LNCS 4762

Kedar S. Namjoshi
Tomohiro Yoneda
Teruo Higashino
Yoshio Okamura (Eds.)

Automated Technology for Verification and Analysis

5th International Symposium, ATVA 2007
Tokyo, Japan, October 2007
Proceedings

Lecture Notes in Computer Science 4762

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Kedar S. Namjoshi   Tomohiro Yoneda
Teruo Higashino   Yoshio Okamura (Eds.)

Automated Technology for Verification and Analysis

5th International Symposium, ATVA 2007
Tokyo, Japan, October 22-25, 2007
Proceedings

Springer

Volume Editors

Kedar S. Namjoshi
Alcatel-Lucent
Bell Labs
600 Mountain Avenue, Murray Hill, NJ 07974, USA
E-mail: kedar@research.bell-labs.com

Tomohiro Yoneda
National Institute of Informatics
Information Systems Architecture Research Division
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: yoneda@nii.ac.jp

Teruo Higashino
Osaka University
Department of Information Networking
Graduate School of Information Science and Technology
Suita, Osaka 565-0871, Japan
E-mail: higashino@ist.osaka-u.ac.jp

Yoshio Okamura
Semiconductor Technology Academic Research Center (STARC)
17-2, Shin Yokohama 3-chome, Kohoku-ku, Yokohama 222-0033, Japan
E-mail: okamura.yoshio@starc.or.jp

Library of Congress Control Number: 2007937234

CR Subject Classification (1998): B.1.2, B.5.2, B.6, B.7.2, C.2, C.3, D.2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-75595-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75595-1 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12173525 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at ATVA 2007, the 5th International Symposium on Automated Technology for Verification and Analysis, which was held on October 22–25, 2007 at the National Center of Sciences in Tokyo, Japan.

The purpose of ATVA is to promote research on theoretical and practical aspects of automated analysis, verification and synthesis in East Asia by providing a forum for interaction between the regional and the international research communities and industry in the field. The first three ATVA symposia were held in 2003, 2004 and 2005 in Taipei, and ATVA 2006 was held in Beijing.

The program was selected from 88 submitted papers, with 25 countries represented among the authors. Of these submissions, 29 regular papers and 7 short papers were selected for inclusion in the program. In addition, the program included keynote talks and tutorials by Martin Abadi (University of California, Santa Cruz and Microsoft Research), Ken McMillan (Cadence Berkeley Labs), and Moshe Vardi (Rice University), and an invited talk by Atsushi Hasegawa (Renesas Technology). A workshop on Omega-Automata (OMEGA 2007) was organized in connection with the conference.

ATVA 2007 was sponsored by the National Institute of Informatics, the Kayamori Foundation of Information Science Advancement, the Inoue Foundation for Science, and the Telecommunications Advancement Foundation. We are grateful for their support.

We would like to thank the program committee and the reviewers for their hard work and dedication in putting together this program. We would like to thank the Steering Committee for their considerable help with the organization of the conference. We also thank Michihiro Koibuchi for his help with the local arrangements.

October 2007

Kedar Namjoshi
Tomohiro Yoneda
Teruo Higashino
Yoshio Okamura

Conference Organization

General Chairs

Teruo Higashino	Osaka University, Japan
Yoshio Okamura	STARC, Japan

Program Chairs

Kedar S. Namjoshi	Bell Labs, USA
Tomohiro Yoneda	National Institute of Informatics, Japan

Program Committee

Rajeev Alur	University of Pennsylvania
Christel Baier	University of Dresden
Jonathan Billington	University of South Australia
Sung-Deok Cha	Korea Advanced Inst. of Sci. and Techn.
Ching-Tsun Chou	Intel
Jin Song Dong	National University of Singapore
E. Allen Emerson	University of Texas at Austin
Masahiro Fujita	University of Tokyo
Susanne Graf	VERIMAG
Wolfgang Grieskamp	Microsoft Research
Aarti Gupta	NEC Labs America
Teruo Higashino	Osaka University
Kiyoharu Hamaguchi	Osaka University
Moonzoo Kim	KAIST
Orna Kupferman	Hebrew University
Robert P. Kurshan	Cadence
Insup Lee	University of Pennsylvania
Xuandong Li	Nanjing University
Shaoying Liu	Hosei University
Zhiming Liu	IIST/United Nations University
Mila E. Majster-Cederbaum	University of Mannheim
Shin Nakajima	National Institute of Informatics
Akio Nakata	Hiroshima City University
Kedar S. Namjoshi	Bell Labs
Mizuhito Ogawa	JAIST
Olaf Owe	University of Oslo
Doron A. Peled	University of Warwick and Bar Ilan University
Mike Reed	UNU-IIST, Macao

Hiroyuki Seki	NAIST
Xiaoyu Song	Portland State University
Yih-Kuen Tsay	National Taiwan University
Irek Ulidowski	University of Leicester
Bow-Yaw Wang	Academia Sinica
Farn Wang	National Taiwan University
Yi Wang	Uppsala University
Baowen Xu	Southeast University of China
Hsu-Chun Yen	National Taiwan University
Tomohiro Yoneda	National Instituite of Informatics
Shoji Yuen	Nagoya University
Wenhui Zhang	Chinese Academy of Sciences
Lenore Zuck	University of Illinois at Chicago

Steering Committee

E. Allen Emerson	University of Texas at Austin, USA
Oscar H. Ibarra	University of California, Santa Barbara, USA
Insup Lee	University of Pennsylvania, USA
Doron A. Peled	University of Warwick, UK and Bar Ilan University, Israel
Farn Wang	National Taiwan University, Taiwan
Hsu-Chun Yen	National Taiwan University, Taiwan

Referees

Benjamin Aminof
Madhukar Anand
David Arney
Colin Atkinson
Louise Avila
Syed Mahfuzul Aziz
Noomene Ben Henda
Domagoj Babic
Hanene Ben-Abdallah
Armin Biere
Lei Bu
Lin-Zan Cai
Wen-Chin Chan
Yu-Fang Chen
Chunqing Chen
Zhenbang Chen
Chang-beom Choi
Jyotirmoy Deshmukh
Nikhil Dinesh
Johan Dovland
Arvind Easwaran
Sebastian Fischmeister
Felix Freiling
Carsten Fritz
Guy Gallasch
Malay Ganai
Jim Grundy
Yi Hong
Reiko Heckel
Monika Heiner
Nao Hirokawa
Geng-Dian Huang
John Håkansson
Keigo Imai
Franjo Ivancic
Einar Broch Johnsen
Vineet Kahlon
Yuichi Kaji
Yunho Kim
Dmitry Korchemny
Piotr Kosiuczenko
Pavel Krcal
Keiichirou Kusakari
Marcel Kyas
Yuan Fang Li
Guoqiang Li
Nimrod Lilith
Xinxin Liu
Lin Liu
Chi-Jian Luo
Yoad Lustig
Michael J. May
Christoph Minnameier
Van Tang Nguyen
Peter Csaba Olveczky
Geguang Pu
Zvonimir Rakamaric

Roopsha Samanta
Gerardo Schneider
Nishant Sinha
Martin Steffen
Volker Stolz
Ryo Suetsugu
Jun Sun
Yoshiaki Takata
Murali Talupur
Kai-Fu Tang
Ming-Hsien Tsai
Emilio Tuosto
Kazunori Ueda
Thomas Wahl
Chao Wang
Verena Wolf
Rong-Shiun Wu
Cong Yuan
Naijun Zhan
Miaomiao Zhang
Jianhua Zhao

Table of Contents

Invited Talks

Policies and Proofs for Code Auditing 1

Recent Trend in Industry and Expectation to DA Research 15

Toward Property-Driven Abstraction for Heap Manipulating Programs 17

Branching vs. Linear Time: Semantical Perspective 19

Regular Papers

Mind the Shapes: Abstraction Refinement Via Topology Invariants 35

Complete SAT-Based Model Checking for Context-Free Processes 51

Bounded Model Checking of Analog and Mixed-Signal Circuits Using an SMT Solver 66

Model Checking Contracts – A Case Study 82

On the Efficient Computation of the Minimal Coverability Set for Petri Nets 98

Analog/Mixed-Signal Circuit Verification Using Models Generated from Simulation Traces 114

Automatic Merge-Point Detection for Sequential Equivalence Checking of System-Level and RTL Descriptions 129

Proving Termination of Tree Manipulating Programs 145

Symbolic Fault Tree Analysis for Reactive Systems 162

Computing Game Values for Crash Games 177

Timed Control with Observation Based and Stuttering Invariant Strategies 192

Deciding Simulations on Probabilistic Automata 207

Mechanizing the Powerset Construction for Restricted Classes of ω-Automata 223

Verifying Heap-Manipulating Programs in an SMT Framework 237

A Generic Constructive Solution for Concurrent Games with Expressive Constraints on Strategies 253

Distributed Synthesis for Alternating-Time Logics.................... 268

Timeout and Calendar Based Finite State Modeling and Verification of Real-Time Systems 284

Efficient Approximate Verification of Promela Models Via Symmetry Markers 300

Latticed Simulation Relations and Games 316

Providing Evidence of Likely Being on Time: Counterexample Generation for CTMC Model Checking 331

Assertion-Based Proof Checking of Chang-Roberts Leader Election in PVS.. 347

Continuous Petri Nets: Expressive Power and Decidability Issues 362

Quantifying the Discord: Order Discrepancies in Message Sequence Charts 378

A Formal Methodology to Test Complex Heterogeneous Systems 394

A New Approach to Bounded Model Checking for Branching Time Logics 410

Exact State Set Representations in the Verification of Linear Hybrid Systems with Large Discrete State Space 425

A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains 441

3-Valued Circuit SAT for STE with Automatic Refinement 457

Bounded Synthesis 474

Short Papers

Formal Modeling and Verification of High-Availability Protocol for Network Security Appliances 489

A Brief Introduction to $\mathcal{THOTL}$ 501

On-the-Fly Model Checking of Fair Non-repudiation Protocols 511

Model Checking Bounded Prioritized Time Petri Nets 523

Using Patterns and Composite Propositions to Automate the Generation of LTL Specifications.. 533

Pruning State Spaces with Extended Beam Search 543

Using Counterexample Analysis to Minimize the Number of Predicates for Predicate Abstraction 553

Author Index 565

Policies and Proofs for Code Auditing

Nathan Whitehead[1], Jordan Johnson[1], and Martín Abadi[1,2]

[1] University of California, Santa Cruz
[2] Microsoft Research

Abstract. Both proofs and trust relations play a role in security decisions, in particular in determining whether to execute a piece of code. We have developed a language, called BCIC, for policies that combine proofs and trusted assertions about code. In this paper, using BCIC, we suggest an approach to code auditing that bases auditing decisions on logical policies and tools.

1 Introduction

Deciding to execute a piece of software can have substantial security implications. Accordingly, a variety of criteria and techniques have been proposed and deployed for making such decisions. These include the use of digital signatures (as in ActiveX [12]) and of code analysis (as in typed low-level languages [5, 9, 10]). The digital signatures can be the basis of practical policies that reflect trust relations—for instance, the trust in certain software authors or distributors. The code analysis can lead to proofs, and thereby to proof-carrying code [11]. Unfortunately, neither trust relations nor proofs are typically sufficient on their own. Trust can be wrong, and code analysis is seldom comprehensive.

We are developing a system for defining and evaluating policies that combine proofs and trusted assertions about code [18, 19, 20]. The core of the system is a logical query language, called BCIC. BCIC is a combination of Binder [4], a logic-programming language for security policies in distributed systems, with Coq's Calculus of Inductive Constructions (CIC) [3], a general-purpose proof framework.

Whereas the focus of most previous work (including our own) is on the decision to execute pieces of code, similar considerations arise in other situations. For instance, from a security perspective, installing a piece of code can be much like executing it. Further upstream, auditing code is also critical to security. Auditing can complement other techniques for assurance, in the course of software production or at various times before execution. Although humans perform the auditing, they are often guided by policies (e.g., what aspects of the code should be audited) and sometime supported by tools (e.g., for focusing attention on questionable parts of the code).

In this paper, using BCIC, we suggest an approach to code auditing that bases auditing decisions on logical policies and tools. Specifically, we suggest that policies for auditing may be expressed in BCIC and evaluated by logical means. Thus, this approach leverages trust relations and proofs, but it also allows

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 1–14, 2007.
© Springer-Verlag Berlin Heidelberg 2007

auditing to complement them. We recognize that this approach is still theoretical and probably incomplete. Nevertheless, it emphasizes the possibility of looking at techniques for verification and analysis in the context of policy-driven systems, and the attractiveness of doing so in a logical setting.

We present two small examples. The first example concerns operating system calls from an application extension language. With a BCIC policy, every operating system call must be authorized by an audit. A policy rule can allow entire classes of calls without separate digital signatures from an authority. In the second example, we consider an information-flow type system [14], specifically a type system that tracks trust (much like Perl's taint mode [6], but statically) due to Ørbæk and Palsberg [13]. The type system includes a form of declassification, in which expressions can be coerced to be trusted (that is, untainted). If any program could use declassification indiscriminately, then the type system would provide no benefit. With a BCIC policy, a trusted authority must authorize each declassification. In both examples, security decisions can rely on nuanced, fine-grained combinations of reason and authority.

We treat these examples in Sections 2 and 3, respectively. We consider implementation details in Section 4. We conclude in Section 5.

2 Example: Auditing Function Calls

In this example we consider the calling behavior of programs in a managed environment of libraries. A base application may allow extensions that provide additional functionality not only to the user but also to other extensions. The extensions may come from many different sources, and accordingly they may be trusted to varying extents. By constraining calls, the security policy can selectively allow different functionality to different extensions.

2.1 Language

For simplicity, we study an interpreted extension language. Specifically, we use an untyped λ-calculus with a special `call` construct that represents operating system calls and calls to other libraries. All calls take exactly one argument, which they may ignore. In order to allow primitive data types, we also include a representation for data constructors (`constr0`, `constr1`, and `constr2`, for constructors that take zero, one, and two arguments respectively). These constructors are enough to handle all the data types that appear in our implementation, including natural numbers, pairs, and lists. Destructors have no special syntax, but are included among the calls.

In Coq notation [2, 3, 16], the syntax of the language is:

```
Inductive exp : Set :=
| var : nat -> exp
| abs : exp -> exp
| app : exp -> exp -> exp
```

```
| call : funcname -> exp -> exp
| constr0 : constrname -> exp
| constr1 : constrname -> exp -> exp
| constr2 : constrname -> exp -> exp -> exp.
```

A detailed knowledge of Coq is not required for understanding this and other definitions in this paper. This definition introduces a class of expressions, inductively by cases with a type for each case; expressions rely on De Bruijn notation, so variables are numbered and binding occurrences of variables are unnecessary. Similarly, other definitions introduce other classes of expressions and propositions, and some parameters for them.

A policy can decide which calls any piece of code may execute. The policy can be expressed in terms of a parameter `audit_maycall`.

```
Parameter audit_maycall : exp -> funcname -> Prop.
```

According to this type, every audit requirement mentions the entire program `exp` that is the context of the audit. Mentioning a subexpression in isolation would not always be satisfactory, and it may be dangerous, as the effects of a subexpression depend on context. The audit requirement also mentions the name of the function being called. We omit any restrictions on the arguments to the function, in order to make static reasoning easier; we assume that the callee does its own checking of arguments. (Section 3 says more on going further with static analysis.)

The predicate `audited_calls` indicates that a piece of code has permission to make all the calls that it could make. This predicate is defined inductively by:

```
Inductive audited_calls : exp -> exp -> Prop :=
| audited_calls_var :
  forall e n,
    audited_calls e (var n)
| audited_calls_app :
  forall e e1 e2,
    audited_calls e e1 ->
    audited_calls e e2 ->
    audited_calls e (app e1 e2)
| audited_calls_abs :
  forall e e1,
    audited_calls e e1 ->
    audited_calls e (abs e1)
| audited_calls_call :
  forall f e e1,
    audited_calls e e1 ->
    audit_maycall e f ->
    audited_calls e (call f e1)
```

```
| audited_calls_constr0 :
  forall e cn,
    audited_calls e (constr0 cn)
| audited_calls_constr1 :
  forall e cn e1,
    audited_calls e e1 ->
    audited_calls e (constr1 cn e1)
| audited_calls_constr2 :
  forall e cn e1 e2,
    audited_calls e e1 ->
    audited_calls e e2 ->
    audited_calls e (constr2 cn e1 e2).
```

Crucially, the case of `audited_calls_call` includes an `audit_maycall` requirement. Every `call` statement requires an audit. A code producer may construct a proof that, given the right audits, the code satisfies the predicate `audited_calls`.

2.2 Policy

Audit statements do not have proofs. They are provided by authorities and trusted to be true through BCIC's policies. The policies employ the special predicate `sat` in order to connect Coq statements and assumptions with Binder rules. Basically, `sat(F)` will be derivable in BCIC when Coq formula `F` has a proof that has been imported into the local context.

The policy writer can express trust in statements that have no proof using a new `believe` predicate, and rules such as:

```
believe(F) :- A says believe(F), trusted(A).
```

This rule expresses that, when a trusted authority says to believe something, the entity that evaluates the policy (in other words, the reference monitor) believes it. In this rule, as in Binder, we use logic-programming constructs plus the special construct `says` [1, 8] for representing the statements of principals.

In order to allow reasoning on beliefs, some additional rules are useful:

```
believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
```

Here, the notation `<term>` includes a CIC term within the policy. The notation `~P` within an included CIC term indicates a free variable that is bound in the policy rule. With these rules, all formulas that have proofs are believed, and belief is closed under modus ponens. (A generalization of the second rule to dependent types, of which implication is a special case, might be attractive but is not needed for our examples.)

Using these constructs and rules, a complete policy of a code consumer may say that a program is allowed to run if it has been properly audited. The policy

can specify which principals are trusted for the auditing. In addition, the policy may collect calls into groups, thus authorizing sets of calls with a single statement from a trusted authority. The complete policy may therefore look as follows:

```
trusted(alice).
trusted(B) :- A says trusted(B), trusted(A).

believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
believe(F) :- A says believe(F), trusted(A).

believe(<audit_maycall ~P ~F>) :-
  A says believe(<audit_maycall ~P ~F>), trusted(A).

classify(setuid, dangerous).
classify(setgid, dangerous).
classify(sbrk, userlowlevel).
classify(printf, io).
classify(sprintf, dangerous).

believe(<audit_maycall ~P ~F>) :-
  classify(F, C), A says allowgroup(P, C), trusted(A).

mayrun(P) :- believe(<audited_calls ~P ~P>).
```

Many variants are of course possible. In particular, a policy may let the auditing requirements depend on CIC proofs about intrinsic properties of the code, with clauses of the form `believe(<audit_maycall ...>) :- sat(...)`. Furthermore, a policy may concern not only the decision to run a program but also any requirements for checks during execution.

2.3 Correctness

This example is simple enough that not too much theory is needed, but there are some subtleties (in part because functions can be higher-order, and recursion is possible). The main theorem about our analysis says that, if a piece of code passes the analysis and is executed, every actual call will have been audited (but it does not say anything about the appropriateness of particular BCIC policies such as that of Section 2.2). Proving this theorem requires defining the operational semantics of the language in such a way that a history of calls is kept. We define a state as a pair that consists of a list of calls and an expression, then define the single-step reduction relation $S \rightarrow S'$ in the usual way. The only nonstandard rule is for calls: $(l, \texttt{call}\ f\, v) \rightarrow (f :: l, O(f, v))$, where we let the

expression $O(f, v)$ represent the return value of calling function f with value v. We assume that the returned values do not contain calls themselves. We obtain:

Theorem 1. e, *audited_calls e e* $([], e) \rightarrow^*$ (l, e'), *audit_maycall e f* f l

The proof relies on a lemma that says if a function name f appears in l then f appears in the program e. The proof of the lemma is by induction on reductions, then by case analysis of all the possible reductions. The substitutions that arise from β reductions constitute the only difficulty. The proof is in Coq.

3 Example: Trust in the λ-Calculus

Ørbæk and Palsberg define a simple type system for a λ-calculus with trust annotations [13]. In this system, one may for instance describe applets for a web server; the type system can help protect the applets from malicious inputs and the web server from malicious applets. The language includes a construct `trust` for untainting, thus supporting a form of declassification. (A dual form of declassification is turning secret data into public data.) In this example we study how to impose auditing requirements on declassifications. We started to consider this example in our work on BLF, a preliminary version of BCIC [20,Appendix B]; here we develop it further in BCIC.

3.1 Language

The type system allows a small set of built-in types and function types; in addition, types include the annotations `tr` or `dis`, which indicate trust and distrust, respectively. Data from unknown outside parties, annotated with `dis`, can be treated with the proper suspicion. For instance, `int` is the type of distrusted integers, while (`int` → `int`) is the type of trusted functions that take distrusted integer inputs and return trusted integer results.

Figure 1 defines the types, annotations, and relations between types. Types are defined as `bare_type`, which become `annotated_type` when annotated by `tr` or `dis`. Subtyping is defined straightforwardly for both bare and annotated types. Auxiliary functions `trust_lte` and `join` represent the partial order of the trust annotations and give the greatest lower bound of two trust levels, respectively.

Expressions in the language are similar to those of $\lambda_{<:}$, the simply typed lambda calculus with subtyping, with the addition of `trust`, `distrust`, and `check` expressions. Each `trust` is tagged with an identifier so it can be referenced by an auditor. We made typecasts explicit in both origin and destination types for simplicity in the typechecker. We also included `int` and `bool` as built-in types. Figure 2 shows the definition of expressions and typechecking. As usual, typechecking is done with respect to typing assumptions in a context.

```
Inductive trust_type : Set :=
| tr : trust_type
| dis : trust_type.

Inductive bare_type : Set :=
| usertype : nat -> bare_type
| arrow : annotated_type -> annotated_type -> bare_type

with annotated_type : Set :=
| annote : bare_type -> trust_type -> annotated_type.

Inductive trust_lte : trust_type -> trust_type -> Prop :=
| trust_lte_refl : forall (T : trust_type), trust_lte T T
| trust_lte_trdis : trust_lte tr dis.

Inductive bare_lte : bare_type -> bare_type -> Prop :=
| bare_lte_refl :
  forall (T : bare_type), bare_lte T T
| bare_lte_arrow :
  forall (X Y A B : annotated_type),
    ann_lte Y B -> ann_lte A X ->
    bare_lte (arrow X Y) (arrow A B)

with ann_lte : annotated_type -> annotated_type -> Prop :=
| ann_lte_refl :
  forall (T : annotated_type), ann_lte T T
| ann_lte_annote :
  forall (A B : bare_type)(U V : trust_type),
    bare_lte A B -> trust_lte U V ->
    ann_lte (annote A U) (annote B V).

Definition join (U V : trust_type) : trust_type :=
  match U with
    | tr => V
    | dis => dis
  end.
```

Fig. 1. Types and relations between types

3.2 Policy

By formalizing the type system in Coq, we can write BCIC security policies that rely on type safety according to this type system. A simple example is the following policy:

```
mayrun(C, P) :- sat(<has_type ~C ~P (annote (usertype 0) tr)>)
```

```
Definition trustid : Set := nat.

Inductive expr : Set :=
| const_int : nat -> expr
| const_bool : bool -> expr
| var : nat -> expr
| abs : annotated_type -> expr -> expr
| app : expr -> expr -> expr
| trust : trustid -> expr -> expr
| distrust : expr -> expr
| check : expr -> expr
| cast : expr -> annotated_type -> annotated_type -> expr.

Inductive context : Set :=
| nilctx : context
| cons : annotated_type -> context -> context.

Inductive has_type : context -> expr -> annotated_type -> Prop :=
  | type_int : forall n C,
      has_type C (const_int n) (annote (usertype 0) tr)
  | type_bool : forall p C,
      has_type C (const_bool p) (annote (usertype 1) tr)
  | type_varz : forall T C,
      has_type (cons T C) (var 0) T
  | type_varn : forall n T T2 C,
      has_type C (var n) T ->
      has_type (cons T2 C) (var (S n)) T
  | type_abs : forall C E T T2,
      has_type (cons T C) E T2 ->
      has_type C (abs T E) (annote (arrow T T2) tr)
  | type_app : forall C E1 E2 T1 T2 U V,
      has_type C E1 (annote (arrow T1 (annote T2 U)) V) ->
      has_type C E2 T1 ->
      has_type C (app E1 E2) (annote T2 (join U V))
  | type_trust : forall C E T U id,
      has_type C E (annote T U) ->
      has_type C (trust id E) (annote T tr)
  | type_distrust : forall C E T U,
      has_type C E (annote T U) ->
      has_type C (distrust E) (annote T dis)
  | type_check : forall C E T,
      has_type C E (annote T tr) ->
      has_type C (check E) (annote T tr)
  | type_cast : forall C E T T2,
      has_type C E T ->
      ann_lte T T2 ->
      has_type C (cast E T T2) T2.
```

Fig. 2. Expressions in the language and typechecking rules

This policy says that program `P` may run in context `C` if it typechecks and has the trusted type `int`˙ . This policy does not exclude the possibility that this type is trivially obtained by an application of `trust`, however.

The next step is to require that all applications of the `trust` operator be audited. Since the `trust` operator is basically a way to escape the type system, the value of the type system for security would be questionable at best if any program could use `trust` indiscriminately, hence the desire for auditing.

Figure 3 defines a predicate `trusts_audited`. This predicate recursively traverses an expression while remembering the entire expression. In this definition, occurrences of `trust` impose extra requirements. In particular, every occurrence of `trust` must have a corresponding `audit` statement. The `audit` predicate is defined to be satisfied externally, in our case by assertions in the policy rather than by proofs.

```
Parameter audit : expr -> trustid -> Prop.
```

An expression `E` has all its `trusts` audited when `trusts_audited E E` holds.

As a small example, consider the context `C` that has two elements, a distrusted function `f` of type (`int`˙ $\rightarrow$ `int`˙)˙˙ and a trusted integer `x`. The expression `(trust`^{0} `f) x` (which is actually `app (trust 0 (var 0)) (var 1)` in our syntax but which we write as `(trust`^{0} `f) x` for simplicity) will pass `trusts_audited` when there is an audit statement `audit ((trust`^{0} `f) x) 0`. Suppose that the code producer `bob` provides the code `(trust`^{0} `f) x` and a proof of:

```
audit ((trust^0 f) x) 0
  ->
trusts_audited ((trust^0 f) x) ((trust^0 f) x)
```

A trusted authority `alice` signs the statement:

```
believe(<audit ((trust^0 f) x) 0>)
```

The policy of a code consumer may be:

```
trusted(alice).

believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
believe(F) :- A says believe(F), trusted(A).

mayrun(C, P) :-
  believe(<has_type ~C ~P (annote (usertype 0) tr)>),
  believe(<trusts_audited ~P ~P>).
```

```
Inductive trusts_audited : expr -> expr -> Prop :=
  | audited_int :
    forall P n,
      trusts_audited P (const_int n)
  | audited_bool :
    forall P b,
      trusts_audited P (const_bool b)
  | audited_var :
    forall P n,
      trusts_audited P (var n)
  | audited_abs :
    forall P T E,
      trusts_audited P E ->
      trusts_audited P (abs T E)
  | audited_app :
    forall P E1 E2,
      trusts_audited P E1 ->
      trusts_audited P E2 ->
      trusts_audited P (app E1 E2)
  | audited_trust :
    forall P E id,
      audit P id ->
      trusts_audited P E ->
      trusts_audited P (trust id E)
  | audited_distrust :
    forall P E,
      trusts_audited P E ->
      trusts_audited P (distrust E)
  | audited_check :
    forall P E,
      trusts_audited P E ->
      trusts_audited P (check E)
  | audited_cast :
    forall P E T1 T2,
      trusts_audited P E ->
      trusts_audited P (cast E T1 T2).
```

Fig. 3. Auditing trusts predicate

After importing `alice`'s statement, the code consumer will have:

$$\texttt{alice says believe(<audit ((trust}^{\mathit{0}}\texttt{ f) x) 0>)}$$

After checking `bob`'s proof, it will have:

$$\texttt{sat}\left(\begin{array}{l}\texttt{<audit ((trust}^{\mathit{0}}\texttt{ f) x) 0}\\ \quad\texttt{->}\\ \texttt{trusts_audited ((trust}^{\mathit{0}}\texttt{ f) x) ((trust}^{\mathit{0}}\texttt{ f) x)>}\end{array}\right)$$

With a little reasoning, the code consumer obtains:

$$\texttt{mayrun}\left(\texttt{<(int}^{tr} \rightarrow \texttt{int}^{tr})^{dis} \texttt{ :: int}^{tr} \texttt{ :: nil>, <(trust}^{0} \texttt{ f) x>}\right)$$

Going further, as in Section 2, we have much flexibility in defining policies. In particular, each audit requirement need not be discharged with an explicit assertion specific to the requirement. We can allow broader assertions. In the extreme case, once a trusted authority vouches for a program, no auditing is required. We can express this policy with a BCIC rule:

```
mayrun(C, P) :- A says vouchfor(C, P), trusted(A).
```

We can also express policies that distinguish users. First, we modify `mayrun` to include an explicit user argument: `mayrun(U, C, P)` is satisfied when user `U` is allowed to run program `P` in context `C`. Since not all users might trust the same auditors and other authorities, we also introduce predicates `believes` and `trusts` as generalizations of `believe` and `trusted`, respectively, with user arguments. Because provability is absolute, not relative to particular users, no user annotation is needed for `sat`. With these variants, we may for example write the policy:

```
trusts(bob, alice).
trusts(charlie, bob).

believes(U, F) :- sat(F).
believes(U, Q) :- believes(U, P), believes(U, <~P -> ~Q>).
believes(U, F) :- A says believe(F), trusts(U, A).

mayrun(U, C, P) :-
  believes(U, <has_type ~C ~P (annote (usertype 0) tr)>),
  believes(U, <trusts_audited ~P ~P>).
```

However, some classes of policies require more advanced techniques. For instance, we may want to focus the auditing on expressions of certain types, or to exclude expressions that satisfy a given static condition established by a particular static-analysis algorithm. While static conditions can certainly be programmed as Coq predicates, once those predicates are present there need to be corresponding proofs.

Fortunately we have a way of encoding decision procedures in Coq signatures in order to allow BCIC to incorporate those decisions procedures. Our approach is based on proof by reflection (also known as proof by computation) [2, 7], a technique applicable to any theorem prover based on typing that includes convertibility to normal forms. The idea of this technique is that details related to computations are elided from proof terms themselves, and are instead handled automatically by the conversion rules. Using this technique, BCIC can integrate various static analyses. (Some details of our use of proofs by reflection appear in another document [17].)

3.3 Correctness

Much as in Section 2, the proof of correctness relies on an instrumented operational semantics for the language. In this semantics, annotations record `trust` operations. We define a state as a pair that consists of a list of trust identifiers, l, and an expression, e. We define the single-step reduction $(l, e) \rightarrow (l', e')$ mostly as usual, ignoring types and type annotations, and eliminating casts, `trusts`, `distrusts`, and `checks`. The rule for `trust` records the identifier of the trust: $(l, \texttt{trust}^{\texttt{id}}\ e) \rightarrow (\texttt{id} :: l, e)$. We obtain a correctness result that says that, when programs execute, they perform `trust` operations only as authorized:

Theorem 2. *e* *trusts_audited e e* $([], e) \rightarrow^*$ (l, e') *audit e id* *id* *l*

While helpful, this theorem does not aim to address the proper criteria for declassification. The study of those criteria, and the guarantees that they may offer, is an active research area (e.g., [15]).

4 Implementation

Our implementation of BCIC includes network communication, cryptographic primitives, a simple user interface, and the machinery for logical queries on policies [19]. We have used Coq's automatic program extraction from proofs to produce a certified correct implementation of the BCIC logic engine [18]. The examples we have presented in this paper are written and work in our implementation of BCIC.

In order to flesh out the examples, we have created corresponding proof generators. Given a target program, the proof generators analyze the program and construct Coq proof terms for it. For the example of Section 2, proof terms establish that strings of audit requirements imply `calls_audited` properties. For the example of Section 3, proof terms yield not only `trusts_audited` properties but also well-typing.

Further, in order to execute the programs that have been analyzed, we have created a simple run-time environment, basically a λ-calculus interpreter. During network communication, statements and proofs must be transmitted between principals. Given that audit statements include copies of programs, performance could be problematic. Therefore, we replace CIC terms with their hashes. Thus, a set of statements about a single program will need to transmit the program only once, not once per statement. Additional performance improvements might be obtained by caching.

Despite poor, exponential bounds, queries in the certified implementation are reasonably fast in practice. Most policies we have studied require only one or two steps of reasoning before all possible conclusions have been drawn and a query can be answered. In practice the biggest bottleneck is typechecking terms in Coq, not logical deduction. Our implementation caches calls to Coq in order to avoid redundant typechecking.

The code samples in our implementation come from Scheme programs that do not use many Scheme language constructs. Going further, one may attempt to incorporate more of the Scheme language. For the example of Section 3, which requires static typing, a language such as ML may be a better match than Scheme. In this context, auditing may also be applied to special language features that allow a program to break the type system in a potentially unsafe way (e.g., [illegible] in OCaml).

5 Conclusion

This paper develops the view that an audit assertion may encapsulate a non-formal judgment about code in the context of a formal reasoning system. While this judgment will typically be made by a human, perhaps incorrectly, the formal reasoning system provides the means to specify how it fits with security policies and with static code analysis. The formal reasoning system can thus guide the auditing work and ensure its completeness.

More broadly, our work with BCIC indicates the possibility of looking at techniques for verification and analysis in the context of logical security policies. Combinations of reason and authority arise in practice, for instance in systems with signed and verified code (e.g., [5]), though often with ad hoc policies and mechanisms. General theories and tools should support such systems.

[illegible] We are grateful to Andrei Sabelfeld and Steve Zdancewic for comments on this work. This work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078.

References

1. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. ACM Transactions on Programming Languages and Systems 15(4), 706–734 (1993)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
3. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76(2/3), 95–120 (1988)
4. De Treville, J.: Binder, a logic-based security language. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 105–113 (2002)
5. ECMA. C# and common language infrastructure standards,(2007), Online at http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx
6. Perl Foundation. Perl 5.8.8 documentation: perlsec - Perl security. Online at http://perldoc.perl.org/perlsec.html
7. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
8. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)

9. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification. Addison-Wesley, Reading (1997)
10. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems 21(3), 528–569 (1999)
11. Necula, G.C.: Proof-carrying code. In: POPL 1997. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, pp. 106–119. ACM Press, New York (1997)
12. Microsoft Developer Network. About ActiveX controls,(2007), Online at http://msdn2.microsoft.com/en-us/library/Aa751971.aspx
13. Ørbæk, P., Palsberg, J.: Trust in the λ-calculus. Journal of Functional Programming 7(6), 557–591 (1997)
14. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
15. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. In: CSFW 2005. Proceedings of the 18th IEEE Workshop on Computer Security Foundations, pp. 255–269 (2005)
16. The Coq Development Team. The Coq proof assistant. http://coq.inria.fr/
17. Whitehead, N.: Towards static analysis in a logic for code authorization. (Manuscript)
18. Whitehead, N.: A certified distributed security logic for authorizing code. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 253–268. Springer, Heidelberg (2007)
19. Whitehead, N., Abadi, M.: BCiC: A system for code authentication and verification. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 110–124. Springer, Heidelberg (2005)
20. Whitehead, N., Abadi, M., Necula, G.: By reason and authority: A system for authorization of proof-carrying code. In: CSFW 2004. Proceedings of the 17th IEEE Computer Security Foundations Workshop, pp. 236–250 (2004)

Recent Trend in Industry and Expectation to DA Research

Atsushi Hasegawa

Renesas Technology Corp., System Solution Business Group, System Core Technology Div.
5-20-1 Jousui Hon-cho, Kodaira Tokyo 187-8588, Japan
hasegawa.atsushi@renesas.com

1 History of Semiconductor Design

In 1970s and early 1980s, functional verification in semiconductor design mean gate level logic simulation. Equivalence check between logic circuits and layout design had been done with human eyes and colored pencils. Layout rule checkers helped to point out layout errors, ex. spacing rule errors.

Progress of design tools and methodology have been helped to increase numbers of transistors on silicon chips, following Moore's law steadily. Improvement of productivity allowed us to tackle larger design problem. Verification and test tools and methodology played important role in such improvement of productivity.

In the industry, the progress of design methodology was slower than academic research of DA technology. But the daily works of engineers are totally different from 1970s and 1980s.

In late 1980s, schematic editors and workstations released engineers from pencils and erasers to draw schematics, removed operator jobs to type netlists based on hand written schematics. Eliminating human transformation from hand written schematics to text base netlists reducing bugs dramatically. In both cases, design teams applied functional verification on the netlists. Schematic editors reducing numbers of finding typo in verification.

Late 80s and early 90s, there were many tools and notion forms to describe functionality of design blocks. Those were aiming to reduce writing efforts for same functionality, such as numbers of lines to define the behavior of blocks. The notion forms were also aiming easy understanding of design, boolean formulas, functional tables, state machine notation in several different format.

Finally two major description languages Verilog and VHDL became popular in semiconductor industry. Great progress of synthesis technology makes design result using those languages became faster, smaller, and reliable compare to design result using ordinal handcrafted gate level schematics. The early version of synthesis tools had poor performance. The generated gate netlists were slower and had more gates. Several years later, the synthesis tools became more mature. Our company shifted from proprietary synthesis tools using restricted C language to commercial Verilog language synthesis tools in mid 90s.

For functional validation, we did RTL level simulation and gate level simulation. Gate level simulation also checked timing with extracted gate netlists and parasitical parameters from layout data just before tape out.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 15–16, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Recent Design Flow

Today's design flow starts design activity to create C model of IPs based on the target specification. Then start to write test suites and apply it to C model to validate C model. Later this C model is used as reference model compare against developing Verilog RTL description. In equivalent test bench environment, test suites are applied to compare simulation result of C model and Verilog RTL. Test coverage tools improve test suites to reduce validation holes Random test generators complement directed test suites and eliminates further validation holes.

Synthesis tools are applied on Verilog RTL with timing constraints to generate gate netlist. Verilog RTL and generated gate netlist are compared with formal verification tools to validate functional equivalence. Static timing analysis tool checks generated netlist with timing constraints to assure the netlist meats timing requirements.

After this validation backend design flow are applied on the validated netlist to place and route cells and wires to create chip level layout data. Applying layout check tools, electric characteristic check tools and rule decks on the layout data to validate against layout rules and electric design rules. Also from layout data, an extraction tool generates gate netlist with parasitical data. The formal equivalence check tools validate extracted gate level netlist against synthesized gate level netlist. Static timing tools assure the layout data meat target timing requirement. Power evaluation tools check power consumption of layout data.

On the each phase of design, tools are used to transform input date to output data for next design phase. After transformation, one tool check functional equivalence of input and output data. Other tools check output data meats timing, power consumption and other design criteria.

Without DA tools we may not develop large scale SoC chip within meaningful time period.

2.1 Expectation to Future DA Research

In these years, many attractive DA methodologies are proposed. Theoretically most of those DA methodologies are workable. But applying those methodologies to actual design flow, we need to wait invention of technologies shorten execution time of tools, relaxing memory size requirements and other difficult factor to run tools on affordable computer resources.

There are several strong expectation to improve design productivity of hardware. Other expectations are improving software development productivity, shorten software and system development schedule, and increase reliability of software and system. Here are some examples:

Ten to hundred times productive (highly abstracted) hardware description language or method.

New language or tool to describe system, hardware, and software specification. Executable specification will effect productivity and reliability.

Provide reasonably high performance software execution (simulation) environment to start software debugging before silicon tape out.

Toward Property-Driven Abstraction for Heap Manipulating Programs

K.L. McMillan

Cadence Berkeley Labs

Automated abstraction refinement methods have shown significant promise in analyzing low-level software, such as operating system device drivers, and other control-oriented codes. For example, the SLAM toolkit from Microsoft research [1] has proved effective in finding control errors (such as illegal use of kernel API functions) in real-world device driver codes. SLAM is based on predicate abstraction, using a counterexample-based abstraction refinement heuristic. This gives it the ability to focus the abstraction on state predicates that are relevant to the proof (or falsification) of a given property. This ability allows SLAM and similar tools to scale to real codes of moderate size, albeit only in the case when the property is fairly shallow, in the sense that it requires only a small amount of information about the program's state to prove it.

Predicate abstraction, however, has some important limitations, related to the restrictions it places on the language of inductive invariants that can be derived. Since it is limited to quantifier-free formulas, it cannot synthesize invariants relating to heap data structures and arrays. For example, it cannot synthesize facts such as "x points to an acyclic, linked list", or "every cell in array a satisfies property p". This is a significant limitation: studies have shown that a majority of operating systems code bugs and reported failures are related to memory management [2,8]. Without the ability to reason about heap data structures and arrays, we cannot expect to find and correct such errors, except in the simplest cases. Thus, a substantial contributor to unreliability of operating systems cannot be fully addressed.

This is not to say that the problem of automatic verification of heap data structures has not been extensively studied. A variety of static analysis methods, based for example on abstract shape graphs [7] and separation logic [3,6], are able to verify fairly strong properties of such programs. For example, the tool TVLA can verify correctness of sorting algorithms on linked lists. However, these methods have not scaled well to large programs. A significant reason for this is that there is no available method to focus the abstractions on facts relevant to proving a given property. Thus, for large programs, the abstractions will carry much information about the heap that is not relevant, leading to an explosion of abstract shape graphs (or correspondingly of separation logic formulas). In particular, an explosion of cases can be caused by carrying irrelevant correlations between different parts of the heap. On the other hand, if relevant correlations are lost by using a weak merge operator, we may fail to prove the property.

The solution to this problem is not immediately evident. Counterexample-driven abstraction refinement methods, used in predicate abstraction tools such

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 17–18, 2007.
© Springer-Verlag Berlin Heidelberg 2007

as SLAM, are typically based on propagating the weakest precondition operator. For heap-manipulating programs, these are likely to diverge, producing an infinite sequence of refinements. Ad-hoc widenings may solve this problem, but may also fail to prove the property. Moreover, it is less clear in the case of shape abstractions how to form an appropriate Cartesian decomposition of the abstract domain to avoid carrying unnecessary correlations, and how to refine this decomposition.

In this talk, I will consider how Craig interpolation methods [5] might be applied to theses problems. These methods are applied, for example, in the BLAST model checker for predicate abstraction refinement [4]. They allow us to exploit the relevance heuristics in a theorem prover or decision procedure to focus refinements on relevant facts at given program locations. Moreover, by using a suitably restricted prover, we can prevent the divergence of refinements that occurs in methods based on weakest preconditions.

The primary challenge in the approach is to handle the richer logical framework needed to express properties of heaps. In particular, we must be able to handle quantifiers (to reason about heaps of unbounded size) and reachability of linked structures (to express properties such as acyclicity). Moreover, we must be able to restrict the quantifier structure of interpolants to prevent divergence. The talk will consider potential solutions to these problems, and show how interpolation in first-order logic with transitive closure might be used for shape abstraction refinement in verifying heap-manipulating programs. The ability to focus such abstraction on facts relevant to a given property may in turn allow the methods to scale up to programs of practical size.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
2. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.R.: An empirical study of operating system errors. In: Symposium on Operating Systems Principles, pp. 73–88 (2001)
3. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
4. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Neil, D., Jones, N.D., Leroy, X. (eds.) POPL, pp. 232–244. ACM Press, New York (2004)
5. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
6. Nanevski, A., Magill, S., Clarke, E., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs. In: Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE) (2006)
7. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
8. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability - a study of field failures in operating systems. In: 21st Int. Symp. on Fault-Tolerant Computing (FTCS-21), pp. 2–9 (1991)

Branching vs. Linear Time: Semantical Perspective*

Sumit Nain and Moshe Y. Vardi

Rice University, Department of Computer Science, Houston, TX 77005-1892, USA

Abstract. The discussion in the computer-science literature of the relative merits of linear- versus branching-time frameworks goes back to early 1980s. One of the beliefs dominating this discussion has been that the linear-time framework is not expressive enough semantically, making linear-time logics lacking in expressiveness. In this work we examine the branching-linear issue from the perspective of process equivalence, which is one of the most fundamental notions in concurrency theory, as defining a notion of process equivalence essentially amounts to defining semantics for processes. Over the last three decades numerous notions of process equivalence have been proposed. Researchers in this area do not anymore try to identify the "right" notion of equivalence. Rather, focus has shifted to providing taxonomic frameworks, such as "the linear-branching spectrum", for the many proposed notions and trying to determine suitability for different applications.

We revisit here this issue from a fresh perspective. We postulate three principles that we view as fundamental to any discussion of process equivalence. First, we borrow from research in denotational semantics and take observational equivalence as the primary notion of equivalence. This eliminates many testing scenarios as either too strong or too weea. Second, we require the description of a process to fully specify all relevant behavioral aspects of the process. Finally, we require observable process behavior to be reflected in its input/output behavior. Under these postulates the distinctions between the linear and branching semantics tend to evaporate. As an example, we apply these principles to the framework of transducers, a classical notion of state-based processes that dates back to the 1950s and is well suited to hardware modeling. We show that our postulates result in a unique notion of process equivalence, which is trace based, rather than tree based.

1 Introduction

One of the most significant recent developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [17,39,51,59]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired property by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this property (see [19]). Model-checking tools have enjoyed a substantial and growing use over the last few years, showing ability to discover subtle flaws that result from extremely improbable events. While early on these tools were viewed as of academic interest only, they are now routinely used in industrial applications [29].

* Work supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, CCF-0613889, and ANI-0216467, by BSF grant 9800096, and by gift from Intel.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 19–34, 2007.
© Springer-Verlag Berlin Heidelberg 2007

A key issue in the design of a model-checking tool is the choice of the temporal language used to specify properties, as this language, which we refer to as the *temporal property-specification language*, is one of the primary interfaces to the tool. (The other primary interface is the modeling language, which is typically the hardware description language used by the designers). One of the major aspects of all temporal languages is their underlying model of time. Two possible views regarding the nature of time induce two types of temporal logics [38]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing the behavior of a single computation of a program. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program.

In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connectives G ("always"), F ("eventually"), X ("next"), and U ("until"). The branching temporal logic CTL* augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. Note that LTL has implicit universal path quantifiers in front of its formulas. Thus, LTL is essentially the linear fragment of CTL*.

The discussion of the relative merits of linear versus branching temporal logics in the context of system specification and verification goes back to 1980 [49,38,23,6,50,25, 24,53,16,14,56,57]. As analyzed in [50], linear and branching time logics correspond to two distinct views of time. It is not surprising therefore that LTL and CTL are expressively incomparable [16,24,38]. The LTL formula FGp is not expressible in CTL, while the CTL formula $AFAGp$ is not expressible in LTL. On the other hand, CTL seems to be superior to LTL when it comes to algorithmic verification, as we now explain.

Given a transition system M and a linear temporal logic formula φ, the model-checking problem for M and φ is to decide whether φ holds in all the computations of M. When φ is a branching temporal logic formula, the problem is to decide whether φ holds in the computation tree of M. The complexity of model checking for both linear and branching temporal logics is well understood: suppose we are given a transition system of size n and a temporal logic formula of size m. For the branching temporal logic CTL, model-checking algorithms run in time $O(nm)$ [17], while, for the linear temporal logic LTL, model-checking algorithms run in time $n2^{O(m)}$ [39]. Since LTL model checking is PSPACE-complete [52], the latter bound probably cannot be improved.

The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking made CTL a popular choice, leading to efficient model-checking tools for this logic [18]. Through the 1990s, the dominant temporal specification language in industrial use was CTL. This dominance stemmed from the phenomenal success of SMV, the first symbolic model

checker, which was CTL-based, and its follower VIS, also originally CTL-based, which served as the basis for many industrial model checkers.

In [58] we argued that in spite of the phenomenal success of CTL-based model checking, CTL suffers from several fundamental limitations as a temporal property-specification language, all stemming from the fact that CTL is a branching-time formalism: the language is unintuitive and hard to use, it does not lend itself to compositional reasoning, and it is fundamentally incompatible with semi-formal verification. In contrast, the linear-time framework is expressive and intuitive, supports compositional reasoning and semi-formal verification, and is amenable to combining enumerative and symbolic search methods. Indeed, the trend in the industry during this decade has been towards linear-time languages, such as ForSpec [4], PSL [22], and SVA [60].

In spite of the pragmatic arguments in favor of the linear-time approach, one still hears the arguments that this approach is not expressive enough, pointing out that in semantical analyses of concurrent processes, e.g., [28], the linear-time approach is considered to be the weakest semantically. In this paper we address the semantical arguments against linear time and argue that even from a semantical perspective the linear-time approach is quite adequate for specifying systems.

The gist of our argument is that branching-time-based notions of process equivalence are *not* reasonable notions of process equivalence, as they distinguish between processes that are not observationally distinguishable. In contrast, the linear-time view does yield an appropriate notion of observational equivalence.

2 The Basic Argument Against Linear Time

The most fundamental approach to the semantics of programs focuses on the notion of equivalence. Once we have defined a notion of equivalence, the semantics of a program can be taken to be its equivalence class. In the context of concurrency, we talk about process equivalence. The study of process equivalence provides the basic foundation for any theory of concurrency [46], and it occupies a central place in concurrency-theory research, cf. [28].

The linear-time approach to process equivalence focuses on the traces of a process. Two processes are defined to be *trace equivalent* if they have the same set of traces. It is widely accepted in concurrency theory, however, that trace equivalence is too weak a notion of equivalence, as processes that are trace equivalent may behave differently in the same context [45]. An an example, using CSP notation, the two processes

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x)\Box(\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

$$\mathbf{if}(a?x \rightarrow h!x)\Box(b?x \rightarrow h!x)\mathbf{fi}$$

have the same set of communication traces, but only the first one may deadlock when run in parallel with a process such as $b!0$.

In contrast, it is known that CTL characterizes *bisimulation*, in the sense that two states in a transition system are bisimilar iff they satisfy exactly the same CTL formulas [13] (see also [34]), and bisimulation is a highly popular notion of equivalence between processes [46,48,54].

This contrast, between the pragmatic arguments in favor of the adequate expressiveness of the linear-time approach [58] and its accepted weakness from a process-equivalence perspective, calls for a re-examination of process-equivalence theory.

3 Process Equivalence Revisited

While the study of process equivalence occupies a central place in concurrency-theory research, the answers yielded by that study leave one with an uneasy feeling. Rather than providing a definitive answer, this study yields a plethora of choices [3]. This situation led to statement of the form “It is not the task of process theory to find the ‘true’ semantics of processes, but rather to determine which process semantics is suitable for which applications” [28]. This situation should be contrasted with the corresponding one in the study of sequential-program equivalence. It is widely accepted that two programs are equivalent if they behave the same in all contexts, this is referred to as *contextual* or *observational* equivalence, where behavior refers to input/output behavior [61]. In principle, the same idea applies to processes: two processes are equivalent if they pass the same tests, but there is no agreement on what a test is and on what it means to pass a test.

We propose to adopt for process-semantics theory precisely the same principles accepted in program-semantics theory.

Principle of Contextual Equivalence: Two processes are equivalent if they behave the same in all contexts, which are processes with “holes”.

As in program semantics, a context should be taken to mean a process with a “hole”, into which the processes under consideration can be “plugged”. This agrees with the point of view taken in *testing equivalence*, which asserts that tests applied to processes need to themselves be defined as processes [20]. Furthermore, all tests defined as processes should be considered. This excludes many of the “button-pushing experiments” of [45]. Some of these experiments are too strong–they cannot be defined as processes, and some are too weak–they consider only a small family of tests [20].

In particular, the tests required to define bisimulation equivalence [2,45] are widely known to be too strong [8,9,10,30]. In spite of its mathematical elegance [54], bisimulation is *not* a reasonable notion of process equivalence, as it makes distinctions that cannot be observed. Bisimulation is a structural similarity relation between states of the processes under comparison, rather than a behavioral relation. Expecting an implementation to be bisimilar to a specification is highly unrealistic, as it requires the implementation to be too similar structurally to the specification. From this point of view, the term “observational equivalence” for weak bisimulation equivalence in [45] is perhaps unfortunate.

Remark 1. One could argue that bisimulation equivalence is not only a mathematically elegant concept; it also serves as the basis for useful sound proof techniques for establishing process equivalence, cf. [34]. The argument here, however, is not against bisimulation as a useful mathematical concept; such usefulness ought to be evaluated on its own merits, cf. [27]. Rather, the argument is against viewing bisimulation-based notions of equivalence as reasonable notions of process equivalence.

The Principle of Contextual Equivalence does not fully resolve the question of process equivalence. In additional to defining the tests to which we subject processes, we need to define the observed behavior of the tested processes. It is widely accepted, however, that linear-time semantics results in important behavioral aspects, such as deadlocks and livelocks, being non-observable [45]. It is this point that contrasts sharply with the experience that led to the adoption of linear time in the context of hardware model checking [58]; in today's synchronous hardware all relevant behavior, including deadlock and livelock is observable (observing livelock requires the consideration of infinite traces). Compare this with our earlier example, where the process

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x)\Box(\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

may deadlock when run in parallel with a process such as $b!0$. The problem here is that the description of the process does not tell us what happens when the first guard is selected in the context of the parallel process $b!0$. The deadlock here is not described explicitly; rather it is implicitly inferred from a lack of specified behavior. This leads us to our second principle.

Principle of Comprehensive Modeling: A process description should model all relevant aspects of process behavior.

The rationale for this principle is that relevant behavior, where relevance depends on the application at hand, should be captured by the description of the process, rather than inferred from lack of behavior by a semantical theory proposed by a concurrency theorist. It is the usage of inference to attribute behavior that opens the door to numerous interpretations, and, consequently, to numerous notions of process equivalence.

Remark 2. It is useful to draw an analogy here to another theory, that of *nonmonotonic logic*, whose main focus is on inferences from absence of premises. The field started with some highly influential papers, advocating, for example "negation as failure" [15] and "circumscription" [43]. Today, however, there is a plethora of approaches to nonmonotonic logic, including numerous extensions to negation as failure and to circumscription [42]. One is forced to conclude that there is no universally accepted way to draw conclusions from absence of premises. (Compare also to the discussion of negative premises in transition-system specifications [10,30].)

Going back to our problematic process

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x)\Box(\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

The problem here is that the process is not *receptive* to communication on channel b, when it is in the left branch. The position that processes need to be receptive to all allowed inputs from their environment has been argued by many authors [1,21,40]. It can be viewed as an instance of our Principle of Comprehensive Modeling, which says that the behavior that results from a write action on channel b when the process is in the left branch needs to be specified explicitly. From this point of view, process-algebraic formalisms such as CCS [45] and CSP [35] are *underspecified*, since they leave important

behavioral aspects unspecified. For example, if the distinction between normal termination and deadlocked termination is relevant to the application, then this distinction ought to be explicitly modeled.

It is interesting to note that *transducers*, which were studied in an earlier work of Milner [44], which led to [45], are receptive. Transducers are widely accepted models of hardware. We come back to transducers in the next section.

The Principle of Comprehensive Modeling requires a process description to model all relevant aspects of process behavior. It does not spell out how such aspects are to be modeled. In particular, it does not address the question of what is observed when a process is being tested. Here again we propose to follow the approach of program semantics theory and argue that only the input/output behavior of processes is observable. Thus, observable relevant aspects of process behavior ought to be reflected in its input/output behavior.

Principle of Observable I/O: The observable behavior of a tested process is precisely its input/output behavior.

Of course, in the case of concurrent processes, the input/output behavior has a temporal dimension. That is, the input/output behavior of a process is a trace of input/output actions. The precise "shape" of this trace depends of course on the underlying semantics, which would determine, for example, whether we consider finite or infinite traces, the temporal granularity of traces, and the like. It remains to decide how nondeterminism is observed, as, after all, a nondeterministic process does not have a unique behavior. This leads to notions such as *may testing* and *must testing* [20]. We propose here to finesse this issue by imagining that a test is being run several times, eventually exhibiting *all* possible behaviors. Thus, the input/output behavior of a nondeterministic test is its full set of input/output traces.

In the next section we apply our approach to transducers; we show that once our three principles are applied we obtain that trace-based equivalence is adequate and fully abstract; that is, it is precisely the unique observational equivalence for transducers.

4 Case Study: Transducers

Transducers constitute a fundamental model of discrete-state machines with input and output channels [32]. They are still used as a basic model for sequential computer circuits [31]. We use nondeterministic transducers as our model for processes. We define a synchronous composition operator for such transducers, which provides us a notion of context. We then define linear observation semantics and give adequacy and full-abstraction results for trace equivalence in terms of it.

4.1 Nondeterministic Transducers

A nondeterministic transducer is a state machine with input and output channels. The state-transition function depends on the current state and the input, while the output depends solely on the current state (thus, our machines are Moore machines [32]).

Definition 1. *A transducer is a tuple, $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$, where*

- *Q is a countable set of states.*
- *q_0 is the start state.*
- *I is a finite set of input channels.*
- *O is a finite set of output channels.*
- *Σ is a finite alphabet of actions (or values).*
- *$\sigma : I \cup O \rightarrow 2^{\Sigma} - \{\emptyset\}$ is a function that allocates an alphabet to each channel.*
- *$\lambda : Q \times O \rightarrow \Sigma$ is the output function of the transducer. $\lambda(q, o) \in \sigma(o)$ is the value that is output on channel o when the transducer is in state q.*
- *$\delta : Q \times \sigma(i_1) \times \ldots \times \sigma(i_n) \rightarrow 2^Q$, where $I = \{i_1, \ldots, i_n\}$, is the transition function, mapping the current state and input to the set of possible next states.*

Both I and O can be empty. In this case δ is a function of state alone. This is important because the composition operation that we define usually leads to a reduction in the number of channels. Occasionally, we refer to the set of allowed values for a channel as the channel alphabet. This is distinct from the total alphabet of the transducer (denoted by Σ).

We represent a particular input to a transducer as an assignment that maps each input channel to a particular value. Formally, an *input assignment* for a transducer $(Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ is a function $f : I \rightarrow \Sigma$, such that for all $i \in I$, $f(i) \in \sigma(i)$. The entire input can then, by a slight abuse of notation, be succinctly represented as $f(I)$.

We point to three important features of our definition. First, note that transducers are receptive. That is, the transition function $\delta(q, f)$ is defined for all states $q \in Q$ and input assignments f. There is no implicit notion of deadlock here. Deadlocks need to be modeled explicitly, e.g., by a special sink state d whose output is, say, "deadlock". Second, note that inputs at time k take effect at time $k + 1$. This enables us to define composition without worrying about causalilty loops, unlike, for example, in Esterel [7]. Thirdly, note that the internal state of a transducer is observable only through its output function. How much of the state is observable depends on the output function.

4.2 Synchronous Parallel Composition

In general there is no canonical way to compose machines with multiple channels. In concrete devices, connecting components requires as little as knowing which wires to join. Taking inspiration from this, we say that a composition is defined by a particular set of desired connections between the machines to be composed. This leads to an intuitive and flexible definition of composition.

A connection is a pair consisting of an input channel of one transducer along with an output channel of another transducer. We require, however, sets of connections to be well formed. This requires two things:

- no two output channels are connected to the same input channel, and
- an output channel is connected to an input channel only if the output channel alphabet is a subset of the input channel alphabet. These conditions guarantee that connected input channels only receive well defined values that they can read. We now formally define this notion.

Definition 2 (Connections). *Let $\mathcal{M}$ be a set of transducers. Then*

$$Conn(\mathcal{M}) = \{X \subseteq \mathcal{C}(\mathcal{M}) | (a,b) \in X, (a,c) \in X \Rightarrow b = c\}$$

where $\mathcal{C}(\mathcal{M}) = \{(i_A, o_B) \; |\{A, B\} \subseteq \mathcal{M}, i_A \in I_A, o_B \in O_B, \sigma_B(o_B) \subseteq \sigma_A(i_A)\}$ is the set of all possible input/output connections for $\mathcal{M}$. Elements of $Conn(\mathcal{M})$ are valid connection sets.

Definition 3 (Composition)
Let $\mathcal{M} = \{M_1, \ldots, M_n\}$, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \lambda_k, \delta_k)$, be a set of transducers, and $C \in Conn(\mathcal{M})$. Then the composition *of $\mathcal{M}$ with respect to C, denoted by $||_C(\mathcal{M})$, is a transducer $(Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ defined as follows:*

- $Q = Q_1 \times \ldots \times Q_n$
- $q_0 = q_0^1 \times \ldots \times q_0^n$
- $I = \bigcup_{k=1}^n I_k - \{i \mid (i,o) \in C\}$
- $O = \bigcup_{k=1}^n O_k - \{o \mid (i,o) \in C\}$
- $\Sigma = \bigcup_{k=1}^n \Sigma_k$
- $\sigma(u) = \sigma_k(u)$, *where* $u \in I_k \cup O_k$
- $\lambda(q_1, \ldots, q_n, o) = \lambda_k(q_k, o)$ *where* $o \in O_k$
- $\delta(q_1, \ldots, q_n, f(I)) = \Pi_{k=1}^n(\delta_k(q_k, g(I_k)))$
  where $g(i) = \lambda_j(q_j, o)$ *if* $(i,o) \in C, \; o \in O_j$, *and* $g(i) = f(i)$ *otherwise.*

Definition 4 (Binary Composition). *Let M_1 and M_2 be transducers, and $C \in Conn(\{M_1, M_2\})$. The* binary composition *of M_1 and M_2 with respect to C is $M_1 ||_C M_2 = ||_C(\{M_1, M_2\})$.*

The following theorem shows that a general composition can be built up by a sequence of binary compositions. Thus binary composition is as powerful as general composition and henceforth we switch to binary composition as our default composition operation.

Theorem 1 (Composition Theorem)
Let $\mathcal{M} = \{M_1, \ldots, M_n\}$, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \lambda_k, \delta_k)$, be a set of transducers, and $C \in Conn(\mathcal{M})$. Let $\mathcal{M}' = \mathcal{M} - \{M_n\}$, $C' = \{(i,o) \in C | i \in I_j, o \in O_k, j < n, k < n\}$ and $C'' = C - C'$. Then

$$||_C(\mathcal{M}) = ||_{C''}(\{||_{C'}(\mathcal{M}'), M_n\}).$$

The upshot of Theorem 1 is that in the framework of transducers a general context, which is a network of transducers with a hole, is equivalent to a single transducer. Thus, for the purpose of contextual equivalence it is sufficient to consider testing transducers.

4.3 Executions and Traces

Definition 5 (Execution). *An* execution *for transducer $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ is a countable sequence of pairs $\langle s_i, f_i \rangle_{i=0}^l$ such that $s_0 = q_0$, and for all $i \geq 0$,*

- $s_i \in Q$.
- $f_i : I \to \Sigma$ *such that for all* $u \in I$, $f(u) \in \sigma(u)$.
- $s_i \in \delta(s_{i-1}, f_{i-1}(I))$.

If $l \in N$, the execution is finite and its length is l. If $l = \infty$, the execution is infinite and its length is defined to be ∞. The set of all executions of transducer M is denoted $exec(M)$.

Definition 6 (Trace). *Let $\alpha = \langle s_i, f_i \rangle_{i=0}^{l} \in exec(M)$. The* trace *of α, denoted by $[\alpha]$, is the sequence of pairs $\langle \omega_i, f_i \rangle_{i=0}^{l}$, where for all $i \geq 0$, $\omega_i : O \rightarrow \Sigma$ and for all $o \in O$, $\omega_i(o) = \lambda(s_i, o)$. The set of all traces of a transducer M, denoted by $Tr(M)$, is the set $\{[\alpha] | \alpha \in exec(M)\}$. An element of $Tr(M)$ is called a trace of M.*

Thus a trace is a sequence of pairs of output and input actions. While an execution captures the real underlying behavior of the system, a trace is the observable part of that behavior. The length of a trace α is defined to be the length of the underlying execution and is denoted by $|\alpha|$.

Definition 7 (Trace Equivalence). *Two transducers M_1 and M_2 are trace equivalent, denoted by $M_1 \sim_T M_2$, if $Tr(M_1) = Tr(M_2)$. Note that this requires that they have the same set of input and output channels.*

We now study the properties of trace equivalence. We first define the composition of executions and traces.

Definition 8. *Given $\alpha = \langle s_i, f_i \rangle_{i=0}^{n} \in exec(M_1)$ and $\beta = \langle r_i, g_i \rangle_{i=0}^{n} \in exec(M_2)$, we define the composition of α and β w.r.t $C \in Conn(\{M_1, M_2\})$ as follows*

$$\alpha ||_C \beta = \langle (s_i, r_i), h_i \rangle_{i=0}^{n}$$

where $h_i(u) = f_i(u)$ if $u \in I_1 - \{i | (i, o) \in C\}$ and $h_i(u) = g_i(u)$ if $u \in I_2 - \{i | (i, o) \in C\}$.

Definition 9. *Given $t = \langle \omega_i, f_i \rangle_{i=0}^{n} \in Tr(M_1)$ and $u = \langle \nu_i, g_i \rangle_{i=0}^{n} \in Tr(M_2)$, we define the composition of t and u w.r.t $C \in Conn(\{M_1, M_2\})$ as follows*

$$t ||_C u = \langle \mu_i, h_i \rangle_{i=0}^{n}$$

where $\mu_i(o) = \omega_i(o)$ if $o \in O_1 - \{o | (i, o) \in C\}$ and $\mu_i(o) = \nu_i(o)$ if $o \in O_2 - \{o | (i, o) \in C\}$, and h_i is as defined in Definition 8 above.

Note that the composition operation defined on traces is purely syntactic. There is no guarantee that the composition of two traces is a trace of the composition of the transducers generating the individual traces. The following simple property is necessary and sufficient to achieve this.

Definition 10 (Compatible Traces). *Given $C \in Conn(\{M_1, M_2\})$, $t_1 = \langle \omega_i^1, f_i^1 \rangle_{i=0}^{n} \in Tr(M_1)$ and $t_2 = \langle \omega_i^2, f_i^2 \rangle_{i=0}^{n} \in Tr(M_2)$, we say that t_1 and t_2 are* compatible *with respect to C if for all $(u, o) \in C$ and for all $i \geq 0$, we have*

- *If $u \in I_j$ and $o \in O_k$ then $f_i^j(u) = \omega_i^k(o)$, for all $i \geq 0$ and for $j, k \in \{1, 2\}$.*

Lemma 1. *Let $C \in Conn(\{M_1, M_2\})$, $t \in Tr(M_1)$ and $u \in Tr(M_2)$. Then $t||_C u \in Tr(M_1||_C M_2)$ if and only if t and u are compatible with respect to C.*

We now extend the notion of trace composition to sets of traces.

Definition 11. *Let $T_1 \subseteq Tr(M_1)$, $T_2 \subseteq Tr(M_2)$ and $C \in Conn(\{M_1, M_2\})$. We define*

$$T_1||_C T_2 = \{t_1||_C t_2 \,|\, t_1 \in Tr(M_1), t_2 \in Tr(M_2), |t_1| = |t_2|\}$$

Theorem 2 (Syntactic theorem of traces). *Let $T_1 \subseteq Tr(M_1) \cap Tr(M_3)$ and $T_2 \subseteq Tr(M_2) \cap Tr(M_4)$, and $C \in Conn(\{M_1, M_2\}) \cap Conn(\{M_3, M_4\})$. Then*

$$(T_1||_C T_2) \cap Tr(M_1||_C M_2) = (T_1||_C T_2) \cap Tr(M_3||_C M_4)$$

Using Theorem 2, we show now that any equivalence defined in terms of sets of traces is automatically a congruence with respect to composition, if it satisfies a certain natural property.

Definition 12 (Trace-based equivalence). *Let $\mathcal{M}$ be the set of all transducers. Let $R : \mathcal{M} \rightarrow \{A \subseteq Tr(M) | M \in \mathcal{M}\}$ such that for all $M \in \mathcal{M}$, $R(M) \subseteq Tr(M)$. Then R defines an equivalence relation on $\mathcal{M}$, denoted by $\sim_R$, such that for all $M_1, M_2 \in \mathcal{M}$, $M_1 \sim_R M_2$ if and only if $R(M_1) = R(M_2)$. Such a relation is called a* trace-based equivalence.

Trace-based equivalences enable us to relativize trace equivalence to “interesting” traces. For example, one may want to consider finite traces only, infinite traces only, fair traces only, and the like. Of course, not all such relativizations are appropriate. The next definition addresses this issue.

Definition 13 (Compositionality). *Let $\sim_R$ be a trace-based equivalence. We say that $\sim_R$ is compositional if given transducers M_1, M_2 and $C \in Conn(\{M_1, M_2\})$, the following hold:*

1. *$R(M_1||_C M_2) \subseteq R(M_1)||_C R(M_2)$.*
2. *If $t_1 \in R(M_1)$, $t_2 \in R(M_2)$, and t_1, t_2 are compatible w.r.t. C, then $t_1||_C t_2 \in R(M_1||_C M_2)$.*

The two conditions in Definition 13 are, in a sense, soundness and completeness conditions, as the first ensures that no inappropriate traces are present, while the second ensures that all appropriate traces are present. That is, the first condition ensures that the trace set captured by R is not too large, while the second ensures that it is not too small.

Note, in particular, that trace equivalence itself is a compositional trace-based equivalence. The next theorem asserts that $\sim_R$ is a *congruence* with respect to composition.

Theorem 3 (Congruence Theorem). *Let $\sim_R$ be a compositional trace-based equivalence. Let $M_1 \sim_R M_3$, $M_2 \sim_R M_4$, and $C \in Conn(\{M_1, M_2\}) = Conn(\{M_3, M_4\})$. Then $M_1||_C M_2 \sim_R M_3||_C M_4$.*

An immediate corollary of Theorem 3 is the fact that no context can distinguish between two trace-based equivalent transducers.

Corollary 1. *Let M_1 and M_2 be transducers, R be a compositional trace-based equivalence and $M_1 \sim_R M_2$. Then for all transducers M and all $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, we have that $M||_C M_1 \sim_R M||_C M_2$.*

Finally, it is also the case that some context can always distinguish between two inequivalent transducers. If we choose a composition with an empty set of connections, all original traces of the composed transducers are present in the traces of the composition. If $M_1 \not\sim_R M_2$, then $M_1||_\emptyset M \not\sim_R M_2||_\emptyset M$. We claim the stronger result that given two inequivalent transducers, we can always find a third transducer that distinguishes between the first two, *irrespective* of how it is composed with them.

Theorem 4. *Let M_1 and M_2 be transducers, R be a compositional trace-based equivalence and $M_1 \not\sim_R M_2$. Then there exists a transducer M such that for all $C \in Conn(\{M, M_1\}) \cap Conn(\{M, M_2\})$, we have $M||_C M_1 \not\sim_R M||_C M_2$.*

5 What Is Linear Time Logic?

The discussion so far has focused on the branching- or linear-time view of process equivalence, where we argued strongly in favor of linear time. This should be distinguished from the argument in, say, [58] in favor of linear-temporal logics (such as LTL, ForSpec, and the like). In the standard approach to linear-temporal logics, one interprets formulas in such logics over traces. Thus, given a linear-temporal formula ψ, its semantics is the set $traces(\psi)$ of traces satisfying it. A system S then satisfies ψ if $traces(S) \subseteq traces(\psi)$.

It has recently been shown that this view of linear time is not rich enough [37]. The context for this realization is an analysis of *liveness* properties, which assert that something good will happen eventually. In satisfying liveness properties, there is no bound on the "wait time", namely the time that may elapse until an eventuality is fulfilled. For example, the LTL formula $F\theta$ is satisfied at time i if θ holds at some time $j \geq i$, but $j - i$ is not a priori bounded.

In many applications, such as real-time systems, it is important to bound the wait time. This has given rise to formalisms in which the eventually operator F is replaced by a bounded-eventually operator $F^{\leq k}$. The operator is parameterized by some $k \geq 0$, and it bounds the wait time to k [5,26]. In the context of discrete-time systems, the operator $F^{\leq k}$ is simply syntactic sugar for an expression in which the next operator $\mathbf{X}$ is nested. Indeed, $F^{\leq k}\theta$ is just $\theta \vee \mathbf{X}(\theta \vee \mathbf{X}(\theta \vee \overset{k-4}{\ldots} \vee \mathbf{X}\theta))$.

A drawback of the above formalism is that the bound k needs to be known in advance, which is not the case in many applications. For example, it may depend on the system, which may not yet be known, or it may change, if the system changes. In addition, the bound may be very large, causing the state-based description of the specification (e.g., an automaton for it) to be very large too. Thus, the common practice is to use liveness properties as an abstraction of such safety properties: one writes $F\theta$ instead of $F^{\leq k}\theta$ for an unknown or a too large k.

This abstraction of safety properties by liveness properties is not sound for a logic such as LTL. Consider the system $\mathcal{S}$ described in Figure 1 below. While S satisfies the LTL formula FGq, there is no $k \geq 0$ such that $\mathcal{S}$ satisfies $F^{\leq k}Gq$. To see this, note that

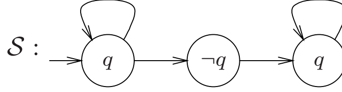


Fig. 1. $\mathcal{S}$ satisfies FGq but does not satisfy $F^{\leq k}Gq$, for all $k \geq 0$

for each $k \geq 0$, the computation that first loops in the first state for k times and only then continues to the second state, satisfies the eventuality Gq with wait time $k + 1$.

In [37], there is a study of an extension of LTL that addresses the above problem. In addition to the usual temporal operators of LTL, the logic PROMPT-LTL has a new temporal operator that is used for specifying eventualities with a bounded wait time. The new operator is called *prompt eventually* and is denoted by $\mathbf{F_p}$. It has the following formal semantics: For a PROMPT-LTL formula ψ and a bound $k \geq 0$, let ψ^k be the LTL formula obtained from ψ by replacing all occurrences of $\mathbf{F_p}$ by $F^{\leq k}$. Then, a system S satisfies ψ iff there is $k \geq 0$ such that S satisfies ψ^k.

Note that while the syntax of PROMPT-LTL is very similar to that of LTL, its semantics is defined with respect to an entire system, and not with respect to computations. For example, while each computation π in the system $\mathcal{S}$ from Figure 1 has a bound $k_\pi \geq 0$ such that Gq is satisfied in π with wait time k_π, there is no $k \geq 0$ that bounds the wait time of all computations. It follows that, unlike LTL, we cannot characterize a PROMPT-LTL formula ψ by a set of traces $L(\psi)$ such that a system S satisfies ψ iff the set of traces of S is contained in $L(\psi)$. Rather, one needs to associate with a formula ψ of PROMPT-LTL an *infinite* family $\mathbf{L}(\psi)$ of sets of traces, so that a system S satisfies ψ if $traces(S) \subseteq L$ for some $L \in \mathbf{L}(\psi)$. This suggests a richer view of linear-time logic than the standard one, which associates a single set of traces with each formula in the logic. Even in this richer setting, we have the desired feature that two trace-equivalent processes satisfy the same linear-time formulas.

6 Discussion

It could be fairly argued that the arguments raised in this paper have been raised before.

– Testing equivalence, introduced in [20], is clearly a notion of contextual equivalence. Their answer to the question, "What is a test?", is that a test is any process that can be expressed in the formalism. So a test is really the counterpart of a context in program equivalence. (Though our notion of context in Section 4, as a network of transducers, is, a priori, richer.) At the same time, bisimulation equivalence has been recognized as being too fine a relation to be considered as contextual equivalence [8,9,10,30].
– Furthermore, it has also been shown that many notions of process equivalence studied in the literature can be obtained as contextual equivalence with respect to appropriately defined notions of directly observable behavior [11,36,41,47]. These notions fall under the title of *decorated trace equivalence*, as they all start with trace semantics and then endow it with additional observables. These notions have the advantage that, like bisimulation equivalence, they are not blind to issues such as deadlock behavior.

With respect to the first point, it should be noted that despite the criticisms leveled at it, bisimulation equivalence still enjoys a special place of respect in concurrency theory as a reasonable notion of process equivalence [28]. In fact, the close correspondence between bisimulation equivalence and the branching-time logic CTL has been mentioned as an advantage of CTL. Thus, it is not redundant, in our opinion, to reiterate the point that bisimulation and its variants are not contextual equivalences.

With respect to the second point we note that our approach is related, but quite different, than that taken in decorated trace equivalence. In the latter approach, the "decorated" of traces is attributed by concurrency theorists. As there is no unique way to decorate traces, one is left with numerous notions of equivalence and with the attitude quoted above that "It is not the task of process theory to find the 'true' semantics of processes, but rather to determine which process semantics is suitable for which applications" [28]. In our approach, only the modelers know what the relevant aspects of behavior are in their applications and only they can decorate traces appropriately, which led to our Principles of Comprehensive Modeling and Observable I/O. In our approach, there is only one notion of contextual equivalence, which is trace equivalence.

Admittedly, the comprehensive-modeling approach is not wholly original, and has been foretold by Brookes [12], who said: "We do not augment traces with extraneous book-keeping information, or impose complex closure conditions. Instead we incorporate the crucial information about blocking directly in the internal structure of traces." Still, we believe that it is valuable to carry Brookes's approach further, substantiate it with our three guiding principles, and demonstrate it in the framework of transducers.

An argument that can be leveled at our comprehensive-modeling approach is that it requires a low-level view of systems, one that requires modeling all relevant behavioral aspects. This issue was raised by Vaandrager in the context of I/O Automata [55]. Our response to this criticism is twofold. First, if these low level details (e.g., deadlock behavior) are relevant to the application, then they better be spelled out by the modeler rather than by the concurrency theorist. Second, one needs to separate the user-level language from its underlying semantics. One could imagine a language such as CSP with handshake communication, where the language does not explicitly address deadlocks. Nevertheless, the underlying semantics, say, in terms of structured operational semantics [33], needs to expose deadlock behavior explicitly and make it observable. This would be analogous to the description of exceptional termination in many programming languages. Note that the alternative is to accept formalisms for concurrency that are not fully specified and admit a plethora of different notions of process equivalence.

In conclusion, this paper puts forward an, admittedly provocative, thesis, which is that process-equivalence theory allowed itself to wander in the "wilderness" for lack of accepted guiding principles. The obvious definition of contextual equivalence was not scrupulously adhered to, and the underspecificity of the formalisms proposed led to too many interpretations of equivalence. While one may not realistic expect a single paper to overwrite about 30 years of research, a more modest hope would be for a renewed discussion on the basic principles of process-equivalence theory.

Acknowledgment. The second author is grateful to P. Cousot and G. Plotkin for challenging him to consider the semantical aspects of the branching vs. linear-time issue, and to S. Abramsky, L. Aceto, S. Brookes, W. Fokkink, P. Panagaden, A. Pitts, and G. Plotkin for discussions and comments on this topic.

References

1. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems 15(1), 73–132 (1993)
2. Abramsky, S.: Observation equivalence as a testing equivalence. Theor. Comput. Sci. 53, 225–241 (1987)
3. Abramsky, S.: What are the fundamental structures of concurrency?: We still don't know! Electr. Notes Theor. Comput. Sci. 162, 37–41 (2006)
4. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The ForSpec temporal logic: A new temporal property-specification logic. In: Katoen, J-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 211–296. Springer, Heidelberg (2002)
5. Beer, I., Ben-David, S., Geist, D., Gewirtzman, R., Yoeli, M.: Methodology and system for practical formal verification of reactive hardware. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 182–193. Springer, Heidelberg (1994)
6. Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. Acta Informatica 20, 207–226 (1983)
7. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
8. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. J. ACM 42(1), 232–268 (1995)
9. Bloom, B., Meyer, A.R.: Experimenting with process equivalence. Theor. Comput. Sci. 101(2), 223–237 (1992)
10. Bol, R.N., Groote, J.F.: The meaning of negative premises in transition system specifications. J. ACM 43(5), 863–914 (1996)
11. Boreale, M., Pugliese, R.: Basic observables for processes. Information and Computation 149(1), 77–98 (1999)
12. Brookes, S.D.: Traces, pomsets, fairness and full abstraction for communicating processes. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 466–482. Springer, Heidelberg (2002)
13. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theoretical Computer Science 59, 115–131 (1988)
14. Carmo, J., Sernadas, A.: Branching vs linear logics yet again. Formal Aspects of Computing 2, 24–59 (1990)
15. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum Press (1978)
16. Clarke, E.M., Draghicescu, I.A.: Expressibility results for linear-time and branching-time logics. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 428–437. Springer, Heidelberg (1989)
17. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)

18. Clarke, E.M., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School). LNCS, vol. 803, pp. 124–175. Springer, Heidelberg (1994)
19. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
20. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
21. Dill, D.L.: Trace theory for automatic hierarchical verification of speed independent circuits. MIT Press, Cambridge (1989)
22. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
23. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Proc. 7th Int. Colloq. on Automata, Languages, and Programming, pp. 169–181 (1980)
24. Emerson, E.A., Halpern, J.Y.: Sometimes and not never revisited: On branching versus linear time. Journal of the ACM 33(1), 151–178 (1986)
25. Emerson, E.A., Lei, C.-L.: Modalities for model checking: Branching time logic strikes back. In: Proc. 12th ACM Symp. on Principles of Programming Languages, pp. 84–96 (1985)
26. Emerson, E.A., Mok, A.K., Sistla, A.P., Srinivasan, J.: Quantitative temporal reasoning. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 136–145. Springer, Heidelberg (1991)
27. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. Formal Methods in System Design 21(1), 39–78 (2002)
28. van Glabbeek, R.J.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra. Ch-1, pp. 3–99. Elsevier, Amsterdam (2001)
29. Goering, R.: Model checking expands verification's scope. Electronic Engineering Today (February 1997)
30. Groote, J.F.: Transition system specifications with negative premises. Theor. Comput. Sci. 118(2), 263–299 (1993)
31. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Kluwer Academic Publishers, Dordrecht (1996)
32. Hartmanis, J., Stearns, R.E.: Algebraic Structure Theory of Sequential Machines. Prentice-Hall, Englewood Cliffs (1966)
33. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
34. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32, 137–161 (1985)
35. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
36. Jonsson, B.: A fully abstract trace model for dataflow networks. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 155–165 (1989)
37. Kupferman, O., Piterman, N., Vardi, M.Y.: From liveness to promptness. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 406–419. Springer, Heidelberg (2007)
38. Lamport, L.: Sometimes is sometimes not never - on the temporal logic of programs. In: Proc. 7th ACM Symp. on Principles of Programming Languages, pp. 174–185 (1980)
39. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proc. 12th ACM Symp. on Principles of Programming Languages, pp. 97–107 (1985)
40. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
41. Main, M.G.: Trace, failure and testing equivalences for communicating processes. Int'l J. of Parallel Programming 16(5), 383–400 (1987)

42. Marek, W.W., Trusczynski, M.: Nonmonotonic Logic: Context-Dependent Reasoning. Springer, Heidelberg (1997)
43. McCarthy, J.: Circumscription - a form of non-monotonic reasoning. Artif. Intell. 13(1-2), 27–39 (1980)
44. Milner, R.: Processes: a mathematical model of computing agents. In: Logic Colloquium, North Holland, pp. 157–173 (1975)
45. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
46. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
47. Olderog, E.R., Hoare, C.A.R.: Specification-oriented semantics for communicating processes. Acta Inf. 23(1), 9–66 (1986)
48. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) Theoretical Computer Science, GI 1981. LNCS, vol. 104, Springer, Heidelberg (1981)
49. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. on Foundations of Computer Science, pp. 46–57 (1977)
50. Pnueli, A.: Linear and branching structures in the semantics and logics of reactive systems. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 15–32. Springer, Heidelberg (1985)
51. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Proc. 8th ACM Symp. on Principles of Programming Languages. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
52. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. Journal of the ACM 32, 733–749 (1985)
53. Stirling, C.: Comparing linear and branching time temporal logics. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 1–20. Springer, Heidelberg (1989)
54. Stirling, C.: The joys of bisimulation. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 142–151. Springer, Heidelberg (1998)
55. Vaandrager, F.W.: On the relationship between process algebra and input/output automata. In: Proc. 6th IEEE Symp. on Logic in Computer Science, pp. 387–398 (1991)
56. Vardi, M.Y.: Linear vs. branching time: A complexity-theoretic perspective. In: Proc. 13th IEEE Sym. on Logic in Computer Science, pp. 394–405 (1998)
57. Vardi, M.Y.: Sometimes and not never re-revisited: on branching vs. linear time. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 1–17. Springer, Heidelberg (1998)
58. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001)
59. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. 1st IEEE Symp. on Logic in Computer Science, pp. 332–344 (1986)
60. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)
61. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)

Mind the Shapes: Abstraction Refinement Via Topology Invariants*

Jörg Bauer[1], Tobe Toben[2], and Bernd Westphal[2]

[1] Technical University of Denmark, Kongens Lyngby, Denmark
joba@imm.dtu.dk**
[2] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
{toben,westphal}@informatik.uni-oldenburg.de

Abstract. Dynamic Communication Systems (DCS) are infinite state systems where an unbounded number of processes operate in an evolving communication topology. For automated verification of properties of DCS, finitary abstractions based on exploiting symmetry can be employed. However, these abstractions give rise to spurious behaviour that often inhibits to successfully prove relevant properties.

In this paper, we propose to combine a particular finitary abstraction with global system invariants obtained by abstract interpretation. These system invariants establish an over-approximation of possible communication topologies occurring at runtime, which can be used to identify and exclude spurious behaviour introduced by the finitary abstraction, which is thereby refined. Based on a running example of car platooning, we demonstrate that our approach allows to verify temporal DCS properties that no technique in isolation is able to prove.

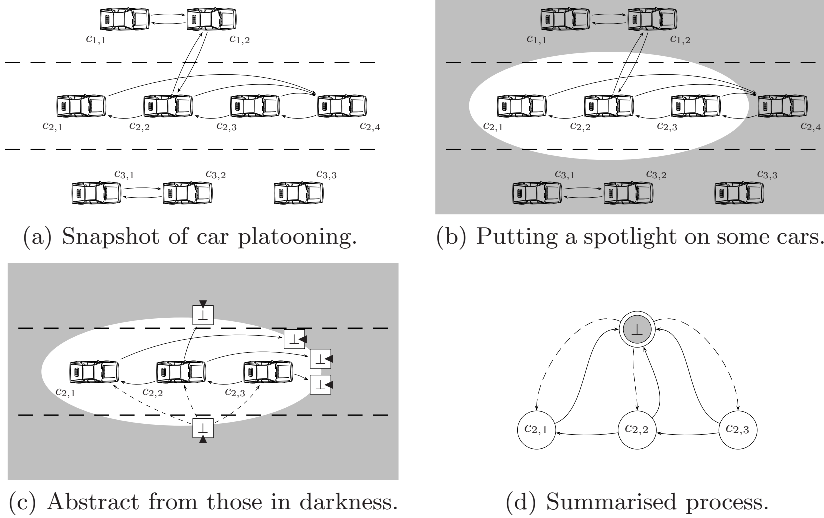
1 Introduction

Formal verification of systems with dynamic process creation is an active research area. In [2], we characterised a certain class of such systems, the so-called Dynamic Communication Systems (DCS), by providing the formal description language *DCS protocols*. DCS protocols are complemented by METT, a variant of temporal logic for requirements specification. Here, we elaborate on an automated procedure for checking whether a DCS protocol satisfies a METT property. A manual procedure was sketched in [2]. By bridging the technical gap between analysis techniques with different strengths and weaknesses, we obtain a fully automated, integrated implementation, which benefits from synergetical effects.

Running Example. DCS are ubiquitous, most prominent among them mobile ad-hoc networks, service-oriented computing scenarios, or traffic control systems based on wireless communication. In order to demonstrate the appropriateness of our automated DCS verification technique we pick the characteristic real-world example *car platooning*, a traffic control system studied by the California

* Partly supported by the German Research Council (DFG), SFB/TR 14 AVACS.
** The author was partly sponsored by the project SENSORIA, IST-2005-016004.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 35–50, 2007.
© Springer-Verlag Berlin Heidelberg 2007



(a) Snapshot of car platooning.
(b) Putting a spotlight on some cars.
(c) Abstract from those in darkness.
(d) Summarised process.

Fig. 1. Car Platooning: Two platoons and one free agent, from concrete to abstract

PATH project [9]. In order to optimise traffic throughput on highways and reduce energy consumption, they propose that cars shall dynamically form platoons (cf. Fig. 1(a)). Conceptionally, each car has a notion of *state*, that is, whether it is currently a *follower* in a platoon (like $c_{1,1}$), a *leader* of a platoon (like $c_{1,2}$), or whether it is driving on its own as a *free agent* (like $c_{3,3}$). In addition, there are *links* between cars, indicated by the arrows in Fig. 1(a). Each follower knows its leader, e.g. to negotiate leaving the platoon, each leader knows its followers, e.g. to announce a braking manoeuvre, and each car may know a lane-neighbour. Interlinked cars communicate by message passing. There is no finite upper bound on the number of cars, as cars freely enter and leave the system "highway".

Example Requirement. There are three elementary actions for car platooning: *merge* and *split*, to build up and separate platoons, and *change lane*. In the following we'll focus on the merge action. If a platoon led by car c_b merges with a platoon in front led by car c_f, then during the merge c_b hands over its followers to c_f. For example, car $c_{3,2}$ merging with $c_{3,3}$ in Fig. 1(a) were an instance of this situation with $c_b := c_{3,2}$ and $c_f := c_{3,3}$. Given a formal DCS protocol model of merge (cf. Sect. 2), a natural requirement to check would concern this handover. Namely, for each follower c of c_b, whenever c_b sends a message '*newld*' (new leader) to c carrying the identity of the new leader c_f as a parameter, then c will finally change its leader link to c_f. Until that point in time, c_b remains c's leader. In the logic METT (cf. Table 1 on page 41), this property is written as

$$\forall\, c_b, c, c_f \,.\, \underbrace{\mathsf{G}\ \left(snd[newld](c_b, c, c_f) \rightarrow (conn[ldr](c, c_b)\ \mathsf{U}\ conn[ldr](c, c_f))\right)}_{=:\mu_0} . \quad (1)$$

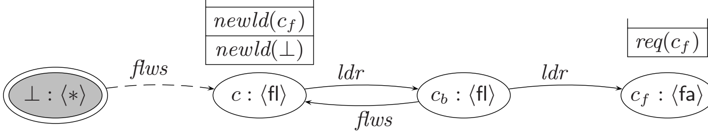


Fig. 2. (Spurious) counter-example: Back leader c_b detected free agent c_f ahead and merged. Summary process $\perp$ sent a spurious '*newld*($\perp$)' before c_b correctly sent '*newld*(c_f)' to follower c. Consumption of the former by c leads to a violation of (1).

Analysis Problem. Faithfully modelling car platooning requires unbounded creation and destruction of processes as we cannot assume finite bounds on the number of cars. DCS are consequently infinite-state systems. In order to employ automated verification techniques for temporal properties, we use a particular abstraction to finite-state transition systems following the *spotlight principle* [18]. The principle is to keep a finite number of processes completely precise and abstract from all others, the *rest*. In the particular case of Data Type Reduction (DTR, cf. Section 3), the number heuristically depends on the considered property and information about the state of the rest is completely dismissed, in particular links from the rest into the spotlight. Links from the spotlight are preserved but may point to the rest. Figures 1(c) and 1(d) illustrate the abstract state that represents Fig. 1(b), if the spotlight is on cars $c_{2,1}, c_{2,2}, c_{2,3}$. Having only the *local* information of the precise processes has the positive effect that the transition relation on abstract states is easily computable from a DCS description [19]. A negative effect is that the aggressive abstraction gives rise to a large amount of spurious behaviour, possibly comprising spurious counter-examples for a given property. As discussed in more detail in Section 3, many critical spurious runs, i.e. runs leading to a bad state in the abstract system, which is unreachable in the concrete one, follow a pattern we call spurious interference. In instances of this pattern, one observes messages from the abstracted *rest* to the concrete part, which are not possible in the concrete system.

For example, consider property (1) with three precise processes and the *rest* as in Fig. 1(d). If the *rest* sends a spurious '*newld*' message with an identity different from c_b and c_f to c, the property is violated (cf. Fig. 2). This situation can manually be identified as spurious by considering the DCS protocol as there is at most one car which may send '*newld*', namely the back leader. Since this is c_b, there cannot be a car in the *rest* sending the message. With this insight, it is desirable to refine the DTR abstraction by explicitly excluding such *communication topologies*, i.e. a global state of a DCS comprising connected processes, like the one shown in Fig. 2. Automating this kind of refinement poses two problems: (i) to automatically obtain information on the set of possible topologies; and (ii) to soundly exclude topologies shown impossible by (i).

Approach. We tackle problem (i) by employing a new static analysis of graph grammars [4], called *Topology Analysis (TA)* below. Using our new encoding of DCS in graph grammars as presented in Section 4, we can thus compute abstract

graphs, *topology invariants*, describing an over-approximation of all topologies possibly occurring in runs of a DCS. Regarding problem (ii), Section 5 provides a logical characterisation of whether an abstract DTR topology like the one shown in Fig. 1(d) is definitely impossible according to the topology invariants computed by TA. These logical formulae can then be used as negative assumptions when model-checking the abstract transition system obtained by DTR. Here the challenge is that both the topologies in the abstract, finite-state transition system obtained by DTR and the topology invariants computed by TA are elements of *different abstract domains*. It is neither obvious how these domains relate formally nor whether an element of one domain represents more concrete topologies than an element of the other domain, since both typically represent infinitely many concrete topologies.

On top of first experimental results in Section 5 proving our approach to be effective, we briefly discuss in Section 6 whether the refinement proposed here could be further refined by, e.g., counter-example guidance.

Related Work. We specify DCS using DCS protocols (cf. Section 2). They were originally inspired by Communicating Finite State Machines (CFSM) of [5] but extend those by dynamic process creation and destruction and flexible communication topologies. Other than DCS protocols, DCS may be modelled using existing techniques such as the π-calculus [14] or variants of I/O automata (see [12] for an overview). The π-calculus is less adequate for our purpose, because crucial high-level features of DCS, like process states, message queues, or explicit graph-like communication topologies require cumbersome and low-level *encodings* into elementary π-actions. In that case, higher-level properties of a DCS are no longer accessible for specially tailored analyses or optimisations. Similar arguments hold for versions of I/O automata dealing with dynamic process creation, although they are admittedly much closer to DCS protocols. However, the better part of research on I/O automata is devoted to features like time or probability, while DCS protocols emphasise the dynamics of DCS.

DTR (cf. Section 3) is an instance of the *spotlight principle* [18] and as such related to different abstractions mapping infinite-state transition systems to finite-state ones comprising, e.g., [11] and [6]. Technically, Topology Analysis (TA, cf. Section 4) is an abstract interpretation of graph grammars, which are well suited to explicitly describe evolving DCS communication topologies. There is only one other abstract interpretation based approach to graph grammar verification [16]. Other approaches to graph grammar analysis use Petri-net based techniques [1]. However, these approaches are mostly concerned with the verification of pointer programs, where updates to graphs occur in a more restricted manner. A static analysis to determine communication topologies for π is given in [17].

The principal idea of Section 5, to refine an abstraction with separately obtained invariants, is not new, e.g. [10], but the combination of TA and DTR is. Moreover, we integrate, in a technically sound way, two verification techniques often regarded as orthogonal: model checking and static program analysis. The synergy effect results in a novel automated verification technique for DCS.

It is noteworthy that hardly any of the specification and verification issues addressed in this work are addressed in the original PATH design [9], which deals with a static number of cars only. In particular, there are neither process creation, destruction, nor evolving communication topologies. In fact, in preliminary work [2], we manually discovered (and remedied) severe flaws in the original specification. With our proposed technique we are able to do so automatically.

2 Dynamic Communication Systems

Dynamic Communication Systems (DCS) can be viewed as a strict generalisation of classical parameterised systems because every process executes the same, finite control part, the *DCS protocol*. But in contrast to classical parameterised systems, where a fixed number of K processes run in parallel and communicate via global shared memory, processes are dynamically created and destroyed in DCS without an upper bound on the number of processes. Furthermore, DCS processes only have local memory, communicate asynchronously by passing messages, which may carry process identities, and in general aren't fully connected, but every process knows only some other processes via links.

In [2], we have introduced *DCS protocols* as an adequate modelling language for DCS, such as the car platooning example.

Syntax. A *DCS protocol* is a seven-tuple $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, \mathit{succ})$ with

- a finite set Q of *states* a process may assume,
- *initial states* $A \subseteq Q$ assumed by newly appeared processes and *fragile states* $\Omega \subseteq Q$, in which processes may disappear,
- a finite set χ of *channels*, each providing potential links to other processes,
- a finite set Σ of *messages* and *environment messages* $\mathcal{E}_{\text{msg}} \subseteq \Sigma$, that is, messages that may non-deterministically be sent by the environment, and
- a *successor relation* '*succ*', determining each processes' behaviour.

The successor relation *succ* comprises four different kinds of labelled transitions between two states from Q, namely send, receive, modify, and conditional transitions. The corresponding (possibly empty) four sets partitioning *succ* are denoted by *Snd*, *Rec*, *Mod*, and *Cnd*. The notation for transitions is as follows.

- $(q, c, m, c', q') \in \mathit{Snd} \subseteq Q \times \chi \times \Sigma \times \chi \mathbin{\dot\cup} \{\mathit{id}\} \times Q$: send over channel c the message m carrying one of the identities stored in channel c' or the own identity if $c' = \mathit{id}$, and change to state q';
- $(q, m, c, \mathit{op}, q') \in \mathit{Rec} \subseteq Q \times \Sigma \times \chi \times \{\mathsf{set}, \mathsf{join}\} \times Q$: consume a message m, store the attached identity by operation op to channel c, change to state q';
- $(q, c_1, \mathit{op}, c_2, q') \in \mathit{Mod} \subseteq Q \times \chi \times \{\mathsf{add}, \mathsf{del}, \mathsf{pick}\} \times \chi \times Q$: combine channels c_1 and c_2 by operation op, store the result in channel c_1, change to state q';
- $(q, \mathit{em}, c, q') \in \mathit{Cnd} \subseteq Q \times \mathbb{B} \times \chi \times Q$: change to state q' if the emptiness constraint em on channel c is satisfied (see below).

Figure 3 shows a high-level model of the merge procedure which is supposed to be executed by each car. At any time, each car is either driving on its own as

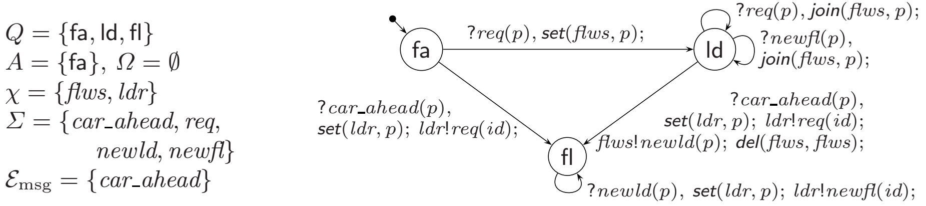


Fig. 3. High-level implementation of "**platoon merge**"

a *free agent* (fa) or participates in a platoon as *leader* (ld) or *follower* (fl). Each follower knows its leader by the channel *ldr* and each leader knows its followers by the channel *flws*. For conciseness, the figure makes use of *complex transitions* [15] where one receive and a couple of send actions are being executed atomically.

The environment message '*car_ahead*' models that some sensors of a free agent discover cars in front and notify the free agent of the identity of such approached cars. The free agent then requests the merge by sending a message '*req*' which carries its identity and becomes a follower by taking the transition to fl. If a free agent receives a merge request, it stores the requester in channel *flws* and changes state to ld. A leader may accept more followers and add them to *flws*, or merge with another free agent or platoon in front. In the latter case, it announces the new leader by a '*newld*' message to its followers and dismisses them, the followers then register with the new leader by a '*newfl*' message.

Semantics. Given a DCS protocol $\mathcal{P}$ and a countably infinite set Id of identities, a *(local) configuration* of a process is a triple (q, C, M) where $q \in Q$ is the local state, $C : \chi \to 2^{Id}$ is a function mapping channels to a set of process identities, and $M = (\Sigma \times Id)^*$ is the message queue. We use $\mathcal{S}(\mathcal{P})$ to denote the set of all local configurations of protocol $\mathcal{P}$. A local configuration (q, C, M) is called *initial* if $q \in A$, C yields the empty set for all channels, and M is the empty word.

A *topology* (or *global configuration*) of $\mathcal{P}$ is a partial function $\mathcal{N} : Id \rightharpoonup \mathcal{S}(\mathcal{P})$ mapping identities to local configurations. In the following we write $\iota \in \mathcal{N}$ to denote that $\mathcal{N}$ is defined for ι. A topology $\mathcal{N}$ *evolves* into $\mathcal{N}'$, written as $\mathcal{N} \rightsquigarrow \mathcal{N}'$, if one of the following conditions is satisfied, where all processes not affected by the evolution are required to remain the same. Note that the first and the last two actions are *environment actions* which are always enabled.

Appearance: A new process is created, starting in an initial state, not connected to anyone else, and with an empty queue, i.e. $\mathcal{N}' = \mathcal{N}[\iota \mapsto (q, C, M)]$ where $dom(\mathcal{N}') = dom(\mathcal{N}) \setminus \{\iota\}$ and configuration (q, C, M) is initial.

SendMessage: A process $\iota \in \mathcal{N}$ takes a send transition (q, c, m, c', q') and thus appends message m carrying an identity as parameter to the message queues of the processes denoted by its channel c, i.e. if $\mathcal{N}(\iota) = (q, C, M)$ we have that $\mathcal{N}'(\iota) = (q', C, M)$ and $\mathcal{N}'(\iota_0) = (q_0, C_0, M_0.(m, \iota'))$ for all $\iota_0 \in C(c)$ with $\mathcal{N}(\iota_0) = (q_0, C_0, M_0)$. If $c' = id$, the process sends its own identity, i.e. $\iota' = \iota$, otherwise it sends an element from channel c', i.e. $\iota' \in C(c')$.

Table 1. METT predicates (p, p_1, p_2 are quantified variables)

$p_1 = p_2$	equality
$instate[q](p)$	in state q
$conn[c](p_1, p_2)$	linked via channel c
$\odot\, p$	just created

$snd[m](p_1, p_2, p)$	message m just sent
$pend[m](p_1, p_2, p)$	message m pending
$rcv[m](p_1, p_2, p)$	message m received
$\otimes\, p$	about to die

ReceiveMessage: A process $\iota \in \mathcal{N}$ takes a receive transition (q, m, c, op, q') and thus stores the identity carried by the message to channel c, i.e. if $\mathcal{N}(\iota) = (q, C, (m, \iota_0).M)$ we have $\mathcal{N}'(\iota) = (q', C', M)$ where $C'(c) = \{\iota_0\}$ if $op = \textsf{set}$, and $C'(c) = C(c) \cup \{\iota_0\}$ if $op = \textsf{join}$, and $C'(c') = C(c')$ for all $c' \neq c$.

ModifyChannel: A process $\iota \in \mathcal{N}$ takes a modify transition (q, c_1, op, c_2, q'), i.e. if $\mathcal{N}(\iota) = (q, C, M)$ we have $\mathcal{N}'(\iota) = (q', C', M)$ where $C'(c_1) = C(c_1) \cup C(c_2)$ if $op = \textsf{add}$, and $C'(c_1) = C(c_1) \setminus C(c_2)$ if $op = \textsf{del}$, and $C'(c_1) = \{\iota_0\}$ for some $\iota_0 \in C(c_2)$ if $op = \textsf{pick}$, and $C'(c) = C(c)$ for all $c \neq c_1$.

Conditional: A process $\iota \in \mathcal{N}$ takes a conditional transition (q, em, c, q'), i.e. if $\mathcal{N}(\iota) = (q, C, M)$ such that $em \leftrightarrow (C(c) = \emptyset)$, then $\mathcal{N}'(\iota) = (q', C, M)$.

SendEnvMessage: A process $\iota \in \mathcal{N}$ obtains a message from $\mathcal{E}_{\text{msg}}$ with an arbitrary identity from $\mathcal{N}$ attached similarly to the sending case above.

Disappearance: A process $\iota \in \mathcal{N}$ in a fragile state is destroyed, i.e. if $\mathcal{N}(\iota) = (q, C, M)$ and $q \in \Omega$ then $\mathcal{N}' = \mathcal{N}\mid_{dom(\mathcal{N})\setminus\{\iota\}}$. Process ι then disappears from all channels, i.e. for all processes $\iota' \in \mathcal{N}'$ with $\mathcal{N}(\iota') = (q, C, M)$ we have $\mathcal{N}'(\iota') = (q, C', M)$ with $C'(c) = C(c) \setminus \{\iota\}$ for all channels c.

The *semantics* of $\mathcal{P}$ is the transition system with the (in general) infinite set of (unbounded) topologies of $\mathcal{P}$ as states, the empty topology as initial state, and transitions given by the evolution relation $\rightsquigarrow$ as defined above. That is, Fig. 3 models that cars freely enter the highway, without an upper bound on the number of cars. Triggered by the environment message '*car_ahead*', free agents or platoons then merge into platoons forming topologies of interlinked cars.

Requirements Specification Language. In [2], DCS is complemented by the requirements specification language METT. A first example of a METT requirement has already been given in Section 1 in formula (1). METT is basically a first-order extension of LTL providing quantification over anonymous objects and the predicates given by Table 1, which refer to processes' local state, topology, and communication. Given a METT formula μ and a DCS protocol $\mathcal{P}$, the satisfaction relation $\mathcal{P} \models \mu$ is inductively defined on the transition system of $\mathcal{P}$ (cf. [2]).

3 Data Type Reduction

By the definitions in Section 2, we have to deal with three dimensions of complexity when verifying METT properties for DCS: finite control within each process, unbounded, evolving topologies of processes, and queue based communication. Here, we focus on the first two layers and restrict message queues to length 1.

Data Type Reduction (DTR) is an abstraction technique, which has originally been introduced for the verification of properties of parameterised systems [13]. In [7], it has been demonstrated that this abstraction applies as well to systems with unbounded dynamic creation and destruction of processes, thus in particular to DCS. Beyond [7,13], for space reasons, we only recall as much of DTR as is necessary to understand the intrinsic problem of spurious counter-examples to be cured with topology invariants (cf. Section 4).

DTR actually applies to quantifier-free properties with free variables, like μ_0 of (1) on page 36. The universally quantified case, i.e. the whole formula (1), follows by symmetry from finitely many cases [13]. DTR employs the spotlight principle [18], that is, it maps each concrete topology $\mathcal{N}$ to an *abstract topology* where a fixed number of processes is kept completely precise and information about the rest is lost (cf. Fig. 1). An abstract topology $\mathcal{N}^\sharp_N$ maps the limited set of identities $Id^\sharp_N = \{u_1, \ldots, u_N, \bot\}$, where N is the number of individuals kept precise, to local configurations from $\mathcal{S}(\mathcal{P})$. Thereby, the set of identities is reduced to a finite set and consequently there are only finitely many abstract topologies. The abstraction of a concrete topology like Fig. 1(d) is obtained from Fig. 1(b) by replacing all identities, in links and messages, belonging to the rest by the special identity $\bot$. The local configuration of $\bot$ is the upper bound of all *possible* local configurations, not only of all the ones present in the original topology; this is graphically illustrated for links by dashed arrows in Fig. 1(d). Existential abstraction of the original system's transition relation yields a finite-state transition system with abstract topologies as states (cf. [18]). Identity $\bot$ is special as it compares inconclusive to itself and unequal to others, i.e. when the leaders of two cars are compared and are $\bot$, then both possible outcomes are explored. With $\mathcal{P}^\sharp_N, \theta \models \mu_0$ denoting that the METT formula μ_0 is valid under assignment θ in the abstract transition system induced by DTR for $\mathcal{P}$ and $N \in \mathbb{N}$, which can heuristically be chosen depending on the formula, we have

Lemma 1 (Soundness of DTR [7]). *Given a DCS protocol $\mathcal{P}$ and a quantifier-free* METT *formula μ_0 over variables $p_1, \ldots, p_n$, DTR is sound for any $N \in \mathbb{N}$ and assignment $\theta : \{p_1, \ldots, p_n\} \to Id^\sharp_N \setminus \{\bot\}$, i.e. $\mathcal{P}^\sharp_N, \theta \models \mu_0 \Rightarrow \mathcal{P}, \theta \models \mu_0$.* ♢

The abstract transition relation is easily obtained by a syntactical transformation of the DCS protocol because only information *local* to the finitely many processes in the spotlight has to be represented [19]. This property is easily lost when trying to explicitly add precision to the abstract topologies. However, the abstraction is rather coarse allowing for many spurious interference initiated from the shadows as described in Section 1.

4 Topology Analysis

As outlined in Section 1, the idea presented in this paper is to add precision to the rather coarse abstract transition system obtained by DTR (cf. Section 3) by forcing it to adhere to certain, independently established, invariants.

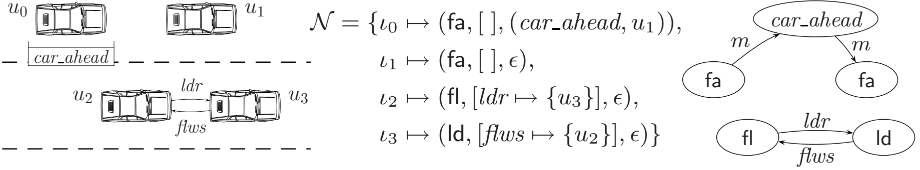


Fig. 4. DCS topology $\mathcal{N}$ and its graph representation

A particular approach to obtain information about legal topologies is Topology Analysis (TA) [4]. Its subject are directed node- and edge-labelled graphs and graph grammars, that is, sets of graph transformation rules. The static analysis of a graph grammar yields a rather precise finite over-approximation, called *topology invariant*, of all graphs possibly generated by the graph grammar. Formally, topology invariants are sets of *abstract clusters*. An instance of an abstract cluster is any graph that can be abstracted to it by *partner abstraction*. Partner abstraction in turn is quotient graph building with respect to partner equivalence, which is motivated by preserving information about *who is talking when to whom*. Intuitively, two processes are partner equivalent if they are in the same state and if they have links to the same kind of processes — regardless of the number of such communication partners. In the context of dynamic *communication* systems, such information is valuable, because it is really this information that determines possible successor topologies of a given topology.

In the following, we rephrase the technique of [4] and contribute an encoding of DCS into graph grammars, thereby making TA amenable to DCS verification.

Topology Analysis. A *graph* is a five-tuple $G = (V, E, s, t, \ell)$ featuring a set of nodes, a set of edges, a source-, a target-, and a labelling function. Source and target functions map edges to their respective source and target nodes, while ℓ maps both, nodes and edges, to labels. A *graph grammar* $\mathfrak{G}$ is a finite set of graph transformation rules. A graph transformation rule consists of two graphs, a *left graph* L and a *right graph* R, and a relation between them indicating which nodes and edges in L and R correspond to each other. In the rule shown in Fig. 5, this correspondence is given implicitly by position. A rule can be *applied* to a graph G, if L is a subgraph of G. The result of the application is the replacement of L's occurrence in G with R. For more details we refer to [4].

Two nodes $u_1, u_2 \in V$ are partner equivalent if $\ell(u_1) = \ell(u_2)$ and if for all edge labels a, $o_G(a, u_1) = o_G(a, u_2)$ and $i_G(a, u_1) = i_G(a, u_2)$. These sets denote the labels of the nodes adjacent to u_1 and u_2, that is $o_G(a, u_1) := \{\ell(v) \mid \exists e \in E : s(e) = u_1, t(e) = v, \ell(e) = a\}$ and analogously for incoming edges.

Given a graph G, the *abstract cluster* $\alpha_{TA}(G)$ is an abstraction of G. It is computed in two steps: First, for each connected component C of G compute the quotient graph with respect to partner equivalence. Doing so, mark equivalence classes consisting of more than one node as a *summary node*. As a second step, summarise isomorphic quotient graphs, that is, keep only one of them.

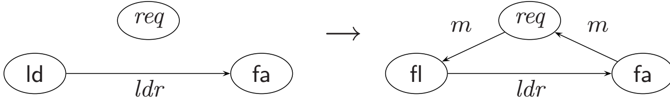


Fig. 5. Graph transformation rule: a platoon approaches a free agent, the platoon leader is in state ld and sends a *req* message to the free agent in front with its identity as parameter. Afterwards the former leader is in state fl.

The set of abstract clusters obtained by the abstract interpretation based on partner abstraction for a graph grammar $\mathfrak{G}$ and the empty graph as initial graph is called *topology invariant* of $\mathfrak{G}$ and denoted by $\mathscr{G}_{\mathfrak{G}}$.

Lemma 2 (Soundness of TA [4]). *Let $\mathfrak{G}$ be a graph grammar and graph G obtained from the empty graph by applying $\mathfrak{G}$. Then $\alpha_{TA}(G) \subseteq \mathscr{G}_{\mathfrak{G}}$.* $\diamondsuit$

DCS Topologies as Graphs. Let $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, succ)$ be a DCS protocol. A topology $\mathcal{N} : Id \rightharpoonup \mathcal{S}(\mathcal{P})$ of $\mathcal{P}$ is encoded as a directed, node- and edge-labelled graph $T(\mathcal{N})$ as follows. For each process in $dom(\mathcal{N})$ and each message in one of the processes' message queues, there is a node. Nodes representing processes are labelled with their local state $q \in Q$, nodes representing messages are labelled with the message name $e \in \Sigma$. For each process u and each channel $c \in \chi$, there are edges labelled with c to each element of $C(c)$.

For each message, there is an edge labelled m from the destination to the message node and from the message to its parameter (cf. Fig. 4). Note that this representation of queues is only feasible for our restriction to finite queue lengths. Unbounded message queues would have to be properly encoded as lists.

Actions as Graph Transformation Rules. Each element of *succ* is translated into a set of graph transformation rules. An example of a graph transformation rule resulting from the "send identity" action $(\mathsf{ld}, req, ldr, id, \mathsf{fl}) \in Snd$ is shown in Fig. 5. The left graph shows a process u_1 in state ld that is connected to process u_2 via channel *ldr*. The result of u_1 sending its own identity attached to message *req* to process u_2 is exactly the right graph. Process u_1 has changed its state to fl, u_2 has an m-labelled edge to message node u_m, and message node u_m is connected to process u_1. Environment interaction like process creation or environment messages are translated similarily. Note that this translation of a DCS protocol into graph transformation rules can be conducted fully automatically.

Topology Invariants. Applying Topology Analysis to the encoding of the platoon merge DCS yields a topology invariant which comprises in particular the four abstract clusters shown in Fig. 6. Intuitively, an abstract cluster denotes that in each topology there may be arbitrary many, that is zero or more, instances of it. For example, $C_1^\sharp$ denotes the possibility of arbitrary many free agents in each topology of the platoon merge DCS. In Fig. 6, summary nodes are drawn with a double-outline. So abstract cluster $C_2^\sharp$ represents an arbitrary number of platoons of size two, whereas $C_3^\sharp$ represents an arbitrary number of platoons of

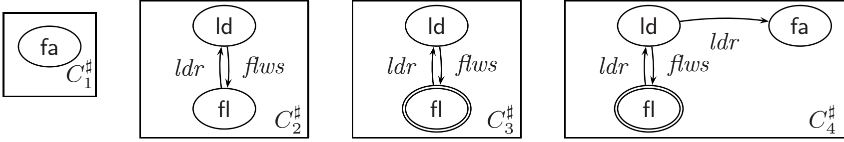


Fig. 6. Abstract clusters. Doubly outlined nodes are summary nodes.

size greater than two. Altogether, a topology invariant denotes any combination of possible instances of the abstract clusters belonging to it.

In terms of a DCS protocol $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, succ)$, a topology invariant is a set $\mathscr{G}_\mathcal{P} = \{C_1^\sharp, \ldots, C_n^\sharp\}$ of abstract clusters. The labelling ℓ_i of $C_i^\sharp$ labels nodes with pairs (st, sm) where st is either a DCS protocol state from Q or a message from Σ. The boolean flag sm indicates whether the node is a summary node or not. Each edge in E_i is labelled by ℓ_i with a channel from χ. Given this encoding we can apply Lemma 2 and obtain:

Corollary 1. *Given DCS protocol $\mathcal{P}$ and topology $\mathcal{N}$ of $\mathcal{P}$, $\alpha(T(\mathcal{N})) \subseteq \mathscr{G}_\mathcal{P}$.* $\diamondsuit$

It proves to be a difficult task to relate the knowledge about valid clusters with the abstract topologies obtained by DTR. We explain our approach to this problem being one of the major contributions of this work in the next section.

5 Putting It Together: Respecting Topology Invariants

Recall from Section 3 that DTR yields a finite-state transition system operating on abstract topologies $\mathcal{N}^\sharp$. They have the form shown in Fig. 1(d), that is, finitely many concrete processes and a summary node representing the rest.

Materialisation. We have briefly discussed in Sections 1 and 3 that this coarse abstraction gives rise to spurious behaviour which can in many cases (manually) be identified as spurious by examining messages that are sent from the summarised rest to concrete processes. For example, consider the abstract topology shown in Fig. 7(a), which is the same as the one in Fig. 2. If in this abstract topology $\mathcal{N}^\sharp$, the summary process $\perp$ sends a message '$newld(\perp)$' to c, we can conclude that the concretisation of $\mathcal{N}^\sharp$ comprises a topology where there is a process capable of sending such a message. Inspecting the DCS protocol (cf. Fig. 3), we see that there must be at least one process ι_{mat} in the rest, which is in state ld, which necessarily has a follower link to c to know the destination, and which has a leader link either to $\perp$ or to itself to know the sent identity. Concerning the link from c_f to $\perp$, we cannot definitely conclude whether it has a link to only one or both of the grayish nodes, that is, there are three cases. Adding all this information to Fig. 7(a) yields Fig. 7(b), which shows one of the six possible materialisations. Figuratively speaking, the view on an abstract topology changes from one with only a single "dark gray" summary process to one with an additional "light gray" process that has been materialised from the summary

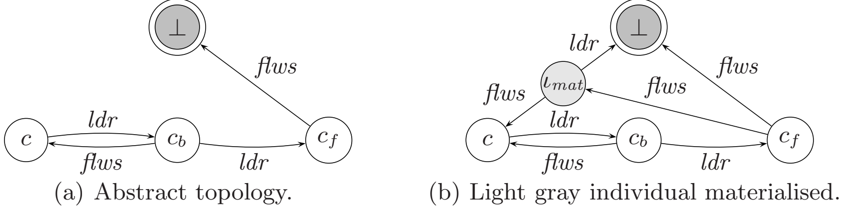


Fig. 7. Materialisation. When a message is sent from the summary process (the dark gray individual), we can conclude on a part of the topology. In the example, we can conclude that there has to be a process having c as follower and c_f as leader.

process. Note that the materialised process is still "gray" in the sense that we typically don't know everything about its configuration but only about the parts involved in the observed action. For example, we cannot conclude whether ι_{mat} has more followers than c.

More formally, if we're in abstract state $\mathcal{N}^\sharp$, representing topology $\mathcal{N}$, and action $ac = (q, c, m, c', q') \in \mathit{Snd}$ is to be executed by the process summary $\perp$ we can conclude that (i) there is a process ι_{mat} summarised by $\perp$, which has local state q, (ii) this process ι_{mat} has ι in channel c, and (iii) it has ι' in channel c'.

Here, the topology invariants automatically established by Topology Analysis (cf. Section 4) come into play as follows. For each abstract topology $\mathcal{N}^\sharp$ in a run of the abstract transition system and each send action $ac \in \mathit{Snd}$ we can derive a (finite) set $\mathit{mat}(\mathcal{N}^\sharp, ac)$ of materialisations following the reasoning above. The run is spurious, if it employs at least one abstract topology and action such that all materialisations are *definitely illegal*. We call a materialisation as the one in Fig. 7(b) definitely illegal if it is *contradictory* to topology invariant $\mathscr{G}_\mathcal{P}$. For this, note that a materialisation $\mathcal{N}^\sharp_{mat}$ again represents a set of concrete topologies which we denote by $\gamma_{DTR}(\mathcal{N}^\sharp_{mat})$. Then materialisation $\mathcal{N}^\sharp_{mat}$ is contradictory to $\mathscr{G}_\mathcal{P}$ iff $\alpha_{TA}(T(\gamma_{DTR}(\mathcal{N}^\sharp_{mat}))) \not\subseteq \mathscr{G}_\mathcal{P}$. Note that this definition doesn't directly yield a decision procedure because $\gamma_{DTR}(\mathcal{N}^\sharp_{mat})$ is in general an infinite set.

Logical Characterisation of Topology Invariants. In the following, we present a solution which isn't based on computing the infinite concretisations but employs a kind of unification between materialised abstract topologies and abstract clusters. This unification is expressed in terms of existentially quantifying predicate logic formulae to be evaluated over materialisations in the 3-valued Kleene interpretation of logic. Using 3-valued logic we treat the fact that materialised nodes are still "light gray", that is, we may not know all of their attributes. References to other predicates of the car, e.g. whether it is currently accelerating, would yield $1/2$ as the definite value cannot be concluded from the observed communication behaviour. Indefiniteness propagates naturally over logical connective, that is, $(1 \wedge 1/2)$ yields $1/2$, while $(0 \wedge 1/2)$ yields 0. A formula evaluating to 1 then indicates that a materialisation is definitely feasible, in case of $1/2$, it is possibly feasible, and in case of 0 it is definitely impossible.

After materialisation, we have finite sets of abstract topologies and abstract clusters $\mathscr{G}_\mathcal{P}$ obtained by Topology Analysis, that is, two kinds of graphs with summary nodes that may match in numerous ways. We only consider the white and light gray processes in the materialisations and individually check whether *their* situation is possible according to $\mathscr{G}_\mathcal{P}$. In other words, if the situation of a white or light gray process in a materialisation $\mathcal{N}^\sharp_{mat}$ doesn't occur in $\mathscr{G}_\mathcal{P}$, then none of the concretisations of $\mathcal{N}^\sharp_{mat}$ are feasible in the concrete system (by Corollary 1). Then if none of the materialisations from $mat(\mathcal{N}^\sharp, ac)$ are feasible, all system runs on which ac applied to $\mathcal{N}^\sharp$ is observed are spurious. We call an abstract topology materialisation $\mathcal{N}^\sharp_{mat}$ *possibly legal* wrt. $\mathscr{G}_\mathcal{P}$ iff for each white and light gray process $\iota \in \mathcal{N}^\sharp$ there is an abstract cluster $C^\sharp = (V, E, s, t, \ell) \in \mathscr{G}_\mathcal{P}$ and a *matching* node $v \in V$ such that $\ell(v)$ is the state of ι, for each c-edge from v to v' there exists a process ι' in state $\ell(v')$ and the channel c of ι comprises ι', and each c'-edge from v' to v implies that the channel c' of ι' comprises ι.

In order to respect the topology invariant $\mathscr{G}_\mathcal{P}$ during model-checking we express the relation induced by (a)–(c) by a logical formula. A process $\iota \neq \bot$ in a materialised abstract topology $\mathcal{N}^\sharp_{mat}$ has $v \in V$ as *matching node* iff there is a bijection $\beta : \{v' \in V \mid (v, v') \in E\} \to dom(\mathcal{N}^\sharp_{mat})$ such that

$$\phi_v(\mathcal{N}^\sharp_{mat}, \iota) := q(\iota) = \ell(v) \wedge \bigwedge_{dom(\beta)} \Big(q(\beta(v')) = \ell(v') \wedge \bigwedge_{\{e \in E \mid s(e)=v \wedge t(e)=v'\}} \ell(e)(\iota, \beta(v')) \wedge \bigwedge_{\{e \in E \mid s(e)=v' \wedge t(e)=v\}} \ell(e)(\beta(v'), \iota) \Big) \quad (2)$$

is not 0, i.e. either 1 or 1/2 (see above), where $q(\iota)$ denotes the state of process ι and $c(\iota, \iota')$ yields true iff there is a link c from ι to ι' in $\mathcal{N}^\sharp_{mat}$. Lifting this characterisation to the level of the whole abstract cluster, the process ι is possibly legal according to $C^\sharp$ if one node of $C^\sharp$ matches, i.e. if

$$\phi_C \ (\mathcal{N}^\sharp_{mat}, \iota) := \exists\, v \in V : \phi_v(\mathcal{N}^\sharp_{mat}, \iota) \quad (3)$$

is not 0. A process ι in $\mathcal{N}^\sharp_{mat}$ is possibly legal according to $\mathscr{G}_\mathcal{P}$ if it is possible legal according to one of the abstract clusters, i.e. if

$$\phi_{\mathscr{G}_\mathcal{P}}(\mathcal{N}^\sharp_{mat}, \iota) := \exists\, C^\sharp \in \mathscr{G}_\mathcal{P} : \phi_C \ (\mathcal{N}^\sharp_{mat}, \iota) \quad (4)$$

is not 0. Finally, the whole materialised abstract topology $\mathcal{N}^\sharp_{mat}$ is possibly legal according to $\mathscr{G}_\mathcal{P}$ if all processes are possibly legal, i.e. if

$$\phi_{\mathscr{G}_\mathcal{P}}(\mathcal{N}^\sharp_{mat}) := \forall\, \iota \in dom(\mathcal{N}^\sharp_{mat}) \setminus \{\bot\} : \phi_{\mathscr{G}_\mathcal{P}}(\mathcal{N}^\sharp_{mat}, \iota). \quad (5)$$

is not 0. Note that the quantifications in (3) - (5) are over finite sets, thus expand to unquantified predicate logic expressions.

For example, let $\mathcal{N}^\sharp_{mat}$ denote the materialised abstract topology from Fig. 7(b). Than $\phi_{C_1}(\mathcal{N}^\sharp_{mat}, \iota_{mat})$ evaluates to 0 for $C^\sharp_1$ from Fig. 6 because ι_{mat} is in state ld and no node in $C^\sharp_1$ has this label. Also $\phi_{C_4}(\mathcal{N}^\sharp_{mat}, \iota_{mat})$ evaluates to 0 although

there is a node labelled with ld in $C_4^\sharp$, but $C_4^\sharp$ requires that there is a leader link back to the ld-node from each follower, which is not the case for ι_{mat}. Actually, none of the abstract clusters in $\mathscr{G}_\mathcal{P}$ matches, thus Fig. 7(b) is definitely illegal.

Assuming Adherence. The indication of spuriousness, namely executing an action ac in abstract topology $\mathcal{N}^\sharp$ whose effect would require the existence of an illegal topology in the concrete, is a non-temporal property depending only on $\mathcal{N}^\sharp$ and ac. As such it can easily be added as a boolean observer to an abstract transition system $\mathcal{P}_N^\sharp$. Then by $\mathcal{P}_N^\sharp, \theta, \mathscr{G}_\mathcal{P} \models \mu_0$, we denote that the abstract transition system obtained by DTR satisfies μ_0 on those system runs, where the observer doesn't indicate spuriousness and we have the following.

Theorem 1 (Soundness of DTR+TA). *Let $\mathcal{P}$ be a DCS protocol and μ_0 a* METT *formula over variables $p_1, \ldots, p_M$. Then given $N \in \mathbb{N}$ and an assignment $\theta = \{p_i \mapsto u_i\}$, DTR+TA is sound, i.e. $\mathcal{P}_N^\sharp, \theta, \mathscr{G}_\mathcal{P} \models \mu_0 \implies \mathcal{P}, \theta \models \mu_0$.* ♢

The proof is based on Corollary 1 by which the restriction to $\mathscr{G}_\mathcal{P}$ only removes transitions from $\mathcal{P}_N^\sharp$ which lead to topologies that aren't possible in the original transition system. Consequently, no original system behaviour is disregarded.

Note that the example discussed below indicates that we indeed obtain a *proper* refinement, i.e. there are properties that cannot be established in result of DTR but can in combination with topology invariants.

Experimental Results. For a proof-of-concept implementation, we had to strengthen the platoon-merge protocol in comparison to the one shown in Fig. 3. The handover of followers to the new leader on a merge, for instance, needs guarding acknowledge messages. This protocol provides for a topology invariant with 77 abstract clusters automatically computed by an implementation of the Topology Analysis [4]; a looser merge protocol easily yields 2000 clusters.[1]

We used the tool-set of [15] to translate the strengthened merge protocol into the input language of the VIS [8] model-checker. Most recently, we have implemented the translation of DCS protocols into graph transformation rules fully automating our tool chain. Without respecting the topology invariants, the model-checker unveils the counter-example discussed in the introduction in about 90 minutes. After encoding the topology invariant following the procedure from the previous section, we were able to prove the property in about 126 minutes. Note that Topology Analysis alone is not able to establish (1) because it comprises a liveness requirement and TA only addresses safety properties.

6 Conclusion

We promote a combination of an easily obtainable but rather coarse abstract transition system with external information obtained by static analysis. Our technique has a number of benefits. It can automatically prove properties that

[1] For the strengthened DCS protocol and the actually employed clusters see the full version [3] of this paper.

neither technique can prove in isolation demonstrating synergy. It formally integrates techniques, namely model checking and static program analysis, that are often considered orthogonal. Moreover, it is able to automatically discover flaws in sophisticated traffic control applications [9] that could only be found manually [2] before. Finally, the technique has been brought to full automation by integrating existing tools.

The technique introduced in Section 5 uses only a small fraction of the information carried by abstract clusters. Namely, for conciseness and to bound the size of the resulting formulae, we only considered *positive* links to distance-1 nodes, that is, the *direct* neighbourhood of nodes, as being relevant to exclude spurious behaviour. Further work comprises the evaluation of two aspects: looking further into the abstract cluster, i.e. to check whether processes up to some distance larger than 1 are legal according to the topology invariant, and considering information about the *absence* of links in abstract clusters.

Additionally, a kind of counter-example guided abstraction refinement could be established where each action of the summary process in the counter-example is checked for validity with respect to the topology invariant, and if one action turns out to be spurious, then in the next run only those abstract clusters are (soundly) considered that relate to the particular spurious interferences.

Regarding applications, we are currently implementing a number of examples from application domains such as mobile ad-hoc networks and service-oriented computing as DCS protocols. This is necessary to gather more experience on usability and scalability of our integrated, automated toolset. In order to facilitate the implementation of examples of realistic size, we plan to write a front-end automatically compiling more high level specifications, e.g. written in UML, into DCS protocols.

References

1. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, Springer, Heidelberg (2004)
2. Bauer, J., Schaefer, I., Toben, T., Westphal, B.: Specification and Verification of Dynamic Communication Systems. In: Proc. ACSD 2006, IEEE, Los Alamitos (2006)
3. Bauer, J., Toben, T., Westphal, B.: Mind the shapes: Abstraction refinement via topology invariants. Reports of SFB/TR 14 AVACS 22, SFB/TR 14 AVACS, (June 2007) available at `http://www.avacs.org` ISSN: 1860-9821
4. Bauer, J., Wilhelm, R.: Static Analysis of Dynamic Communication Systems by Partner Abstraction. In: Nielson, H.R, filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the Association for Computing Machinery 30(2), 323–342 (1983)
6. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)

7. Damm, W., Westphal, B.: Live and let die: LSC-based verification of UML-models. Science of Computer Programming 55(1–3), 117–159 (2005)
8. Brayton, R.K., et al.: VIS: a system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)
9. Hsu, A., Eskafi, F., Sachs, S., Varaiya, P.: The design of platoon maneuver protocols for IVHS. PATH Report UCB-ITS-PRR-91-6, U. California (April 1991)
10. Jain, H., et al.: Using statically computed invariants in the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)
11. Lubachevsky, B.D.: An approach to automating the verification of compact parallel coordination programs. Acta Inf. 21, 125–169 (1984)
12. Lynch, N.A.: Input/output automata: Basic, timed, hybrid, probabilistic, dynamic. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 187–188. Springer, Heidelberg (2003)
13. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Science of Computer Programming 37, 279–309 (2000)
14. Milner, R.: The π Calculus. Cambridge University Press, Cambridge (1999)
15. Rakow, J.: Verification of Dynamic Communication Systems. Master's thesis, Carl von Ossietzky Universität Oldenburg (April 2006)
16. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. 157(1), 39–59 (2006)
17. Venet, A.: Automatic determination of communication topologies in mobile systems. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 152–167. Springer, Heidelberg (1998)
18. Wachter, B., Westphal, B.: The spotlight principle. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 182–198. Springer, Heidelberg (2007)
19. Westphal, B.: LSC verification for UML models with unbounded creation and destruction. In: Proc. SoftMC 2005. ENTCS, 144(3),133–145 (2005)

Complete SAT-Based Model Checking for Context-Free Processes*

Geng-Dian Huang[1,2] and Bow-Yaw Wang[1]

[1] Institute of Information Science
Academia Sinica, Taiwan
[2] Department of Electrical Engineering
National Taiwan University, Taiwan
{gdhuang|bywang}@iis.sinica.edu.tw

Abstract. A complete SAT-based model checking algorithm for context-free processes is presented. We reduce proof search in local model checking to Boolean satisfiability. Bounded proof search can therefore be performed by SAT solvers. Moreover, the completion of proof search is reduced to Boolean unsatisfiability and hence can be checked by SAT solvers. By encoding the local model checking algorithm in [13], SAT solvers are able to verify properties in the universal fragment of alternation-free μ-calculus formula on context-free processes.

1 Introduction

Since pushdown systems give natural representations of program control flows, problems in program analysis can be reduced to verification problems on the infinite-state model. In the past years, efficient verification algorithms for pushdown systems have been proposed [11,12]. Experimental results suggest that the BDD-based algorithm for the succinct model could be much more space-efficient than those for finite-state systems in program analysis [12].

Meanwhile, hardware verification has been influenced by the development of practical satisfiability (SAT) solvers [4,3]. Thanks to various heuristics, SAT solvers are very efficient in both time and space. By reducing bounded verification problems to Boolean satisfiability, the technique can detect flaws in finite-state models unattainable by explicit-state or BDD-based algorithms.

SAT-based verification algorithms for finite-state systems make the experiment in [12] regretfully obsolete. Since `bebop` uses a BDD-based algorithm [2], it is unclear how the explicit-state [5,11] or BDD-based [12] algorithms for pushdown systems compare with SAT-based algorithms for finite-state systems. Moreover, the explicit-state and BDD-based algorithms for pushdown systems might suffer from the same capacity problem as in finite-state systems. An SAT-based algorithm for pushdown systems could be more scalable.

* The work is partly supported by NSC grands 95-3114-P-001-002-Y02, 95-2221-E-001-024-MY3, and the SISARL thematic project of Academia Sinica.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 51–65, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In this paper, we give a complete SAT-based model checking algorithm for the universal fragment of alternation-free μ-calculus formulae on context-free processes. Given a context-free grammar, one may view derivations as system evolutions. A context-free grammar thus defines the transition system of a context-free process [6,13]. Although the languages recognized by context-free grammars and pushdown automata coincide, pushdown systems are in fact more expressive than context-free processes [8]. Nevertheless, pushdown systems with only one control state are context-free processes. Problems in program analysis can thus be modeled by context-free processes. Moreover, our preliminary experimental results show that the new algorithm performs better than a BDD-based algorithm for large random models. We feel that our SAT-based verification algorithm could still be useful in program analysis.

Based on the explicit-state algorithm in [6], a local model checking algorithm for alternation-free μ-calculus formulae on context-free processes is developed [13]. We construct a Boolean formula whose satisfiability is equivalent to a bounded proof. If no proof within certain bounds can be found, the completion of proof search is then established by the unsatisfiability of another formula. Our SAT-based model checking algorithm therefore reduces proof search of the local model checking algorithm in [13] to Boolean (un)satisfiability. The universal fragment of alternation-free μ-calculus formulae on context-free processes can hence be verified by the absence of proofs of their negations using SAT solvers.

An explicit-state model checking algorithm for context-free processes is given in [6]. Second-order assertions specify properties on sets of states under contextual assumptions. Since the given property is of main concern, formulae in its closure are sufficient for contextual assumptions. The model checking problem is solved by computing contextual assumptions on finite representations. Employing second-order assertions, a local model checking algorithm for alternation-free μ-calculus formulae on context-free processes is developed in [13].

Complete SAT-based model checking algorithms for finite-state models can be found in literature [15,1,18]. Interpolation is exploited to verify invariants [15]. In [1], SAT solvers are used to detect cycles and check properties in linear temporal logic. A similar technique based on local model checking is able to verify the universal fragment of μ-calculus properties by SAT solvers [18].

Verification algorithms for pushdown systems have also been proposed [16,5,11,12,17]. Model checking monadic second-order logic properties is known to be decidable but with a non-elementary upper bound [16]. Verifying μ-calculus properties is DEXPTIME-complete for pushdown systems [5]. For linear properties, the problem can be solved in polynomial time but requires polynomial space [11]. A BDD-based algorithm is compared with the software verification tool `bebop` in [12]. Finally, a game-theoretic algorithm is given in [17].

The paper is organized as follows. Section 2 gives backgrounds. Our reduction of proof search to Boolean satisfiability is presented in Section 3. The SAT-based model checking algorithm for the universal fragment of alternation-free μ-calculus formulae is shown in Section 4. Preliminary experimental results are reported in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

A context-free process is a finite-state automaton with procedure calls and two designated locations.[1] Procedure invocation is denoted by names. The unique entry and exit points of procedures are represented by the start and end locations respectively.

Definition 1. *A* context-free process $P = \langle \Sigma, N, Act, \rightarrow_P, \delta, \epsilon \rangle$ *is a tuple where*

- Σ *is a finite set of* locations*;*
- N *and* Act *are finite sets of* names *and* actions *respectively;*
- $\rightarrow_P \subseteq \Sigma \times (N \cup Act) \times \Sigma$ *is its* transition relation*; and*
- δ *and* ϵ *are are the* start *and* end *locations respectively.*

For clarity, we write $\sigma \xrightarrow{\alpha} \sigma'$ for $(\sigma, \alpha, \sigma') \in \rightarrow$. A context-free process is *guarded* if for all $\delta \xrightarrow{\alpha} \sigma$, we have $\alpha \in Act$. We only consider guarded context-free processes in the following presentation.

A context-free process system is a set of recursively defined context-free processes. Let $\underline{n}$ be the name set $\{0, 1, \ldots, n\}$ and the name i denote the invocation of process P_i. Since all context-free processes share the same name set, mutual recursion can be modeled easily. In our setting, context-free processes and basic process algebra are in fact equivalent [9,7].

Definition 2. *A* context-free process system $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ *consists of context-free processes* $P_0, \ldots, P_n$ *where*

- $P_0, \ldots, P_n$ *share the sets of names* $\underline{n}$ *and actions* Act*;*
- P_0 *is the* main process.

A context-free process system serves as a finite representation of a process graph. A process graph is a transition system with designated start and end states, and may have an infinite number of states.

Definition 3. *A* process graph $G = \langle S, Act, \rightarrow, s_0, s_e \rangle$ *is a tuple where*

- S *is the set of* states*;*
- Act *is the finite set of* actions*;*
- $\rightarrow \subseteq S \times Act \times S$ *is its* transition relation*; and*
- s_0 *and* s_e *are the* start *and* end *states respectively.*

The process graph represented by a context-free system is obtained by expanding recursive calls. Observe that copies of context-free processes can be made infinitely many times. A location in a context-free process may correspond to an infinite number of states in the process graph.

Definition 4. *Let* $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ *be a context-free process system with* $P_i = \langle \Sigma_i, \underline{n}, Act, \rightarrow_i, \delta_i, \epsilon_i \rangle$ *for* $0 \leq i \leq n$*. The* process graph $PG(\mathbb{P})$ of $\mathbb{P}$ *is obtained by expanding* $s_0 \xrightarrow{0} s_e$ *recursively as follows.*

[1] The term "location" is called "state class" in [6,13].

1. *For each transition $s \xrightarrow{i} s'$, make a copy of the context-free process P_i; and*
2. *Identify δ_i and ϵ_i with s and s' respectively.*

When a copy of P_i is made, a state s is added for each location σ in Σ_i except δ_i and ϵ_i. We hence say s is an *instance* of σ. The notation $s \in \sigma$ denotes that s is an instance of σ or s is identified with σ when $\sigma = \epsilon_i$. Further, s'' is the *return state* of s (denoted by $end(s)$) if s is added while expanding $s' \xrightarrow{p} s''$. Both $end(s_0)$ and $end(s_e)$ are defined to be s_e.

Given the set Var of *relational variables*, $X \in Var$, and $A \subseteq Act$. The syntax of μ-calculus formulae is defined as follows

$$\phi ::= \mathsf{tt} \big| X \big| \neg\phi \big| \phi_0 \wedge \phi_1 \big| \langle A \rangle \phi \big| \mu X.\phi$$

Relational variables must be bound positively by least fixed point operators in $\mu X.\phi$. We adopt the following abbreviation: ff for $\neg\mathsf{tt}$, $\phi_0 \vee \phi_1$ for $\neg(\neg\phi_0 \wedge \neg\phi_1)$, $[A]\phi$ for $\neg\langle A \rangle \neg\phi$, and $\nu X.\phi$ for $\neg\mu X.\neg\phi[\neg X/X]$. A μ-calculus formula is *alternation-free* if all its fixed point subformulae do not have free relational variables bound by fixed point operators of the other type. The *negative normal form* of a μ-calculus formula is obtained by applying De Morgan's laws repeatedly so that negations appear only before tt. The *universal fragment of μ-calculus formulae* consists of μ-calculus formulae whose negative normal forms do not have existential modal operators ($\langle a \rangle \bullet$). Let ψ be a universal μ-calculus formula. It is easy to verify that the negative normal form of $\neg\psi$ does not have universal modal operators ($[a]\bullet$). In the following, we assume all formulae are in their negative normal forms.

Given a process graph $G = \langle S, Act, \rightarrow, s_0, s_e \rangle$, an environment e is a mapping from Var to 2^S. The notation $e[X \mapsto U]$ denotes the environment that maps X to U and Y to $e(Y)$ for $Y \neq X$. The semantic function $[\![\phi]\!]_e^G$ for the μ-calculus formula ϕ is defined as follows.

$$\begin{aligned}
[\![\mathsf{tt}]\!]_e^G &= S \\
[\![X]\!]_e^G &= e(X) \\
[\![\phi_0 \wedge \phi_1]\!]_e^G &= [\![\phi_0]\!]_e^G \cap [\![\phi_1]\!]_e^G \\
[\![\phi_0 \vee \phi_1]\!]_e^G &= [\![\phi_0]\!]_e^G \cup [\![\phi_1]\!]_e^G \\
[\![[A]\phi]\!]_e^G &= \{s \in S \mid \forall a, s'.a \in A \wedge s \xrightarrow{a} s' \Longrightarrow s' \in [\![\phi]\!]_e^G\} \\
[\![\langle A \rangle \phi]\!]_e^G &= \{s \in S \mid \exists a, s'.a \in A \wedge s \xrightarrow{a} s' \wedge s' \in [\![\phi]\!]_e^G\} \\
[\![\nu X.\phi]\!]_e^G &= \bigcup \{U \subseteq S \mid U \subseteq [\![\phi]\!]_{e[X \mapsto U]}^G\} \\
[\![\mu X.\phi]\!]_e^G &= \bigcap \{U \subseteq S \mid U \supseteq [\![\phi]\!]_{e[X \mapsto U]}^G\}
\end{aligned}$$

We say *s satisfies ϕ in process graph G* (denoted by $G, s \models \phi$) if $s \in [\![\phi]\!]_\emptyset^G$. Let Φ be a set of μ-calculus formulae. Define $G, s \models \Phi$ if $G, s \models \phi$ for all $\phi \in \Phi$. If $\mathbb{P}$ is a context-free system, we say $\mathbb{P}$ *satisfies* ϕ, $\mathbb{P} \models \phi$, if $PG(\mathbb{P}), s_0 \models \phi$. When there is no ambiguity, we write $s \models \phi$ and $s \models \Phi$ for $G, s \models \phi$ and $G, s \models \Phi$.

Since different instances of a location may be instantiated in different invocations, one cannot naively expect all instances to satisfy the same property. Contextual assumptions are hence used in the specification of locations. They are chosen from closures of μ-calculus formulae and postulated during process invocation [6,13].

Definition 5. *The* closure $CL(\phi)$ *of a μ-calculus formula ϕ is inductively defined as follows.*

$$\begin{aligned} CL(\mathsf{tt}) &= \emptyset \\ CL(\phi_0 \wedge \phi_1) &= \{\phi_0 \wedge \phi_1\} \cup CL(\phi_0) \cup CL(\phi_1) \\ CL(\phi_0 \vee \phi_1) &= \{\phi_0 \vee \phi_1\} \cup CL(\phi_0) \cup CL(\phi_1) \\ CL([A]\phi) &= \{[A]\phi\} \cup CL(\phi) \\ CL(\langle A \rangle\phi) &= \{\langle A \rangle\phi\} \cup CL(\phi) \\ CL(\nu X.\phi) &= \{\nu X.\phi\} \cup CL(\phi[\nu X.\phi/X]) \\ CL(\mu X.\phi) &= \{\mu X.\phi\} \cup CL(\phi[\mu X.\phi/X]) \end{aligned}$$

Given a μ-calculus formula ϕ and a set of μ-calculus formulae $\Theta \subseteq CL(\phi)$, the pair $\langle \phi, \Theta \rangle$ is called a *second-order assertion*. Define $\sigma \models \langle \phi, \Theta \rangle$ if $s \models \phi$ for $s \in \sigma$ provided $end(s) \models \Theta$. Intuitively, a location satisfies a second-order assertion $\langle \phi, \Theta \rangle$ if its instances under the given contextual assumptions Θ satisfy the μ-calculus formula ϕ.

We now describe the local model checking algorithm in [13]. Let $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ be a context-free process system with $P_i = \langle \Sigma_i, \underline{n}, Act, \rightarrow_i, \delta_i, \epsilon_i \rangle$ for $0 \leq i \leq n$, $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph, and ϕ a μ-calculus formula. A *sequent* is of the form $s \vdash \phi$ or $\sigma \vdash \langle \phi, \Theta \rangle$. We call the former *first-order* and the latter *second-order* sequents respectively. Let Ω be a set of sequents and ω a sequent. An *inference rule* is represented as $\frac{\Omega}{\omega}$. The sequents in Ω and ω are the *premises* and *conclusion* of the inference rule respectively. For clarity, we write $\frac{\omega}{\omega'}$ and $\frac{\omega_0 \qquad \omega_1}{\omega'}$ for $\frac{\{\omega\}}{\omega'}$ and $\frac{\{\omega_0, \omega_1\}}{\omega'}$ respectively. A *proof* is a tree rooted at a given sequent and constructed according to the inference rules in Figure 1. The start rule first guesses initial contextual assumptions Θ for the given property ϕ in the second-order assertion $\langle \phi, \Theta \rangle$. The assumptions Θ must be satisfied after the invocation of the main process. Similarly, contextual assumptions are chosen in modality rules. There are only finitely many possible contextual assumptions for any μ-calculus formula ϕ because $CL(\phi)$ is finite.

To show a location satisfies a conjunction in a second-order assertion, one proves that both conjuncts are satisfied under the same contextual assumptions. Symmetrically, a disjunct under the same assumptions in a second-order assertion must be satisfied in a disjunction. For fixed points, the inference rules simply unroll the formula. The unrolling need be justified on the leaves of the proof (Definition 6 (vi)). A (d, r)-*proof* is a proof which applies the fixed point and modality rules at most d and r times along any path from the root to a leaf respectively.

Definition 6. *A leaf of a proof is* successful *if it has one of the following forms.*

(i) $s_e \vdash \mathsf{tt}$*;*
(ii) $s_e \vdash [A]\phi$*;*
(iii) $\sigma \vdash \langle \mathsf{tt}, \Theta \rangle$*;*
(iv) $\sigma \vdash \langle [A]\phi, \Theta \rangle$ *where* $\sigma \in \Sigma_i$ *is not an end location and there is no* σ' *with* $\sigma \xrightarrow{a}_i \sigma'$ *for any* $a \in A$*, or* $\sigma \xrightarrow{j}_i \sigma'$*;*
(v) $\epsilon \vdash \langle \phi, \Theta \rangle$ *and* $\phi \in \Theta$*; or*
(vi) $\sigma \vdash \langle \phi(\nu X.\psi), \Theta \rangle$ *where* $\phi(\nu X.\psi) \in CL(\nu X.\psi)$ *and the same sequent re-occurs on the path from the root to itself.*

A finite proof is *successful* if all its leaves are successful. A sequent is *derivable* if there is a successful proof rooted in the sequent. A formula ϕ is *derivable* if the sequent $s_0 \vdash \phi$ is derivable. The following theorem shows the inference rules in Figure 1 are sound and complete.

Theorem 1. *([13]) An alternation-free* μ*-calculus formula* ϕ *is derivable for a context-free process system* $\mathbb{P}$ *iff it is satisfied in* $\mathbb{P}$*.*

An exemplary run of the local model checking algorithm is shown in Figure 2. In the figure, a simple context-free process P with an infinite number of states is considered. After performing the action a, the process P can either call itself recursively or terminates by executing b. We verify that the start state s_0 satisfies $\nu X.[a, b]X$ by the successful proof in the figure. Observe there are two nondeterministic choices of contextual assumptions in the applications of start and modality rules. They happen to be the same in the proof.

3 Proof Search by SAT

Suppose $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ is a context-free process system with $P_i = \langle \Sigma_i, \underline{n}, Act, \rightarrow_i, \delta_i, \epsilon_i \rangle$ for $0 \leq i \leq n$, and $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph. Since the number of locations in $\Sigma_0 \cup \cdots \cup \Sigma_n$ is finite, we can use a Boolean vector of size $\lg(\sum_{i=0}^{n} |\Sigma_i|)$ to encode locations. The Boolean vector representing the location σ is denoted by $\overline{\sigma}$. Moreover, we assume a fixed linear order on $CL(\phi)$ for the μ-calculus formula ϕ and denote the i-th formula in $CL(\phi)$ by ϕ_i. Any subset Θ of $CL(\phi)$ can hence be encoded by a Boolean vector $\bar{z}$ of size $|CL(\phi)|$ such that $\bar{z}[i] = \mathsf{tt}$ if and only if $\phi_i \in \Theta$. The Boolean vector representing the subset $\Theta \subseteq CL(\phi)$ is denoted by $\overline{\Theta}$.

Let ϕ be an alternation-free μ-calculus formula without universal modal operators. Figure 3 gives our encoding of proof search in the local model checking algorithm. In the figure, the Boolean variable vectors $\bar{u}, \bar{v}, \bar{w}$ encode locations and are of size $\lg(\Sigma_{i=0}^{n}|\Sigma_i|)$. The Boolean variable vectors $\bar{z}$ and $\bar{z}'$ encode a subset of $CL(\phi)$ and hence of size $|CL(\phi)|$. The list Γ consists of triples of the form $(\bar{u}, \phi, \bar{z})$. It records all second-order sequents $\sigma \vdash \langle \phi, \Theta \rangle$ from the root to the current sequent. The notation $(\bar{u}, \phi, \bar{z}) :: \Gamma$ represents the list whose elements are $(\bar{u}, \phi, \bar{z})$ followed by those in Γ. $|\Gamma|$ denotes the size of the list Γ. Intuitively,

Start rule

$$\frac{\{\delta_0 \vdash \langle \phi, \Theta \rangle\} \cup \{s_e \vdash \theta \mid \theta \in \Theta\}}{s_0 \vdash \phi}$$

End rules

$$\frac{s_e \vdash \phi_0 \qquad s_e \vdash \phi_1}{s_e \vdash \phi_0 \wedge \phi_1}$$

$$\frac{s_e \vdash \phi_0}{s_e \vdash \phi_0 \vee \phi_1} \qquad \frac{s_e \vdash \phi_1}{s_e \vdash \phi_0 \vee \phi_1}$$

$$\frac{s_e \vdash \phi[\mathsf{tt}/X]}{s_e \vdash \nu X.\phi} \qquad \frac{s_e \vdash \phi[\mathsf{ff}/X]}{s_e \vdash \mu X.\phi}$$

Conjunction and disjunction rules

$$\frac{\sigma \vdash \langle \phi_0, \Theta \rangle \qquad \sigma \vdash \langle \phi_1, \Theta \rangle}{\sigma \vdash \langle \phi_0 \wedge \phi_1, \Theta \rangle}$$

$$\frac{\sigma \vdash \langle \phi_0, \Theta \rangle}{\sigma \vdash \langle \phi_0 \vee \phi_1, \Theta \rangle} \qquad \frac{\sigma \vdash \langle \phi_1, \Theta \rangle}{\sigma \vdash \langle \phi_0 \vee \phi_1, \Theta \rangle}$$

Fixed point rule

$$\frac{\sigma \vdash \langle \phi[\nu X.\phi/X], \Theta \rangle}{\sigma \vdash \langle \nu X.\phi, \Theta \rangle} \qquad \frac{\sigma \vdash \langle \phi[\mu X.\phi/X], \Theta \rangle}{\sigma \vdash \langle \mu X.\phi, \Theta \rangle}$$

Modality rules

$$\frac{\{\sigma' \vdash \langle \phi, \Theta \rangle \mid a \in A, \sigma \xrightarrow{a}_i \sigma'\} \cup \bigcup_{j:\sigma \xrightarrow{j}_i \sigma'} (\{\delta_j \vdash \langle [A]\phi, \Psi_j \rangle\} \cup \{\sigma' \vdash \langle \psi, \Theta \rangle \mid \psi \in \Psi_j\})}{\sigma \vdash \langle [A]\phi, \Theta \rangle}$$

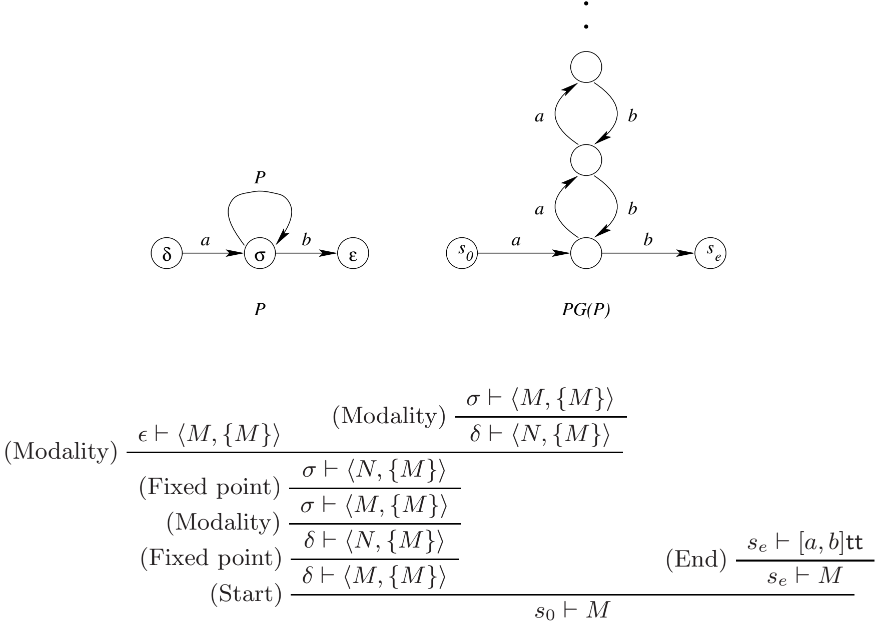
$$\frac{\sigma' \vdash \langle \phi, \Theta \rangle}{\sigma \vdash \langle \langle A \rangle \phi, \Theta \rangle}\ a \in A, \sigma \xrightarrow{a}_i \sigma' \qquad \frac{\{\delta_j \vdash \langle \langle A \rangle \phi, \Psi \rangle\} \cup \{\sigma' \vdash \langle \psi, \Theta \rangle \mid \psi \in \Psi\}}{\sigma \vdash \langle \langle A \rangle \phi, \Theta \rangle}\ \sigma \xrightarrow{j}_i \sigma'$$

Weakening rule

$$\frac{\sigma \vdash \langle \phi, \Theta' \rangle}{\sigma \vdash \langle \phi, \Theta \rangle}\ (\Theta' \subseteq \Theta)$$

Fig. 1. Local Model Checking Algorithm

the idea is to construct a Boolean formula whose satisfiability is equivalent to a bounded proof. Another Boolean formula whose unsatisfiability is equivalent to completion will be built later. The model checking problem for context-free processes is therefore reduced to Boolean (un)satisfiability.



where $M = \nu X.[a, b]X$ and $N = [a, b]\nu X.[a, b]X$

Fig. 2. An Example of Local Model Checking

The following lemma states that our encoding of the end rules is correct.

Lemma 1. *$s_e \vdash \phi'$ is derivable iff* $\lambda(s_e, \phi') = \mathtt{tt}$.

To encode the derivation of the second-order sequent $\sigma \vdash \langle\langle A\rangle\phi', \Theta\rangle$, we let SAT solvers choose the contextual assumption Ψ in modality rules. The new assumption Ψ is represented by $\bar{z}'$ in the Boolean formula Π ($\bar{u}$, A, ϕ', $\bar{z}$, Γ, d, r). Given an assignment ρ, the valuation $[\![\bar{u}]\!]_\rho$ maps a Boolean variable vector $\bar{u}$ to a Boolean vector. The function $\chi(\bar{u}, A, \bar{v})$ in $\pi(\bar{u}$, A, ϕ', $\bar{z}$, Γ, d, $r)$ is $\mathtt{tt}$ with respect to an assignment ρ if and only if $[\![\bar{u}]\!]_\rho = \overline{\sigma}$, $[\![\bar{v}]\!]_\rho = \overline{\sigma}'$, and $\sigma \xrightarrow{a}_i \sigma'$ for some i and $a \in A$. Similarly, $\zeta(\bar{u}, \bar{v}, \bar{w}) = \mathtt{tt}$ with respect to an assignment ρ if and only if $[\![\bar{u}]\!]_\rho = \overline{\sigma}$, $[\![\bar{v}]\!]_\rho = \delta_j$, $[\![\bar{w}]\!]_\rho = \overline{\sigma}'$, and $\sigma \xrightarrow{j}_i \sigma'$ for some i. For the derivation of the sequent $\sigma \vdash \langle\eta X.\phi', \Theta\rangle$ where η is either the least or greatest fixed point operator, we simply unroll the formula and let $Successful_leaf(\bar{u}, \phi', \bar{z}, \Gamma)$ check whether the sequent is successful or not. The variable c_i's in fixed point and modality rules are called *expansion variables*. Intuitively, c_i's indicate a proof needs to apply more fixed point and modality rules. The following lemma shows that a satisfying assignment ρ for $\Lambda(\bar{u}, \phi', \bar{z}, [], d, r) \wedge \bigwedge_{i=0}^{l} \neg c_i$ corresponds to a successful proof for the sequent $\sigma \vdash \langle\phi', \Theta\rangle$ with $\overline{\sigma} = [\![\bar{u}]\!]_\rho$ and $\overline{\Theta} = [\![\bar{z}]\!]_\rho$.

Auxiliaries

$$\Upsilon_0(\bar{u}) \triangleq \bigvee_{i=0}^{n}(\overline{\epsilon}_i = \bar{u}) \qquad \Upsilon_1(\phi_k, \bar{z}) \triangleq \bar{z}[k]$$

$$\Omega_0(\bar{u}, \phi', \bar{z}, \Gamma) \triangleq \bigvee_{k=0}^{|\Gamma|-1}(\bar{u}, \phi', \bar{z}) = \Gamma[k]$$

$$Not_leaf(\bar{u}, \phi', \bar{z}, \Gamma) \triangleq \neg\Upsilon_0(\bar{u}) \wedge \neg\Omega_0(\bar{u}, \phi', \bar{z}, \Gamma)$$

$$Successful_leaf(\bar{u}, \phi', \bar{z}, \Gamma) \triangleq \begin{cases} (\Upsilon_0(\bar{u}) \wedge \Upsilon_1(\phi', \bar{z})) \vee \Omega_0(\bar{u}, \phi', \bar{z}, \Gamma) & \text{if } \phi' \in CL(\nu X.\psi) \\ \Upsilon_0(\bar{u}) \wedge \Upsilon_1(\phi', \bar{z}) & \text{if } \phi' \notin CL(\nu X.\psi) \end{cases}$$

Start rule

$$\alpha(\phi, d, r) \triangleq \bar{u} = \bar{\delta}_0 \wedge \Lambda(\bar{u}, \phi, \bar{z}, [], d, r) \wedge \bigwedge_{k=0}^{|\bar{z}|-1}(\bar{z}[k] \Rightarrow \lambda(s_e, \phi_k))$$

where $\bar{z}$: a vector of fresh Boolean variables of size $|CL(\phi)|$

End rules

$$\lambda(s_e, \phi'_0 \wedge \phi'_1) \triangleq \lambda(s_e, \phi'_0) \wedge \lambda(s_e, \phi'_1) \qquad \lambda(s_e, \phi'_0 \vee \phi'_1) \triangleq \lambda(s_e, \phi'_0) \vee \lambda(s_e, \phi'_1)$$

$$\lambda(s_e, \nu X.\phi') \triangleq \lambda(s_e, \phi'[\mathsf{tt}/X]) \qquad \lambda(s_e, \mu X.\phi') \triangleq \lambda(s_e, \phi'[\mathsf{ff}/X])$$

$$\lambda(s_e, \langle a \rangle \phi') \triangleq \mathsf{ff} \qquad \lambda(s_e, \mathsf{tt}) \triangleq \mathsf{tt} \qquad \lambda(s_e, \mathsf{ff}) \triangleq \mathsf{ff}$$

Conjunction and disjunction rules

$$\Lambda(\bar{u}, \phi'_0 \wedge \phi'_1, \bar{z}, \Gamma, d, r) \triangleq$$
$$Successful_leaf(\bar{u}, \phi'_0 \wedge \phi'_1, \bar{z}, \Gamma) \vee (\Lambda(\bar{u}, \phi'_0, \bar{z}, \Gamma', d, r) \wedge \Lambda(\bar{u}, \phi'_1, \bar{z}, \Gamma', d, r))$$

where $\Gamma' = (\bar{u}, \phi'_0 \wedge \phi'_1, \bar{z}) :: \Gamma$

$$\Lambda(\bar{u}, \phi'_0 \vee \phi'_1, \bar{z}, \Gamma, d, r) \triangleq$$
$$Successful_leaf(\bar{u}, \phi'_0 \vee \phi'_1, \bar{z}, \Gamma) \vee (\Lambda(\bar{u}, \phi'_0, \bar{z}, \Gamma', d, r) \vee \Lambda(\bar{u}, \phi'_1, \bar{z}, \Gamma', d, r))$$

where $\Gamma' = (\bar{u}, \phi'_0 \vee \phi'_1, \bar{z}) :: \Gamma$

Fixed point rule

$$\Lambda(\bar{u}, \eta X.\phi', \bar{z}, \Gamma, d, r) \triangleq$$
$$\begin{cases} Successful_leaf(\bar{u}, \eta X.\phi', \bar{z}, \Gamma) \vee (Not_leaf(\bar{u}, \eta X.\phi', \bar{z}, \Gamma) \wedge c_i) & \text{if } d = 0 \\ Successful_leaf(\bar{u}, \eta X.\phi', \bar{z}, \Gamma) \vee \Lambda(\bar{u}, \phi'[\eta X.\phi'/X], \bar{z}, \Gamma', d-1, r)) & \text{if } d > 0 \end{cases}$$

where c_i : a fresh Boolean variable and $\Gamma' = (\bar{u}, \eta X.\phi', \bar{z}) :: \Gamma$

Modality rules

$$\Lambda(\bar{u}, \langle A \rangle \phi', \bar{z}, \Gamma, d, r) \triangleq$$
$$\begin{cases} Successful_leaf(\bar{u}, \langle A \rangle \phi', \bar{z}, \Gamma) \vee (Not_leaf(\bar{u}, \langle A \rangle \phi', \bar{z}, \Gamma) \wedge c_i) & \text{if } r = 0 \\ Successful_leaf(\bar{u}, \langle A \rangle \phi', \bar{z}, \Gamma) \vee \pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r) \vee \Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r) & \text{if } r > 0 \end{cases}$$

where c_i : a fresh Boolean variable

$$\pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r) \triangleq \chi(\bar{u}, A, \bar{v}) \wedge \Lambda(\bar{v}, \phi', \bar{z}, \Gamma', d, r)$$

$$\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r) \triangleq$$
$$\zeta(\bar{u}, \bar{v}, \bar{w}) \wedge \Lambda(\bar{v}, \langle A \rangle \phi', \bar{z}', \Gamma', d, r-1) \wedge \bigwedge_{k=0}^{|\bar{z}'|-1}(\bar{z}'[k] \Rightarrow \Lambda(\bar{w}, \phi_k, \bar{z}, \Gamma', d, r-1))$$

$\bar{v}, \bar{w}$: vectors of fresh Boolean variables of size $\lg(\sum_{i=0}^{n} |\Sigma_i|)$

where $\bar{z}'$: a vector of fresh Boolean variables of size $|CL(\phi)|$

$\Gamma' = (\bar{u}, \langle A \rangle \phi', \bar{z}) :: \Gamma$

Atomic rules

$$\Lambda(\bar{u}, \mathsf{tt}, \bar{z}, \Gamma, d, r) \triangleq \mathsf{tt} \qquad \Lambda(\bar{u}, \mathsf{ff}, \bar{z}, \Gamma, d, r) \triangleq \mathsf{ff}$$

Fig. 3. Proof Search in Boolean Satisfiability

Lemma 2. *Let $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ be a context-free process system, $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph, and $c_0, \ldots, c_l$ the expansion variables in $\Lambda(\bar{\delta}_0, \phi', \bar{z}, [], d, r)$ with $d, r \in \mathbb{N}$. If $\bar{u} = \bar{\delta}_0 \wedge \Lambda(\bar{u}, \phi', \bar{z}, [], d, r) \wedge \bigwedge_{i=0}^{l} \neg c_i$ is satisfied by the assignment ρ, there is a successful proof for $\delta_0 \vdash \langle \phi', \Theta \rangle$ with $[\![\bar{z}]\!]_\rho = \overline{\Theta}$.*

On the other hand, the formula $\Lambda(\bar{u}, \phi', \bar{z}, [], d, r) \wedge \bigwedge_{i=0}^{l} \neg c_i$ can be satisfied by the assignment ρ if a successful (d, r)-proof for the second-order sequent $\sigma \vdash \langle \phi', \Theta \rangle$ with $\overline{\sigma} = [\![\bar{u}]\!]_\rho$ and $\overline{\Theta} = [\![\bar{z}]\!]_\rho$ exists.

Lemma 3. *Let $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ be a context-free process system, $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph, and $c_0, \ldots, c_l$ the expansion variables in $\bar{u} = \overline{\sigma} \wedge \Lambda(\bar{u}, \phi', \bar{z}, [], d, r)$ where $d, r \in \mathbb{N}$. If there is a successful (d, r)-proof for $\sigma \vdash \langle \phi', \Theta \rangle$, then there is a satisfying assignment ρ for $\bar{u} = \overline{\sigma} \wedge \Lambda(\bar{u}, \phi', \bar{z}, [], d, r) \wedge \bigwedge_{i=0}^{l} \neg c_i$ such that $[\![\bar{u}]\!]_\rho = \overline{\sigma}$ and $[\![\bar{z}]\!]_\rho = \overline{\Theta}$.*

By Lemma 2 and 3, a satisfying assignment ρ for $\Lambda(\bar{u}, \phi', \bar{z}, [], d, r)$ and a successful (d, r)-proof for the second-order sequent $\sigma \vdash \langle \phi', \Theta \rangle$ are related by $[\![\bar{u}]\!]_\rho = \overline{\sigma}$ and $[\![\bar{z}]\!]_\rho = \overline{\Theta}$. If the start rule is furthermore taken into consideration, we have the following theorem.

Theorem 2. *Let $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ be a context-free process system, $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph, and $c_0, \ldots, c_l$ the expansion variables in $\bar{u} = \alpha(\phi, d, r)$ with $d, r \in \mathbb{N}$. Define $\Xi_{-}(\phi, d, r) \triangleq \alpha(\phi, d, r) \wedge \bigwedge_{i=0}^{l} \neg c_i$.*

(i) If $\Xi_{-}(\phi, d, r)$ is satisfiable, then there is a successful proof for $s_0 \vdash \phi$.
(ii) If there is a successful (d, r)-proof for $s_0 \vdash \phi$, then $\Xi_{-}(\phi, d, r)$ is satisfiable.

Given two integers d and r, Theorem 2 shows that a successful (d, r)-proof for a second-order sequent exists exactly when the Boolean formula $\Xi_{-}(\phi, d, r)$ is satisfiable. But we have no information when the Boolean formula is unsatisfiable. Particularly, we do not know if any (d, r)-proof exists for larger d or r. The following lemma states that we need not continue the proof search when a similar formula is unsatisfiable.

Lemma 4. *If there is a successful (d', r')-proof for $\sigma \vdash \langle \phi', \Theta \rangle$ with $d' > d$ or $r' > r$, and $c_0, \ldots, c_l$ are the expansion variables in $\Lambda(\bar{u}, \phi', \bar{z}, [], d, r)$, then there is a satisfying assignment ρ for $\bar{u} = \overline{\sigma} \wedge \Lambda(\bar{u}, \phi', \bar{z}, [], d, r) \wedge \bigwedge_{i=0}^{l} c_i$ with $[\![\bar{z}]\!]_\rho = \overline{\Theta}$.*

By considering the start location and adding the start rule, we have the following criteria for the completion of proof search.

Theorem 3. *Let $\mathbb{P} = \langle P_0, \ldots, P_n \rangle$ be a context-free process system, $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$ its process graph, and $c_0, \ldots, c_l$ the expansion variables in $\alpha(\phi, d, r)$ with $d, r \in \mathbb{N}$. Define $\Xi_{+}(\phi, d, r) \triangleq \alpha(\phi, d, r)$. If there is a successful (d', r')-proof for $s_0 \vdash \phi$ with $d' > d$ or $r' > r$, then there is an assignment ρ satisfying $\Xi_{+}(\phi, d, r)$.*

The unsatisfiability of the formula $\Xi_{+}(\phi, d, r)$ corresponds to the absence of proof in local model checking. Again, one must show that $\Xi_{+}(\phi, d, r)$ is eventually unsatisfiable if the property fails at the initial state. Our technical results are now summarized in the following two theorems.

Theorem 4. *(Soundness and completeness)*

1. *If $\Xi_-(\phi, d, r)$ is satisfiable, then there is a successful (d, r)-proof for $s_0 \vdash \phi$.*
2. *If there is a successful (d, r)-proof for $s_0 \vdash \phi$, then $\Xi_-(\phi, d, r)$ is satisfiable.*

Theorem 5. *(Completion and termination)*

1. *If $\Xi_+(\phi, d, r)$ is unsatisfiable, then there is no successful (d', r')-proof for $s_0 \vdash \phi$ with $d' > d$ or $r' > r$.*
2. *If there is no successful proof for $s_0 \vdash \phi$, then there are some d and r such that $\Xi_+(\phi, d, r)$ is unsatisfiable.*[2]

4 Algorithm

Our SAT-based model checking algorithm for context-free processes is shown in Figure 4. The algorithm first computes the negative normal form of the negation of the given property. Proofs of refutation and completion are checked incrementally. If a proof of the negated property is found, the algorithm reports an error. If, on the other hand, the completion criteria is satisfied, it reports success.

Given a context-free process system $\mathbb{P}$ and $PG(\mathbb{P}) = \langle S, Act, \rightarrow, s_0, s_e \rangle$
Let ψ be a universal alternation-free μ-calculus formula
Let ϕ be the negative normal form of $\neg\psi$
$d \leftarrow 0$
loop
  if $\Xi_-(\phi, d, d)$ is satisfiable **then**
    return "$s_0 \nvdash \psi$"
  if $\Xi_+(\phi, d, d)$ is unsatisfiable **then**
    return "$s_0 \vdash \psi$"
  $d \leftarrow d + 1$
end

Fig. 4. Model Checking Algorithm

By Theorem 4, a proof of the negated property can always be found should it exist. Otherwise, the algorithm checks whether the search should continue by Theorem 5 (1). If a proof is possible, the algorithm increments the bounds and repeats. It will terminate if there is no proof (Theorem 5 (2)). Applying Theorem 1, we have the following theorem.

Theorem 6. *The algorithm in Figure 4 is correct. Namely, it has the following properties.*

- *It always terminates.*
- *It reports "$s_0 \vdash \phi$" if and only if $s_0 \models \phi$.*

[2] Intuitively, $Successful_leaf(\bar{u}, \phi', \bar{z}, \Gamma)$ is unsatisfiable when there is no successful proof. But $Not_leaf(\bar{u}, \phi', \bar{z}, \Gamma)$ will become unsatisfiable eventually.

5 Experiments

5.1 Implementation

We have implemented our SAT-based model checking algorithm for context-free processes in Objective Caml [14]. Given a context-free process and a formula in the universal fragment of alternation-free μ-calculus, our implementation creates instances of the Boolean satisfiability and solves them using MiniSat [10].

Most of our implementation is straightforward, but care must be taken for modality rules in Figure 3. The formula $\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r)$ presumes process invocation for any location represented by $\bar{u}$. Hence it may construct $\Lambda(\bar{w}, \phi_k, \bar{z}, \Gamma', d, r-1)$ for $1 \leq k \leq |CL(\phi)|$ unnecessarily. Precisely, the formulae $\Lambda(\bar{w}, \phi_k, \bar{z}, \Gamma', d, r-1)$ are not needed when $\zeta(\bar{u}, \bar{v}, \bar{w})$ is unsatisfiable.

To avoid creating unneeded formulae in $\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r)$, we compute the set of locations reachable by r transitions from the start location of the main process P_0. If there is no location in the set that may invoke other processes, $\zeta(\bar{u}, \bar{v}, \bar{w})$ is unsatisfiable and $\Lambda(\bar{w}, \phi_k, \bar{z}, \Gamma', d, r-1)$ can be omitted. More formally, let δ be the start state of P_0. Define $\Delta_0 \triangleq \{\delta\}$ and $\Delta_{i+1} \triangleq \{\sigma' : \sigma \xrightarrow{\alpha} \sigma'$ for some $\sigma \in \Delta_i\}$. Modify $\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r)$ as follows.

$$\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r) \triangleq \begin{cases} \mathsf{ff} & \text{if } \forall \sigma \in \Delta_r, i \in \underline{n}. \sigma \stackrel{i}{\not\to} \\ \zeta(\bar{u}, \bar{v}, \bar{w}) \wedge \Lambda(\bar{v}, \langle A \rangle \phi', \bar{z}', \Gamma', d, r-1) \wedge & \text{otherwise} \\ \quad \bigwedge_{k=0}^{|\bar{z}'|-1} (\bar{z}'[k] \Rightarrow \Lambda(\bar{w}, \phi_k, \bar{z}, \Gamma', d, r-1)) & \end{cases}$$

Intuitively, Δ_i contains locations reachable from the start location δ_0 with i transitions. If none of the locations in Δ_i performs process invocation, it is unnecessary to expand $\Pi(\bar{u}, A, \phi', \bar{z}, \Gamma, d, r)$ further. The simple modification avoids creating unneeded formulae and can improve the performance significantly.

5.2 Experimental Results

We compare our SAT-based algorithm with a BDD-based algorithm on randomly generated context-free process systems in [12]. A location may be sequential, branching, and looping with probabilities 0.6, 0.2, and 0.2 respectively. Moreover, context-free processes may be called with probability 0.2 on the sequential location.

Two properties are checked against the randomly generated system. The liveness property checks if a random location σ of the main process is reachable on all paths:

$$\mu X.([A]\mathsf{tt} \vee [Act]X), \text{ where } A \text{ is the specific set of actions for } \sigma.$$

The safety property checks if a random process P_i is never called on all paths:

$$\nu X.([A]\mathsf{ff} \wedge [Act]X), \text{ where } A \text{ is the specific set of actions for } \delta_i.$$

Table 1. Performance Comparison

#process/ avg. #location	*liveness*			*safety*		
	ans.	BDD (sec.)	SAT (sec.)	ans.	BDD (sec.)	SAT (sec.)
3/1k	*No*	0.02	0.03	*Yes*	0.02	0.12
4/2k	*No*	0.10	0.14	*No*	0.12	0.24
5/4k	*No*	0.09	0.16	*Yes*	0.18	1.70
6/8k	*No*	0.38	0.54	*Yes*	0.54	7.78
7/16k	*No*	1.51	1.44	*No*	2.57	3.54
8/32k	*No*	19.07	7.07	*No*	35.22	7.61
9/64k	*No*	23.44	9.21	*No*	46.89	8.31
10/128k	*No*	O/M	49.17	*No*	O/M	55.11

(measured in a 1.6 GHz Intel machine with 512Mb memory)

Table 2. Profiling data

#process/ avg. #location	*liveness*			*safety*		
	read	create	solve	read	create	solve
3/1k	0.02	0.01	<0.01	0.05	0.07	<0.01
4/2k	0.14	0.00	<0.01	0.16	0.08	0.01
5/4k	0.15	0.01	<0.01	0.18	1.27	0.24
6/8k	0.52	0.01	<0.01	0.58	4.56	2.63
7/16k	1.42	0.01	<0.01	1.53	1.68	0.32
8/32k	7.03	0.03	<0.01	6.94	0.59	0.06
9/64k	9.17	0.03	<0.01	8.21	0.08	0.01
10/128k	48.69	0.39	0.08	50.78	4.04	0.35

(execution time in seconds)

In table 1, the performance data of our SAT-based algorithm and the BDD-based algorithm are shown. Our SAT-based algorithm outperforms the BDD-based algorithm for larger context-free process systems (8/32k, 9/64k, 10/128k) on both properties. For the largest case (10/128k), our SAT-based algorithm is capable of finding errors while the BDD-based algorithm running out of memory.

Our experiments show that the execution time of the BDD-based algorithm increases consistently with the sizes of the context-free process systems. On the other hand, SAT-based algorithms are known to be very efficient in bug detection [4,3]. Indeed, our SAT-based algorithm takes more time in proving the safety property for the 6/8k system than falsifying it for the 7/16k system.

Table 2 gives time distribution of our implementation. We measure the time

- for reading the input file and model building;
- for creating instances of Boolean satisfiability; and
- for solving the instances.

When the property does not hold, our implementation spends most of the time reading the input and building models. For the verification of the safety property

on 5/4k, 6/8k, and 7/16k, the majority of time is used for creating instances. This suggests further improvement could still be possible for our implementation.

6 Conclusion

Because of its capacity and scalability, SAT solvers have found many applications in hardware verification. On the other hand, alternative computational models have shown their promises in software verification. In this paper, a complete SAT-based model checking algorithm is developed to take advantages of SAT solvers and context-free processes. By combining the scalability of SAT solvers and the succinctness of context-free processes, the proposed algorithm could potentially analyze more programs. To the best of our knowledge, this is the first SAT-based model checking algorithm for context-free processes.

In comparison with other SAT-based techniques, our proof-theoretic approach is very general. Instead of exploiting characteristics in specification logics or computational models, we reduce the proof search in local model checking algorithms to the satisfiability problem. The same idea is used to develop an SAT-based model checking algorithm for finite-state models in [18]. The present work demonstrates the applicability of our approach even for context-free processes.

In the experiments, we show that our SAT-based algorithm outperforms in some cases. We would like to conduct more experiments to support our preliminary findings in the future.

Although we believe context-free processes should be sufficient for program analysis, a scientific study will be very welcome. Finally, it would be interesting to see if our technique can be applied to other infinite-state models.

Acknowledgment. The authors would like to thank anonymous reviewers for their comments and suggestions in revising the paper.

References

1. Awedh, M., Somenzi, F.: Proving more properties with bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 96–108. Springer, Heidelberg (2004)
2. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000 Workshop on Model Checking of Software. LNCS, vol. 1885, Springer, Heidelberg (2000)
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC, pp. 317–320. ACM Press, New York (1999)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.)TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Boujjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Applications to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

6. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
7. Burkart, O., Esparza, J.: More infinite results. In: Paun, G., Rozenberg, G., Salomaa, A. (eds.) Current Trends in Theoretical Computer Science, Entering the 21th Century, pp. 480–503. World Scientific (2001)
8. Caucal, D., Monfort, R.: On the transition graphs of automata and grammars. In: Möhring, R.H. (ed.) WG 1990. LNCS, vol. 484, pp. 311–337. Springer, Heidelberg (1991)
9. Christensen, S., Hüttel, H.: Decidability issues for infinite-state processes - a survey. Bulletin of the European Association for Theoretical Computer Science 51, 156–166 (1993)
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
12. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
13. Hungar, H., Steffen, B.: Local model checking for context-free processes. Nordic Journal of Computing 1(3), 364–385 (1994)
14. Leroy, X.: The Objective Caml system: Documentation and user's manual (With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon) (2000)
15. McMillan, K.L.: Interpolation and sat-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
16. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and sceond-order logic. Theoretical Computer Science 37, 51–75 (1985)
17. Walukiewicz, I.: Pushdown processes: Games and model-checking. Information and Computation 164(2), 234–263 (2001)
18. Wang, B.Y.: Proving $\forall\mu$-calculus properties with SAT-based model checking. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 113–127. Springer, Heidelberg (2005)

Bounded Model Checking of Analog and Mixed-Signal Circuits Using an SMT Solver*

David Walter, Scott Little, and Chris Myers

University of Utah, Salt Lake City, UT 84112, USA
{dwalter,little,myers}@vlsigroup.ece.utah.edu

Abstract. This paper presents a bounded model checking algorithm for the verification of *analog and mixed-signal* (AMS) circuits using a *satisfiability modulo theories* (SMT) solver. The systems are modeled in *VHDL-AMS*, a hardware description language for AMS circuits. In this model, system safety properties are specified as assertion statements. The VHDL-AMS description is compiled into *labeled hybrid Petri nets* (LHPNs) in which analog values are modeled as continuous variables that can change at rates in a bounded range and digital values are modeled using Boolean signals. The verification method begins by transforming the LHPN model into an SMT formula composed of the initial state, the transition relation unrolled for a specified number of iterations, and the complement of the assertion in each set of state variables. When this formula evaluates to true, this indicates a violation of the assertion and an error trace is reported. This method has been implemented and preliminary results are promising.

1 Introduction

To date, there has been relatively little research in the formal verification of *analog and mixed-signal* (AMS) circuits. Perhaps the first work in this area is from Kurshan and McMillan in which analog circuits are represented as finite state models [1]. Hartong et al. verify analog circuits by dividing the continuous state space into regions that are represented in a Boolean manner [2]. This allows them to use Boolean-based verification but with significant loss in accuracy. Hybrid system tools have also been adapted to verify AMS circuits. Gupta et al. utilize `CheckMate` to verify a tunnel diode oscillator and a delta-sigma modulator [3]. In [4], Dang et al. use d/dt to verify a biquad low-pass filter. In [5], Frehse et al. use `PHAVer` to verify analog oscillator circuits. These tools are very accurate but also very computationally complex. These approaches also require a user to describe an AMS circuit using a *hybrid automaton* which is unfamiliar to most AMS designers. In [6], Little et al. use *difference bound matrices* (DBMs) to verify AMS circuits. This method, however, only supports constant rates of change for continuous variables and conservatively abstracts the continuous state space. In [7], Walter et al. present a BDD model checking algorithm for verifying AMS circuits. This method, however, can have substantial memory requirements.

* Support from SRC contract 2005-TJ-1357 and an SRC Graduate Fellowship.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 66–81, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The goal of this paper is to develop a method for verifying AMS circuits using a *satisfiability modulo theories* (SMT) solver. The SMT problem is a generalization of the *Boolean Satisfiability* (SAT) problem where Boolean variables are replaced by predicates from various background theories [8]. These theories may include linear arithmetic over reals and integers, uninterpreted functions, and the theories of various data structures such as lists, arrays, and bit vectors [9,10,11,12,13,14,15]. Initial SMT solver implementations functioned by translating SMT instances into SAT instances and passing those SAT instances to a SAT solver. For example, to support integer arithmetic, multiple Boolean variables are used as a bit representation for integers and the necessary integer theories are specified as Boolean operations on those individual bit variables. This can result in extremely large SAT instances; however, existing SAT solvers can be used directly without modification. Therefore, as SAT solvers improve, so do the SMT solvers. This approach, however, can be severely restricting. The loss of higher level knowledge of the underlying theories requires the SAT solver to work harder to discover simple concepts [16]. This problem is made even more difficult by the large SAT instances that result.

More recent SMT solvers [17,15,18] closely integrate theory-specific solvers with a DPLL (Davis-Putnam-Logemann-Loveland) approach to Boolean satisfiability [8]. These types of SMT solvers are often referred to as DPLL(T) [15]. In this type of architecture, the DPLL-based SAT solver passes conjunctions of predicates belonging to theory T to a specialized solver. The specialized solver is then responsible for deciding feasibility of those predicates. Additionally, the particular theory solver must be able to explain the reasons for infeasibility. Recent work applies SMT solvers to the bounded model checking of software [16]. There are a number of SMT solvers including `Barcelogic` [15,18], `MathSAT` [17], and `Yices` [19]. The `Barcelogic` solver supports difference logic over integers and equality with uninterpreted functions. The `MathSAT` solver currently supports theories of equality, uninterpreted functions, separation logic, and linear arithmetic over reals and integers. `Yices` includes an incremental Simplex algorithm for the theory of linear arithmetic that is tightly integrated within the DPLL framework. `Yices` strong ability to work with the theory of linear arithmetic makes it particularly well suited for hybrid system model checking. For this reason, `Yices` is selected as the SMT solver for the bounded model checker described in this paper.

This paper describes a bounded model checking algorithm for the verification of AMS circuits. The model checker begins with a VHDL-AMS description of an AMS circuit that is compiled into a *labeled hybrid Petri net* (LHPN). Next, the LHPN model is converted into an SMT formula which includes a set of Boolean and continuous state variables for each of the specified number of iterations. The formula is composed of the initial state, the transition relation, and the negation of the assertion statement. The SMT solver `Yices` is used to evaluate this formula. When a satisfying assignment is found, this indicates a failure, and an error trace is reported. This method has been implemented and preliminary results are promising.

2 Motivating Example

The switched capacitor integrator shown in Figure 1 is used as a running example throughout this paper. This circuit takes as input a 5 kHz square wave that varies from -1000 mV to 1000 mV and generates a triangle wave as output representing the integral of the input voltage. Discrete-time integrators typically utilize switched capacitor circuits to accumulate charge which can cause gain errors in the integrator due to capacitor mismatch. Therefore, the output voltage in our model is allowed to have a slew rate anywhere between 18 to 22 mV/μs to represent a ± 10 percent variance in circuit parameters. The verification goal is to ensure that $Vout$ never saturates (i.e., it is always between -2000 mV and 2000 mV). An experienced analog circuit designer may realize the potential of this circuit to fail. However, a very specific SPICE simulation is required to demonstrate this failure where the output voltage always increases at a faster rate than it decreases. Furthermore, it is highly unlikely that a simulation allowing for random uncertainty in the system variables would reveal the error [20]. Therefore, a formal verification approach is beneficial.

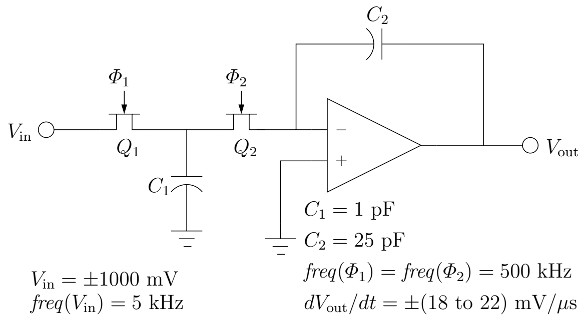


Fig. 1. Circuit diagram of a switched capacitor integrator

VHDL-AMS is a hardware description language that includes extensions specifically for describing analog and mixed-signal circuits. VHDL-AMS was designed to allow a textual description of AMS circuits which can be simulated. By providing a VHDL-AMS front-end to our tool, many of the hurdles associated with verification can potentially be avoided because designers who are already familiar with VHDL-AMS are not required to learn abstract modeling methods. Our VHDL-AMS compiler is built using methods described in [21] and currently works with a subset of the VHDL-AMS language. Methods for generating LHPNs from many VHDL statements for representing digital systems are described in [21]. Specifically, variables of types **std_logic** for representing Boolean signals are allowed and sequential behavior can be specified using **process** statements without sensitivity lists. Within a **process**, supported statements are **wait**, signal assignment, **if-use**, **case**, and **while-loop**.

Simulators that support the AMS extensions to VHDL seem to vary in the semantics that are implemented. Therefore, a subset of the AMS extensions have been selected such that the semantics seem to be fairly consistent across simulators. The supported subset of VHDL-AMS allows the creation of a continuous value using a **quantity** of type **real**, the initialization of continuous variables using **break** statements, and the assignments of rates to real quantities using the **'dot** notation within simultaneous **if-use** and **case-use** statements. Additionally, the use of **'above** to test the value of real quantities, and the specification of properties using **assert** statements is allowed. For convenience, VHDL-AMS descriptions also use procedures defined in the `handshake` and `nondeterminism` packages [22]. The **assign** procedure performs an assignment to a signal at some random time within a bounded range specified by its parameters and waits until the assignment has been performed before returning. The **span** procedure takes two real values and returns a random value within that range. The **span** procedure is used to assign a range of rates to a continuous variable. Figure 2 shows a VHDL-AMS description for the circuit in Figure 1. The **break** statement sets the initial value for *Vout*. The **if-use** statement determines the rate of *Vout*. When *Vin* is **false**, *Vout* increases at a rate between 18 and 22 mV/μs. When *Vin* is **true**, it decreases at a rate between -22 and -18 mV/μs. The **process** statement controls *Vin*. Finally, an **assert** statement checks if *Vout* saturates.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.handshake.all;
use work.nondeterminism.all;
entity integrator is
end integrator;
architecture switchCap of integrator is
   quantity Vout:real;
   signal Vin:std_logic := '0';
begin
   break Vout => -1000.0; --Initial value
   if Vin='0' use
      Vout'dot == span(18.0, 22.0);
   elsif Vin = '1' use
      Vout'dot == span(-22.0, -18.0);
   end use;
   process begin
      assign(Vin,'1',100,100);
      assign(Vin,'0',100,100);
   end process;
   assert (Vout'above(-2000.0) and not Vout'above(2000.0))
      report ``error'' severity failure;
end switchCap;
```

Fig. 2. VHDL-AMS for a switched capacitor integrator

3 Labeled Hybrid Petri Nets

Our VHDL-AMS descriptions are compiled automatically into LHPN models. LHPNs were developed specifically to model AMS circuits. The model is inspired by features in both hybrid Petri nets [23] and hybrid automata [24]. While LHPNs are only described briefly here, a complete definition with formal semantics can be found in [25]. An LHPN is defined as a directed graph with labels on places and transitions. An LHPN is a tuple $N = \langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$:

- P : is a finite set of places;
- T : is a finite set of transitions;
- B : is a finite set of Boolean signals;
- V : is a finite set of continuous variables;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- L : is a tuple of labels defined below;
- $M_0 \subseteq P$ is the set of initially marked places;
- S_0 : is the set of initial Boolean signal values;
- Q_0 : is the set of initial ranges of values for each continuous variable and;
- R_0 : is the set of initial ranges of rates for each continuous variable.

The *preset* of a transition t (denoted $\bullet t$) represents the set of places feeding t (i.e., $\bullet t = \{p \mid (p, t) \in F\}$). The *postset* of a transition t (denoted $t\bullet$) represents the set of places that t feeds (i.e., $t\bullet = \{p \mid (t, p) \in F\}$).

A key component of LHPNs are the labels. Some labels contain *hybrid separation logic* (HSL) formulas which are a Boolean combination of Boolean variables and separation predicates. HSL is an extension of *separation logic* [26] (sometimes referred to as *difference logic*) that allows for non-unit slopes on the separation predicates. These formulas satisfy the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid c_1 x_1 \geq c_2 x_2 + c_3$$

where b_i are Boolean variables, x_1 and x_2 are continuous variables, and c_1, c_2, and c_3 are rational constants in $\mathbb{Q}$. Note that any inequality between two real variables can be formed with $\geq$ and negations of $\geq$ inequalities. Each transition $t \in T$ is labeled using the functions defined in $L = \langle En, D, BA, VA, RA \rangle$:

- $En : T \to \phi$ labels each transition $t \in T$ with an enabling condition;
- $D : T \to |\mathbb{Q}| \times (|\mathbb{Q}| \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper bound $[d_{\mathrm{l}}, d_{\mathrm{u}}]$ on the delay for t to fire;
- $BA : T \to 2^{(B \times \{0,1\})}$ labels each transition $t \in T$ with Boolean assignments made when t fires;
- $VA : T \to 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a continuous variable assignment range, consisting of a lower and upper bound $[a_l, a_u]$, that is made when t fires;
- $RA : T \to 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a range of rates, consisting of a lower and upper bound $[r_l, r_u]$, that are assigned when t fires.

The LHPN shown in Figure 3 is automatically generated from the VHDL-AMS model in Figure 2. This model tracks the real quantity *Vout* that represents the output voltage. The **if-use** statement is compiled into the LHPN in Figure 3a. The **process** statement is compiled into the LHPN in Figure 3b. Initially *Vout* is -1000 mV and increasing between 18 and 22 mV/μs. After 100 μs, *Vin* is assigned to **true** by the **assign** function which causes *Vout* to begin decreasing at a rate of -22 to -18 mV/μs. The **assert** statement is used to check if *Vout* falls below -2000 mV or goes above 2000 mV and is compiled into the LHPN shown in Figure 3c which fires a transition to set the Boolean signal *fail* to true when the assertion is violated.

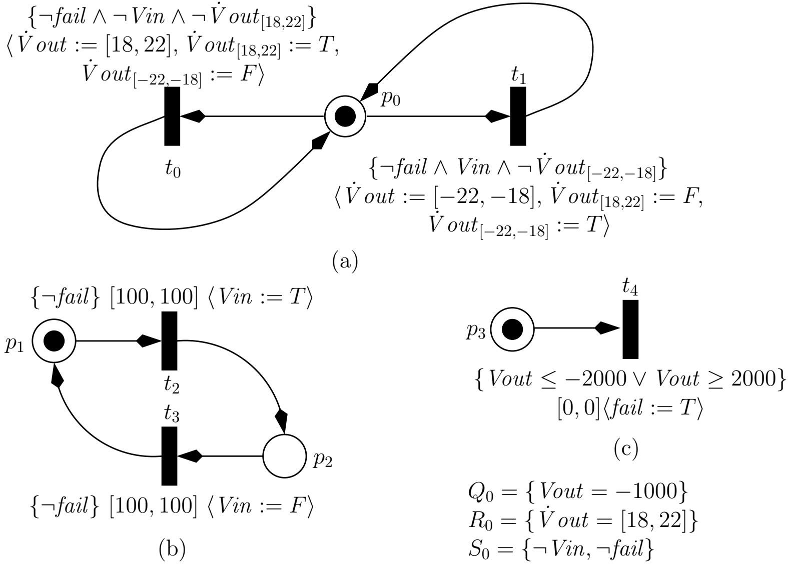


Fig. 3. LHPN of the switched capacitor integrator generated from VHDL-AMS

4 Symbolic Model of LHPNs

In order for analysis to proceed, a symbolic model is generated from the LHPN that contains the essential information for analysis. The symbolic model consists of three components: an *invariant*, a set of *possible rates*, and a set of *guarded commands*. Before constructing the symbolic model, a set of real variables and two additional sets of Boolean variables are created in addition to the sets defined for an LHPN. The set of real variables, C, are used to track the values of the clocks on each transition. The *transition clock* for transition t is denoted by c_t. The first set of Boolean variables are known as *clock active variables*, A, and are used to keep track of whether or not the clocks on transitions are active. The clock active variable for transition t is denoted by a_t. The second set of Boolean variables are known as *Boolean rate variables*, BR, used for determining the

current rate of change for each continuous variable. Boolean rate variables are denoted by $\dot{v}_{[r_l,r_u]}$ for the variable corresponding to the continuous variable v currently advancing at a range of rates $[r_l, r_u]$.

The invariant ($\phi_{\mathcal{I}}$) is an HSL statement that must be satisfied in every state of the system and is calculated as shown in Equation 1.

$$\phi_{\mathcal{I}} = \Phi \wedge \bigwedge_{t \in T} (a_t \Rightarrow \bullet t \wedge En(t) \wedge 0 \leq c_t \leq d_u(t)) \wedge (\overline{a_t} \Rightarrow \overline{\bullet t} \vee \widetilde{En(t)}) \quad (1)$$

The invariant first states that only the reachable discrete states (represented by Φ) are allowed. The formula Φ is found by performing a state space exploration of the LHPN while neglecting the continuous variables. The discrete state space exploration is based on the Petri net algorithm described in [27] with extensions to include values of Boolean signals and Boolean rate variables in the state space [25]. In other words, Φ is a formula over the Boolean variables for the Petri net marking, Boolean signals, and Boolean rate variables.

After calculating the discrete state space, Φ, the next step in constructing the system invariant, $\phi_{\mathcal{I}}$, is to insert known information about the continuous state space. This is performed using the clock active variables. Specifically, for a transition's clock to be active, the preset must be marked, the enabling condition must be satisfied, and the clock must be greater than zero but not greater than its upper bound. This portion of $\phi_{\mathcal{I}}$ prevents an active clock from exceeding its upper bound. The last part of $\phi_{\mathcal{I}}$ states that if a transition's clock is not active it must either have an unmarked place in its preset (denoted $\overline{\bullet t}$) or the *non-strict inverse* ($\widetilde{En(t)}$) of the enabling condition must be satisfied. In the non-strict inverse, all $\geq$ separation predicates become $\leq$ separation predicates and vice-versa. For example, the non-strict inverse of the HSL formula $a \wedge x \leq 2000$ is $\overline{a} \vee x \geq 2000$. The non-strict inverse is used to allow for the existence of a time of overlap when a clock is both allowed to be active and inactive at which time the clock's state can change. The last two portions of $\phi_{\mathcal{I}}$ when taken together enforce the activation or deactivation of a clock if a changing continuous variable should cause an enabling condition to change evaluation.

The set of possible rates ($\mathcal{R}$) consist of an HSL statement indicating a possible Boolean rate assignment and the set of rate assignments to continuous variables corresponding to the statement ($\langle \phi_R, R \rangle$). This set is constructed from Φ, the Boolean state set, by existentially abstracting all non-rate Boolean variables. Each product term in Φ corresponds to a ϕ_R of a pair in $\mathcal{R}$.

The set of guarded commands ($\mathcal{C}$) is used to determine in each state which transitions are enabled and the effect on the state due to the firing of a transition. It is constructed using a set of *primary guarded commands* ($\mathcal{C}_P$) and a set of *secondary guarded commands* ($\mathcal{C}_S$). Each guarded command consists of a *guard*, ϕ_G, represented using an HSL formula and a set of *commands*, $\mathcal{A}$, to be performed when the guard is satisfied.

A primary guarded command is created for each transition $t \in T$. The guard for transition t ensures that the preset for t is marked, the enabling condition on t is satisfied, and the clock associated with t is active and exceeds its lower bound.

The commands for transition t cause the postset of t to become marked and apply the assignments associated with t. Formally, the set of primary guarded commands is defined as follows:

$$\mathcal{C}_P = \{\langle \phi_G \;\; (t), \mathcal{A}_P(t)\rangle \mid t \in T\} \tag{2}$$

where $\phi_G \;\; (t) = (\bullet t \wedge \overline{t \bullet - \bullet t} \wedge En(t) \wedge a_t \wedge c_t \geq d_l(t))$ and $\mathcal{A}_P(t) = \{(\bullet t - t\bullet) := F, (t\bullet) := T, a_t := F, c_t := [-\infty, \infty], BA(t), VA(t), RA(t)\}$. The primary guarded command for transition t_2 in Figure 3 is:

$$\begin{aligned} \phi_G \;\; (t_2) &= p_1 \wedge \overline{p_2} \wedge \overline{fail} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\ \mathcal{A}_P(t_2) &= \{p_1 := F, p_2 := T, Vin := T, \\ &\qquad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\} \end{aligned}$$

Two secondary guarded commands are created for each transition $t \in T$. The first one activates the clock for t and sets it to zero when its preset is marked and its enabling condition is true. The second one deactivates the clock when t is no longer enabled and sets its values to $[-\infty, \infty]$. This removes the clock from the state space. The set of secondary guarded commands is defined as follows:

$$\mathcal{C}_S = \{\langle \phi_G \;\;\; (t), \mathcal{A}_{SA}(t)\rangle, \langle \phi_G \;\;\; (t), \mathcal{A}_{SD}(t)\rangle \mid t \in T\} \tag{3}$$

where $\phi_G \;\;\; (t) = \bullet t \wedge En(t) \wedge \overline{a_t}$, $\mathcal{A}_{SA}(t) = \{a_t := T, c_t := [0, 0]\}$, $\phi_G \;\;\; (t) = (\overline{\bullet t} \vee \overline{En(t)}) \wedge a_t$, and $\mathcal{A}_{SD}(t) = \{a_t := F, c_t := [-\infty, \infty]\}$. The activating and deactivating guarded commands for transition t_1 in Figure 3 are:

$$\begin{aligned} \phi_G \;\;\; (t_1) &= p_0 \wedge \overline{fail} \wedge Vin \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a_{t_1}} \\ \mathcal{A}_{SA}(t_1) &= \{a_{t_1} := T, c_{t_1} := [0, 0]\} \\ \phi_G \;\;\; (t_1) &= (\overline{p_0} \vee fail \vee \overline{Vin} \vee \dot{V}out_{[-22,-18]}) \wedge a_{t_1} \\ \mathcal{A}_{SD}(t_1) &= \{a_{t_1} := F, c_{t_1} := [-\infty, \infty]\} \end{aligned}$$

The sets $\mathcal{C}_P$ and $\mathcal{C}_S$ are merged to form the set $\mathcal{C}$. It is necessary to merge these commands because the firing of a transition may result in the activation or deactivation of clocks associated with other transitions by changing the marking or the values of the Boolean or continuous variables. Due to space limitations, only a brief description of the merging process is given. A complete algorithm is described in [25]. The basic idea is that for each transition, t, the effect of its assignments associated with its primary guarded command $\mathcal{A}_P(t)$ must be checked against the guards $\phi_G \;\;\; (t')$ and $\phi_G \;\;\; (t')$ for each other transition t' to determine if the assignment may have enabled the guard [25]. If the assignments have no effect on the guard or disable it, then the secondary for t' is not merged with the primary for t. If the assignment would make the guard true, then the commands associated with the secondary must be combined with those for the primary. Finally, if the assignment may have changed the guard's evaluation, then two guarded commands must be constructed. One is for the case in which the guard for the secondary is true in which the commands are merged, and

the other is for when the guard is false in which the secondary commands are not merged. Note that after performing the merge operation, secondary guarded commands whose guards contain inequalities are inserted into the final guarded command set. This is necessary because as time moves forward, the secondary guarded commands could become enabled and cause clocks to be activated or deactivated. However, before the secondary guarded commands are added, their guards must be modified to enforce the threshold on the continuous variables. For example, consider a situation where a transition has the enabling condition $x \geq 5$. The clock on this transition can be activated either when its preset becomes marked when x is already greater than or equal to five, or by x becoming equal to five while the preset is already marked. The first case is handled by the merged guarded command while the second case should be handled by a secondary guarded command that ensures that x is equal to five and continues to increase above five, i.e., when $x \geq 5 \wedge x \leq 5 \wedge \mathtt{incr}(x)$ where $\mathtt{incr}(x)$ returns the disjunction of the Boolean rate variables where the rates are increasing. Similarly, $\mathtt{decr}(x)$ returns the disjunction of the Boolean rate variables where the rates for x are decreasing. In the integrator example, since t_2 assigns Vin to true and marks p_2, it activates the clocks for t_1 and t_3. This results in the following merged guarded command:

$$\begin{aligned}
\phi_G &= p_0 \wedge p_1 \wedge \overline{p_2} \wedge \overline{fail} \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a_{t_1}} \wedge a_{t_2} \wedge \overline{a_{t_3}} \wedge c_{t_2} \geq 100 \\
\mathcal{A} &= \{p_1 := F, p_2 := T, Vin := T, \\
&\quad a_{t_1} := T, c_{t_1} := [0,0], a_{t_3} := T, c_{t_3} := [0,0], \\
&\quad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\}
\end{aligned}$$

5 SMT Based Bounded Model Checking

The basic algorithm for performing SMT based bounded model checking of LHPNs is shown in Figure 4. The algorithm proceeds by creating an SMT instance in which statements are asserted. The first step is to create a set of state variables for each iteration of the exploration. The state variables for each iteration, i, are defined using the tuple $\langle M^i, S^i, Q^i, C^i, A^i, BR^i \rangle$. The next step is to assert the initial state (ϕ_{init}) in terms of the initial iteration's variables (i.e., $i = 0$). At this point, the SMT formula is constructed one iteration at a time. For each iteration, it is necessary to assert the invariant in terms of that iteration's set of state variables. Then, each iteration's next states are calculated by firing transitions or elapsing time. This is performed by asserting a disjunction of the guarded commands and a time elapse formula. Finally, a failure condition is asserted in terms of state variables from each iteration. After asserting each of these components, the SMT satisfiability check is performed. Satisfiability indicates that the property is violated because there is an assignment indicating that the failure condition is reachable. Unsatisfiability indicates that the property could not be violated in that number of iterations. This does not necessarily indicate that the property cannot be violated, however, so this is a bounded model checker.

```
smtCheck(φ_init, φ_I, C, R, maxIterations)
   SMTInstance ins(maxIterations);
   i = 0
   ins.assert(φ^0_init)
   while (i < maxIterations)
      ins.assert(φ^i_I)
      trans = true
      for each ⟨φ_G, A⟩ ∈ C
         trans = trans ∨ mkExprForGC(φ_G, A, i, i + 1)
      trans = trans ∨ mkExprForTimeElapse(R, i, i + 1)
      ins.assert(trans)
      i + +
   ins.assert(mkExprForFailProp(maxIterations))
   if (ins.check == true) then return ''Property Violated''
   else return ''Property Not Violated''
```

Fig. 4. Algorithm for SMT based bounded model checking

The remainder of this section describes the SMT based bounded model checking algorithm in greater detail.

An assertion for the next state calculation is made based on the disjunction of each guarded command and the time elapse assertion. The transition relation portion of the next state assertion makes use of the merged guarded command set ($\mathcal{C}$) to calculate the values of the next state variables based on the values of the current iteration, i, variables. The algorithm for constructing the assertion statement for a given guarded command is shown in Figure 5. Essentially, the guard portion (ϕ_G) of the guarded command is asserted in terms of the current state while the assignment portion of the guarded command makes use of both the current and the next iteration variables. Assignments that are specified in the assignment set ($\mathcal{A}$) are performed on the next iteration variables while variables that have no assignment performed on them are simply assigned the same value as in the current iteration. There is one exception, however. If a clock is assigned the range $[-\infty, \infty]$, no assignment is made to that clock variable. This allows the clock to remain undefined in the next iteration. In the integrator example, the SMT assertion statement for the merged guarded command that fires t_2 and activates the clocks for t_1 and t_3, given the current iteration i and the next iteration j, is:

$$
\begin{aligned}
&p_0^i \wedge p_1^i \wedge \overline{p_2^i} \wedge \overline{fail^i} \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a_{t_1}^i} \wedge a_{t_2}^i \wedge \overline{a_{t_3}^i} \wedge c_{t_2}^i \geq 100 \wedge \\
&p_0^j = p_0^i \wedge p_1^j = \mathbf{false} \wedge p_2^j = \mathbf{true} \wedge p_3^j = p_3^i \wedge Vin^j = \mathbf{true} \wedge fail^j = fail^i \wedge \\
&a_{t_0}^j = a_{t_0}^i \wedge a_{t_1}^j = \mathbf{true} \wedge a_{t_2}^j = \mathbf{false} \wedge a_{t_3}^j = \mathbf{true} \wedge a_{t_4}^j = a_{t_4}^i \wedge \\
&\dot{V}out_{[18,22]}^j = \dot{V}out_{[18,22]}^i \wedge \dot{V}out_{[-22,-18]}^j = \dot{V}out_{[-22,-18]}^i \wedge \\
&c_{t_0}^j = c_{t_0}^i \wedge c_{t_1}^j = 0 \wedge c_{t_3}^j = 0 \wedge c_{t_4}^j = c_{t_4}^i \wedge Vout^j = Vout^i \wedge \delta^{i,j} = 0
\end{aligned}
$$

```
mkExprForGC(φ_G, 𝒜, i, j)
    result = φ_G^i          // Guard in terms of current iteration variables.
    foreach b ∈ {M ∪ S ∪ A ∪ BR}      // Perform Boolean assignments.
        if ((b := true) ∈ 𝒜) then result = result ∧ (b^j = true)
        else if ((b := false) ∈ 𝒜) then result = result ∧ (b^j = false)
        else result = result ∧ (b^j = b^i)
    foreach v ∈ {C ∪ Q}              // Perform real assignments.
        if ((v := [−∞, ∞]) ∈ 𝒜) then   // Do Nothing.
        else if ((v := [a_l, a_u]) ∈ 𝒜) then
            result = result ∧ (v^j ≥ a_l) ∧ (v^j ≤ a_u)
        else
            result = result ∧ (v^j = v^i)
    result = result ∧ (δ^{i,j} = 0)         // No time advancement
    return result
```

Fig. 5. Algorithm to generate an SMT statement for a guarded command

Note that $c_{t_2}^j$ is not assigned any value, since it is to be assigned the value $[-\infty, \infty]$. By not performing any assignment on $c_{t_2}^j$, it can take any value.

The time elapse portion of the next state assertion makes use of the possible rate set ($\mathcal{R}$) to calculate the values of real variables as a result of time moving forward. This algorithm is shown in Figure 6. In calculating the next state via time elapse, a new real variable is created representing the amount of time that has elapsed. This variable is referred to as $\delta^{i,j}$, and it represents the amount of time that has elapsed between iterations i and j. Since time is moving forward, $\delta^{i,j}$ is always greater than or equal to zero. All clock variables increase by exactly $\delta^{i,j}$. Next, based on the current values of the Boolean rate variables, the real variables change by some multiple of $\delta^{i,j}$. Lastly, all Boolean variables in the next iteration have the same value as in the current iteration. The complete time elapse assertion for the integrator, given the current iteration i and next iteration j, is:

$$
\begin{aligned}
&\delta^{i,j} \geq 0 \wedge c_{t_0}^j = c_{t_0}^i + \delta^{i,j} \wedge c_{t_1}^j = c_{t_1}^i + \delta^{i,j} \wedge c_{t_2}^j = c_{t_2}^i + \delta^{i,j} \wedge \\
&c_{t_3}^j = c_{t_3}^i + \delta^{i,j} \wedge c_{t_4}^j = c_{t_4}^i + \delta^{i,j} \wedge \\
&((\dot{V}out^i_{[18,22]} \wedge \overline{\dot{V}out^i_{[-22,-18]}} \wedge 18\delta^{i,j} + Vout^i \leq Vout^j \leq 22\delta^{i,j} + Vout^i) \vee \\
&(\overline{\dot{V}out^i_{[18,22]}} \wedge \dot{V}out^i_{[-22,-18]} \wedge -22\delta^{i,j} + Vout^i \leq Vout^j \leq -18\delta^{i,j} + Vout^i)) \wedge \\
&p_0^j = p_0^i \wedge p_1^j = p_1^i \wedge p_2^j = p_2^i \wedge p_3^j = p_3^i \wedge Vin^j = Vin^i \wedge fail^j = fail^i \wedge \\
&a_{t_0}^j = a_{t_0}^i \wedge a_{t_1}^j = a_{t_1}^i \wedge a_{t_2}^j = a_{t_2}^i \wedge a_{t_3}^j = a_{t_3}^i \wedge a_{t_4}^j = a_{t_4}^i \wedge \\
&\dot{V}out^j_{[18,22]} = \dot{V}out^i_{[18,22]} \wedge \dot{V}out^j_{[-22,-18]} = \dot{V}out^i_{[-22,-18]}
\end{aligned}
$$

The last step in the construction of the SMT formula is to assert that the property is violated (i.e., *fail* becomes true during some iteration). This is

```
mkExprForTimeElapse(𝓡, i, j)
    result = δ^{i,j} ≥ 0
    foreach c ∈ C   // Increment all clocks by δ
        result = result ∧ (c^j = c^i + δ^{i,j})
    rates = false
    foreach ⟨φ_R, R⟩ ∈ 𝓡   // Increment real variables based on 𝓡
        rate = φ_R
        foreach (v̇ := [r_l, r_u]) ∈ R
            rate = rate ∧ (v^j ≥ v^i + r_l δ^{i,j}) ∧ (v^j ≤ v^i + r_u δ^{i,j})
        rates = rates ∨ rate
    foreach b ∈ {M ∪ S ∪ A ∪ BR}   // Boolean variables stay same value
        result = result ∧ (b^j = b^i)
    result = result ∧ rates
    return result
```

Fig. 6. Algorithm to generate an SMT statement for the time elapse calculation

accomplished by constructing a disjunction of the *fail* variables over all iterations. For five iterations of the integrator example, the result is:

$$fail^0 \vee fail^1 \vee fail^2 \vee fail^3 \vee fail^4$$

The final step of the model checker is to apply the SMT checking procedure. If a satisfiable solution is found, this indicates that it is possible to reach the violating condition. In this event, the SMT solver generates a satisfying solution to the current context. This solution corresponds to a trace over all iteration's state variables beginning from the initial state to the error condition. Since this is a bounded model checker, if the property is not violated within the specified number of iterations, the property may still be violated after more iterations.

6 Results

The VHDL-AMS to LHPN compiler, the symbolic model generator, and the SMT bounded model checker have been implemented within the LEMA tool. This section compares the SMT model checker with BDD and DBM model checkers within LEMA. All results use a 2Ghz Intel CoreDuo with 2GB of memory.

The results for the integrator are shown in the top part of Table 1 in which the ranges of rate for the change of *Vout* are varied. In particular, when the lower and upper bound for these rates are equal, all three model checkers determine in a few seconds that the property is satisfied (i.e., the circuit does not saturate). Results for the SMT model checker are presented for both 10 and 20 iterations. When the lower and upper bounds are not equal, both SMT and BDD model checkers find a violation of the property. For example, if the rising slew rate of *Vout* is consistently larger than the falling slew rate, there can be a build up of charge leading to saturation of *Vout*. Note that the DBM model checker cannot directly support ranges of rates. Therefore, a piecewise approximate model must

Table 1. Switched capacitor integrator verification results

Example	Exp. Result	SMT Time (s)	SMT Iter.	BDD Time (s)	BDD Iter.	DBM Time (s)	DBM Zones
Original ([20, 20])	Pass	< 1	10	< 1	7	< 1	4
Original ([20, 20])	Pass	7	20	–	–	–	–
Original ([18, 22])	Fail	< 2	15	< 2	11	n/a	n/a
Piecewise ([18, 22])	Fail	60	20	< 1	6	< 1	9
Corrected	Pass	28	10	6*	6*	n/a	n/a
Corrected	Pass	388	20	–	–	–	–
Corrected piecewise	Pass	249	10	OOM	3	< 1	54
Corrected piecewise	Pass	980	20	–	–	–	–

* Verification result does not match expected result.

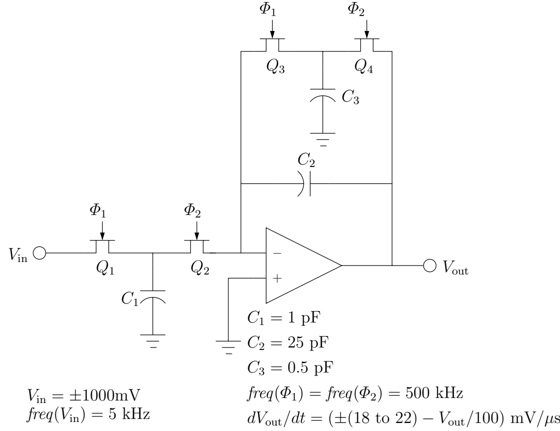


Fig. 7. Circuit diagram of a corrected switched capacitor integrator

first be generated in which the rate of *Vout* initially increases at 18 mV/μs. After some random amount of time, the rate may switch to 22 mV/μs. Decreasing rates for *Vout* are modeled in a similar way.

Saturation of the integrator can be prevented using the circuit shown in Figure 7. This circuit uses a switched capacitor resistor inserted in parallel with the feedback capacitor to cause *Vout* to drift back to 0 V. In other words, if *Vout* is increasing, it increases faster below 0 V than above. In this circuit's model, the range for *Vout* is 28 to 37 mV/μs when below -1000 mV, 18 to 32 mV/μs when below 0 mV, 8 to 22 mV/μs when below 1000 mV, and 3 to 12 mV/μs when above 1000 mV. Similar rates are used when *Vout* is decreasing. The verification results for the corrected integrator are shown in the bottom part of Table 1. The SMT model checker correctly determines that this circuit does not violate the property for 10 and 20 iterations. For this model, the BDD model checker finds a failure erroneously. This false negative is due to inexactness that results from

not adding transitivity constraints at all necessary phases of the the analysis. If transitivity constraints are added at each step, BDD analysis quickly runs out of memory. Since the DBM model checker does not support ranges of rates directly, it cannot be applied to this model. Again, an approximate piecewise model can be verified by the DBM model checker. Ironically, the SMT model checker performs better on the more accurate model, since the added transitions in the piecewise model significantly increase the complexity of the SMT formula.

7 Conclusions

This paper describes an SMT bounded model checking algorithm for AMS circuits. These circuits can be described using VHDL-AMS and automatically compiled into an LHPN representation for analysis. This LHPN model is translated into a symbolic model composed of an invariant, possible rates set, and guarded commands. This symbolic model is then automatically converted into an SMT formula for a given number of iterations. If this SMT formula is satisfiable, the satisfying assignment represents an error trace for the circuit being verified.

One promising abstraction and refinement approach is to combine the BDD and SMT model checkers. The BDD model checker is capable of performing an unbounded full state space exploration, but it often runs out of memory due to the large number of BDD variables created. The SMT model checker efficiently determines if the full model violates the property, but it can never guarantee that the property is not violated. Therefore, the BDD model checker could be applied to an abstract model. If the BDD model checker determines that the property is violated in the abstract model, the SMT model checker can be used with the full model to ensure that the failure is not a false negative. In this case, the BDD model checker would specify the number of iterations that are required for the abstract model to fail. If the SMT model checker verifies that the full model does fail, verification is complete. If the full model does not violate the property, the violation is a false negative and the unsatisfying core can be used to refine the abstract model. This process repeats until a true failure is found or the BDD model checker determines that the abstract model does not violate the property. Another interesting approach to consider would be to apply the proof-based iterative abstraction and refinement method from [28].

References

1. Kurshan, R.P., McMillan, K.L.: Analysis of digital circuits through symbolic reduction. IEEE Transactions on CAD 10(11), 1356–1371 (1991)
2. Hartong, W., Hedrich, L., Barke, E.: Model checking algorithms for analog verification. In: Proc. of DAC, pp. 542–547 (2002)
3. Gupta, S., Krogh, B.H., Rutenbar, R.A.: Towards formal verification of analog designs. In: Proc. of ICCAD, pp. 210–217 (2004)
4. Dang, T., Donze, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid systems techniques. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 21–36. Springer, Heidelberg (2004)

5. Frehse, G., Krogh, B.H., Rutenbar, R.A.: Verifying analog oscillator circuits using forward/backward refinement. In: Proc. of DATE, pp. 257–262 (2006)
6. Little, S., Seegmiller, N., Walter, D., Myers, C.J.: Verification of analog/mixed-signal circuits using labeled hybrid petri nets. In: Proc. of ICCAD, pp. 275–282 (2006)
7. Walter, D., Little, S., Seegmiller, N., Myers, C., Yoneda, T.: Symbolic model checking of analog/mixed-signal circuits. In: Proc. of ASPDAC, pp. 316–323 (2007)
8. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
9. Armando, A., Castellini, C., Giunchiglia, E.: SAT-based procedures for temporal reasoning. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 97–108. Springer, Heidelberg (2000)
10. Armando, A., Castellini, C., Giunchiglia, E., Maratea, M.: A sat-based decision procedure for the boolean combination of difference constraints. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, Springer, Heidelberg (2005)
11. Barrett, C., Dill, D., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to sat. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
12. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 317–333. Springer, Heidelberg (2005)
13. de Moura, L., Rue, H.: Lemmas on demand for satisfiability solvers. In: Proc. of SAT (2002)
14. Filliâtre, J.C., Owre, S., Rue, H.: ICS: Integrated Canonization and Solving (Tool presentation). In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 246–249. Springer, Heidelberg (2001)
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
16. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
17. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Ranise, S., van Rossum, P., Sebastiani, R.: Efficient satisfiability modulo theories via delayed theory combination. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 335–349. Springer, Heidelberg (2005)
18. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and Abstract DPLL Modulo Theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
19. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
20. Myers, C.J., Harrison, R.R., Walter, D., Seegmiller, N., Little, S.: The case for analog circuit verification. Electronic Notes Theoretical Computer Science 153(3), 53–63 (2006)
21. Zheng, H.: Specification and compilation of timed systems. Master's thesis, University of Utah (1998)
22. Myers, C.: Asynchronous Circuit Design. Wiley, Chichester (2001)

23. David, R., Alla, H.: On hybrid petri nets. Discrete Event Dynamic Systems: Theory and Applications 11, 9–40 (2001)
24. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems, pp. 209–229 (1992)
25. Walter, D.: Verification of Analog and Mixed-Signal Circuits Using Symbolic Methods. PhD thesis, University of Utah (2007)
26. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. In: 7th Symposium of Logics in Computer Science, pp. 394–406. IEEE Computer Scienty Press, Los Alamitos (1992)
27. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) PNPM 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994), citeseer.ist.psu.edu/pastor94petri.html
28. McMillan, K., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)

Model Checking Contracts – A Case Study*

Gordon Pace[1], Cristian Prisacariu[2], and Gerardo Schneider[2]

[1] Dept. of Computer Science and AI, University of Malta, Msida, Malta
[2] Department of Informatics – University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
gordon.pace@um.edu.mt, {cristi,gerardo}@ifi.uio.no

Abstract. Contracts are agreements between distinct parties that determine rights and obligations on their signatories, and have been introduced in order to reduce risks and to regulate inter-business relationships. In this paper we show how a conventional contract can be written in the contract language $\mathcal{CL}$, model the contract and verify properties of the model using the NuSMV model checking tool.

1 Introduction

Internet-based applications involving one or more entities participating in inter-business collaborations, virtual organisations, and web services, usually communicate through service exchanges. Such exchanges are subject to certain understanding on the different roles the participants play, including assumptions on their correct and incorrect behaviours, and their rights and obligations in order to avoid misunderstanding and ambiguities in such business relationships. This motivates the need of establishing an agreement before any transaction is performed, through a *contract*, guaranteeing the rights and duties of each signatory. Such documents may also contain clauses determining penalties in case of contract violations, and be as unambiguous as possible to avoid conflicting interpretations. *Conventional contracts* are documents written in natural language, as one may find in usual judicial or commercial traditional activities. On the other hand, *electronic contracts* (or e-contracts for short) are machine-oriented and as such they must be "understood" by the software responsible for controlling and monitoring the service exchanges. E-contracts might be seen in two different ways: (1) As the executable version of a conventional contract, obtained from the translation of the "paper" version into the electronic one; (2) As contracts by themselves obtained directly from certain software applications, like web services and virtual organisations. For our current purposes, the difference above is irrelevant, though our case study is based on a conventional contract.

Ideally, e-contracts should be shown to be contradiction-free both internally, and with respect to the governing policies under which the contract is enacted.

* Partially supported by the Nordunet3 project "Contract-Oriented Software Development for Internet Services".

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 82–97, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Moreover, there must be a run-time system ensuring that the contract is respected. In other words, contracts should be amenable to formal analysis allowing both static and dynamic verification, and thus written in a formal language. In this paper we are interested only in the analysis of the contract itself (statically), and we are not concerned with its relation with policies nor with its enforcement at run-time.

A formal language for writing contracts should be designed as to avoid most of the philosophical problems of deontic logic [11]. Moreover, it should be possible to represent conditional obligations, permissions and prohibitions, as well as *contrary-to-duty obligations* (CTD) and *contrary-to-prohibitions* (CTP). CTDs are statements representing obligations that might not be respected, whereas CTPs are similar statements dealing with prohibitions that might be violated. Both constructions specify the obligation/prohibition to be fulfilled and which is the *reparation/penalty* to be applied in case of violation.

A formal language for writing (untimed) contracts is $\mathcal{CL}$ [13]. The language is tailored to e-contracts, following an action-based approach, and having the following properties: (1) The language avoids most of the classical paradoxes of deontic logic; (2) It is possible to express in the language (conditional) obligations, permissions and prohibitions over concurrent actions keeping their intuitive meaning; (3) It is possible to express CTDs and CTPs; (4) The language has a formal semantics given in a variant of the modal μ-calculus.

The main contribution of this paper is to show how model checking techniques can be applied in the context of contract-oriented software development, in order to determine whether a given contract stipulates what it is supposed to. $\mathcal{CL}$ is used as an intermediary between the contract clauses in plain English and the system specification required by the model checking tool. This use of $\mathcal{CL}$ increases the confidence in the initial formulation of the contract clauses. The model checking method that we present requires to pursue the following steps:

1. Model the conventional contract written in English into the formal language $\mathcal{CL}$;
2. Translate syntactically the $\mathcal{CL}$ specification into the extended μ-calculus $\mathcal{C}\mu$;
3. Obtain a Kripke-like model (a labelled transition system with state propositions — LTS) of the $\mathcal{C}\mu$ formulae;
4. Translate the LTS into the input language of NuSMV;
5. Perform model checking using NuSMV;
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied;
7. Finally, repair the original contract by adding a corresponding clause, if applicable.

The paper is organised as follows. In Section 2 we start by presenting the language $\mathcal{CL}$, including an example of the kind of contracts we are dealing with, from which we will extract our case study. Section 3 is the main part of the paper where we first formalise the case study in $\mathcal{CL}$, and afterwards we show how to use model checking and the NuSMV tool to determine whether the contract is

correct with respect to certain desired properties, and how to get feedback as to write the "correct" contract. In Section 4 we analyse related works and conclude by discussing our choice of the model checking tool as well as future work.

2 A Formal Language for Contracts

We present in Fig. 1 a part of a conventional contract between a service provider and a client, where the provider gives access to Internet to the client. We analyse part of this contract in the following section. First we recall the contract language $\mathcal{CL}$; for a more detailed presentation see [13].

Definition 1 (Contract Language Syntax). *A contract is defined by:*

$$
\begin{aligned}
\mathcal{C}ontract &:= \mathcal{D}\ ;\ \mathcal{C} \\
\mathcal{C} &:= \phi \mid \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc \mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O &:= O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P &:= P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F &:= F(\delta) \mid \mathcal{C}_F \vee [\delta]\mathcal{C}_F
\end{aligned}
$$

The syntax of $\mathcal{CL}$ closely resembles the syntax of a modal (deontic) logic. Though this similarity is clearly intentional since we are driven by a logic-based approach, $\mathcal{CL}$ is *not* a logic. The semantics of $\mathcal{CL}$ are given in an extension of μ-calculus [8] which we call $\mathcal{C}\mu$. In what follows we provide an intuitive explanation of the $\mathcal{CL}$ syntax.

A contract consists of two parts: *definitions* ($\mathcal{D}$) and *clauses* ($\mathcal{C}$). We deliberately let the definitions part underspecified in the syntax above. $\mathcal{D}$ specifies the *assertions* (or conditions) and the atomic actions present in the clauses. ϕ denotes assertions and ranges over boolean expressions including the usual boolean connectives, and arithmetic comparisons like "the budget is more than 200\$". We let the atomic actions underspecified, which for our purposes can be understood as consisting of three parts: the proper action, the subject performing the action, and the target of (or, the object receiving) such an action. Note that, in this way, the parties involved in a contract are encoded in the actions.

$\mathcal{C}$ is the general *contract clause*. $\mathcal{C}_O$, $\mathcal{C}_P$, and $\mathcal{C}_F$ denote respectively *obligation*, *permission*, and *prohibition* clauses. $O(\cdot)$, $P(\cdot)$, and $F(\cdot)$, represents the obligation, permission or prohibition of performing a given action. $\wedge$ and $\oplus$ may be thought as the classical conjunction and exclusive disjunction, which may be used to combine obligations and permissions. For prohibition $\mathcal{C}_F$ we have $\vee$, again with the classical meaning of the corresponding operator. α is a compound action (i.e., an expression containing one or more of the following operators: choice "+"; sequence "$\cdot$"; concurrency "&", and test "?" —see [13]), while δ denotes a compound action not containing any occurrence of $+$. Note that syntactically $\oplus$ cannot appear between prohibitions and $+$ cannot occur under the scope of F.

This deed of **Agreement** is made between:
1. **[name]**, from now on referred to as **Provider** and
2. **[name]**, from now on referred to as the **Client**.

INTRODUCTION
3. The **Provider** is obliged to provide the **Internet Services** as stipulated in this **Agreement**.
5. DEFINITIONS
5.1. j) **Internet traffic** may be measured by both **Client** and **Provider** by means of Equipment and may take the two values **high** and **normal**.

OPERATIVE PART
7. CLIENT'S RESPONSIBILITIES AND DUTIES
7.1. The **Client** shall not:
a) supply false information to the Client Relations Department of the **Provider**.
7.2. Whenever the Internet Traffic is **high** then the **Client** must pay [*price*] immediately, or the **Client** must notify the **Provider** by sending an e-mail specifying that he will pay later.
7.3. If the **Client** delays the payment as stipulated in 7.2, after notification he must immediately lower the Internet traffic to the **normal** level, and pay later twice ($2 * [price]$).
7.4. If the **Client** does not lower the Internet traffic immediately, then the **Client** will have to pay $3 * [price]$.
7.5. The **Client** shall, as soon as the Internet Service becomes operative, submit within seven (7) days the Personal Data Form from his account on the **Provider**'s web page to the Client Relations Department of the **Provider**.
8. CLIENT'S RIGHTS
8.1. The **Client** may choose to pay either:
a) each month; b) each three (3) months; c) each six (6) months;
9. PROVIDER'S SERVICE
9.2. As part of the Service offered by the **Provider** the **Client** has the right to an e-mail and an user account.
9.3. **Provider** is obliged to offer with no limitation and within a period of seven (7) days a password and any other Equipment Specific to Client, necessary for the correct usage of the user account, upon receiving of all the necessary data about the client from the Client Relations Department of the **Provider**.
9.4. Each month the **Client** pays the *bill* the **Provider** is obliged to send a Report of Internet Usage to the Client.
10. PROVIDER'S DUTIES
10.1. The **Provider** takes the obligation to return the personal data of the client to the original status upon termination of the present **Agreement**, and afterwards to delete and not use for any purpose any whole or part of it.
10.2. The **Provider** guarantees that the Client Relations Department, as part of his administrative organisation, will be responsive to requests from the **Client** or any other Department of the **Provider**, or the **Provider** itself within a period less than two (2) hours during *working hours* or the day after.
11. PROVIDER'S RIGHTS
11.1. The **Provider** takes the right to alter, delete, or use the *personal data* of the **Client** only for statistics, monitoring and internal usage in the confidence of the **Provider**.
11.2. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:
b) Suspend Internet Services immediately if **Client** is in breach of Clause 7.1;
13. TERMINATION
13.1. Without limiting the generality of any other *Clause* in this *Agreement* the **Client** may terminate this *Agreement* immediately without any notice and being vindicated of any of the Clause of the present Agreement if:
a) the **Provider** does not provide the Internet Service for seven (7) days consecutively.
13.2. The **Provider** is forbidden to terminate the present Agreement without previous written notification by normal post and by e-mail.
13.3. The **Provider** may terminate the present Agreement if:
a) any payment due from **Client** to **Provider** pursuant to this **Agreement** remains unpaid for a period of fourteen (14) days;
16. GOVERNING LAW
16.1. The **Provider** and the present **Agreement** are governed by and construed according to the Law Regulating Internet Services and to the Law of the State.
a) The Law of the State stipulates that any **ISP Provider** is obliged, upon request to seize any activity until further notice from the State representatives.

Fig. 1. Part of a contract between an Internet provider and a client

We borrow from propositional dynamic logic [6] the syntax $[\alpha]\phi$ to represent that after performing α (if it is possible to do so), ϕ must hold. The $[\cdot]$ notation allows having a *test*, where $[\phi?]\mathcal{C}$ must be understood as $\phi \Rightarrow \mathcal{C}$. $\langle\alpha\rangle\phi$ captures the idea that it exists the possibility of executing α, in which case ϕ must hold afterwards. Following temporal logic (TL) notation we have $\mathcal{U}$ (*until*), $\bigcirc$ (*next*), and $\Box$ (*always*), with intuitive semantics as in TL [12]. Thus $\mathcal{C}_1 \mathcal{U} \mathcal{C}_2$ states that $\mathcal{C}_1$ holds until $\mathcal{C}_2$ holds. $\bigcirc\mathcal{C}$ intuitively states that $\mathcal{C}$ holds in the next moment, usually after something happens, and $\Box\mathcal{C}$ expressing that $\mathcal{C}$ holds in every moment. We can define $\Diamond\mathcal{C}$ (*eventually*) for expressing that $\mathcal{C}$ holds sometimes in a future moment.

To express CTDs we provide the following notation, $O_\varphi(\alpha)$, which is syntactic sugar for $O(\alpha) \wedge [\overline{\alpha}]\varphi$ stating the obligation to execute α, and the reparation φ in case the obligation is violated, i.e. whenever α is not performed. The reparation may be any contract clause. Similarly, CTP statements $F_\varphi(\alpha)$ can be defined as $F_\varphi(\alpha) = F(\alpha) \wedge [\alpha]\varphi$, where φ is the penalty in case the prohibition is violated. Notice that it is possible to express nested CTDs and CTPs.

In $\mathcal{CL}$ we can write *conditional* obligations, permissions and prohibitions in two different ways. Just as an example let us consider conditional obligations. The first kind is represented as $[\alpha]O(\beta)$, which may be read as "after performing α, one is obliged to do β". The second kind is modelled using the test operator ?: $[\varphi?]O(\alpha)$, representing "If φ holds then one is obliged to perform α". Similarly for permission and prohibition. For convenience, in what follows we use the notation $\phi \Rightarrow \mathcal{C}$ instead of the $\mathcal{CL}$ syntax $[\phi?]\mathcal{C}$.

3 A Contract Case Study

In what follows we consider part 7 of the contract given in Fig. 1 between a service provider and a client, where the provider gives access to the Internet to the client. We consider two parameters of the service: *high* and *normal*, which denote the client's Internet traffic. We will consider only the following clauses of the contract.

7.1. The **Client** shall not:
a) supply false information to the Client Relations Department of the **Provider**.
7.2. Whenever the Internet Traffic is **high** then the **Client** must pay [*price*] immediately, or the **Client** must notify the **Provider** by sending an e-mail specifying that he will pay later.
7.3. If the **Client** delays the payment as stipulated in 7.2, after notification he must immediately lower the Internet traffic to the **normal** level, and pay later twice ($2 * [price]$).
7.4. If the **Client** does not lower the Internet traffic immediately, then the **Client** will have to pay $3 * [price]$.
7.5. The **Client** shall, as soon as the Internet Service becomes operative, submit within seven (7) days the Personal Data Form from his account on the **Provider**'s web page to the Client Relations Department of the **Provider**.

We also add clause 11.2 as it is strongly related to clause 7.1 and the two should be taken together:

11.2. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:
b) Suspend Internet Services immediately if **Client** is in breach of Clause 7.1;

In what follows we formalise the above contract clauses. As part of the formalisation of a contract in $\mathcal{CL}$ we first have to define the assertions and actions:

ϕ = the Internet traffic is high
fi = client supplies false information to Client Relations Department
h = client increases Internet traffic to *high* level
p = client pays [price]
d = client delays payment
n = client notifies by e-mail
l = client lowers the Internet traffic
sfD = client sends the Personal Data Form to Client Relations Department
o = provider activates the Internet Service (it becomes operative)
s = provider suspends service

Note that we have the action h which does not appear explicitly in the example clauses. Action h is implicit as it makes the proposition ϕ valid (the Internet becomes *high* only if the client increases it). Action h can be considered as the complement of action l which makes ϕ false (lowers the Internet traffic). The six clauses above are written in $\mathcal{CL}$ as follows:

1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \overline{l}\,]\Diamond O(p\&p\&p))$
5. $\Box([o]O(sfD))$

Clause 1 has a concise syntax and represents a *contrary-to-prohibition.* More precisely, the CTP represents the prohibition $F(fi)$ (clause 7.1) and the reparation which should be enforced in case the prohibition is violated (in this case $P(s)$; the right of the provider to suspend the Internet service, clause 11.2).

Note that all the clauses are supposed to hold throughout the whole contract because of the $\Box$. Clause 2 models clause 7.2 of the contract example and it represents the fact that whenever the assertion ϕ holds (the Internet traffic of the client is at the *high* level) then it must be the case that the client is obliged to choose (+) between either paying immediately (p) or delaying the payment by sending the notification ($d\&n$).

Clauses 3 and 4 refer to the clauses 7.3 and 7.4 of the contract example. They both refer to the moment after the client has delayed the payment ($[d\&n]$). Clause 3 states that the client has the obligation to lower the Internet traffic ($O(l)$) and that after lowering the client should pay twice the price. On the other hand, clause 4 specifies the obligation of the client to pay three times the price in case he does not lower the Internet traffic ($\overline{l}$). The two formulae may be combined in a single formula using CTDs: $\Box([d\&n](O_{\varphi}(l) \wedge [l]\Diamond(O(p\&p))$ where $\varphi = O(p\&p\&p)$. Clause 5 formally represents clause 7.5 of the contract example. It represents the obligation of the client to submit the form ($O(sfD)$) after the Internet service becomes operative ($[o]$).

Table 1. The translation function $f^{\mathcal{T}}$ from $\mathcal{CL}$ to $\mathcal{C}\mu$

$$
\begin{array}{lrcl}
(1) & f^{\mathcal{T}}(O(\&_{i=1}^{n} a_i)) & = & \langle\{a_1,\ldots,a_n\}\rangle(\wedge_{i=1}^{n} O_{a_i}) \\
(2) & f^{\mathcal{T}}(\mathcal{C}_O \oplus \mathcal{C}_O) & = & f^{\mathcal{T}}(\mathcal{C}_O) \wedge f^{\mathcal{T}}(\mathcal{C}_O) \\
(3) & f^{\mathcal{T}}(P(\&_{i=1}^{n} a_i)) & = & \langle\{a_1,\ldots,a_n\}\rangle(\wedge_{i=1}^{n} \neg\mathcal{F}_{a_i}) \\
(4) & f^{\mathcal{T}}(\mathcal{C}_P \oplus \mathcal{C}_P) & = & f^{\mathcal{T}}(\mathcal{C}_P) \wedge f^{\mathcal{T}}(\mathcal{C}_P) \\
(5) & f^{\mathcal{T}}(F(\&_{i=1}^{n} a_i)) & = & [\{a_1,\ldots,a_n\}](\wedge_{i=1}^{n} \mathcal{F}_{a_i}) \\
(6) & f^{\mathcal{T}}(F(\delta) \vee [\beta]F(\delta)) & = & f^{\mathcal{T}}(F(\delta)) \vee f^{\mathcal{T}}([\beta]F(\delta)) \\
(7) & f^{\mathcal{T}}(\mathcal{C}_1 \wedge \mathcal{C}_2) & = & f^{\mathcal{T}}(\mathcal{C}_1) \wedge f^{\mathcal{T}}(\mathcal{C}_2) \\
(8) & f^{\mathcal{T}}(\bigcirc\mathcal{C}) & = & [\mathbf{any}]f^{\mathcal{T}}(\mathcal{C}) \\
(9) & f^{\mathcal{T}}(\mathcal{C}_1 \mathcal{U} \mathcal{C}_2) & = & \mu Z.f^{\mathcal{T}}(\mathcal{C}_2) \vee (f^{\mathcal{T}}(\mathcal{C}_1) \wedge [\mathbf{any}]Z \wedge \langle\mathbf{any}\rangle\top) \\
(10) & f^{\mathcal{T}}(\Box\mathcal{C}) & = & \nu Z.\mathcal{C} \wedge [\mathbf{any}]Z \\
(11) & f^{\mathcal{T}}([\&_{i=1}^{n} a_i]\mathcal{C}) & = & [\{a_1,\ldots,a_n\}]f^{\mathcal{T}}(\mathcal{C}) \\
(12) & f^{\mathcal{T}}([(\&_{i=1}^{n} a_i)\alpha]\mathcal{C}) & = & [\{a_1,\ldots,a_n\}]f^{\mathcal{T}}([\alpha]\mathcal{C}) \\
(13) & f^{\mathcal{T}}([\alpha+\beta]\mathcal{C}) & = & f^{\mathcal{T}}([\alpha]\mathcal{C}) \wedge f^{\mathcal{T}}([\beta]\mathcal{C}) \\
(14) & f^{\mathcal{T}}([\varphi?]\mathcal{C}) & = & f^{\mathcal{T}}(\varphi) \Rightarrow f^{\mathcal{T}}(\mathcal{C})
\end{array}
$$

3.1 Translating the $\mathcal{CL}$ Specification into $\mathcal{C}\mu$

We extract a model from the $\mathcal{CL}$ clauses by first translating the language specification into the extended μ-calculus $\mathcal{C}\mu$ where the semantics is given as a special labelled transition system. The translation function $f^{\mathcal{T}}$ which takes a $\mathcal{CL}$ formula and returns a formula in the $\mathcal{C}\mu$ is shown in Table 1. The special syntax $[\mathbf{any}]$ (or the dual $\langle\mathbf{any}\rangle$) represents the fact that any action can be executed. To represent obligations and prohibitions of a given action a we need the special propositional constants O_a and $\mathcal{F}_a$.

We briefly mention here the semantics of $\mathcal{C}\mu$, see [13] for more details. The formulae are interpreted over a labelled transition system (LTS). The labels of the transitions are represented by multi-sets of actions (e.g. $\{p,p,p\}$ is a label corresponding to the $\mathcal{CL}$ concurrent action term $p\&p\&p$). The formulae are interpreted over states as usual in modal logics with semantics on LTSs. For example the expression $\phi \Rightarrow \langle p\rangle O_p$ is interpreted in a state and should be understood as: if the assertion ϕ holds in the state then $\langle p\rangle O_p$ should hold in the same state. $[p]\mathcal{C}$ and $\langle p\rangle\mathcal{C}$ are interpreted as holding in the current state if and only if in the next state reachable by action p the formula corresponding to the translation of $\mathcal{C}$ holds. In $\mathcal{C}\mu$ the difference between the two operators is that $\langle p\rangle\varphi$ requires the existence of at least one next state reachable by p where φ holds, where $[p]\varphi$ is quantified universally, and thus the formula also holds in case the set of states reachable by p is empty.

We will now translate the five $\mathcal{CL}$ clauses corresponding to the contract given above, into $\mathcal{C}\mu$. Note that we use the $\Box$ and $\Diamond$ with their classical interpretation from temporal logics; the last not being included in the Table 1. It is known [2] that $f^{\mathcal{T}}(\Diamond\mathcal{C}) = f^{\mathcal{T}}(\top\mathcal{U}\mathcal{C}) = \mu Z.\mathcal{C} \vee ([\mathbf{any}]Z \wedge \langle\mathbf{any}\rangle\top)$. In order to translate the first clause of the $\mathcal{CL}$ representation above we can proceed as follows:

$$
\begin{array}{rl}
 & f^{\mathcal{T}}(\Box F_{P(s)}(fi)) = \nu Z.f^{\mathcal{T}}(F_{P(s)}(fi)) \wedge [\mathbf{any}]Z, \\
\text{where:} & f^{\mathcal{T}}(F_{P(s)}(fi)) = f^{\mathcal{T}}(F(fi) \wedge [fi]P(s)) = [fi]\mathcal{F}_{fi} \wedge [fi]\langle s\rangle\neg F_s.
\end{array}
$$

In this manner, we use $\Box$ operator in the clauses below simply as syntactic sugar which is reduced to the ν operator in μ-calculus.

1. $\Box[fi]\mathcal{F}_{fi} \wedge [fi]\langle s\rangle\neg F_s$
2. $\Box[h](\phi \Rightarrow (\langle p\rangle O_p \wedge \langle\{d,n\}\rangle(O_d \wedge O_n)))$
3. $\Box[\{d,n\}](\langle l\rangle O_l \wedge [l](\mu Z.\langle\{p,p\}\rangle O_p \vee ([\mathbf{any}]Z \wedge \langle\mathbf{any}\rangle\top)))$
4. $\Box[\{d,n\}][\bar{l}](\mu Z.\langle\{p,p,p\}\rangle O_p \vee ([\mathbf{any}]Z \wedge \langle\mathbf{any}\rangle\top))$
5. $\Box[o]\langle sfD\rangle O_{sfD}$

3.2 From $\mathcal{C}\mu$ to the LTS

In Fig. 2 we have pictured one model of the above clauses where we denote by *else* all other actions different than the ones from the current node (e.g. for the state s_7 in the picture $else = \mathbf{any} \setminus \{fi\}$).

Note that because of the semantics of the prohibition $F(fi)$ (i.e., $[fi]\mathcal{F}_{fi}$), we would not need to explicitly add a transition from each state labelled with fi to a state with the propositional constant $\mathcal{F}_{fi}$. However, in the presence of a CTP, as it is the case with clause 1, we need to do so in order to represent the reparation $P(s)$.

We attempt to build a model in the form of an LTS — in a certain sense an implementation of the contract as specified. The process is done manually and prone to error — to ensure correctness of the automata we build, we model check them against the contract specification. Furthermore, multiple models satisfying the contract specification exist, ranging from the weakest being equivalent to the specification itself, to stronger and more concrete implementations. In this paper we are not concerned with achieving the weakest model.

Although the weakest model is desirable to have, we can still reason about our contract based on a (correct) model we build. Given a model $\mathcal{M}$ and contract specification $\mathcal{C}$, we start off by proving that the model really implements the contract: $\mathcal{M} \models \mathcal{C}$. We note that when the model does not satisfy a property π: $\mathcal{M} \not\models \pi$, it immediately follows that neither does the contract: $\mathcal{C} \not\models \pi$, thus enabling us to discover bugs in our specification as translated from the natural language, or in the original natural language contract itself. On the other hand, using this approach we cannot prove the correctness of the original contract. Were we able to obtain the weakest model, we would have been able to reason directly about the contract specification itself.

In what follows, we will specify this model using the input language of NuSMV, and prove that it is indeed a model of the $\mathcal{CL}$ formulae.

3.3 From the LTS to the NuSMV Input Syntax

In NuSMV [4], a model can be specified in two ways: either using *assignments* or by *direct specification*. We choose to use the direct specification technique as it enables us to translate our system more directly into NuSMV.

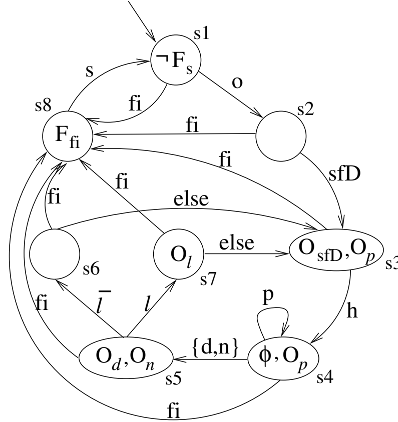


Fig. 2. Example of a model for the five clauses written in $\mathcal{CL}$

NuSMV uses *state variables* to identify states; the number of states is determined by the product of the number of different values each state variable can take. There is also a second kind of variables, *input variables* which are meant to specify labels of a labelled transition system. Since we have actions as labels, we make substantial use of the input variables in our application.

We have defined an input variable for each atomic action of the $\mathcal{CL}$ specification. The type of the input variables is `boolean` so that if the value of `d = false` then `d` is not an active label of the transition. Whenever a variable is left unspecified then NuSMV interprets it as having any value so it creates a transition (or a state in case of state variables) for each value of the variable.

In NuSMV it is easy to simulate the concurrent labels $\{d, n\}$ of $\mathcal{C}\mu$ which mean that the transition is taken if both actions d and n are executed concurrently: we activate both input variables `d = true ; n = true`. We can also represent the resource-awareness of the labels (i.e. the $p\&p$ of $\mathcal{CL}$, or the $\{p, p\}$ of $\mathcal{C}\mu$) by defining the input variable with the type *range of integers*. If `p = 0` then the transition is not labelled with the action p; if `p = 1` then the transition is labelled with one normal action p (like in the case of `boolean` type); but if `p = 2` then we take the transition if two copies of the action p are executed concurrently. We have then the following declaration of variables:

```
IVAR
  d : boolean ;
  n : boolean ;
  p : 0 .. 3 ;
```

Note that we may have *empty transitions* (with no label) by giving to all the input variables the value `false` (or `p = 0`). Moreover, we may represent the special action **any** of $\mathcal{C}\mu$ by leaving all input variables unspecified.

We have defined a state variable named `state` of enumeration type so it can take only eight values, corresponding to the eight states depicted in Fig. 2.

```
VAR
  state : {s1,s2,s3,s4,s5,s6,s7,s8} ;
```

Other variables are declared accordingly (e.g., `high : boolean`). Moreover, we define a state variable of type `boolean` for each input variable. This is required by the $\mathcal{C}\mu$ where we have a propositional constant O_a or $\mathcal{F}_a$ associated to each atomic action a which enters under the scope of an obligation or of a prohibition respectively:

```
  F_s : boolean ;  F_fi : boolean ;
  O_p : boolean ;  O_d : boolean ;  O_n : boolean ;
  O_l : boolean ;  O_sfD : boolean ;
```

As an example, we show below the encoding of the initial state, and one of its outgoing transitions, of the automaton in Fig. 2. We call the initial state `s1`.

```
INIT
  (state = s1) & !high &
  !F_fi & !O_p & !O_d & !O_n & !O_l & !O_sfD & !F_s ;
```

The transitions are specified using the `TRANS` keyword followed by a propositional formula which determines the pairs of states that form the transition relation. The propositional formula contains names of state variable (which are tested in the current state) and `next` expressions which refer to the value of the state variables in the next state. It also contains the input variables to model the labels of the transitions. Remember that any variable that is missing from the formula is interpreted as having any value and will give rise to a number of different transitions equal to the number of values it can take.

```
TRANS
--state variables of the current state
  ((state = s1) & !high &
   !F_fi & !O_p & !O_d & !O_n & !O_l & !O_sfD & !F_s &
--input variables as the labels
   (!fi & p = 0 & !d & !n & !l & !negl & !sfD & o & !s) &
--the values of the state variables in the next states
   (next(state) = s6) & !next(high) &
   next(!F_fi & !O_p & !O_d & !O_n & !O_l & !O_sfD & !F_s))
```

3.4 Model Checking the Contract

We propose to combine the contract specification and the model we build in different ways with model checking techniques to help us improve the contract and increase our confidence in our model.

Proving that the model satisfies the original clauses: Clearly, to have confidence that we are reasoning using a correct model, we need to prove that the automaton

of Fig. 2, specified in NuSMV[1] respects the five $\mathcal{CL}$ clauses representing the statements from the contract example. For this we have specified each clause as a special LTL specification in NuSMV:

```
G ((fi -> X F_fi) & (fi -> X (s & X !F_s)))
G (h -> X (high -> ((p = 1 -> X O_p) &
                    ((d & n) -> X (O_d & O_n)))))
G ((d & n) -> X ((l -> X O_l) & l -> X F (p = 2 -> X O_p)))
G ( (d & n) -> X (l -> X F (p = 3 -> X O_p)))
G (o -> X (sfD -> X O_sfD))
```

The first, second and fourth properties go through immediately. The third fails, but upon investigation, it turns out that the actual contract wording gave a dependency between the second and third properties — the $d\&n$ action in the third property only refers to ones produced in the context of the second property (just after the Internet traffic going high and the user paying once). This indicates that the two ought to be combined together either by adding extra logic to indicate the dependency, or by merging then into a single property. We choose the latter, obtaining:

```
G (h -> X (high -> ((p = 1 -> X O_p) &  ((d & n) ->
                  X (O_d & O_n & (l -> X O_l) &
                     l -> X F (p = 2 -> X O_p))))))
```

This new property can be verified of our model.

Finally, the fifth property fails, suggesting that our model is incorrect. However, upon inspection it was realised that nothing in the contract specifies that the activation of the service happens once, or that the user's obligation is only valid the first time the activation occurs. We choose to revise the original contract to state that: "The first time the service becomes operative, the client is obliged to send the Personal Data Form to Client Relations Department". This is formulated as the following property, which model checks:

```
(!o) U (o -> X(can_sfD & (sfD -> X O_sfD)))
```

An alternative solution is to ensure that the contract is only in force once the Internet Service becomes operative, and simplify the property accordingly.

Verifying a property about client obligations: The first desirable property we want to check on the contract model can be expressed in English as: "It is always the case that whenever the Internet traffic is high, if the clients pays immediately, then the client is *not* obliged to pay again immediately afterwards". The property is expressed in $\mathcal{CL}$-like syntax[2] as: $\Box\neg(\phi \Rightarrow [p]\neg O(p))$. The property proves to be false, as can be seen in the transcript below, which includes a counter-example:

[1] The NuSMV code we have used is available on Nordunet3 project homepage: `http://www.ifi.uio.no/~gerardo/nordunet3/software.shtml`

[2] Notice that formally in $\mathcal{CL}$ there is no negation at the clause level.

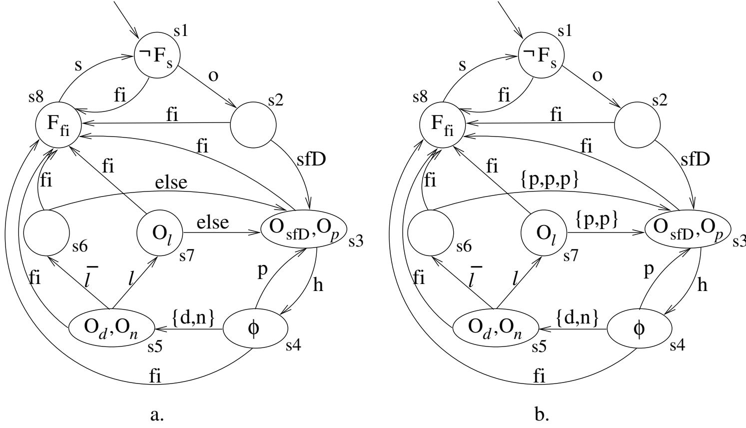


Fig. 3. The model of Fig. 2 corrected.

```
NuSMV > check_ltlspec
-- specification
     G (!high | (p = 1 ->  X (p = 1 ->  X !O_p))) is false
-- as demonstrated by the following execution sequence
-> State: 2.1 <-
  state = s1; o = 1
-> State: 2.2 <-
  state = s2; sfD = 1
-> State: 2.3 <-
  state = s3; O_p = 1; O_sfD = 1; h = 1
-- Loop starts here
-> State: 2.4 <-
  state = s4; high = 1; O_sfD = 0; p = 1
-- Loop starts here
-> State: 2.5 <-
  p = 1
```

The above counter-example shows that in state `s4` of Fig. 2 the client must fulfil one of the following obligations: or to pay (`p`), or to delay payment and notify (`d,n`). However, after paying once, the automaton is still in a state with high traffic (state `s4`), and thus the client is still obliged to pay again.

We give in Fig. 3-a the new model, which is proved correct with respect to the above property. The difference is the transition $s_4 \xrightarrow{p} s_3$ which replaces the one labelled with p from s_4 to itself. From this it is easy now to modify the original contract by introducing the following clause: "The provider guarantees that if the Internet traffic of the Client reaches a high level and the Client pays the [price] then it will not be obliged to pay the [price] again".

Notice that though we have obtained a new model that satisfies the property (and a clause in the original contract solving the above problem), the solution is still not satisfactory, as the contract does not specify what happens after the

client pays but does not decrease the Internet traffic. In the new model shown in Fig. 3-a this is reflected by the fact that after taking the new added transition (from `s4` to `s3`), there is an implicit assumption that the Internet traffic is low. For brevity we do not further analyse the contract in order to obtain the right contract concerning this problem, though it can be done following a similar approach as above.

Verifying a property about payment in case of increasing Internet traffic: The checking of the previous property was done for the benefit of the client. We now perform model checking in order to increase the confidence of the provider of the service.

We are interested in proving that: "It is always the case that whenever Internet traffic is high, if the client delays payment and notifies, and afterwards lowers the Internet traffic, then the client is forbidden to increase Internet traffic unless she/he pays twice". This complicated English clause is specified in $\mathcal{CL}$-like syntax as: $\Box(\phi \Rightarrow [d\&n \cdot l](F(h)\ \mathcal{U}\ \mathrm{done}_{p\&p}))$.

Here $\mathrm{done}_{p\&p}$ is an assertion added to specify that the client has paid twice. Notice that in order to prove the property we need to extend the NuSMV model of the contract with a propositional constant corresponding to $\mathrm{done}_{p\&p}$ which is true only after a transition labelled $\{p, p\}$ is taken.

In Fig. 3-a we show the control structure of the LTS. The additional state variable $\mathrm{done}_{p\&p}$ is added to the NuSMV model, thus effectively introducing two states for every one in Fig. 3-a, with different values for the state variable.

The original property proves to be false, since from state s_4 (where ϕ holds), after $d\&n \cdot l$, it is possible to increase Internet traffic in state s_7 (due to the *else* label), so neither $F(h)$ nor $\mathrm{done}_{p\&p}$ hold.

Though it was not apparent at first sight, and confirmed by the result given by the tool, the above clause allow the client to go from normal to high Internet traffic many times and pay the penalty ($2 * [price]$) only once. The problem is that after the client lowers the Internet traffic, he might get a high traffic again and postpone the payment till a future moment. This problem comes from the ambiguity of the language. Note that the $\mathcal{CL}$ formalisation in the clauses 3 and 4 use the $\Diamond$ to model the fact that a statement will hold eventually in the future but not necessarily *immediately* (expressions "pay later" in clause 7.3 and "will have to pay" in clause 7.4 are the ambiguities). The *eventually* was translated with the help of the special syntax *else* that we see in Fig. 3-a. We use the counter-example given by NuSMV to construct the model in Fig. 3-b where the property holds. The difference is at the transition from s_7 to s_3 where we have changed the label to the multi-set label $\{p, p\}$. In $\mathcal{CL}$ the solution is to add a new clause corresponding to the property above, and the original contract should be extended with the English version of the property as expressed above. Note that a similar property can be stated for the clause 4 for which we have given the solution in Fig. 3-b also by replacing the label of the transition from s_6 to s_3 by the multi-set label $\{p, p, p\}$.

4 Final Discussion

In this paper we have shown how model checking techniques and tools can be applied to analyse contracts. In particular, we have used NuSMV [4] to model check conventional contracts specified using the language $\mathcal{CL}$. In this paper, we presented multiple uses of model checking for reasoning about contracts. Firstly, we use model checking to increase our confidence in the correctness of the model with respect to the original natural language contract. Secondly, by finding errors in the model, we can identify problems with the original natural language contract or its interpretation into $\mathcal{CL}$. Finally, we enable the signatories to safeguard their interests by ensuring certain desirable properties (and lack of undesirable ones).

About NuSMV: NuSMV [4] is the successor of the milestone symbolic model checker SMV [10]. Symbolic model checking [3] is based on the clever encoding of the states using binary decision diagrams or related techniques, but still relies on the classical model checking algorithm. NuSMV allows the checking of properties specified in CTL, LTL, or PSL. More recently NuSMV has included *input variables* with which it is possible to specify directly a labelled transition system. This feature of NuSMV has been very useful in our context.

Related Work: To our knowledge, model checking contracts is quite an unexplored area where only few works can be found [15,5]. The main difference with our approach is that in [15] there is no language for writing contracts, instead automata are used to model the different participants of a contract, i.e. there is no model of the contract itself but only of the behaviour of the contract signatories. Many safety and liveness properties identified as common to e-contracts are then verified in a purchaser/supplier case study using SPIN [7]. Similarly, in [5] Petri nets are used to model the behaviour of the participants of a contractual protocol. Though in [15] it is claimed that modelling the signatories gives modularity, adding clauses to a given contract implies modifying the automata. In our case, adding clauses to a contract is done as in any declarative language, without changing the rest. Though in our current implementation we would also need to rewrite the verification model, this should not be seen as a disadvantage; given that $\mathcal{CL}$ has formal semantics in $\mathcal{C}\mu$ the model could be obtained automatically after the modifications. An advantage of our approach is the possibility of explicitly writing conditional obligations, permissions and prohibitions, as well as CTDs and CTPs. We are not aware of any other work on model checking e-contracts along the same line as ours. See [13] and [15] (and references therein) for further discussions, and other approaches, on formalisations of contracts.

Future Work: The approach we have followed has few drawbacks. First notice that the way we have obtained the model for the least fix-point in the $\mathcal{C}\mu$ formula 3 in Section 3.1 was modelled as the cycle $(s_7, s_3, s_4, s_5)^*$, which may indeed be an infinite loop as we do not have accepting conditions in our labelled Kripke structure nor fairness constraints. This of course would need to be refined in

order to guarantee that the cycle will eventually finish. Moreover, in order to be able to prove properties about actions which must have been performed, we should extend our language with a constructor $done(\cdot)$ to be applied to actions, meaning that the action argument was performed (as with the $\text{done}_{p\&p}$ in the example). This will definitely facilitate specifying properties like the last one of the previous section concerning the prohibition on actions by the client. We are currently working on improving the above aspects in order to make a more precise analysis.

We have presented a manual translation from the $\mathcal{C}\mu$ semantics of the contract written in $\mathcal{CL}$ into the input language of NuSMV. We plan to implement a tool to automatically model check contracts written in $\mathcal{CL}$. We can benefit from the counter-example generation to fix the original contract, as we have briefly shown in Section 3.4. The underlying model checker of such tool could be NuSMV or another existing μ-calculus model checker (e.g., [1,9]).

With such a tool the whole model checking process will be accelerated facilitating its use and thus making it easy to prove other interesting general properties about e-contracts, as suggested in [15]. Besides such classical liveness or safety properties we are also interested in properties more specific to e-contracts, such as: finding the obligations or prohibitions of one of the parties in the contract; listing of all the rights that follow after the fulfilling of an obligation; what are the penalties for whenever violating an obligation or prohibition; determining whether a given participant is obliged to perform contradictory actions.

The generation of the (automata-like) model that we did by hand in Section 3 can be done automatically along the lines of existing LTL-to-Büchi automata translators (like `ltl2smv` or `ltl2ba`). [14] presents a comprehensive overview of the state-of-the-art of such tools.

In the current state of development, the language $\mathcal{CL}$ cannot explicitly express timing constraints. We intend to extend the language with such features in order to be able to specify and verify real-time properties.

Acknowledgements. We would like to thank Martin Steffen for suggestions on an early draft of this paper.

References

1. Biere, A.: mu-cke - efficient mu-calculus model checking. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 468–471. Springer, Heidelberg (1997)
2. Bradfield, J., Stirling, C.: Modal Logics and Mu-Calculi: an Introduction, pp. 293–330. Elsevier, Amsterdam (2001)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS 1990, pp. 428–439 (1990)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
5. Daskalopulu, A.: Model checking contractual protocols. In: JURIX 2000, Frontiers in Artificial Intelligence and Applications Series, pp. 35–47 (2000)

6. Fischer, M.J., Ladner, R.E.: Propositional modal logic of programs. In: STOC 1977, pp. 286–294 (1977)
7. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
8. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)
9. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. 46, 255–281 (2003)
10. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
11. McNamara, P.: Deontic logic. In: Handbook of the History of Logic, vol. 7, pp. 197–289. North-Holland Publishing, Amsterdam (2006)
12. Pnueli, A.: The temporal logic of programs. In: FOCS 1977, pp. 46–57 (1977)
13. Prisacariu, C., Schneider, G.: A formal language for electronic contracts. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 174–189. IFIP (2007)
14. Rozier, K.Y., Vardi, M.Y.: Ltl satisfiability checking. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 182–200 (2007)
15. Solaiman, E., Molina-Jiménez, C., Shrivastava, S.K.: Model checking correctness properties of electronic contracts. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 303–318. Springer, Heidelberg (2003)

On the Efficient Computation of the Minimal Coverability Set for Petri Nets*

Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin**

Université Libre de Bruxelles (U.L.B.), Computer Science Department
CPI 212, Campus Plaine, Boulevard du Triomphe, B-1050 Bruxelles, Belgium

Abstract. The *minimal coverability set* (MCS) of a Petri net is a finite representation of the downward-closure of its reachable markings. The minimal coverability set allows to decide several important problems like coverability, semi-liveness, place boundedness, etc. The classical algorithm to compute the MCS constructs the Karp&Miller tree [8]. Unfortunately the K&M tree is often huge, even for small nets. An improvement of this K&M algorithm is the Minimal Coverability Tree (MCT) algorithm [1], which has been introduced 15 years ago, and implemented since then in several tools such as Pep [7]. Unfortunately, we show in this paper that the MCT is flawed: it might compute an under-approximation of the reachable markings. We propose a new solution for the efficient computation of the MCS of Petri nets. Our experimental results show that this new algorithm behaves much better in practice than the K&M algorithm.

1 Introduction

Petri nets [10] are a very popular formalism for the modeling and verification of parametric and concurrent systems [6]. The underlying transition graph of a Petri net is potentially infinite. Nevertheless, a large number of interesting verification problems are decidable on Petri nets. Among these decidable problems are the *coverability* problem (to which many safety verification problem can be reduced); the *boundedness* problem (is the number of reachable markings finite ?); the *place boundedness* problem (is the maximal reachable number of tokens bounded for some place p ?); the *semi-liveness* problem (is there a reachable marking in which some transition t is enabled).

In order to decide the aforementioned problems, one can use the *minimal coverability set* (MCS), which is a finite representation of some over-approximation of the reachable markings. The MCS is thus a very useful tool for the analysis of Petri nets, and an efficient algorithm to compute it is highly desirable.

Karp and Miller have shown, in their seminal paper [8], that the minimal coverability set is computable. The main idea of the Karp and Miller (K&M) algorithm is to build a finite tree that summarizes the potentially infinite unfolding of the reachability graph of the Petri net. In particular, this algorithm relies on an acceleration technique, which computes the limit of repeating any number of times some sequences of transitions that

* This research was supported by the Belgian FNRS grant 2.4530.02 of the FRFC project "Centre Fédéré en Vérification" and by the project "MoVES", an Interuniversity Attraction Poles Programme of the Belgian Federal Government.

** Laurent Van Begin is "Chargé de recherche" at FNRS, Belgium.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 98–113, 2007.
© Springer-Verlag Berlin Heidelberg 2007

strictly increase the number of tokens in certain places. The acceleration technique is sound because Petri nets are *strictly monotonic*, i.e. a sequence of transitions which can be fired from a marking $\mathbf{m}$ can be fired from all markings $\mathbf{m}'$ such that $\mathbf{m} \preccurlyeq \mathbf{m}'$ (where $\preccurlyeq$ is a partial order for the markings). Furthermore sequences of transitions have constant effect, i.e. they add and subtract in each place the same number of tokens no matter from which marking they are fired. At the end of the execution of the K&M algorithm, one obtains a *coverability tree*, from which the MCS can be extracted.

Unfortunately, the K&M algorithm is often useless in practice because the finite tree that it builds is often much larger than the minimal coverability set, and cannot be constructed in reasonable time. As a consequence, a more efficient algorithm is needed. In [1], such an algorithm is proposed. The minimal coverability tree (MCT) builds on the idea of K&M but tries to take advantage more aggressively of the strict monotonicity of Petri nets. The main idea is to construct a tree where all markings that label nodes are incomparable wrt $\preccurlyeq$. To achieve this goal, reduction rules are applied at each step of the algorithm: each time a new marking is computed, it is compared to the other markings. If the new marking is smaller than an existing marking, the construction is not pursued from this new marking. If the new marking is larger than an existing marking, the subtree starting from that smaller marking is removed. The informal justification for this is as follows: the markings that are reachable from removed markings will be covered by markings reachable from the marking that was used for the removal, by the monotonicity property of Petri nets. While this idea is appealing and leads to small trees in practice, we show in this paper that, unfortunately, it is not correct: the MCT algorithm is not complete and can compute a strict under-approximation of the minimal coverability set. The flaw is intricate and we do not see an easy way to get rid of it.

So, instead of trying to fix the MCT algorithm, we consider the problem from scratch and propose a new efficient method to compute the MCS. It is based on novel ideas: first, we do not build a tree but handle sets of pairs of markings. Second, in order to exploit monotonicity property, we define an adequate order on pairs of markings that allows us to maintain sets of maximal pairs only. We give in this paper a detailed proof of correctness for this new method, and explain how to turn it into an efficient algorithm for the computation of the MCS of practically relevant Petri nets. We have implemented our algorithm in a prototype and compared its performance with the K&M algorithm. Our algorithm is orders of magnitude faster than the K&M algorithm.

The rest of the paper is organized as follows. In Section 2, we recall necessary preliminaries. In Section 3, we recall the KM as well as the MCT algorithms. In Section 4, we expose the bug in the MCT algorithm using an example and explain the essence of the flaw. In Section 5, we define the covering sequence, a sequence of sets of pairs of ω-markings that allows to compute the MCS. In Section 6, we discuss a prototype implementation of this new method. Due to the lack of space, we refer the reader to [5] (available on the web at: `http://www.ulb.ac.be/di/ssd/cfv/`) for the missing proofs.

2 Preliminaries

Petri nets. Let us first recall the model of Petri nets, and fix several notations.

Definition 1. *A* Petri net *[10] (*PN *for short) is a tuple $\mathcal{N} = \langle P, T\rangle$, where $P = \{p_1, p_2, \ldots, p_{|P|}\}$ is a finite set of places and $T = \{t_1, t_2, \ldots, t_{|T|}\}$ is a finite set of transitions. Each transition is a tuple $\langle I, O\rangle$, where $I : P \mapsto \mathbb{N}$ and $O : P \mapsto \mathbb{N}$ are respectively the input and output functions of the transition.*

To define the semantics of PN, we first introduce the notion of ω-marking. An ω-*marking* $\mathbf{m}$ is a function $\mathbf{m} : P \mapsto (\mathbb{N} \cup \{\omega\})$ that associates a number of *tokens* to each place ($\mathbb{N}$ is the set of natural numbers including 0 and ω means 'any natural number'). An ω-marking $\mathbf{m}$ is denoted either as $\langle \mathbf{m}(p_1), \mathbf{m}(p_2), \ldots, \mathbf{m}(p_{|P|})\rangle$ (vector) , or as $\{\mathbf{m}(p_{i_1})p_{i_1}, \mathbf{m}(p_{i_2})p_{i_2}, \ldots, \mathbf{m}(p_{i})p_{i}\}$ (multiset), where $p_{i_1}, p_{i_2}, \ldots, p_{i}$ are exactly the places that contain at least one token (we omit $\mathbf{m}(p)$ when it is equal to 1). For example, $\langle 0, 1, 0, \omega, 2\rangle$ and $\{p_2, \omega p_4, 2p_5\}$ denote the same ω-marking. An ω-marking $\mathbf{m}$ is a *marking* iff $\forall p \in P : \mathbf{m}(p) \neq \omega$.

Let $\mathcal{N} = \langle P, T\rangle$ be a PN, $\mathbf{m}$ be an ω-marking of $\mathcal{N}$ and $t = \langle I, O\rangle \in T$ be a transition. Then, t is *enabled* in $\mathbf{m}$ iff $\mathbf{m}(p) \geq I(p)$ for any $p \in P$ (we assume that $\omega \geq \omega$ and $\omega > c$ for any $c \in \mathbb{N}$). In that case, t can *fire* and transforms $\mathbf{m}$ into a new ω-marking $\mathbf{m}'$ s.t. for any $p \in P$: $\mathbf{m}'(p) = \mathbf{m}(p) - I(p) + O(p)$ (assuming that $\omega - c = \omega = \omega + c$ for any $c \in \mathbb{N}$). We denote this by $\mathbf{m} \xrightarrow{t} \mathbf{m}'$, and extend the notation to sequences of transitions $\sigma = t_1 t_2 \cdots t_k \in T^*$, i.e., $\mathbf{m} \xrightarrow{\sigma} \mathbf{m}'$ iff either $\sigma = \varepsilon$ (the empty sequence) and $\mathbf{m} = \mathbf{m}'$, or there are $\mathbf{m}_1, \ldots, \mathbf{m}_{k-1}$ s.t. $\mathbf{m} \xrightarrow{t_1} \mathbf{m}_1 \xrightarrow{t_2} \cdots \xrightarrow{t} \mathbf{m}'$. Given an ω-marking $\mathbf{m}$ of some PN $\mathcal{N} = \langle P, T\rangle$, we let $\mathsf{Post}(\mathbf{m}) = \{\mathbf{m}' \mid \exists t \in T : \mathbf{m} \xrightarrow{t} \mathbf{m}'\}$ and $\mathsf{Post}^*(\mathbf{m}) = \{\mathbf{m}' \mid \exists \sigma \in T^* : \mathbf{m} \xrightarrow{\sigma} \mathbf{m}'\}$. Given a sequence of transitions $\sigma = t_1 t_2 \cdots t_k$ with $t_i = \langle I_i, O_i\rangle$ for any $1 \leq i \leq k$, we let, for any place p, $\sigma(p) = \sum_{i=1}^{k}(I_i(p) - O_i(p))$, i.e., the effect of σ on p.

In the following, we use the partial order $\preccurlyeq$ for ω-markings.

Definition 2. *Let P be a set of places of some* PN*. Then, $\preccurlyeq \subseteq (\mathbb{N} \cup \{\omega\})^{|P|} \times (\mathbb{N} \cup \{\omega\})^{|P|}$ is s.t. for any $\mathbf{m}_1, \mathbf{m}_2$: $\mathbf{m}_1 \preccurlyeq \mathbf{m}_2$ iff for any $p \in P$: $\mathbf{m}_1(p) \leq \mathbf{m}_2(p)$.*

We write $\mathbf{m} \prec \mathbf{m}'$ when $\mathbf{m} \preccurlyeq \mathbf{m}'$ but $\mathbf{m} \neq \mathbf{m}'$. Finally, it is well-known that PN are *strictly monotonic*. That is, if $\mathbf{m}_1$, $\mathbf{m}_2$ and $\mathbf{m}_3$ are three markings and t is a transition of some PN $\mathcal{N}$ s.t. $\mathbf{m}_1 \xrightarrow{t} \mathbf{m}_2$ and $\mathbf{m}_1 \prec \mathbf{m}_3$, then, t is enabled in $\mathbf{m}_3$ and the marking $\mathbf{m}_4$ with $\mathbf{m}_3 \xrightarrow{t} \mathbf{m}_4$ is s.t. $\mathbf{m}_2 \prec \mathbf{m}_4$.

Covering and coverability sets. Given a set M of ω-markings, we define the set of maximal elements of M as $\mathsf{Max}^{\preccurlyeq}(M) = \{\mathbf{m} \in M \mid \nexists \mathbf{m}' \in M : \mathbf{m} \prec \mathbf{m}'\}$. Given an ω-marking $\mathbf{m}$ (ranging over set of places P), its *downward-closure* is the set *of markings* $\downarrow^{\preccurlyeq}(\mathbf{m}) = \{\mathbf{m}' \in \mathbb{N}^{|P|} \mid \mathbf{m}' \preccurlyeq \mathbf{m}\}$. Given a set M of ω-markings, we let $\downarrow^{\preccurlyeq}(M) = \cup_{\mathbf{m} \in M} \downarrow^{\preccurlyeq}(\mathbf{m})$. A set $D \subseteq \mathbb{N}^{|P|}$ is *downward-closed* iff $\downarrow^{\preccurlyeq}(D) = D$. Then:

Definition 3. *Let $\mathcal{N} = \langle P, T\rangle$ be a* PN *and let $\mathbf{m}_0$ be the initial ω-marking of $\mathcal{N}$. The* covering set *of $\mathcal{N}$, denoted as* $\mathsf{Cover}(\mathcal{N}, \mathbf{m}_0)$ *is the set* $\downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_0))$.

Given a PN $\mathcal{N}$ with initial marking $\mathbf{m}_0$, a *coverability set* for $\mathcal{N}$ and $\mathbf{m}_0$ is a finite sub-set $S \subseteq (\mathbb{N} \cup \{\omega\})^{|P|}$ such that $\downarrow^{\preccurlyeq}(S) = \mathsf{Cover}(\mathcal{N}, \mathbf{m}_0)$. Such a set always exists because any down.-cl. set of markings can be represented by a finite set of ω-markings[1]:

Lemma 1 ([12]). *For any subset $D \subseteq \mathbb{N}^k$ such that $\downarrow^{\preccurlyeq}(D) = D$ there exists a finite subset $S \subset (\mathbb{N} \cup \{\omega\})^k$ such that $\downarrow^{\preccurlyeq}(S) = D$.*

It is also well-known [1] that there exists one minimal (in terms of $\subseteq$) coverability set (called the *minimal coverability set*).

Labeled trees. Finally, let us introduce the notion of *labeled tree*:

Definition 4. *Given a set of places P, a labeled tree is a tuple $\mathcal{T} = \langle N, B, root, \Lambda \rangle$, s.t. $\langle N, B, root \rangle$ forms a tree (N is the set of nodes, $B \subseteq N \times N$ is the set of edges and $root \in N$ is the root node) and $\Lambda : N \mapsto (\mathbb{N} \cup \{\omega\})^{|P|}$ is a labeling function of the nodes by ω-markings.*

Given two nodes n and n' in N, we write respectively $B(n, n')$, $B^*(n, n')$ $B^+(n, n')$ instead of $(n, n') \in B$, $(n, n') \in B^*$, $(n, n') \in B^+$.

3 The Karp Miller and the MCT Algorithms

The Karp and Miller algorithm. The Karp&Miller algorithm [8] is a well-known solution to compute a coverability set of a PN. It consists in building a labeled tree whose root is labeled by $\mathbf{m}_0$. The tree is obtained by unfolding the transition relation of the PN, and by applying *accelerations*, which exploit the strict monotonicity property of PN. That is, let us assume that $\mathbf{m}_1$ and $\mathbf{m}_2$ are two ω-markings s.t. $\mathbf{m}_1 \prec \mathbf{m}_2$ and there exists a sequence of transitions σ with $\mathbf{m}_1 \xrightarrow{\sigma} \mathbf{m}_2$. By (strict) monotonicity, σ is firable from $\mathbf{m}_2$ and produces a ω-marking $\mathbf{m}_3$ s.t. $\mathbf{m}_2 \prec \mathbf{m}_3$. As a consequence, all the places p s.t. $\mathbf{m}_1(p) < \mathbf{m}_2(p)$ are unbounded. Hence, the ω-marking $\mathbf{m}_\omega$ defined as $\mathbf{m}_\omega(p) = \omega$ if $\mathbf{m}_1(p) < \mathbf{m}_2(p)$, and $\mathbf{m}_\omega(p) = \mathbf{m}_1(p)$ otherwise, has the property that $\downarrow^{\preccurlyeq}(\mathbf{m}_\omega) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_1))$. This can be generalized to the case where we consider an ω-marking $\mathbf{m}$ and a set S of ω-markings s.t. for any $\mathbf{m}' \in S$: $\mathbf{m} \in \mathsf{Post}^*(\mathbf{m}')$. Hence, the following acceleration function:

$$\forall p \in P : \mathsf{Accel}(S, \mathbf{m})(p) = \begin{cases} \omega & \text{if } \exists \mathbf{m}' \in S : \mathbf{m}' \prec \mathbf{m} \text{ and } \mathbf{m}'(p) < \mathbf{m}(p) \\ \mathbf{m}(p) & \text{Otherwise} \end{cases}$$

The Karp&Miller procedure (see Algorithm 1) relies on this function: when developing the successors of a node n, it calls the acceleration function on every $\mathbf{m} \in \mathsf{Post}(\Lambda(n))$, by letting S be the set of all the markings that are met along the branch ending in n. This procedure terminates and computes a coverability set:

Theorem 1 ([8]). *For any PN $\mathcal{N} = \langle P, T \rangle$ with initial ω-marking $\mathbf{m}_0$, the KM procedure produces a finite labeled tree $\mathcal{T} = \langle N, B, root, \Lambda \rangle$, s.t. $\downarrow^{\preccurlyeq}(\{\Lambda(n) \,|\, n \in N\}) = \mathsf{Cover}(\mathcal{N}, \mathbf{m}_0)$.*

[1] Since a given set of ω-markings represents one and only one downward-closed set, we sometimes confuse such a set of ω-marking with the downward-closed set of markings it represents.

Algorithm 1. The KM algorithm

```
Data: A PN 𝒩 = ⟨P, T⟩ and an initial ω-marking m_0.
Result: The minimal coverability set of 𝒩 for m_0.
KM (𝒩, m_0) begin
    𝒯 ← ⟨N, B, n_0, Λ⟩ where N = {n_0}, B = ∅ and Λ(n_0) = m_0 ;
    to_treat ← {n_0} ;
    while to_treat ≠ ∅ do
        Choose and remove a node n from to_treat ;
        if ∄n̄ : B⁺(n̄, n) ∧ Λ(n̄) = Λ(n) then
            foreach m ∈ Post(Λ(n)) do
                S ← {Λ(n') | B*(n', n)} ;
                Let n' be a new node s.t. Λ(n') = Accel(S, m) ;
                N ← N ∪ {n'} ; B ← B ∪ {(n, n')} ;
                to_treat ← to_treat ∪ {n'} ;
    return({Λ(n) | n ∈ N ∧ ∄n' ∈ N : Λ(n') ≻ Λ(n)}) ;
end
```

Properties of the Karp&Miller tree. Let $n \neq root$ be a node of some Karp&Miller tree. Hence, $\Lambda(n)$ has been obtained by calling Accelerate with parameters S and $\mathbf{m}$. In this case, we say that n has been obtained *by the acceleration of* $\mathbf{m}$ (with S). For any node $n \neq root$ of any Karp&Miller tree, we assume that $\mathsf{M}(n)$ is the marking $\mathbf{m}$ s.t. $\Lambda(n)$ has been obtained by the acceleration of $\mathbf{m}$. Remark that $\forall n \neq root$, $\mathsf{M}(n) \in \mathsf{Post}(\Lambda(n'))$ where n' is the father of n and it might be the case that $\Lambda(n) = \mathsf{M}(n)$.

Let $\mathcal{N} = \langle P, T\rangle$ be a PN with initial marking $\mathbf{m}_0$ and let $\mathcal{T} = \langle N, B, root, \Lambda\rangle$ be its Karp&Miller tree. Then, $\varsigma : N \mapsto T^*$ is a function that associates a sequence of transitions to every node n, as follows. (i) If $n = root$, then $\varsigma(n)$ returns the empty sequence. (ii) If there is no $n' \in N$ s.t. $B^+(n', n)$, $\Lambda(n') \neq \Lambda(n)$ and $\Lambda(n') \preccurlyeq \Lambda(n)$ (hence, n is such that $\Lambda(n) = \mathsf{M}(n)$), then $\varsigma(n)$ returns the empty sequence. (iii) Otherwise, n has been obtained by the acceleration of $\mathsf{M}(n)$. Let $P_a = \{p \in P \mid \Lambda(n)(p) = \omega$ and $\mathsf{M}(n)(p) \neq \omega\}$ and let $P_\omega = \{p \in P \mid \Lambda(n)(p) = \mathsf{M}(n)(p) = \omega\}$. In that case, $\varsigma(n)$ returns one of the finite non-empty sequences s.t. for any $p \in P_a$: $\varsigma(n)(p) > 0$; for any $p \in P \setminus (P_a \cup P_\omega)$: $\varsigma(n)(p) = 0$; and $\varsigma(n)$ is firable from $\mathsf{M}(n)$.

The existence of $\varsigma(n)$ in the third case is guaranteed by the following lemma, that can be extracted from the main proof of the Algorithm 1, in [8]:

Lemma 2 ([8]). *Let $\mathcal{N} = \langle P, T\rangle$ be a* PN *with initial ω-marking $\mathbf{m}_0$ and let $\mathcal{T} = \langle N, B, root, \Lambda\rangle$ be its Karp&Miller tree. Let $n \neq root$ be a node of $\mathcal{T}$. Let $P_a = \{p \in P \mid \Lambda(n)(p) = \omega$ and $\mathsf{M}(n)(p) \neq \omega\}$ and $P_\omega = \{p \in P \mid \Lambda(n)(p) = \mathsf{M}(n)(p) = \omega\}$. Then, there exists a sequence of transitions $\sigma \in T^*$ s.t.: (i) for any $p \in P_a$: $\sigma(p) > 0$. (ii) for any $p \in P \setminus (P_a \cup P_\omega)$: $\sigma(p) = 0$. (iii) σ is firable from $\mathsf{M}(n)$.*

The MCT algorithm. The *minimal coverability tree algorithm* (MCT for short) has been introduced by Finkel in [1], as an optimization of the Karp&Miller algorithm. It is recalled in Algorithm 2, and relies on two auxiliary functions: given a labeled tree $\mathcal{T}$ and a node n of $\mathcal{T}$, removeSubtree$(n, \mathcal{T})$ removes the subtree rooted by n from $\mathcal{T}$. The function removeSubtreeExceptRoot$(n, \mathcal{T})$ is similar to removeSubtree$(n, \mathcal{T})$

Algorithm 2. The MCT algorithm [1]

Data: A PN $\mathcal{N} = \langle P, T \rangle$ and an initial marking $\mathbf{m}_0$
Result: The minimal coverability set of $\mathcal{N}$.
MCT $(\mathcal{N}, \mathbf{m}_0)$ **begin**
 $\mathcal{T} \leftarrow \langle N, B, n_0, \Lambda \rangle$ where $N = \{n_0\}$, $B = \emptyset$ and $\Lambda(n_0) = \mathbf{m}_0$;
 $to_treat \leftarrow \{n_0\}$;
 while $to_treat \neq \emptyset$ **do**
  Choose and remove a node n from to_treat ;
(a) **if** $\nexists \overline{n} \in N$ *s.t.* $\Lambda(\overline{n}) = \Lambda(n)$ **then**
   foreach $\mathbf{m} \in \mathsf{Post}(\Lambda(n))$ **do**
(b)   **if** $\exists \overline{n} : B^*(\overline{n}, n)$ *and* $\Lambda(\overline{n}) \prec \mathbf{m}$ **then**
     Let $\overline{n}$ be the highest node s.t. $B^*(\overline{n}, n) \wedge \Lambda(\overline{n}) \prec \mathbf{m}$;
     $\Lambda(\overline{n}) \leftarrow \mathsf{Accel}(\{n' \in N \mid B^*(n', n)\}, \mathbf{m})$;
     $to_treat \leftarrow \left(to_treat \setminus \{n' \mid B^*(\overline{n}, n')\}\right) \cup \{\overline{n}\}$;
     removeSubtreeExceptRoot$(\overline{n}, \mathcal{T})$;
     break ;
(c)   **else if** $\nexists \overline{n} \in N$ *s.t.* $\mathbf{m} \prec \Lambda(\overline{n})$ **then**
     Let n' be a new node s.t. $\Lambda(n') = \mathbf{m}$;
     $N \leftarrow N \cup \{n'\}$; $B \leftarrow B \cup (n, n')$; $to_treat \leftarrow to_treat \cup \{n'\}$;
(d)   **while** $\exists n_1, n_2 \in N : \Lambda(n_1) \prec \Lambda(n_2)$ **do**
    $to_treat \leftarrow to_treat \setminus \{n \mid B^*(n_1, n)\}$;
    removeSubtree$(n_1, \mathcal{T})$;
 `return`$(\{\Lambda(n) \mid n \in N\})$;
end

except that the root node n is not removed. The main idea consists in exploiting the monotonicity property of PN in order to avoid developing part of the nodes of the Karp&Miller tree, as well as removing some subtrees during the construction. With respect to the Karp&Miller algorithm, three main differences can be noted. Let n be a node picked from to_treat. First, when there already exists another node $\overline{n}$ with $\Lambda(n) = \Lambda(\overline{n})$ in the tree, n, is not developed (line (a)). Second, when n is accelerated (line (b)), the result of the acceleration is assigned to the label of its highest ancestor $\overline{n}$ s.t. $\Lambda(\overline{n}) \prec \Lambda(n)$, and the whole subtree of $\overline{n}$ is removed from the tree. Third, the algorithm avoids adding a node n' to the tree if there is another node $\overline{n}$ s.t. $\Lambda(\overline{n}) \succcurlyeq \Lambda(n')$ (line (c)). Moreover, the adjunction of n' to the tree (when it happens) triggers the deletion of all the subtrees rooted in some node n'' s.t. $\Lambda(n'') \prec \Lambda(n')$ (line (d)).

Remark that this algorithm is *non-deterministic*, in the sense that no ordering is imposed on the nodes in to_treat. Hence, any strategy that fixes the exploration order (which can possibly improve the efficiency of the algorithm) can be chosen.

4 Counter-Example to the MCT Algorithm

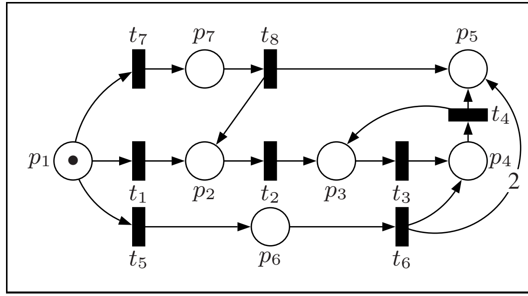
In this section, we introduce a PN on which the MCT algorithm might compute a *strict under-approximation* of the covering set (see Fig. 1). Fig 1(a) is the PN on which we run the MCT algorithm, and Fig. 1(b) through 1(f) are the key points of the execution.

Let us briefly comment on this execution. First remark that place p_5 of the PN in Fig. 1(a) is unbounded, because marking $\{p_3\}$ is reachable from the initial marking $\mathbf{m}_0 = \{p_1\}$ by firing t_1t_2, and the sequence t_3t_4 can be fired an arbitrary number of times from $\{p_3\}$, and strictly increases the markings of p_5. Then, one possible execution of MCT is as follows (markings in the frontier are underlined): **Fig. 1(b)**: The three successors of $\mathbf{m}_0$ are computed. Then, the branch rooted in $\{p_2\}$ is unfolded, by firing t_2, t_3 and t_4. At that point, two comparable markings $\{p_3\}$ and $\{p_3, p_5\}$ are met with and an acceleration occurs (line (b) of Algorithm 2) . The result is $\{p_3, \omega p_5\}$, which is put into to_treat. **Fig. 1(c)**: The subtree rooted in $\{p_6\}$ is unfolded. After the firing of t_6t_4, one obtains $\{p_3, 3p_5\}$, which is smaller than $\{p_3, \omega p_5\}$. Hence, $\{p_3, 3p_5\}$ is not put into to_treat and the branch is stopped (line (c)). **Fig. 1(d)**: The subtree rooted in $\{p_7\}$ is developed. The unique successor $\{p_2, p_5\}$ of $\{p_7\}$ is larger than $\{p_2\}$. Hence, the subtree rooted in $\{p_2\}$ (including $\{p_3, \omega p_5\}$, still in the frontier) is removed (line (d)). **Fig. 1(e) and 1(f)**: The tree (actually a single branch) rooted in $\{p_2, p_5\}$ (only node in the frontier) is further developed through the firing of t_2 and t_3. The resulting node $\{p_4, p_5\}$ is strictly smaller than $\{p_4, 2p_5\}$. Hence, that branch is stopped too (line (c)), and the frontier becomes empty. The final result of the algorithm is shown in Fig. 1(f). It is not difficult to see that the set of labels of this tree does not form a coverability set, because it contains no marking $\mathbf{m}$ s.t. $\mathbf{m}(p_5) = \omega$.

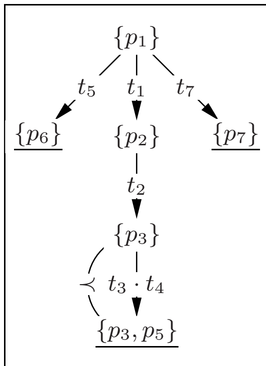
Comment on the counter-example. This counter-example allows us to identify a flaw in the logic of the MCT algorithm. The algorithm stops the development of a node n or removes the subtree rooted in n because its has found another node n' s.t. $\Lambda(n')$ is larger than $\Lambda(n)$. In that case, we say that n' is *a proof for* n. The intuition behind this notion is that, by monotonicity, all the successors of n should be covered by some successor of n'. Thus, when n' is a proof for n, the algorithm makes implicitly the hypothesis that either all the successors of n' will be fully developed, or that they will be covered by some other nodes of the tree.

In our counter-example, that reasoning fails because *cycles* appear in 'proofs'. In Fig. 1, we have drawn a thick gray arrow from n to n' when n' is a proof for n. On Fig. 1(d), the node labeled by $\{p_3, \omega p_5\}$, which is a proof for $\{p_3, 3p_5\}$ is deleted, because of $\{p_2, p_5\}$. Hence, $\{p_2, p_5\}$, becomes the proof of $\{p_3, 3p_5\}$ (see Fig. 1(e)). The cycle clearly appears in Fig. 1(f): all the successors of $\{p_4, 2p_5\}$ will be eventually covered under the assumption that all the successors of $\{p_2, p_5\}$ are covered. However, this happens only if all the successors of $\{p_4, 2p_5\}$ are eventually covered.

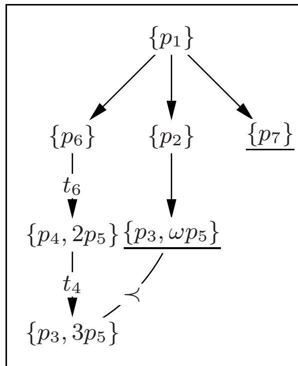
Implementation of the MCT in the Pep tool. Actually, the flaw in the MCT algorithm has already been independently discovered by the team of Prof. Peter Starke. They have implemented in INA (a component of the toolkit Pep [7]) a variation of the MCT which is supposed to correct the aforementioned bug. To the best of our knowledge, this implementation (and the discovery of the bug) has been documented only in a master's thesis in German [9]. Unfortunately, their version of the MCT contains a flaw too, because it offers no guarantee of termination [11,3], although [9] contains a proof of termination. See [2] for a counter-example to termination. Thus, from our point of view, fixing the bug of the MCT algorithm seems to be a difficult task.



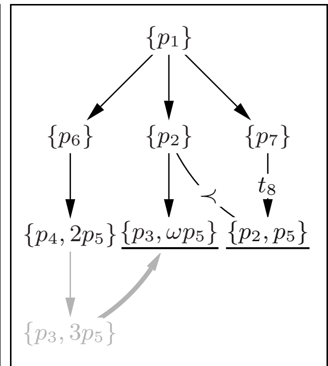
(a) The PN.



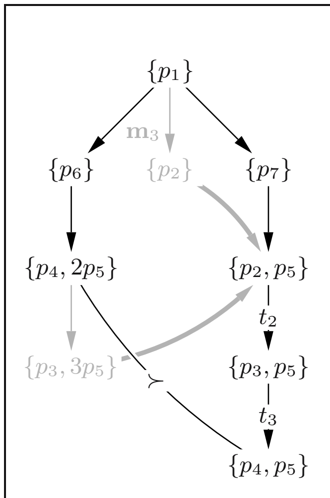
(b) Step 1.



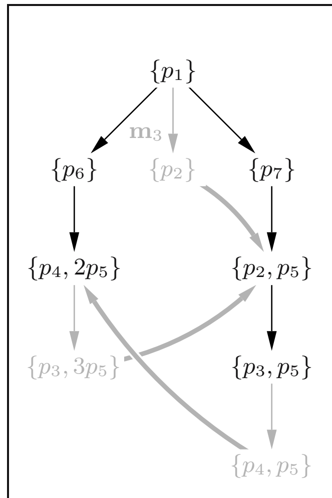
(c) Step 2.



(d) Step 3.



(e) Step 4.



(f) The result.

Fig. 1. A counter-example to the MCT algorithm. Underlined markings are in the frontier. A gray arrow from n to n' means that n' is a 'proof' for n.

5 The Covering Sequence

Instead of trying to fix the bug in the MCT algorithm, we propose a different solution based on novel ideas. To introduce our new solution, let us look back at the basics. Remember that we want to compute an effective representation of $\downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_0))$. It is easy to show that this set is the limit of the following infinite sequence of sets: $X_0 = \downarrow^{\preccurlyeq}(\{\mathbf{m}_0\})$, and for $i \geq 1$, $X_i = \downarrow^{\preccurlyeq}(X_{i-1} \cup \mathsf{Post}(X_{i-1}))$. Note that by strict monotonicity of Petri nets, we can instead consider the following sequence that handles maximal elements only: $Y_0 = \mathsf{Max}^{\preccurlyeq}(\{\mathbf{m}_0\})$, and $Y_i = \mathsf{Max}^{\preccurlyeq}(Y_{i-1} \cup \mathsf{Post}(Y_{i-1}))$ for any $i \geq 1$. Unfortunately, this is not an effective way to compute the minimal coverability set as we do not know how to compute the limit of this sequence. To compute that limit, we need accelerations. Accelerations are computed from pairs of markings. Our solution constructs a sequence of sets of pairs of markings on which we systematically apply a variant of the Post operator and a variant of the acceleration function. By defining an adequate order $\sqsubseteq$ on pairs of markings, we show that we can concentrate on maximal elements for $\sqsubseteq$.

Preliminaries. Let $\mathbf{m}_1$ and $\mathbf{m}_2$ be two ω-markings. Then, $\mathbf{m}_1 \ominus \mathbf{m}_2$ is a function $P \mapsto \mathbb{Z} \cup \{-\omega, \omega\}$ s.t. for any place p: $(\mathbf{m}_1 \ominus \mathbf{m}_2)(p)$ is equal to ω if $\mathbf{m}_1(p) = \omega$; $-\omega$ if $\mathbf{m}_2(p) = \omega$ and $\mathbf{m}_1(p) \neq \omega$; $\mathbf{m}_1(p) - \mathbf{m}_2(p)$ otherwise. Then, given two pairs of ω-markings $(\mathbf{m}_1, \mathbf{m}_2)$ and $(\mathbf{m}'_1, \mathbf{m}'_2)$, we have $(\mathbf{m}_1, \mathbf{m}_2) \sqsubseteq (\mathbf{m}'_1, \mathbf{m}'_2)$ iff $\mathbf{m}_1 \preccurlyeq \mathbf{m}'_1$, $\mathbf{m}_2 \preccurlyeq \mathbf{m}'_2$ and for any place p: $(\mathbf{m}_2 \ominus \mathbf{m}_1)(p) \leq (\mathbf{m}'_2 \ominus \mathbf{m}'_1)(p)$.

For any $(\mathbf{m}_1, \mathbf{m}_2)$, we let $\downarrow^{\sqsubseteq}((\mathbf{m}_1, \mathbf{m}_2)) = \{(\mathbf{m}'_1, \mathbf{m}'_2) \mid (\mathbf{m}'_1, \mathbf{m}'_2) \sqsubseteq (\mathbf{m}_1, \mathbf{m}_2)\}$. We extend this to sets of pairs R as follows: $\downarrow^{\sqsubseteq}(R) = \cup_{(\mathbf{m}_1, \mathbf{m}_2) \in R} \downarrow^{\sqsubseteq}((\mathbf{m}_1, \mathbf{m}_2))$. Given a set R of pairs of markings, we let $\mathsf{Max}^{\sqsubseteq}(R) = \{(\mathbf{m}_1, \mathbf{m}_2) \in R \mid \nexists (\mathbf{m}'_1, \mathbf{m}'_2) \in R : (\mathbf{m}_1, \mathbf{m}_2) \neq (\mathbf{m}'_1, \mathbf{m}'_2) \wedge (\mathbf{m}_1, \mathbf{m}_2) \sqsubseteq (\mathbf{m}'_1, \mathbf{m}'_2)\}$

Our new solution relies on a weaker acceleration function than that of Karp&Miller (because its first argument is restricted to a single marking instead of a set of markings). Given two ω-markings $\mathbf{m}_1$ and $\mathbf{m}_2$ s.t. $\mathbf{m}_1 \preccurlyeq \mathbf{m}_2$, we let $\mathsf{AccelPair}(\mathbf{m}_1, \mathbf{m}_2) = \mathbf{m}_\omega$ s.t. for any place p, $\mathbf{m}_\omega(p) = \mathbf{m}_1(p)$ if $\mathbf{m}_1(p) = \mathbf{m}_2(p)$; $\mathbf{m}_\omega(p) = \omega$ otherwise. According to the following lemma, this acceleration function is sound:

Lemma 3. *Let $\mathcal{N}$ be a* PN *and let $\mathbf{m}_1$ and $\mathbf{m}_2$ be two ω-markings of $\mathcal{N}$ that respect $\mathbf{m}_1 \preccurlyeq \mathbf{m}_2$ and $\downarrow^{\preccurlyeq}(\mathbf{m}_2) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_1))$. Then, $\downarrow^{\preccurlyeq}(\mathsf{AccelPair}(\mathbf{m}_1, \mathbf{m}_2)) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_2))$.*

Moreover, given a labeled tree $\mathcal{T} = \langle N, B, root, \Lambda \rangle$, we let, for any $n \in N$, $\mathsf{Anc}(\mathcal{T}, n) = \{n' \mid B^*(n', n)\}$ (that is, $\mathsf{Anc}(\mathcal{T}, n)$ is the set of 'ancestors' of n in $\mathcal{T}$, n included). Then, the following lemma draws a link between $\mathsf{AccelPair}$ and the Karp&Miller acceleration. It shows that, although $\mathsf{AccelPair}$ is weaker, it can produce the same results than the Karp&Miller acceleration, when properly applied.

Lemma 4. *Let $\mathcal{N} = \langle P, T \rangle$ be a Petri net with initial marking $\mathbf{m}_0$ and let $\mathcal{T} = \langle N, B, root, \Lambda \rangle$ be its Karp&Miller tree. Let $n \neq root$ be a node of $\mathcal{T}$. Let $\mathbf{m}'$ be s.t. $\mathsf{M}(n) \xrightarrow{\varsigma(n)} \mathbf{m}'$. Then, $\Lambda(n) \preccurlyeq \mathsf{AccelPair}(\mathsf{M}(n), \mathbf{m}')$.*

Proof. Let $P_a = \{p \in P \mid \Lambda(n)(p) = \omega \wedge \mathsf{M}(n)(p) \neq \omega\}$. By construction, for any place p, $\Lambda(n)(p) = \omega$ if $p \in P_a$; $\Lambda(n)(p) = \mathsf{M}(n)$ otherwise. Moreover, by definition

of AccelPair, for any place p, $\mathsf{AccelPair}(\mathsf{M}(n), \mathbf{m}')(p) = \omega$ if $\varsigma(n)(p) > 0$; and $\mathsf{AccelPair}(\mathsf{M}(n), \mathbf{m}')(p) = \mathsf{M}(n)(p)$ otherwise. By def. of $\varsigma(n)$, $p \in P_a$ implies $\varsigma(n)(p) > 0$, and $p \notin P_a$ implies $\Lambda(n)(p) = \mathsf{M}(n)(p)$. □

Finally, we introduce several operators that work directly on pairs of ω-markings. Given a set R of pairs of ω-markings, we let $\mathsf{Flatten}(R) = \{\mathbf{m} \mid \exists \mathbf{m}' : (\mathbf{m}', \mathbf{m}) \in R\}$. Given a pair of ω-markings $(\mathbf{m}_1, \mathbf{m}_2)$, we let $\overline{\mathsf{Post}}((\mathbf{m}_1, \mathbf{m}_2)) = \{(\mathbf{m}_1, \mathbf{m}'), (\mathbf{m}_2, \mathbf{m}') \mid \mathbf{m}' \in \mathsf{Post}(\mathbf{m}_2)\}$ and $\overline{\mathsf{Accel}}((\mathbf{m}_1, \mathbf{m}_2)) = \{(\mathbf{m}_2, \mathsf{AccelPair}(\mathbf{m}_1, \mathbf{m}_2))\}$ if $\mathbf{m}_1 \prec \mathbf{m}_2$; and $\overline{\mathsf{Accel}}((\mathbf{m}_1, \mathbf{m}_2))$ is undefined otherwise. We extend these two functions to sets R of pairs in the following way: $\overline{\mathsf{Post}}(R) = \cup_{(\mathbf{m}_1, \mathbf{m}_2) \in R} \overline{\mathsf{Post}}((\mathbf{m}_1, \mathbf{m}_2))$ and $\overline{\mathsf{Accel}}(R) = \cup_{(\mathbf{m}_1, \mathbf{m}_2) \in R, \mathbf{m}_1 \prec \mathbf{m}_2} \{\overline{\mathsf{Accel}}(\mathbf{m}_1, \mathbf{m}_2)\}$.

Definition of the sequence. We are now ready to introduce the covering sequence. We will define the sequence in a way that allows for optimizations. To incorporate those optimizations elegantly, we allow our construction to be helped by an oracle which is a procedure that produces pairs of markings. This oracle potentially allows for the early convergence of the covering sequence. However, we will prove that our sequence converges even if the oracle is trivial (returns always an empty set of pairs of markings). In the next section, we will show that the oracle can be implemented by a recursive call to the covering sequence, by considering ω-markings where the number of ω is increasing as initial ω-markings in the recursive call. This will lead to an efficient procedure as we will see in the next section.

In the following, given a Petri net $\mathcal{N}$ and an initial ω-marking $\mathbf{m}_0$, we call an *oracle* any function $\mathsf{Oracle} : \mathbb{N} \mapsto (\mathbb{N} \cup \{\omega\})^{|P|} \times (\mathbb{N} \cup \{\omega\})^{|P|}$ that returns, for any $i \geq 0$, a set of pairs of ω-markings s.t.

$$\downarrow^{\preccurlyeq}(\mathsf{Post}(\mathsf{Flatten}(\mathsf{Oracle}(i)))) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}(\mathsf{Oracle}(i))) \quad (1)$$

and

$$\downarrow^{\preccurlyeq}(\mathsf{Flatten}(\mathsf{Oracle}(i))) \subseteq \mathsf{Cover}(\mathcal{N}). \quad (2)$$

Let $\mathcal{N} = \langle P, T \rangle$ be a PN, $\mathbf{m}_0$ be an initial marking, and Oracle be an oracle. Then, the covering sequence of $\mathcal{N}$, noted $\mathsf{CovSeq}(\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle})$ is the infinite sequence $(V_i, F_i, O_i)_{i \geq 0}$, defined as follows:

- $V_0 = \emptyset$, $O_0 = \emptyset$ and $F_0 = \{(\mathbf{m}_0, \mathbf{m}_0)\}$;
- For any $i \geq 1$: $O_i = \mathsf{Max}^{\sqsubseteq}(O_{i-1} \cup \mathsf{Oracle}(i))$;
- For any $i \geq 1$: $V_i = \mathsf{Max}^{\sqsubseteq}(V_{i-1} \cup F_{i-1}) \setminus \downarrow^{\sqsubseteq}(O_i)$;
- For any $i \geq 1$: $F_i = \mathsf{Max}^{\sqsubseteq}\left(\overline{\mathsf{Post}}(F_{i-1}) \cup \overline{\mathsf{Accel}}(F_{i-1})\right) \setminus \downarrow^{\sqsubseteq}(V_i \cup O_i)$.

It is not difficult to see that this sequence enjoys the following three properties:

Lemma 5. *Let $\mathcal{N}$ be a* PN, *$\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, and let* $\mathsf{CovSeq}(\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i \geq 0}$. *Then, for all $i \geq 0$:*

1. $\overline{\mathsf{Post}}(V_i) \cup \overline{\mathsf{Accel}}(V_i) \subseteq \downarrow^{\sqsubseteq}(V_i \cup F_i \cup O_i)$;
2. $\downarrow^{\preccurlyeq}(\mathsf{Flatten}(V_i \cup O_i)) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}(V_{i+1} \cup O_{i+1}))$;
3. *for all* $(\mathbf{m}_1, \mathbf{m}_2) \in F_i \cup V_i$: $\downarrow^{\preccurlyeq}(\mathbf{m}_2) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_1))$.

Completeness of the sequence. For all marking $\mathbf{m}$ computed by the Karp&Miller algorithm, we show that there exists a finite value k s.t. $\downarrow^{\preccurlyeq}(\mathbf{m}) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (V_k))$, where k depends on the depth of the node labeled by $\mathbf{m}$ in the Karp&Miller tree:

Lemma 6. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, $\mathcal{T} = \langle N, B, root, \Lambda\rangle$ be the K&M tree of $\mathcal{N}$ and* $\mathsf{CovSeq}\ (\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i\geq 0}$*. Then $\forall n \in N$: $\forall k \geq \sum_{n'\in \mathsf{Anc}(\mathcal{T},n)}(|\varsigma\,(n')\,|+3)$: $\downarrow^{\preccurlyeq}(\Lambda\,(n)) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (V_k \cup O_k))$.*

Proof. Sketch. We show by induction on the depth ℓ of nodes in the Karp&Miller tree that the lemma holds for all $n \in N$. For a node n at depth ℓ, we prove that there exists i and a pair $(\mathbf{m}_1, \mathbf{m}_2) \in V_i \cup O_i$ s.t. $\downarrow^{\preccurlyeq}(\Lambda\,(n)) \subseteq \downarrow^{\preccurlyeq}(\mathbf{m}_2)$, as follows. By induction hypothesis, there is j s.t. $\downarrow^{\preccurlyeq}(\Lambda\,(n')) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (O_j \cup V_j))$, where n' is the father of n. The value i s.t. $\Lambda\,(n) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (O_i \cup V_i))$ depends on j and the length of $\varsigma\,(n)$. Hence, a second induction on the size of $\varsigma\,(n)$ is used. That induction is applied in the case where $|\ \varsigma\,(n)\ | > 0$ and allows to prove that there is a pair $(\mathbf{m}_3, \mathbf{m}_4) \in V_{i-1} \cup O_{i-1}$ such that $(\mathsf{M}(n)\,, \mathbf{m}) \sqsubseteq (\mathbf{m}_3, \mathbf{m}_4)$ where $\mathbf{m}$ is such that $\mathsf{M}(n) \xrightarrow{\varsigma(n)} \mathbf{m}$. Once that result is obtained, we have by Lemma 4 that $\Lambda\,(n) \preccurlyeq \overline{\mathsf{Accel}}\,((\mathsf{M}(n)\,, \mathbf{m}))$, since $\mathsf{M}(n) \prec \mathbf{m}$, and that $\overline{\mathsf{Accel}}\,((\mathsf{M}(n)\,, \mathbf{m})) \preccurlyeq \overline{\mathsf{Accel}}\,((\mathbf{m}_3, \mathbf{m}_4)) = \mathbf{m}'$ by definition of $\overline{\mathsf{Accel}}$ and $\sqsubseteq$. Hence $(\mathbf{m}_4, \mathbf{m}') \in \downarrow^{\sqsubseteq}(O_{i-1} \cup F_{i-1} \cup V_{i-1})$ by Lemma 5.1. Finally, by construction of O_i and V_i, we conclude that $(\mathbf{m}_4, \mathbf{m}') \in \downarrow^{\sqsubseteq}(O_i \cup V_i)$, with $\Lambda\,(n) \preccurlyeq \mathbf{m}'$, hence $\Lambda\,(n) \in \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (O_i \cup V_i))$. □

As a consequence, the covering sequence is complete:

Corollary 1. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, and* $\mathsf{CovSeq}\ (\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i\geq 0}$*. There exists $k \geq 0$ such that for all $k' \geq k$ we have* $\mathsf{Cover}\ (\mathcal{N}, \mathbf{m}_0) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (V_{k'} \cup O_{k'}))$.

Soundness of the sequence. In order to show that the covering sequence is correct, it remains to show that any marking $\mathbf{m}$ produced by the sequence is s.t. $\downarrow^{\preccurlyeq}(\mathbf{m}) \subseteq \mathsf{Cover}\ (\mathcal{N}, \mathbf{m}_0)$. For that purpose, we need the two following lemmata:

Lemma 7. *Let $\mathcal{N}$ be a* PN *let A and B be two sets of ω-markings of $\mathcal{N}$. Then, $\downarrow^{\preccurlyeq}(A) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*\ (B))$ implies that $\downarrow^{\preccurlyeq}(\mathsf{Post}^*\ (A)) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*\ (B))$.*

Lemma 8. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, and let* $\mathsf{CovSeq}\ (\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i\geq 0}$*. Then, $\forall i \geq 1$: $\forall \mathbf{m} \in \mathsf{Flatten}\ (T_i \cup F_i \cup O_i)$, $\downarrow^{\preccurlyeq}(\mathbf{m}) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*\ (\mathbf{m}_0))$.*

As a consequence, we directly obtain our soundness result:

Corollary 2. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, and* $\mathsf{CovSeq}\ (\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i\geq 0}$*. Then, $\forall i \geq 1$, $\downarrow^{\preccurlyeq}(\mathsf{Flatten}\ (V_i \cup O_i)) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*\ (\mathbf{m}_0))$.*

Corollary 1 and 2 allow us to obtain the next Theorem.

Theorem 2. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be its initial marking,* Oracle *be an oracle, and* $\mathsf{CovSeq}\,(\mathcal{N}, \mathbf{m}_0, \mathsf{Oracle}) = (V_i, F_i, O_i)_{i \geq 0}$*. Then, there exists $k \geq 0$ such that*

1. *for all* $1 \leq i < k : {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_i \cup O_i)) \subset {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_{i-1} \cup O_{i-1}))$*;*
2. *for all* $i \geq k : {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_i \cup O_i)) = \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$.

Proof. By Corollary 1 and 2, we conclude that there exists at least one $k \in \mathbb{N}$ such that ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k)) = {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_{k+1} \cup O_{k+1}))$. Let us consider the smallest $k \in \mathbb{N}$ that satisfies that condition and let us prove that ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k)) = \mathsf{Cover}\,(\mathcal{N})$. Note that by Lemma 5.2 we have for all $0 \leq i < k : {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_i \cup O_i)) \subset {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_{i+1} \cup O_{i+1}))$.

First, we prove that ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(F_k)) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$. By construction, ${\downarrow^{\sqsubseteq}}(F_k) \subseteq {\downarrow^{\sqsubseteq}}(V_{k+1} \cup O_{k+1})$. Hence, ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(F_k)) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_{k+1} \cup O_{k+1}))$, by definition of $\sqsubseteq$. However, ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_{k+1} \cup O_{k+1})) = {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$, by definition of k.

By Lemma 5.1, $\overline{\mathsf{Post}}\,(V_k) \cup \overline{\mathsf{Accel}}\,(V_k) \subseteq {\downarrow^{\sqsubseteq}}(V_k \cup F_k \cup O_k)$, which implies that ${\downarrow^{\preccurlyeq}}\left(\mathsf{Flatten}\,\left(\overline{\mathsf{Post}}\,(V_k) \cup \overline{\mathsf{Accel}}\,(V_k)\right)\right) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup F_k \cup O_k))$. In particular, ${\downarrow^{\preccurlyeq}}\left(\mathsf{Flatten}\,\left(\overline{\mathsf{Post}}\,(V_k)\right)\right) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup F_k \cup O_k))$. Since ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(F_k)) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$, we have ${\downarrow^{\preccurlyeq}}\left(\mathsf{Flatten}\,\left(\overline{\mathsf{Post}}\,(V_k)\right)\right) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$. This means that ${\downarrow^{\preccurlyeq}}(\mathsf{Post}\,(\mathsf{Flatten}\,(V_k))) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$. Furthermore, by (1) and definition of O_k, ${\downarrow^{\preccurlyeq}}(\mathsf{Post}\,(\mathsf{Flatten}\,(O_k))) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(O_k))$. We conclude that ${\downarrow^{\preccurlyeq}}(\mathsf{Post}\,(\mathsf{Flatten}\,(V_k \cup O_k))) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$. Then, by Lemma 5.2, and since $\mathbf{m}_0 \in {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_1 \cup O_1))$, we have $\mathbf{m}_0 \in {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$. Hence, ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k))$ is a Post fixpoint that covers $\mathbf{m}_0$. Thus ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k)) \supseteq {\downarrow^{\preccurlyeq}}(\mathsf{Post}^*\,(\mathbf{m}_0))$. Since, by Corollary 2, ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(O_k \cup V_k)) \subseteq \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$, we have: ${\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k)) = \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$. Finally, by Lemma 5.2 and Corollary 2, $\forall i > k : {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_i \cup O_i)) = {\downarrow^{\preccurlyeq}}(\mathsf{Flatten}\,(V_k \cup O_k)) = \mathsf{Cover}\,(\mathcal{N})$. □

6 Practical Implementation

To implement the method in practice, we have to instantiate the oracle. First, note that the *empty oracle*, i.e. $\mathsf{Oracle}\,(i) = \emptyset$ for all $i \geq 1$, is a correct oracle. Indeed, ${\downarrow^{\preccurlyeq}}(\mathsf{Post}\,(\emptyset)) = \emptyset \subseteq {\downarrow^{\preccurlyeq}}(\emptyset)$ and ${\downarrow^{\preccurlyeq}}(\emptyset) \subseteq {\downarrow^{\preccurlyeq}}(\mathsf{Post}^*\,(\mathbf{m}_0))$ for all $\mathbf{m}_0$. Thus, the oracle can be regarded as an optional optimization of our algorithm. Yet, this optimization can be very powerful, as we show now. When using the empty oracle our method performs a breadth first search. In particular, if several accelerations can be applied from an ω-marking $\mathbf{m}$, each of them putting ω's in different places (for instance a first acceleration puts one ω in place p_1 and a second one puts an ω is p_2), then all the possible orders for their application will be investigated (i.e. first put the ω in place p_1 in $\mathbf{m}$ and then an ω in p_2; and vice-versa). However, all the possible orders lead to the same ω-marking (with an ω in p_1 and p_2) that covers the intermediate ones (where there is one ω either in p_1 or p_2). Thus, in order to improve our method, only one possible order should be explored. To achieve that goal, we present in the next paragraph the CovProc procedure where the oracle is implemented as a recursive call on ω-markings resulting from an acceleration. As a consequence, the initial breadth first search is mixed with a depth first search that allows to develop first the ω-markings resulting from an acceleration.

Algorithm 3. The CovProc algorithm

```
Data: A PN 𝒩 = ⟨P, T⟩, an ω-marking m₀
Result: A set of pairs of markings.
CovProc (𝒩, m₀) begin
    i := 0 ; Ō₀ := ∅ ; V̄₀ := ∅ ; F̄₀ := {(m₀, m₀)} ;
    repeat
        i := i + 1 ;
        Rᵢ := ∪_{m∈S} CovProc (𝒩, m) where S ⊆ Flatten (Accel‾ (F_{i−1})) ;
        Ōᵢ := Max^⊑ (Ō_{i−1} ∪ Rᵢ) ;
        V̄ᵢ := Max^⊑ (V̄_{i−1} ∪ F̄_{i−1}) \ ↓^⊑(Ōᵢ) ;
        F̄ᵢ := Max^⊑ (Post‾ (F̄_{i−1}) ∪ Accel‾ (F̄_{i−1})) \ ↓^⊑(Ōᵢ ∪ V̄ᵢ) ;
    until ↓^≼(Flatten (Ōᵢ ∪ V̄ᵢ)) ⊆ ↓^≼(Flatten (Ō_{i−1} ∪ V̄_{i−1})) ;
    return(Ōᵢ ∪ V̄ᵢ) ;
end
```

The CovProc *procedure.* The CovProc procedure is shown in Algorithm 3. It closely follows the definition of the covering sequence. At each step i, the oracle is implemented as a finite number of recursive calls to CovProc, where the initial ω-markings are the results of the accelerations occurring at this step. Note that a recursive call is not applied on all the accelerated ω-markings but a non-deterministically chosen subset S. Indeed, in practice, if we have two accelerated ω-markings $\mathbf{m}_1$ and $\mathbf{m}_2$ with $\downarrow^{\preccurlyeq}(\mathbf{m}_2) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Post}^*(\mathbf{m}_1))$ then it is not necessary to apply CovProc on $\mathbf{m}_2$ to explore the ω-markings that are reachable from $\mathbf{m}_2$.

This strategy allows to mix the breadth-first exploration of the covering sequence and the depth-first exploration due to the recursive calls which favor ω-markings with more ω. It turns out to be very efficient in practice (see hereunder). Since, for any pair $(\mathbf{m}_1, \mathbf{m}_2)$, $\mathsf{AccelPair}(\mathbf{m}_1, \mathbf{m}_2)$ contains strictly more ω's than $\mathbf{m}_1$ and $\mathbf{m}_2$, and since the number of places of the PN is bounded, the depth of recursion is bounded too, which ensures termination.

Let us show that this solution is correct and terminates. For any ω-marking $\mathbf{m}$, we let $\mathsf{Nb}\omega(\mathbf{m}) = |\{p \mid \mathbf{m}(p) = \omega\}|$. Then:

Lemma 9. *Let $\mathcal{N}$ be a* PN*, $\mathbf{m}_0$ be a ω-marking, and let $\overline{F}_i$ be the sets computed by* $\mathsf{CovProc}(\mathcal{N}, \mathbf{m}_0)$*. Then, $\forall i \geq 0$: for any $\mathbf{m} \in \mathsf{Flatten}(\overline{F}_i)$: $\mathsf{Nb}\omega(\mathbf{m}) \geq \mathsf{Nb}\omega(\mathbf{m}_0)$.*

Theorem 3. *For any* PN *$\mathcal{N}$ and any ω-marking $\mathbf{m}_0$:* $\mathsf{CovProc}(\mathcal{N}, \mathbf{m}_0)$ *terminates and* $\downarrow^{\preccurlyeq}(\mathsf{Flatten}(\mathsf{CovProc}(\mathcal{N}, \mathbf{m}_0))) = \mathsf{Cover}(\mathcal{N}, \mathbf{m}_0)$.

Proof. The proof works by induction on $\mathsf{Nb}\omega(\mathbf{m}_0)$.

Base case ($\mathsf{Nb}\omega(\mathbf{m}_0) = |P|$) In that case, $\mathsf{CovProc}(\mathcal{N}, \mathbf{m}_0)$ finishes after two iterations and returns $\overline{O}_2 \cup \overline{V}_2 = \{(\mathbf{m}_0, \mathbf{m}_0)\}$. Remark that no recursive call is performed because $R_1 = \overline{\mathsf{Accel}}(\overline{F}_0) = \emptyset$ and $R_2 = \overline{\mathsf{Accel}}(\{(\mathbf{m}_0, \mathbf{m}_0)\}) = \emptyset$. Moreover, $\downarrow^{\preccurlyeq}(\mathsf{Flatten}(\mathsf{CovProc}(\mathcal{N}, \mathbf{m}_0))) = \downarrow^{\preccurlyeq}(\mathbf{m}_0) = \mathsf{Cover}(\mathcal{N}, \mathbf{m}_0)$.

Inductive case ($\mathsf{Nb}\omega(\mathbf{m}_0) = k < |P|$) We consider two cases. First, assume that the algorithm terminates after ℓ iterations, i.e., assume that $\downarrow^{\preccurlyeq}(\mathsf{Flatten}(\overline{O}_\ell \cup \overline{V}_\ell)) =$

$\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{O}_{\ell-1} \cup \overline{V}_{\ell-1}\big)\big)$, but for any $1 \leq j \leq \ell - 1$: $\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{O}_j \cup \overline{V}_j\big)\big) \neq \downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{O}_{j-1} \cup \overline{V}_{j-1}\big)\big)$. By Lemma 9, for any $1 \leq j \leq \ell$, for any $(\mathbf{m}_1, \mathbf{m}_2) \in \overline{F}_j$: $\mathsf{Nb}\omega\,(\mathbf{m}_2) \geq k$. Hence, for any $1 \leq j \leq \ell$, for any $\mathbf{m} \in \mathsf{Flatten}\,\big(\overline{\mathsf{Accel}}\,\big(\overline{F}_j\big)\big)$: $\mathsf{Nb}\omega\,(\mathbf{m}) \geq k + 1$. Thus, by induction hypothesis, for any $1 \leq j \leq \ell$, for any $\mathbf{m} \in \mathsf{Flatten}\,\big(\overline{\mathsf{Accel}}\,\big(\overline{F}_j\big)\big)$, $\mathsf{CovProc}\,(\mathcal{N}, \mathbf{m})$ terminates and returns a set of pairs such that $\downarrow^{\preccurlyeq}(\mathsf{Flatten}\,(\mathsf{CovProc}\,(\mathcal{N}, \mathbf{m}))) = \mathsf{Cover}\,(\mathcal{N}, \mathbf{m})$. As a consequence, and since $\mathsf{Flatten}\,\big(\overline{\mathsf{Accel}}\,\big(\overline{F}_j\big)\big)$ is a finite set for any $1 \leq j \leq \ell$, we conclude that R_j is computed in a finite amount of time and that $\downarrow^{\preccurlyeq}(\mathsf{Post}\,(\mathsf{Flatten}\,(R_j))) \subseteq \downarrow^{\preccurlyeq}(\mathsf{Flatten}\,(R_j)) \subseteq \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$, for any $1 \leq j \leq \ell$.

Let Ω denote the function s.t., for any $1 \leq j \leq \ell$: $\Omega(j) = R_j$, and, for any $j > \ell$, $\Omega(j) = \emptyset$. Thus, Ω is an oracle. Let us assume that $\mathsf{CovSeq}\,(\mathcal{N}, \mathbf{m}_0, \Omega) = (V_i, F_i, O_i)_{i \geq 1}$. Clearly, for any $0 \leq j \leq \ell$, $\overline{V}_j = V_j$ and $\overline{O}_j = O_j$. Thus, by Theorem 2, there exists k s.t. $\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_{k-1} \cup \overline{O}_{k-1}\big)\big) = \downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_k \cup \overline{O}_k\big)\big) = \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$, and s.t. for every $1 \leq j \leq k - 1$: $\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_{j-1} \cup \overline{O}_{j-1}\big)\big) \subset \downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_j \cup \overline{O}_j\big)\big)$. Hence, $k = \ell$, and we conclude that $\mathsf{CovProc}\,(\mathcal{N}, \mathbf{m}_0)$ terminates and returns $\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_\ell \cup \overline{O}_\ell\big)\big) = \mathsf{Cover}\,(\mathcal{N}, \mathbf{m}_0)$.

In the latter case, we assume that the algorithm does not terminate and derive a contradiction. This can happen either because the test of the **repeat** loop is never fulfilled, or because some step j of the loop takes an infinite time to complete. The latter is not possible: Indeed, R_j is computed in a finite amount of time and the functions $\mathsf{Max}^{\sqsubseteq}$, $\mathsf{Flatten}$, $\overline{\mathsf{Post}}$, $\overline{\mathsf{Accel}}$, as well as the guard of the loop are computable. Thus, if the algorithm does not terminate, it computes an infinite sequence of sets $(\overline{V}_i, \overline{F}_i, \overline{O}_i)_{i \geq 0}$. Symmetrically to the first part of this proof, we build an oracle Ω s.t. $\Omega(j) = R_j$ for any $j \geq 1$. Let us assume that $\mathsf{CovSeq}\,(\mathcal{N}, \mathbf{m}_0, \Omega) = (V_i, F_i, O_i)_{i \geq 0}$. Clearly, for any $j \geq 0$, we have $V_j = \overline{V}_j$ and $O_j = \overline{O}_j$. Hence, by Theorem 2, we conclude that there exists $k \geq 1$ s.t. $\downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_k \cup \overline{O}_k\big)\big) = \downarrow^{\preccurlyeq}\big(\mathsf{Flatten}\,\big(\overline{V}_{k-1} \cup \overline{O}_{k-1}\big)\big)$, which compels the algorithm to terminate at step k. Contradiction. □

Empirical evaluation. We have implemented a prototype that computes the coverability set of a PN, thanks to the covering sequence method and the Karp&Miller algorithm. We have selected bounded and unbounded PN that describe (mutual exclusion) protocols (bounded PN), parameterized systems and communication protocols (unbounded PN). The prototype has been written in the PYTHON programming language in a very straightforward way. As a consequence, running times are given for the sake of comparison only. Nevertheless, the prototype performs very well on our examples[2] (Table 1).

We have compared two implementations of the covering sequence to the KM algorithm. The former (**Cov. Seq. w/o oracle**) is the covering sequence where $\mathsf{Oracle}\,(i) = \emptyset$ for any $i \geq 0$. In that case the sets of pairs built by our algorithm are small (column Max P.) wrt the size of the K&M tree (column Nodes), although the *total* number of pairs (column Tot. P.) is not dramatically small wrt the K&M tree. This shows the efficiency of our approach based on *pairs* and on the $\sqsubseteq$ order, wrt the classical approach.

The latter implementation is the CovProc procedure (Algorithm 3) with the following oracle: we consider the accelerated ω-markings one by one. If an accelerated ω-marking is already covered by the Flatten of the pairs computed by previous recursive

[2] See `http://www.ulb.ac.be/di/ssd/ggeeraer/eec` for a complete description.

Table 1. Empirical evaluation of the covering sequence. Experiments on an INTEL XEON 3GHz. Times in seconds (× = no result within 20 minutes). P = number of places; T = number of transitions; MCS = size of the minimal coverability set ; Tp = Bounded or Unbounded PN; Max P. = $\max\{|V_i \cup O_i \cup F_i|, i \geq 1\}$; Tot. P. = tot. number of pairs created along the whole execution.

Example					KM		Cov. Seq. w/o Oracle			CovProc		
Name	P	T	MCS	Tp	Nodes	Time	Max P.	Tot. P.	Time	Max P.	Tot. P.	Time
RTP	9	12	9	B	16	0.18	47	47	0.10	47	47	0.13
lamport	11	9	14	B	83	0.18	115	115	0.17	115	115	0.17
peterson	14	12	20	B	609	2.19	170	170	0.21	170	170	0.25
dekker	16	14	40	B	7,936	258.95	765	765	1.13	765	765	1.03
readwrite	13	9	41	B	11,139	529.91	1,103	1,103	1.43	1,103	1,103	1.75
manuf.	13	6	1	U	32	0.19	9	101	0.18	2	47	0.14
kanban	16	16	1	U	9,839	1221.96	593	9,855	95.05	4	110	0.19
basicME	5	4	3	U	5	0.10	5	5	0.12	5	5	0.12
CSM	14	13	16	U	$>2.40 \cdot 10^6$	×	371	3,324	14.38	178	248	0.34
FMS	22	20	24	U	$>6.26 \cdot 10^5$	×	>4,460	×	×	477	866	2.10
PNCSA	31	36	80	U	$>1.02 \cdot 10^6$	×	>5,896	×	×	2,617	13,408	113.79
multipoll	18	21	220	U	$>1.16 \cdot 10^6$	×	>7,396	×	×	14,034	14,113	365.90
mesh2x2	32	32	256	U	$>8.03 \cdot 10^5$	×	>6,369	×	×	10,483	12,735	330.95

calls, it is forgotten; otherwise a recursive call is applied on it. In the case of *bounded* PN, CovProc performs as the covering sequence with trivial oracle, which is not surprising since no acceleration occur. In the case of *unbounded* PN, the oracle-based optimization is successful: the sets of pairs built by CovProc are much smaller than the respective K&M trees, and negligible with respect to the sets built with the trivial oracle. The CovProc procedure always terminates within 20 minutes and outperforms the covering sequence with trivial oracle. Finally, the execution times of CovProc are several order of magnitudes smaller than those of the KM procedure.

Acknowledgments. The authors are grateful to Prof. Peter Starke and Prof. Alain Finkel for their friendly cooperation.

References

1. Finkel, A.: The minimal coverability graph for Petri nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1993. LNCS, vol. 674, pp. 210–243. Springer, Heidelberg (1993)
2. Finkel, A., Geeraerts, G., Raskin, J.F., Van Begin, L.: A counter-example to the minimal coverability tree algorithm. Technical Report 535, Université Libre de Bruxelles (2005)
3. Geeraerts, G.: Coverability and Expressiveness Properties of Well-structured Transition Systems. PhD thesis, Université Libre de Bruxelles, Belgium (2007)
4. Geeraerts, G., Raskin, J.F., Van Begin, L.: Well-structured languages. Act. Inf. 44(3-4)
5. Geeraerts, G., Raskin, J.F., Van Begin, L.: On the efficient computation of the minimal coverability set for Petri nets. Technical Report CFV 2007.81
6. German, S., Sistla, A.: Reasoning about Systems with Many Processes. J. ACM 39(3) (1992)

7. Grahlmann, B.: The PEP tool. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 440–443. Springer, Heidelberg (1997)
8. Karp, R.M., Miller, R.E.: Parallel Program Schemata. JCSS 3, 147–195 (1969)
9. Luttge, K.: Zustandsgraphen von Petri-Netzen. Master's thesis, Humboldt-Universität (1995)
10. Reisig, W.: Petri Nets. An introduction. Springer, Heidelberg (1986)
11. Starke, P.: Personal communication
12. Van Begin, L.: Efficient Verification of Counting Abstractions for Parametric systems. PhD thesis, Université Libre de Bruxelles, Belgium (2003)

Analog/Mixed-Signal Circuit Verification Using Models Generated from Simulation Traces*

Scott Little, David Walter, Kevin Jones, and Chris Myers

University of Utah, Salt Lake City, UT 84112, USA
{little,dwalter,kjones,myers}@vlsigroup.ece.utah.edu

Abstract. Formal and semi-formal verification of analog/mixed-signal circuits is complicated by the difficulty of obtaining circuit models suitable for analysis. We propose a method to generate a formal model from simulation traces. The resulting model is conservative in that it includes all of the original simulation traces used to generate it plus additional behavior. Information obtained during the model generation process can also be used to refine the simulation and verification process.

1 Introduction

Increased interest in system on a chip design has resulted in a need to improve validation methods for analog/mixed-signal (AMS) circuits. Validation of digital circuits has changed dramatically in the past ten years while AMS circuit validation remains largely the same. AMS circuit validation is still largely driven by designers using many simulation traces to validate specific properties of a circuit. While this methodology has been used with success for many years, recent trends are stretching it beyond its capacity. Increase in process variations and use of mixed-signal circuits present challenges that this simulation only methodology is not well prepared to address.

Currently, most AMS designers use an informal approach to circuit verification. With the aid of a simulator, the designer creates a circuit that under ideal conditions meets a set of specifications. A major concern for circuit designers using today's process technologies is the circuit's resilience to process variation. To help understand how the circuit operates under global variation, corner simulations are run. These simulations evaluate the circuit performance under combinations of change for common global variations such process, voltage, and temperature. There may also be local transistor to transistor process variation. To understand how this variation affects the circuit, Monte Carlo simulation is employed. These methods for exploring global and local variation are very expensive. This expense increases dramatically as more sources of variation are explored. As a result, only the most common sources of variation of the most critical specifications of the most critical circuits are thoroughly validated. The design team also has no real measure of the quality of the verification performed

* Support from SRC contract 2005-TJ-1357 and an SRC Graduate Fellowship.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 114–128, 2007.
© Springer-Verlag Berlin Heidelberg 2007

on the design. The correctness of the design is almost solely the responsibility of each designer. The lack of feedback to the designer and large cost to verify the circuit under variation are major concerns when using this simulation only methodology.

Based on the success of formal methods for digital circuits there has been an increasing body of work in formal methods for AMS circuits. Several tools and methods have been developed to explore the continuous state space of these systems [1,2,3,4,5,6]. These methods work well on small examples and have shown some promise to work on larger circuits. One challenge for these methods is the significant effort required to create an appropriate formal model for each different system. These methods also suffer from high computation costs for the analysis of the model. The more accurately the method explores the state space of the system the more computationally intensive it is.

In response to these challenges, there has been recent work in verifying formal properties within the framework of simulation. There are currently two main approaches for using simulation as a verification aid. The first approach attempts to find a finite number of simulation traces that are sufficient to represent all trajectories of the system and therefore prove correctness of the circuit [7,8,9,10]. The second approach uses simulation traces to generate a formal model which is then analyzed using a state space exploration engine [11]. This paper describes a new method using the second approach.

Dastidar, et al. [11] generate a finite state machine (FSM) from a systematic set of simulation traces. This FSM includes currents, voltages, and time as state variables to generate an acyclic FSM. The state space of the system is divided into symmetric state divisions. After each delta time step, the current state of the simulator is determined and rounded to the center of the appropriate state division. The simulator is then started from this point and run for the next delta time step. This process continues until the global time reaches a user specified maximum. Conversely, our approach uses *Labeled Hybrid Petri Nets* (LHPNs) [4,5] as the model. The state space is divided as specified by user provided thresholds. A global timer is not a part of the state space which results in graphs that may include cycles. Simulation traces are run from start to finish without stopping allowing our model to preserve the original simulation trace.

The novelty of our approach is that the model allows for dynamic variation of parameters. Standard simulation based methods allow for changes in initial conditions and parameters, but these values are then fixed for the duration of the simulation run. Our model explores the system under ranges of initial conditions as well as ranges of dynamically changing parameter values. This additional behavior improves our ability to uncover variation induced errors.

The verification flow supported by our tool, `LEMA`, is shown in Fig. 1. Our previous work [4,5,12,6] describes how a subset of VHDL-AMS can be compiled into an LHPN and analyzed using one of our model checkers. Each model checker uses a different data structure to represent that state space including: *difference bound matrices* (DBMs) [4], *binary decision diagrams* (BDDs) [5], and *satisfiability modulo theories* (SMT) formulas [6]. This paper describes the flow on

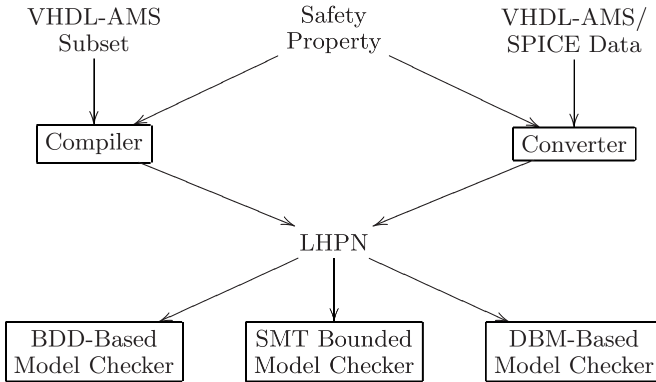


Fig. 1. LEMA: LHPN Embedded/Mixed-signal Analyzer

the right side of Fig. 1 which takes simulation data, generates an LHPN, and uses one of these model checkers to verify the given property of the system. The remainder of this paper gives a brief introduction to LHPNs, describes the algorithms used to generate an LHPN model from a set of simulation traces, and concludes with a discussion of interesting metrics that can be extracted from the simulation data during model generation.

2 Motivating Example

The switched capacitor integrator circuit shown in Fig. 2 is a circuit used as a component in many AMS circuits such as ADCs and DACs. Although only a small piece of these complex circuits, the switched capacitor integrator proves to be a useful example illustrating the type of problems that can be present in AMS circuit designs. Discrete-time integrators typically utilize switched capacitor circuits to accumulate charge. Capacitor mismatch can cause gain errors in integrators. Also, the CMOS switch elements in switched capacitor circuits inject charge when they transition from closed to open. This charge injection is difficult to control with any precision, and its voltage-dependent nature leads to circuits that have a weak signal-dependent behavior. This can cause integrators to have slightly different gains depending on their current state and input value. Circuits using integrators run the risk of the integrator saturating near one of the power supply rails. Therefore, the verification property to check for this circuit is whether or not the voltage V_{out} can rise above 2000mV or fall below −2000mV. It is essential to ensure that this never happens during operation under any possible permutation of component variations. For simplicity, we assume for this example that the major source of uncertainty is that the capacitor C_2 can vary dynamically by ±10 percent from its nominal value. This circuit, therefore, must be verified for all values in this range [13].

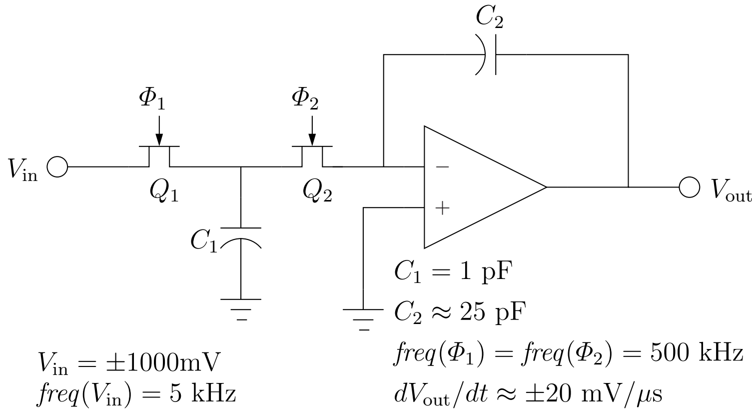


Fig. 2. A schematic of a switched capacitor integrator

3 Labeled Hybrid Petri Nets

An LHPN is a Petri net model developed to represent AMS circuits [4,12]. The model is inspired by features in both hybrid Petri nets [14] and hybrid automata [15]. An LHPN is a tuple $N = \langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$ where:

P : is a finite set of places;

T : is a finite set of transitions;

B : is a finite set of Boolean signals;

V : is a finite set of continuous variables;

$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;

L : is a tuple of labels defined below;

$M_0 \subseteq P$ is the set of initially marked places;

S_0 : is the set of initial Boolean signal values;

Q_0 : is the set of initial ranges of values for each continuous variable and;

R_0 : is the set of initial ranges of rates for each continuous variable.

A key component of LHPNs are the labels. Some labels contain *hybrid separation logic* (HSL) formulas which are a Boolean combination of Boolean variables and separation predicates (inequalities relating continuous variables to constants). These formulas satisfy the following grammar:

$$\phi ::= \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid c_i x_i \geq c_j x_j + c$$

where b_i are Boolean variables, x_i and x_j are continuous variables, and c_i, c_j, and c are rational constants in $\mathbb{Q}$. Note that any inequality between two real variables

can be formed with $\geq$ and negations of $\geq$ inequalities. The labels permitted in LHPNs are represented using a tuple $L = \langle En, D, BA, VA, RA\rangle$:

$En : T \rightarrow \phi$ labels each transition $t \in T$ with an enabling condition;

$D : T \rightarrow |\mathbb{Q}| \times (|\mathbb{Q}| \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper bound $[d_\text{l}, d_\text{u}]$ on the delay for t to fire;

$BA : T \rightarrow 2^{(B \times \{0,1\})}$ labels each transition $t \in T$ with Boolean assignments made when t fires;

$VA : T \rightarrow 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a continuous variable assignment range, consisting of a lower and upper bound $[a_l, a_u]$, that is made when t fires;

$RA : T \rightarrow 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a range of rates, consisting of a lower and upper bound $[r_l, r_u]$, that are assigned when t fires.

The semantics of the LHPN model are briefly illustrated using an LHPN model of the switched capacitor integrator shown in Fig. 3. A formal description of the semantics for LHPNs can be found in [12]. The output voltage, V_out, is modeled by the LHPN shown in Fig. 3a. The rate of the output voltage changes based on the value of V_out and the input voltage. The square wave input voltage, V_in, is modeled using the LHPN shown in Fig. 3b. V_in is modeled as a stable, multi-valued continuous quantity. Stable, multi-valued continuous quantities are modeled using continuous variables with a rate of zero and are updated using a variable assignment after a time delay. The LHPN shown in Fig. 3c is used to detect a failure. The enabling condition on the transition is the negation of an HSL formula for the safety property being verified. When this transition is enabled and fires, a failure is detected. In the initial state, p_0, p_1, and p_6 are marked; *fail* is **false**; V_out is -1000mV; V_in is -1000mV; the rate of V_in is 0; and the rate of V_out is 18 to 22 mV/μs. Initially, t_1 is the only enabled transition. However, as time passes, V_out crosses 0V enabling t_6 which fires immediately moving the token from p_6 to p_3. After 100 to 101μs from the initial state, t_1 fires and sets V_in to 1000mV. This change on V_in enables transition t_3 which fires immediately and sets the rate of V_out to be between -22 and -17 mV/μs. Transition t_4 fires next in zero time when $V_\text{out} < 0V$. After this firing, transition t_2 fires after being enabled 99 to 100μs. This firing sets V_in to -1000mV and enables transition t_5 which fires immediately and sets the rate of V_out to be between 17 and 22 mV/μs. This behavior continues until the range of V_out enables transition t_0 which fires and sets *fail* to **true**.

4 LHPN Model Generation

During the course of traditional analog circuit verification, designers run many different simulations to check that the circuit meets its specification. The goal of this work is to automatically generate an LHPN such as the one shown in Fig. 3 from simulation data. The generated LHPN model of the circuit is conservative and models all the provided simulation traces plus additional behavior. By using

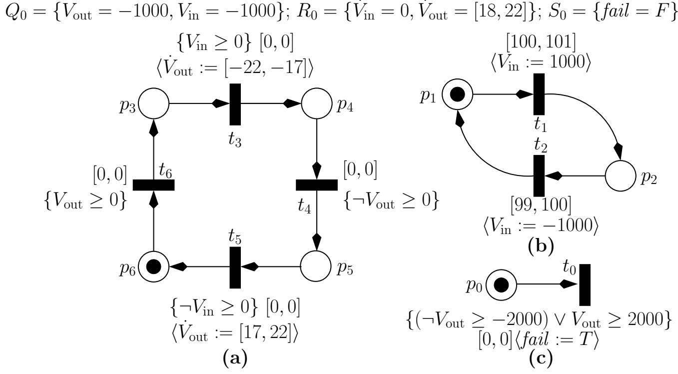


Fig. 3. A simple LHPN example for the switched capacitor integrator

---

Algorithm 1. GenerateLHPNfromData(*data*, *thresholds*, *property*)

---

1 $binData$ = `binData`($data$, $thresholds$);
2 $rates$ = `calculateRates`($data$, $binData$);
3 $delays$ = `calculateTimes`($data$, $binData$, $rates$);
4 N = `generateLHPN`($binData$, $rates$, $delays$, $property$);

---

simulations already produced by the designer, no additional simulation time is required. However, the quality of the model is directly related to the simulations used to create it. If the designer has inadequately simulated the design, the model may not exhibit the full behavior of the system. In this case, there is a potential that the actual circuit may have a failing behavior that is not included in the generated model. To help address this issue, Section 6 proposes the use of coverage metrics.

Algorithm 1 describes the process of taking simulation data and generating an LHPN. The input to our algorithm is time series simulation data, thresholds on the state space of the system, and the safety property to be checked specified using an HSL formula. The data is first sorted into bins based on the thresholds. Next, ranges of rates are calculated for each continuous variable within each bin. The algorithm assumes nothing about the dependence or independence of the rates. Each rate is calculated individually for each bin. It is expected that the rates change during different phases of operation. For this reason, it is important that thresholds are selected to separate the different phases of operation into distinct bins. At this point, continuous variables which are mostly stable but occasionally change are identified as variables that can be approximated by discrete transitions. Finally, after these calculations, the LHPN is generated.

Algorithm 1 is illustrated using two simulations of the switched capacitor integrator. In particular, the switched capacitor integrator is simulated with capacitance values of 23pF and 27pF for capacitor C_2. The simulation data is recorded for the nodes representing the input voltage, V_{in}, and output voltage,

Table 1. Partial simulation result with $C_2 = 23pF$ for the integrator

Time (μs)	V_{in} (mV)	V_{out} (mV)	Bin	$\Delta V_{\mathrm{in}}/\Delta t$ (mV/μs)	$\Delta V_{\mathrm{out}}/\Delta t$ (mV/μs)	V_{in} time
0.00	-1000	-1000	00	-227.85	21.29	0.0
0.50	-1000	-999	00	0.0	21.74	0.5
⋮	⋮	⋮	⋮	⋮	⋮	⋮
46.48	-1000	-0.4	00	-	-	46.48
46.98	-1000	10	01	0.0	21.74	46.98
47.48	-1000	21	01	0.0	21.74	47.48
⋮	⋮	⋮	⋮	⋮	⋮	⋮
100.50	-1000	1174	01	-	-	100.50
100.54	-840	1174	01	-	-	100.54
100.62	-520	1176	01	-	-	100.70
100.78	120	1176	11	275.00	-21.08	0.08
101.00	1000	1174	11	0.0	-21.74	0.30
101.03	1.0	1173	11	0.0	-21.74	0.33
⋮	⋮	⋮	⋮	⋮	⋮	⋮
154.98	1000	0.3	11	-	-	54.28
155.48	1000	-11	10	0.0	-21.74	54.78
155.98	1000	-21	10	0.0	-21.74	55.28
⋮	⋮	⋮	⋮	⋮	⋮	⋮
200.00	1000	-978	10	-	-	99.30
200.04	840	-979	10	-	-	99.34
200.12	520	-980	10	-	-	99.50
200.28	-120	-981	00	-275.00	21.08	0.08
200.50	-1000	-978	00	0.0	21.74	0.30
200.53	-1000	-976	00	0.0	21.74	0.33
⋮	⋮	⋮	⋮	⋮	⋮	⋮
400.00	1000	-957	10	-	-	99.34

V_{out}, during 400μs of transient simulation for each capacitance value. Part of the data from these simulations is shown in Tables 1 and 2.

The first step of Algorithm 1 is to bin the data based upon the thresholds provided. For this example, the thresholds chosen for both V_{in} and V_{out} are 0V. Each data file is analyzed and each time point is assigned to a bin based upon the values of V_{in} and V_{out}. In the data shown in Tables 1 and 2, each digit in the fourth column represents a bin. The first digit represents the V_{in} bin and the second digit represents the V_{out} bin. For instance, at time 100.50μs in Table 1, the bin assigned is `01` indicating that V_{in} is below 0V and V_{out} is above 0V. When V_{in} moves above 0V at time 100.78μs, the bin assignment changes to `11`.

The second step of Algorithm 1 calculates rate of change for each continuous variable. The rate of change is calculated for each bin using two time points within the same bin separated by a given interval. In the data shown in Tables 1

Table 2. Partial simulation result with $C_2 = 27pF$ for the integrator

Time (μs)	V_{in} (mV)	V_{out} (mV)	Bin	$\Delta V_{\mathrm{in}}/\Delta t$ (mV/μs)	$\Delta V_{\mathrm{out}}/\Delta t$ (mV/μs)	V_{in} time
0.00	-1000	-1000	00	-227.85	18.14	0.0
0.50	-1000	-999	00	0.0	18.52	0.5
⋮	⋮	⋮	⋮	⋮	⋮	⋮
54.48	-1000	-0.3	00	-	-	54.48
54.98	-1000	9	01	0.0	18.52	54.98
55.48	-1000	18	01	0.0	18.52	55.48
⋮	⋮	⋮	⋮	⋮	⋮	⋮
100.50	-1000	852	01	-	-	100.50
100.54	-840	852	01	-	-	100.54
100.62	-520	853	01	-	-	100.70
100.78	120	854	11	275.00	-17.96	0.08
101.00	1000	852	11	0.0	-18.52	0.30
101.03	1000	850	11	0.0	-18.52	0.33
⋮	⋮	⋮	⋮	⋮	⋮	⋮
146.98	1000	0.3	11	-	-	46.28
147.48	1000	-9	10	0.0	-18.52	46.78
147.98	1000	-18	10	0.0	-18.52	47.28
⋮	⋮	⋮	⋮	⋮	⋮	⋮
200.00	1000	-981	10	-	-	99.30
200.04	840	-982	10	-	-	99.34
200.12	520	-983	10	-	-	99.50
200.28	-120	-984	00	-275.00	17.96	0.08
200.50	-1000	-981	00	0.0	18.52	0.30
200.53	-1000	-980	00	0.0	18.52	0.33
⋮	⋮	⋮	⋮	⋮	⋮	⋮
400.00	1000	-963	10	-	-	99.34

and 2 the interval is set to a length of ten. For example, the rate of change for V_{out} at time 46.98μs in Table 1 is calculated by looking at its value at this time point and the value ten points later. This value is determined to be 21.74mV/μs. After all the rates have been calculated, the minimum and maximum rates for each bin are determined. These values are the specified rate of change whenever the model is in that specific bin. The range of rates for each bin found from these two simulation runs for V_{out} from the switched capacitor integrator are shown in Table 3.

The third step of Algorithm 1 examines the rates for each continuous variable to determine if it can be reasonably approximated with a multi-valued continuous variable that makes discrete changes. This is true if a variable remains stable for large portions of time (i.e., has a rate of change that is nearly 0). In the switched capacitor integrator example, the square wave input voltage, V_{in}, is an example

Table 3. Rates for V_{out} from the switched capacitor integrator

Bin	Place	Range of rates	Comment
00	p_6	[17,22]	$V_{\text{in}} < 0V$; $V_{\text{out}} < 0V$
01	p_3	[17,22]	$V_{\text{in}} < 0V$; $V_{\text{out}} \geq 0V$
11	p_4	[-22,-17]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} \geq 0V$
10	p_5	[-22,-17]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} < 0V$

of this type of signal. In Tables 1 and 2, V_{in} has a rate of change of 0 mV/μs at most times. For these discrete signals, the algorithm determines the amount of time that they spend at each discrete value. This is shown in the last column of Tables 1 and 2. The value of this continuous variable is then assigned to change at that specified time. For example, V_{in} is set to -1000mV and remains there for 100μs to 101μs after which is changes to 1000mV and remains there for 99μs to 100μs. This cycle then repeats.

Using the information derived in the first three steps, Algorithm 1 can now generate an LHPN that models the provided simulation traces. A place is created for each bin discovered in the simulation traces. While in this example a place is produced for every bin assignment, in larger examples, many bins may never be encountered during simulation, so places are not generated for these unreachable bins. The places created for each bin from the integrator example are shown in the second column of Table 3. Next, transitions between bins are created when a transition between two bins is found in the simulation traces. It is theoretically possible that this could result in a fully connected graph, but in practice this is highly unlikely. Each transition is given an enabling condition representing the threshold that is being crossed to move from the first bin to the second. The delay for the transition is set to [0,0] to make it fire immediately as the state of the system moves from one bin to the next. Finally, each transition is given a rate assignment to set the rate to the value for that bin as shown in Table 3. For the integrator example, the result is the LHPN shown in Fig. 3a. Note that the rate assignment is omitted for transition t_6, since the range of rates for p_6 and p_3 are the same. Similarly, the rate assignment for t_4 can be omitted.

Next, a separate net is created for each discrete multi-valued continuous signal. A place is added for each discrete value of this variable. For the integrator, place p_1 is added for V_{in} equal to -1000mV, and p_2 is added to represent that V_{in} is equal to 1000mV. A transition is added for each discrete change found in the simulation data. The delay of this transition is determined by the time calculated in the previous step. Finally, this transition includes a continuous variable assignment to execute the discrete change. For the integrator example, the LHPN generated to control V_{in} is shown in Fig. 3b.

Finally, the last step is to create an LHPN to check the safety property provided as an HSL formula. This net has a single initially marked place and a single transition. The transition's enabling condition is the complement of the safety property. This transition has a delay of [0,0], and it sets a special Boolean signal *fail* to true when it fires. Therefore, to verify this safety property, a model

checker only needs to determine if there exists any state in which *fail* is true. For the integrator example, the LHPN generated to check if the circuit can saturate is shown in Fig. 3c. Note that to cause analysis to terminate sooner, a Boolean condition, $\neg fail$, can be added to each transition in the LHPNs. This results in a deadlock once a failure is detected.

The LHPNs generated from simulation traces include ranges of rates. While these LHPNs can be directly analyzed using the BDD and SMT model checkers, they cannot be directly analyzed using the DBM method. To enable analysis of these LHPN models by the DBM model checker, a piecewise approximation of the range of rates is created by performing a transformation on the LHPN. In particular, this transformation allows the rate to change nondeterministically between the lower and upper bound on the range of rates. By exploring all of the possible nondeterministic rate changes the state space of the system for the entire range of rates is explored.

To simplify the description, the expansion process is illustrated using the LHPN in Fig. 4a which only has a threshold for V_{in} at 0V and no threshold for V_{out} resulting in just two bins represented with two places. The rate expansion proceeds by adding an additional transition and Boolean signal for each range of rates present in the unexpanded LHPN. The original transition is modified by changing the rate assignment to assign the lower bound of the range. Also, additional Boolean signal assignments are added to enable the firing of the upper bound of the rate and disable all other upper bound rate assignments. For example, in Fig. 4b the rate assignment on t_0 is changed from $[-22, -17]$ to -22. The Boolean signal $v1$ is set to true enabling the firing of t_2. The delay bound on t_2 is $[0, \infty]$ allowing t_2 to fire at any time in the future while the enabling condition remains satisfied. When t_2 fires it sets the rate to the upper bound, -17 and sets the Boolean signal $v1$ to false. This translation method has been implemented in the `LEMA` tool enabling LHPNs with ranges of rates to be analyzed by any of the model checkers in the tool.

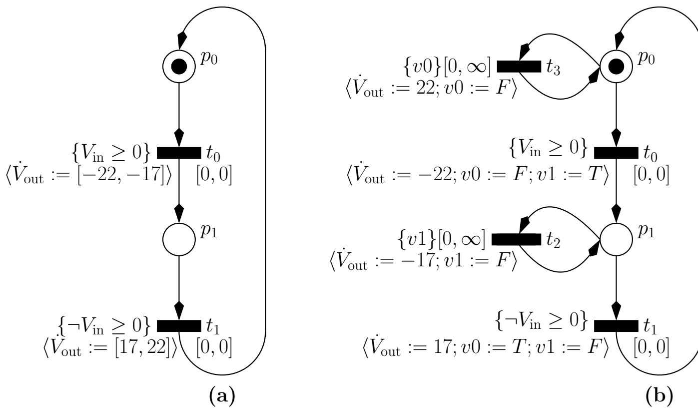


Fig. 4. LHPN demonstrating piecewise approximation of a range of rates

5 Case Study

Using Algorithm 1, two simulation traces of the switched capacitor integrator result in the LHPN shown in Fig. 3. Although neither of the simulation traces indicate a problem with saturation of the integrator, a state space analysis using the DBM model checker finds in less than a second that there is a potential for the circuit to fail. This failure can occur when the integrator charges the capacitor, C_2, at a rate that is on average faster than the rate of discharge. This situation causes charge to build up on the capacitor and eventually results in V_{out} reaching a voltage above 2000mV. The reason that this method can find this failure is that the LHPN model represents not only each simulation trace, but also the union of the traces. It is this behavior explored by unioning the traces that allows the analyzer to find the flaw in the circuit.

Saturation of the integrator can be prevented using the circuit shown in Fig. 5. In this circuit, a resistor in the form of a switched capacitor is inserted in parallel with the feedback capacitor. This causes V_{out} to drift back to 0V. In other words, if V_{out} is increasing, it increases faster when it is far below 0V than when it is near or above 0V. Using the same simulation parameters and thresholds for this circuit, Algorithm 1 obtains an LHPN with the same structure as the one shown in Fig. 3, but the ranges of rates for each bin are as shown in Table 4. This LHPN also fails the property as the thresholds are too simple to capture the effect of the additional switched capacitor. Due to the addition of this switched

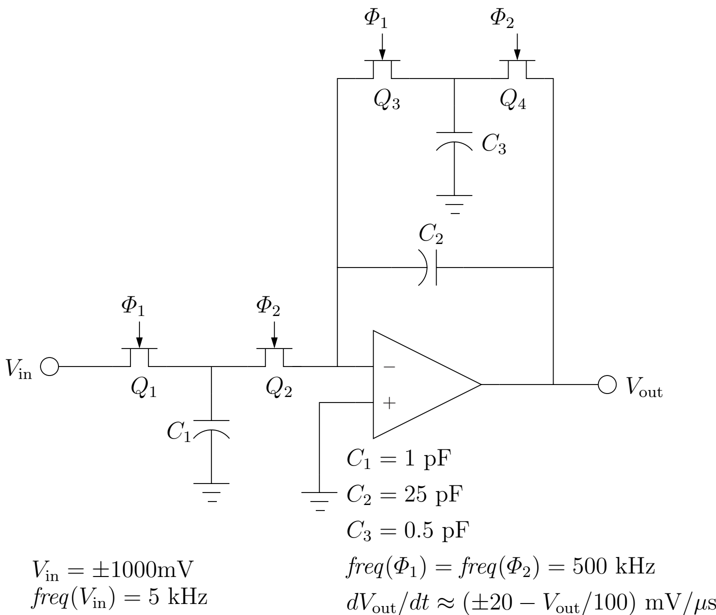


Fig. 5. Circuit diagram of a corrected switched capacitor integrator

Table 4. Rates for V_{out} in the corrected integrator using two bins

Bin	Place	Range of rates	Comment
00	p_6	[18,32]	$V_{\text{in}} < 0V$; $V_{\text{out}} < 0V$
01	p_3	[9,22]	$V_{\text{in}} < 0V$; $V_{\text{out}} \geq 0V$
11	p_4	[-22,-9]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} \geq 0V$
10	p_5	[-32,-18]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} < 0V$

Table 5. Rates for V_{out} in the corrected integrator using four bins

Bin	Place	Range of rates	Comment
00	p_9	[23,32]	$V_{\text{in}} < 0V$; $V_{\text{out}} < -500mV$
01	p_7	[18,27]	$V_{\text{in}} < 0V$; $-500mV \leq V_{\text{out}} < 0V$
02	p_5	[14,22]	$V_{\text{in}} < 0V$; $0 \leq V_{\text{out}} < 500mV$
03	p_3	[9,16]	$V_{\text{in}} < 0V$; $V_{\text{out}} \geq 500mV$
10	p_{10}	[-16,-9]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} < -500mV$
11	p_8	[-22,-14]	$V_{\text{in}} \geq 0V$; $-500mV \leq V_{\text{out}} < 0V$
12	p_6	[-27,-18]	$V_{\text{in}} \geq 0V$; $0 \leq V_{\text{out}} < 500mV$
13	p_4	[-32,-23]	$V_{\text{in}} \geq 0V$; $V_{\text{out}} \geq 500mV$

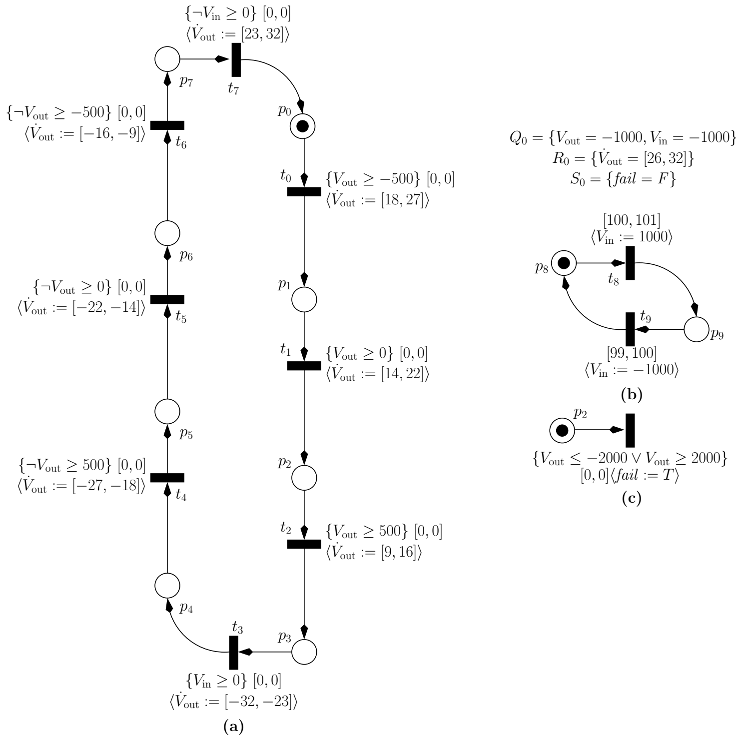


Fig. 6. LHPN for the corrected switched Integrator Example

capacitor resistor, the rate of change is now very dependent on the value of V_{out}. In particular, this variation slows the rate of the voltage change as it approaches the power supply rail. This prevents saturation of the integrator. Based on this knowledge, the thresholds on V_{out} are changed to -500mV, 0V, and 500mV. These new thresholds result in the rates shown in Table 5. The LHPN for this table is shown in Fig. 6, and this LHPN is found to satisfy the property in less than a second using the DBM model checker. Finally, to explore the scalability of our algorithms, Table 6 shows how the size of the LHPN and model checking time scales as the number of thresholds increases.

Table 6. Scalability of model checking as number of thresholds increase

No. thresholds	Places	Transitions	Model Checking Time
1	7	7	0.03s
3	11	11	0.06s
5	14	14	0.19s
7	18	19	0.31s
9	22	27	0.62s

6 Coverage Metrics

Our proposed method takes simulation traces from the designer and generates an LHPN. While the generated LHPN represents the behaviors that the designer deems to be important, it may miss problems not foreseen by the designer. Therefore, coverage metrics would be very useful to warn the designer about these unexplored portions of the state space where pitfalls may lie. Coverage information gives a quantitative metric about the quality of a set of simulation traces. This promises to aid the simulation only verification methodology as well as our model generation. We propose a coverage metric where each simulation trace is given a score. A higher score is achieved by a simulation trace that exhibits behavior not previously seen. From the perspective of the LHPN model some obvious examples of new behavior are entering a previously unvisited bin, taking a previously untaken bin to bin transition, or altering the overall rate of a bin. More complex measures of new behavior could be used such as the distance of the new trace from previously seen traces. A metric of this type gives a qualitative measure of the utility of an additional simulation trace. This type of metric could be used as an aid to determine the benefit of doing further simulations. A global metric for the entire set of simulation traces is also useful and could be created in a similar fashion.

For the integrator example, using just the simulation trace shown in Table 1 with C_2 equal to 23pF would result in the LHPN shown in Fig. 3. Adding the simulation trace shown in Table 2 with C_2 equal to 27pF results in the exact same LHPN structure, but the ranges of rate for V_{out} would be changed. Therefore, the value of the second trace run is less than that of the first, but it still has some value. Finally, if a third trace with C_2 equal to 25pF is added at this point, the

resulting LHPN would not change at all as the rates generated from this trace would be contained in those generated from the first two. Therefore, this trace adds no new knowledge, so the coverage metric would say that it has no value. As a final example, if a trace is added that changes V_{in} at twice the frequency (i.e., every 50μs), it now becomes possible for V_{in} to change before V_{out} goes above 0V. This means that the LHPN generated would now have a new transition from p_6 to p_5. This LHPN would also have a wider range of delays for when V_{in} changes. Therefore, this additional trace provides new information.

7 Conclusion

Interest in formal and semi-formal methods for validating AMS circuits is increasing. Many of these methods are seriously handicapped by the difficulty of generating formal models. This paper develops a method to generate a conservative, trace preserving formal LHPN model from a set of simulation traces and thresholds on the state space. This LHPN model can be used by several different model checking engines to prove safety properties about the entire state space of the model. Using two variations of the switched capacitor integrator circuit, this paper shows how an adequate LHPN model can be created using two simulation traces and a basic set of thresholds. The model is analyzed using a DBM based model checker to obtain the expected verification results.

While the current version of the tool requires the user to provide thresholds, it would be interesting to explore automated methods to determine important thresholds. Our initial investigations into the autogeneration of thresholds attempt to increase the number of thresholds in regions where the rates change rapidly. Automatic generation of thresholds may also provide the designer with useful information about the circuit.

Another potential benefit of the method described in this paper is that an LHPN model can be translated into a VHDL-AMS or Verilog-AMS model. One problem for AMS designers is creation of abstract models of their circuit for use in a digital or mixed-mode simulation flow. Models can be created by hand but must be updated to remain consistent as circuits change. Using this method, the models could maintain their consistency by running the needed set of simulations after changes and regenerating the HDL model from the LHPN.

References

1. Hartong, W., Hedrich, L., Barke, E.: On discrete modeling and model checking for nonlinear analog systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 401–413. Springer, Heidelberg (2002)
2. Dang, T., Donzé, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid systems techniques. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 21–36. Springer, Heidelberg (2004)
3. Frehse, G., Krogh, B.H., Rutenbar, R.A.: Verifying analog oscillator circuits using forward/backward refinement. In: Proc. Design, Automation and Test in Europe (DATE), pp. 257–262. IEEE Computer Society Press, Los Alamitos (2006)

4. Little, S., Seegmiller, N., Walter, D., Myers, C., Yoneda, T.: Verification of analog/mixed-signal circuits using labeled hybrid petri nets. In: Proc. International Conference on Computer Aided Design (ICCAD), pp. 275–282. IEEE Computer Society Press, Los Alamitos (2006)
5. Walter, D., Little, S., Seegmiller, N., Myers, C.J., Yoneda, T.: Symbolic model checking of analog/mixed-signal circuits. In: Asia and South Pacific Design Automation Conference (ASPDAC), pp. 316–323 (2007)
6. Walter, D., Little, S., Myers, C.: Bounded model checking of analog and mixed-signal circuits using an SMT solver. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 66–81. Springer, Heidelberg (2007)
7. Donzé, A., Maler, O.: Systematic simulation using sensitivity analysis. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC. LNCS, vol. 4416, Springer, Heidelberg (2007)
8. Dang, T., Nahhal, T.: Randomized simulation of hybrid systems for circuit validation. Technical report, VERIMAG (May 2006)
9. Girard, A., Pappas, G.J.: Verification using simulation. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 272–286. Springer, Heidelberg (2006)
10. Fainekos, G.E., Girard, A., Pappas, G.J.: Temporal logic verification using simulation. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 171–186. Springer, Heidelberg (2006)
11. Dastidar, T.R., Chakrabarti, P.P.: A verification system for transient response of analog circuits using model checking. In: VLSI Design, pp. 195–200. IEEE Computer Society Press, Los Alamitos (2005)
12. Walter, D.C.: Verification of analog and mixed-signal circuits using symbolic methods. PhD thesis, University of Utah (May 2007)
13. Myers, C.J., Harrison, R.R., Walter, D., Seegmiller, N., Little, S.: The case for analog circuit verification. Electronic Notes Theoretical Computer Science 153(3), 53–63 (2006)
14. David, R., Alla, H.: On hybrid petri nets. Discrete Event Dynamic Systems: Theory and Applications 11(1–2), 9–40 (2001)
15. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) Hybrid Systems. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)

Automatic Merge-Point Detection for Sequential Equivalence Checking of System-Level and RTL Descriptions

Bijan Alizadeh and Masahiro Fujita

VLSI Design and Education Center (VDEC), University of Tokyo, Japan
alizadeh@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract. In this paper, we propose a novel approach to verify equivalence of C-based system level description versus Register Transfer Level (RTL) model by looking for merge points as early as possible to reduce the size of equivalence checking problems. We tackle exponential path enumeration problem by identifying merge points as well as equivalent nodes automatically. It will describe a hybrid bit- and word-level representation called Linear Taylor Expansion Diagram (LTED) [1] which can be used to check the equivalence of two descriptions in different levels of abstractions. This representation not only has a compact and canonical form, but also is close to high-level descriptions so that it can be utilized as a formal model for many EDA applications such as synthesis. It will then show how this leads to more effective use of LTED to verify equivalence of two descriptions in different levels of abstractions. We use LTED package to successfully verify some industrial circuits. In order to show that our approach is applicable to industrial designs, we apply it to 64-point Fast Fourier Transform and Viterbi algorithms that are the most computationally intensive parts of a communication system.

Keywords: Formal Verification, Sequential Equivalence Checking, System on a Chip (SoC), Communication System, Canonical Representation.

1 Introduction

As system on a chip (SoC) designs continue to increase in size and complexity, many companies have paid more attention to design hardware at higher levels of abstraction due to faster design changes and higher simulation speed. In this phase, a C-based high level specification is described and then refined to a Register Transfer Level (RTL) description by adding more and more implementation details at different steps. Therefore there is a significant increase in the amount of verification required to achieve functionally correct description at each step, if traditional dynamic techniques such as simulation are used. This has led to a trend away from dynamic approaches and therefore Sequential Equivalence Checking (SEC) methods have become very important to reduce time-to-market as much as possible. SEC is a process of formally proving functional equivalence of designs that may in general have sequentially different implementations. Examples of sequential differences span the space from

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 129–144, 2007.
© Springer-Verlag Berlin Heidelberg 2007

retimed pipelines, differing latencies and throughputs, and even scheduling and resource allocation differences.

A few approaches have been proposed to perform equivalence checking between C-based specification and RTL description. In symbolic simulation based approaches, loop and conditional statements need to be unrolled and then all paths through the code must be explored [2-7]. If dependencies exist between different iterations of a loop statement, it will increase the run time for symbolic simulation and degrades quality due to the exponential number of paths. For example consider C code of Fig. 1(a). After unrolling *for-loop*, corresponding to each *then* and *else* branch it is necessary to have two execution paths. In general for N number of iterations we have to enumerate 2^N paths and therefore exponential path enumeration problem occurs. On the other hand, the different results computed on the different paths must be tracked that will cause a blow-up in logic if lower level techniques such as BDDs and SAT solvers are utilized.

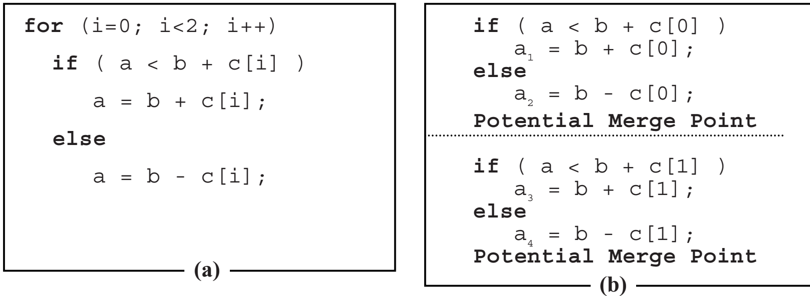


Fig. 1. Path enumeration of conditional statements (a) original source code (b) potential merge points to be detected

To cope with this complexity, the basic idea is to look for merge points as shown in Fig. 1(b), because it is obvious that two branches for *if-then-else* statements can be merged again. In this paper we not only attempt to figure out merge points automatically but also represent word-level arithmetic functions without requiring bit-level encoding due to use of a canonical hybrid bit- and word-level representation, i.e., LTED [1]. Furthermore, we point out how to check the equivalence of a C-based description against a RTL model while there is no information about corresponding equivalent points into two descriptions. Therefore, the main contributions of our paper are as follows:

- Automatic merge point detection as early as possible to overcome exponential path enumeration problem.
- Defining cut-planes (each cut-plane is a set of cut-points) as outputs of different iterations of loops in the C-based description and therefore finding equivalent nodes in the RTL model automatically, rather than specifying them in the two descriptions as done in [2].
- Efficient representation of the C-based description as well as the RTL model to reduce run time for checking their equivalence.

The rest of this paper is structured as follows. Related works are addressed in Section 2. LTED as a hybrid canonical representation is briefly described in Section 3. Automatic merge-point detection approach to check the equivalence between C-based and RTL descriptions is presented in Section 4 and 64-point Fast Fourier Transform and Viterbi decoder algorithms as two case studies are discussed in Section 5. Finally a brief conclusion and future work are shown in Section 6.

2 Related Works

Recently, some techniques have been proposed to apply equivalence checking to the system level and RTL descriptions [2-7]. In [2] an equivalence checking technique to verify system level design descriptions against their implementations in RTL was proposed. It presented an automatic technique to compute high level sequential compare points to compare variables of interest in the candidate design descriptions. They start the two design state machines at the same initial state and step the machines through every cycle, until a sequential compare point is reached. At this point the equivalence of the two state machines is proved using a lower (Boolean) level engine which is zChaff Satisfiability (SAT) solver. One of the limitations of this technique is not to be scalable in the number of cycles. As the number of cycles gets larger, the size of the expression grows quadratically, causing capacity problems for the lower level Satisfiability (SAT) engine. Furthermore it may not be applicable to large designs due to arithmetic encoding. In addition, in this technique corresponding equivalent points between two descriptions should be determined while these points may not be at all obvious due to complex control flow.

The authors in [3] have proposed early cut-point insertion for checking the equivalence of high level software against RTL of combinational components. They introduce cut-points early during the analysis of the software model, rather than after generating a low level hardware equivalent. In this way, they overcome the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. However, it is necessary to synthesize word level information into bit level because of using BDD to represent the symbolic expressions and so the capacity is limited by memory and run time requirements. In addition, it has only focused on combinational equivalence checking and has not addressed how to extend the proposed method for sequential equivalence checking problem. Another approach to equivalence checking between C descriptions is presented in [4]. This approach detects the textual differences in the two target programs, and then performs a dependence analysis using program slicing, to check for the actual differences in the two programs. It then symbolically simulates this difference and reports the equivalence checking results. Since this process uses syntactic information, the similarity of the target descriptions is very essential to its application.

A solution with a C-based bounded model checking (CBMC) engine was proposed in [6] that takes a C program and a Verilog implementation. They described an innovative method to convert the C program, including pointers and nested loops, into Boolean formulas. The Verilog code is also converted to Boolean formulas by a synthesis-like process. Then the two programs are converted into a Boolean satisfiability problem. Since this tool works entirely in the Boolean domain the

capacity of CBMC is limited by space and time considerations. In [7] a method of equivalence checking between the Finite State Machine with Data-path (FSMD) model of the high-level behavioral specification and the FSMD model of the behavior transformed by the scheduler has been proposed. In this method cut-points in one FSMD are introduced and then computations are visualized as concatenation of paths from cut-points to cut-points. Finally equivalent finite path segments in the other FSMD are identified. This technique, however, is not scalable due to its limited application.

In all above approaches BDD or SAT based methods are utilized to represent symbolic expressions while algorithmic specifications such as those for digital signal processing contain a lot of arithmetic operations that should be encoded into bit level operations. Thus lower-level techniques like BDD or SAT are not able to handle these designs due to the large number of Boolean variables or clauses to be generated. In order to improve Boolean SAT-based methods, a Hybrid Satisfiability approach (HSAT) has been introduced [8] to generate functional test vectors for RTL designs. This approach creates linear arithmetic constraints for arithmetic operators and conjunctive normal form (CNF) clauses for Boolean logical operators. It then uses 3-SAT checking to solve the logic equations and integer linear programming (ILP) solver to check the feasibility of the arithmetic equations separately, in different domains. Hence for variables correspond to the interaction between the Boolean and arithmetic domains of the design, an assignment is selected from the CNF-clauses, and the resulting constraints are propagated to the arithmetic domain for the linear program to check for consistency. If variable assignments that satisfy the CNF clauses cause the linear programming constraints in the arithmetic domain to be infeasible, backtracking is needed to select another set of Boolean assignments. Since these two engines operate in separate domains, the performance of HSAT is limited by the heuristics that choose the set of assignments to Boolean variables. In addition, although HSAT is able to model bit- and word-level expressions, it only deals with scalar multiplication due to using integer linear programming. On the contrary, in our previous works [1] and [9], we have proposed a canonical hybrid bit and word levels representation that integrates two domains in one engine and also represents two descriptions to be checked for equivalence, in a way that equivalent nodes could be found automatically without having to specify state or output mappings into two descriptions.

3 Hybrid Bit and Word Levels Representation

The goal of this section is to introduce a new graph-based representation called Linear Taylor Expansion Diagram (LTED) for functions with a mixed Boolean and integer domain and an integer range to represent arithmetic operations at a high level of abstraction, while other proposed *Word Level Decision Diagrams* (WLDDs) are graph-based representations that provide a concise representation of integer-valued functions defined over binary variables as a bit vector. A thorough review of WLDDs can be found in [10]. On the other hand, BDDs or SAT based methods suffer from size explosion problems when the designs grow in size and complexity. BDD-based

verification tools have not been very successful for designs containing large arithmetic data-path units due to prohibitive memory requirements.

In LTED, functions to be represented are maintained as a single graph in strongly canonical form. We assume that the set of variables is totally ordered and that all of the vertices constructed obey this ordering. Maintaining a canonical form requires obeying a set of conventions for vertex creation as well as weight manipulation. These conventions are similar to other word level canonical representations and are not discussed here for brevity. In contrast to TED, LTED is a binary graph-based representation where the algebraic expression F(X,Y,…) is expressed by a first-order linearization of the Taylor series expansion [1]. Suppose variable *X* is the top variable of F(X,Y,…). Equation (1) shows F(X,Y,…), where *const* is independent of variable *X*, while *linear* is coefficient of variable *X*.

$$F(X, Y, \ldots) = const + X*linear. \tag{1}$$

LTED data structure consists of a *Variable* node v that has as attributes an integer variable var(v) and two children const(v) and linear(v). In order to normalize the weights, any common factor is extracted by taking the greatest common divisor (gcd) of the argument weights. In addition, we adopt the convention that the sign of the extracted weight matches that of the *const* part. This assumes that gcd always returns a nonnegative value. Once the weights have been normalized the hash table is looked for an existing vertex or creates a new one. Similar to that of BDDs, each entry in the hash table is indexed by a key formed from the variable and the two children, i.e. *const* and *linear* parts. As long as all vertices are created, the graph will remain in strongly canonical form (see [1] for more details). Fig. 2 illustrates how the following multivariate polynomial expression is represented by LTED.

$$f(X, Y, Z) = 24-8*Z+12*Y*Z-6*X^2-6*X^2*Z$$

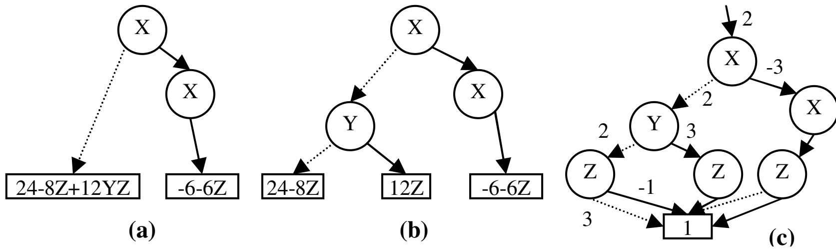


Fig. 2. LTED representation of $24-8*Z+12*Y*Z-6*X^2-6*X^2*Z$ (a) decomposition with respect to variable X (b) decomposition with respect to variables X and Y (c) decomposition with respect to variables X, Y and Z

Let the ordering of variables be X, Y and Z. First the decomposition with respect to variable X is taken into account. As shown in Fig. 2(a), *const* and *linear* parts will be $24-8*Z+12*Y*Z$ and $-6*X^2-6*X^2*Z$ respectively. After that, the decomposition is performed with respect to variable Y of Fig. 2(b). Finally the expressions are decomposed with respect to variable Z and a reduced diagram is depicted. In order to reduce the size of an LTED, redundant nodes are removed and isomorphic sub-graphs are merged as shown in Fig. 2(c). Analogous to TED and *BMDs, LTED is a

canonical representation. In this representation, dashed and solid lines indicate *const* and *linear* parts respectively.

It should be noted that LTED was introduced in [11] as a graph-based representation with application to formal property verification. In order to have a canonical from, all nodes introduced in [11] except *Constant* (C) and *Variable* (V) nodes have been removed. In this representation basic arithmetic operators such as addition, unary addition, subtraction, unary subtraction and multiplication are available that work for symbolic integer variables. In order to represent Boolean functions, logical bitwise operations including NOT, AND, and OR have been provided.

4 Sequential Equivalence Checking

In this section, we describe a sequential equivalence checking algorithm which is based on LTED canonical representation. Moreover, we will discuss merge point and cut-plane identification techniques.

4.1 Merge-Point and Cut-Plane Detection Approaches

Fig. 3 depicts our proposed equivalence checking algorithm. An algorithmic specification in C (ASC) and an RTL description in Verilog (RTL) are treated as inputs to the algorithm. Although set of cut-planes (C), set of variables that are interesting for observation, can be defined by user as done in [2], in this paper it is obtained automatically as outputs of different iterations of loop executions in the ASC as shown by three first lines of Fig. 3. As a matter of fact this automatic decomposition converts the original description to some simpler expressions that can be handled easier even though there are among data dependencies between different loop iterations.

As illustrated in Fig. 3, first of all a cut-plane is chosen. The nearest cut-plane to primary inputs is selected for better performance. A straight-forward way to do this is to sort cut-planes from primary inputs to primary outputs in the ASC. The selected cut-plane (CP) is removed from the set of cut-planes (C) and then all variables in CP are created in LTED. In order to detect merge-points of conditional statements such as *if-then-else statement* and *case statement* appeared in ASC description, variables from different branches of conditional statements, are rewritten by different indices (e.g., variable *n* is defined as n_1, n_2, … , n_m variables for *m* cases as shown in Fig. 3) and then added to CP.

On the other hand, RTL description is synthesized using a high-level synthesis tool and modeled by a Finite State Machine with Datapath (FSMD). The FSMD adds a datapath including variables and operators on communication to the classic FSM. The FSMD is represented as a transition table, where we assume each transition is executed in a single clock cycle. Operations associated with each transition of this model are executed in a sequential form. Each controller transition is defined by the current state, the condition to be satisfied and a set of operations or actions. The condition evaluated true will determine the transition to be done and thus the actions to be executed. In an *inner while loop*, the FSMD is traversed at the current cycle and all variables on the left hand side of the assignments are created in LTED. During representing by LTED, equivalent nodes will be found automatically due to canonical

form of LTED representation. At anytime during this process, if it is found that n_1, n_2, … , n_m are equivalent to some nodes in the RTL model, they will be merged as the original variable, i.e. *n*, and a primary input is introduced in its place. According to this explanation, we will be able to prevent exponential path enumeration problem since it is not necessary to consider different branches of the conditional statements. If equivalent nodes do not belong to $\{n_1, n_2, \dots, n_m\}$, we cut out the equivalent part and introduce new primary inputs in their places. These primary inputs are used while next iteration of an *outer while loop* is executed. In the *inner while loop*, the algorithm proceeds to the next state of RTL model until all variables in the selected cut-plane are checked their equivalence with some nodes in the ASC. In the *outer while loop*, however, the process repeats until no cut-plane is available. If we can carry on this process to outputs of the two descriptions, then we have formally verified equivalence.

```
Sequential_EC (ASC: Algorithmic Level Model; RTL: RTL Model)
  Cut-plane_i = Variables on the left hand side of the assignments in i^th iteration of a loop;
  C (set of Cut-planes) = ∪^(number of iterations) (Cut-plane_i)
  WHILE (C is not empty)
     Select a cut-plane (CP) and remove it from C (C = C – CP);
     ASC = Generate LTED representation of all variables in CP;
     IF (a conditional statement is encountered)
        FOR (each variable n on the left hand side of the assignments)
           Define n_1, n_2, … , n_m for m different cases instead of n;
           CP = CP ∪ {n_1, n_2, … , n_m};
           WHILE (CP is not empty)
              RTL (t) = Generate LTED representation of all variables are assigned to
                             at the current cycle (t) of RTL;
              IF (a set of variables (v) are assigned to at the current cycle)
                 RTL_v = Get LTED representation of v;
                 IF ((RTL_v is equivalent to some nodes in ASC) ⊆ CP)
                    CP = CP – v;
                    IF (v == {n_1, n_2, … , n_m})
                       Merge n_1, n_2, … , n_m points and introduce primary input;
                    ELSE
                       Introduce primary inputs at v and related nodes in ASC;
              Proceed to the next cycle;
```

Fig. 3. Sequential equivalence checking algorithm with cut-plane and merge-point detection

4.2 Example

Fig. 4 illustrates an example containing the heart of Viterbi decoder algorithm called Add-Compare-Select (ACS) block that will be discussed in detail in Section 5.2. The

C code of Fig. 4(a) indicates a high-level model of the ACS block, while another code of Fig. 4(b) describes different cycles of its RTL model. As soon as our proposed verification algorithm encounters *if-then-else* statement in the high-level model, it first represents $a_{01} = c_0$ (variable a_0 in *then* branch) and $a_{02} = c_1$ (variable a_0 in *else* branch) in LTED with respect to in_0, in_1, PI_0 and PI_1 inputs, as shown in Fig. 5(a). After that it is looking for equivalent nodes in the RTL model. At cycle t+2 of RTL model, it found out $e_{01} = f_1$ and $e_{02} = f_0$ that are equivalent to a_{02} and a_{01} respectively as depicted in Fig. 5(b). Therefore, a_0 and e_0 in the high-level and RTL models respectively, are detected as merge point and then can be taken into account as primary inputs in the rest of the two descriptions. In this work, we assume that the condition parts of different conditional statements can be checked using model checking methods and therefore are just skipped.

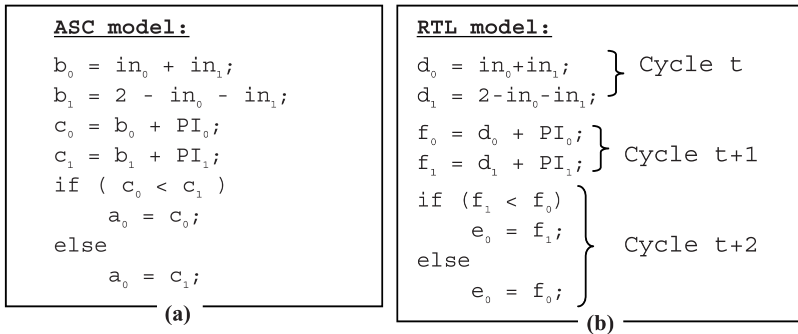


Fig. 4. ACS block in Viterbi benchmark (a) C-based model and (b) RTL model

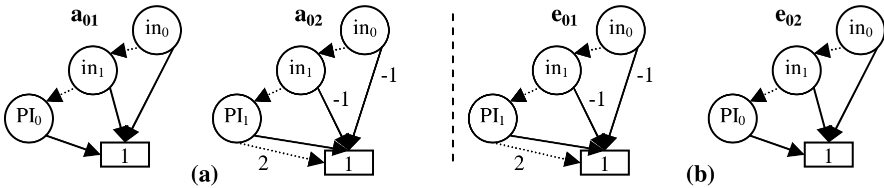


Fig. 5. LTED representations of variables in Fig. 4 (a) a_{01} and a_{02} (b) e_{01} and e_{02}

5 Case Studies

In recent years high speed wireless data communications has found many application areas. Fourth generation wireless and mobile systems are currently focusing on packet-based high-data-rate communication suitable for video transmission and mobile internet applications. Apart from the high speed of operation, the system demands lower power consumption. A general purpose DSP with associated software is not beneficial for this application since its power consumption is an order of magnitude higher compared to a dedicated hardware solution. Fig. 6 shows IEEE 802.11a transmitter and receiver where are many signal processing functions such as

Convolutional coder and inverse fast Fourier transform (IFFT) in the transmitter and, fast Fourier transform (FFT) and Viterbi decoder in the receiver. In addition, it has been shown [12] through extensive simulation that the most computationally intensive parts of such a high-data-rate system are the 64-point IFFT in the transmit direction and the Viterbi decoder in the receive direction. Therefore it is necessary to pay close attention to 64-point FFT and Viterbi decoder blocks.

In order to demonstrate that our approach is applicable to such a complete system solution with application to communication systems, we present experimental results of two case studies (1) 64-point Fast Fourier Transform (*FFT64*) and (2) Viterbi Decoder with K=3 (*Viterbi3*), K=7 (*Viterbi7*) and K=9 (*Viterbi9*). The important point to be noted here is that Boolean SAT based verification is not able to handle all benchmarks discussed here due to a huge number of Boolean variables or clauses to be generated after encoding arithmetic functions into bit-level operations.

General information about the benchmark circuits are given in Table 1. Column *benchmark* gives the benchmark's name, whereas column *#spec* provides the number of lines in C code after unrolling all loops. In column *#impl*, the number of lines in RTL code after synthesizing is reported. The fourth, fifth and sixth columns (*#add*, *#sub* and *#mul*) provide the number of additions, subtractions and multiplications required in each benchmark respectively. For *Viterbi3* benchmark, this information has been provided before (*Viterbi3bmp*) and after (*Viterbi3amp*) identifying merge-points. While before applying merge-point detection technique, the number of additions to be computed is 16474, after detecting merge points they have reduced to 391. For *Viterbi7* and *Viterbi9* benchmarks, it is not possible to prepare information before applying merge-point detection technique due to generating too many branches of ACS blocks. In section 5.2, we will see that 2^{42}-2 and 2^{54}-2 states should be processed for *Viterbi7* and *Viterbi9* respectively if merge-point detection technique is not used.

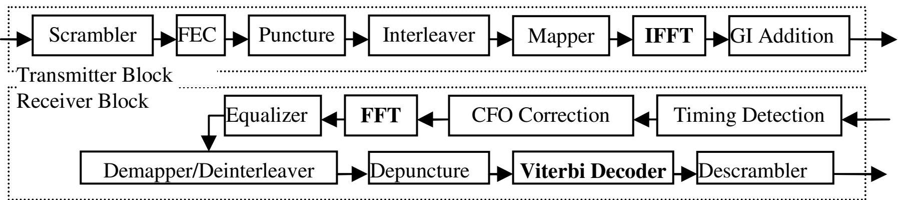


Fig. 6. Block diagram of 802.11a Transmitter and Receiver

Table 1. Industrial benchmark characteristics

Benchmark	#spec	#impl	#add	#sub	#mul
FFT64	1412	1640	1026	1026	1512
Viterbi3bmp	8357	16565	16474	192	0
Viterbi3amp	296	394	391	192	0
Viterbi7amp	4885	9397	9450	324	0
Viterbi9amp	23911	37810	47410	420	0

In the rest of this paper, experimental results are reported while the LTED package was implemented in C++ and has been carried out on an Intel 2.1GHz Core Duo and 1GByte of main memory running Windows XP.

5.1 64-Point FFT Benchmark

The first case study is 64-point Fast Fourier Transform (FFT64) which is one of the most computationally intensive building blocks in communication systems. Although the FFT64 is realized by decomposing it into a two-dimensional structure of 8-point FTTs to reduce the number of required multiplications compared to the conventional radix-2 FFT64, we consider the conventional radix-2 FFT64 in order to have the maximum number of multiplications. Fig. 7 illustrates N-point FFT algorithm which performs the butterfly computations with three main loops. An outside loop counts through the $\log_2(N)$ stages of the FFT computation and it causes huge data-dependent computations. Two inner loops perform the individual butterfly computations of each stage. The heart of this algorithm is the block of code that performs each butterfly computation in the third loop. In this figure, *wr* and *wi* parameters are commonly known as *twiddle factors* and can be computed before the algorithm is performed. But here we have considered them as symbolic variables rather than constant values to increase the number of arithmetic operations. Although there is no conditional statement in Fig. 7 for defining merge points, a lot of data-dependent computations exist that make this test-case a suitable benchmark for proving the claim that our approach is able to deal with real industrial designs even though it only has to determine some cut-planes rather than looking for merge points. As illustrated in Fig. 8, cut-planes have been defined as outputs of different iterations of the outer loop in Fig. 7. In Fig. 8, butterfly diagrams have been shown according to different iterations of the inner loops in Fig. 7.

Table 2 summarizes the results for two configurations, i.e., 64-point FFT without cut-planes (*FFT64nocp*) and 64-point FFT with cut-planes (*FFT64cp*). In this table, columns *#Nodes* and *#InputVar* give the number of LTED nodes and the number of input variables respectively. The memory and CPU time required for equivalence

```
for (s = 0 ; s < log2N ; s++)
  for (i = 0 ; i < N/(2^(s+1)) ; i++)
        C = wr[idx];  S = wi[idx];
        for (j = i ; j < N ; j + = N/2^s)
             tmpr = aar[idx] - aar[idx+N/2^(s+1)];
             tmpi = aai[idx] - aai[idx+N/2^(s+1)];
             aar[idx] = aar[idx] + aar[idx+N/2^(s+1)];
             aai[idx] = aai[idx] + aai[idx+N/2^(s+1)];
             aar[idx+N/2^(s+1)] = tmpr*C – tmpi*S;
             aai[idx+N/2^(s+1)] = tmpr*S + tmpi*C;
        idx = idx + 2^s;
```

Fig. 7. C code of N-point FFT benchmark

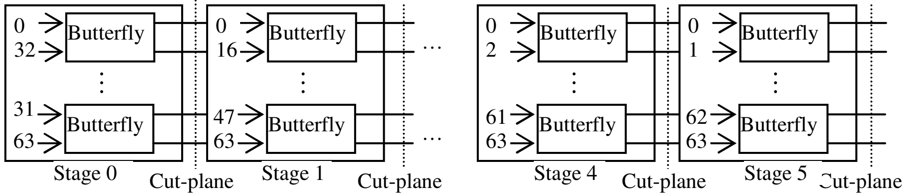


Fig. 8. Cut-planes defined in FFT64 benchmark

checking of the two descriptions are provided in columns *Memory Usage* in MByte and *Run Time* in seconds respectively. After identifying cut-planes as shown in Fig. 8, the number of LTED nodes will decrease to 1668 from 11220, while the number of inputs will increase from 190 to 830. In other words, 830-190 = 640 points were specified as equivalent parts and then new primary inputs were introduced in their places. As expected, after applying cut-plane detection technique, the run time required checking the equivalence between two descriptions has reduced from 3.5 seconds to 0.66 second. Moreover, the memory needed to generate LTED without looking for cut-planes are 10.8 MB, while after applying cut-plane detection method it was reduced to 1.3 MB.

Table 2. FFT64 benchmark experimental results

Type	#Nodes	#InputVar	Memory Usage	Run Time
FFT64nocp	11220	190	10.8	3.5
FFT64cp	1668	830	1.3	0.66

5.2 Viterbi Benchmark

Viterbi decoding is a technique for performing maximum likelihood sequence detection on data that has been convolutionally coded. The decoding problem is to determine the path with the minimum path metric through the trellis, with path metric being defined as the sum of the branch metrics along the path. This is done in a stepwise manner by processing a set of state metrics forward in time, stage by stage over the trellis as shown in Fig. 9. The complexity of the Viterbi algorithm lies in the computation of 2^{K-1} path metrics for a constraint length K decoder at each time stage. For the rate ½ codes (n=2) we are considering, there are just two predecessor states or branches for each state. Thus, state metric computation involves calculation of two branch metrics per state and then a selection of that branch which gives a smaller value of the new state metric. The former operation is done in the Branch Metric Unit (BMU) which takes in the received n-bit blocks of data and generates branch metrics by computing the distance between the received data and the actual codeword. The latter selection operation is performed by Add-Compare-Select (ACS) unit. The ACS unit takes in two state metrics and two branch metrics as input to yield an updated path metric. As the above process is performed, the selected or surviving branches for each state are recorded by storing one survivor bit per state at each trellis stage. The

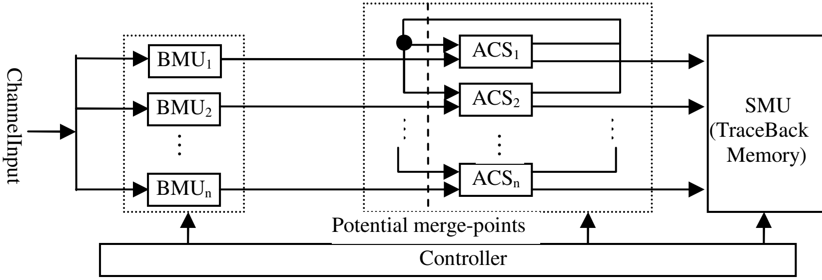


Fig. 9. Block diagram of Viterbi decoder

Survivor Management Unit (SMU) is responsible for tracing back through the trellis using the survivor bits to produce the input data bits.

In order to have a better understanding of Viterbi decoder algorithm consider pseudo code of Fig. 10 where an outside loop is repeated *ChannelLength= K*6-1* times. Inner loops run 2^{K-1} (the number of states) and n=2 (due to rate ½ codes) times respectively. In each iteration of inner loops, the branch metric (BrMetric[i][j]) is added to the current path metric using *Add part* of the ACS block, then two updated path metrics at each node (i.e., A1 and B1 in Fig. 10) are compared (*Compare part* of the ACS block) and finally the smaller is saved and the other is discarded (*Select part* of the ACS block). Thus, the essence of the Viterbi algorithm lies in the relatively simple operations of add, compare, select and trace-back which need to be applied to a large number of states.

To give a glimpse about the complexity and size of Viterbi decoder benchmark, we compute how many states need to be processed after unrolling conditional statements related to all ACS blocks if we do not try to look for merge points. In Fig. 10, it is necessary to check 2 states on the first iteration of the second loop nest, 2^2 states on the second iteration and finally 2^{K*6-1} states on the last iteration. Therefore the total number of states to be checked is $2+2^2+2^3+\ldots+2^{K*6-1} = 2^{K*6}-2$. For K=7 and K=9, they are $2^{42}-2$ and $2^{54}-2$ respectively which are large enough that methods mentioned in the

```
for (t = 0 ; t < ChannelLength ; t++)
   for (i = 0 ; i < 2^(K-1) ; i+=step)
      for (j=0 ; j < n ; j++)
         A1 = AcumErr[nextstate[i][j]][1];
         B1 = AcumErr[i][0]+BrMetric[i][j];
         if (A1 > B1)
               AcumErr[nextstate[i][j]][1] = B1;
               StateHistory[nextstate[i][j]][t] = i;
   for (i = 0 ; i < 2^(K-1) ; i++)
      AcumErr[i][0] = AcumErr[i][1];
      AcumErr[i][1] = MAXINTEGER;
```

Fig. 10. Pseudo code of Viterbi algorithm

literature are not able to handle them easily. After looking for merge points, however, the number of states to be processed are reduced to $2+2^2+\ldots+2^{K-2}+(2^{K-1}+\ldots+2^{K-1})=2*(2^{K-2}-1)+5*K*2^{K-1} = (5*K+1)*2^{K-1}-2$.

Cut-planes and Merge-points in Viterbi Benchmark. In this section we will discuss how to determine cut-planes and merge-points in the C-based description to reduce the size of equivalence checking problem. In Viterbi decoder the first K stages are different from other stages as shown in Fig. 11(a), where K is 7. This is because during the first K stages, there is only one path to achieve each next state from current state. For instance at t=1, there is only one way to reach next states 0, 16, 32 and 48. These stages are outputs of the corresponding iterations of the outer loop of Fig. 10 that are viable candidates to be cut-planes as illustrated in Fig. 11(a).

On the other hand, another decision flow exists for stages K+1 to 6*K-1, where each state can be reachable from two paths. One decision butterfly out of 32 pairs needed for Viterbi decoder K = 7, has been depicted in Fig. 11(b), where S varies from 0 to 31. In this figure each *circle* indicates a state and also corresponds to an ACS operation in Fig. 10. For instance consider state S that can be received through 2S and 2S+1 by different branch metrics. According to Viterbi algorithm described in Fig. 10, to compute accumulated error metric for this state, first of all *AcumErr[2S][0]+BrMetric0* is computed (B1) and then compared to *AcumErr[S][1]* (A1 of Fig. 10). Finally the smaller one is saved as a new value into *AcumErr[S][1]*. This process is repeated when B1 = *AcumErr[2S+1][0]+BrMetric2* is computed and compared to *AcumErr[S][1]*. As illustrated in Fig. 10, after completing the second loop nest, *AcumErr[S][1]* is saved into *AcumErr[S][0]* and gets a very large integer number, i.e., MAXINTEGER, because of beginning another iteration of an outer loop properly (see the fourth loop in Fig. 10). Obviously, each output of ACS units has the potential to be a merge point due to conditional statements.

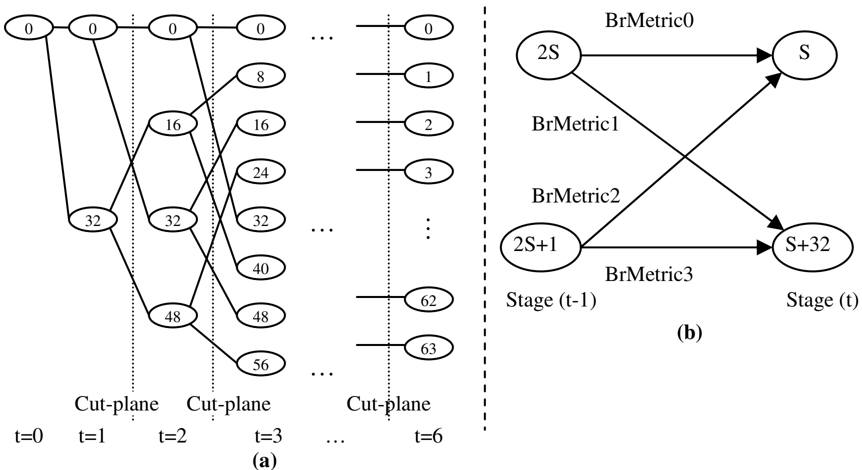


Fig. 11. (a) Seven first stages of Viterbi K=7 (b) Decision butterfly for ACS pair in Viterbi K=7

Experimental Results. Table 3 provides experimental results for six configurations of Viterbi decoder, i.e., Viterbi (K=3) without merge point detection (*Vitbi3nomp*), Viterbi (K=3) with merge point detection (*Vitbi3mp*), Viterbi (K=7) with merge point detection (*Vitbi7mp*), Viterbi (K=7) with merge point and cut-plane detection (*Vitbi7mpcp*), Viterbi (K=9) with merge point detection (*Vitbi9mp*) and Viterbi (K=9) with merge point and cut-plane detection (*Vitbi9mpcp*). In this table, rows *#Nodes* and *#Vars* give the number of LTED nodes and the number of input variables respectively. The memory usage and CPU time needed for equivalence checking of the two descriptions are presented in rows *Mem* (in Mega-Byte) and *Time* (in seconds) respectively. The second and third columns, i.e., *Vitbi3nomp* and *Vitbi3mp*, provide useful information before and after applying automatic merge point detection method to Viterbi K=3 test case. Obviously, in this case after finding merge points automatically, 90-24 = 66 new primary inputs (*#Vars* row in Table 3) have been introduced and the number of LTED nodes (*#Nodes*) has reduced from 52827 to 355. Moreover, memory and run time required for equivalence checking have been reduced from 36.3 MB to 0.4 MB and 57.8 seconds to 0.1 second respectively.

Columns *Vitbi7mp* and *Vitbi7mpcp* in Table 3 represent experimental results of Viterbi K=7. Although we are not sure that LTED package is able to handle this case without merge point detection, the task of preparing the input file for this package is very difficult because it needs to duplicate the number of states on each iteration where the number of iterations and the number of states on the first iteration are K*6-1 = 41 and 2^{K-1} = 64 respectively. Thus here we only report experimental result of Viterbi K=7 after applying merge point detection technique where memory usage and CPU time required to perform equivalence checking are 6.9 MB and 12.6 seconds. While after defining cut-planes, as shown in column *Vitbi7mpcp* of Table 3, they have been reduced to 6 MB and 12 seconds respectively. Fortunately the case study in [2] was Viterbi K=7 that makes it possible to compare results without spending a lot of time to apply Viterbi K=7 to SAT based methods. The authors in [2] have used zChaff as a SAT solver to check the equivalence between expressions computed at every cycle of RTL model and expressions achieved from C-based description. They gave a breakdown of number of clauses in the CNF formula for various blocks. Table 4 provides experimental results of our method in comparison with proposed method in [2]. Although they reported that without their decomposition method, the monolithic Trellis computation would generate a CNF with nearly 1.9 million clauses, after using the decomposed technique, they created 32 independent CNF formulas that were input to zChaff. Each of these formulas had 59136 clauses and 128 variables. In addition the number of clauses in the CNF formula for Trellis computation per butterfly was 57344, while in our method it requires 352 LTED nodes, 0.28 MB memory and 0.06 second run time to check the equivalence between butterflies in the two descriptions. There was no report of memory usage and CPU time for SAT based method proposed in [2], so related entries was left blank in Table 4.

The two last columns in Table 3 give experimental results of Viterbi K=9. After applying merge-point technique, in order to verify the equivalence of two descriptions, 66075 LTED nodes was generated and LTED package spent 190 seconds run time while the memory manager reported that 27.3MB RAM was consumed. This case proves scalability of our approach in comparison with method in [2] that was only applied to Viterbi K=7 and it cannot deal with Viterbi K=9 due to computational explosion problem of lower level SAT-based methods.

Table 3. Experimental results of Viterbi benchmark

Type	*Vitbi3nomp*	*Vitbi3mp*	*Vitbi7mp*	*Vitbi7mpcp*	*Vitbi9mp*	*Vitbi9mpcp*
#Nodes	52827	355	13279	12665	66075	64761
#Vars	24	90	2258	2384	11627	11881
Mem	36.3	0.4	6.9	6	27.3	26.5
Time	57.8	0.1	12.6	12	190	178

Table 4. Experimental results of Trellis computation per butterfly in Viterbi benchmark

Technique	**#Nodes**	**#Var**	**Memory (MByte)**	**Time (Sec)**	**#add**	**#sub**
Our Method	352	66	0.28	0.06	262	8
Method in [2]	57344	66	---	---	---	---

6 Conclusion and Future Work

In this paper, we proposed an automatic merge-point detection technique based on an hybrid bit- and word-level canonical representation called LTED. Then we have used it to check the equivalence between C-based specification and RTL implementation of two large industrial circuits, i.e., 64-point FFT algorithm (FFT64) and Viterbi decoder K=3, 7, 9. This representation is strong enough to handle arithmetic operations at word level representation and there is no need to encode them to bit-level operations. As opposed to low level methods such as Boolean SAT based techniques reported in the literature, the empirical results indicate that our approach not only uses an efficient canonical form to represent symbolic expressions but also is scalable even on large industrial circuits.

Obvious direction for future work is to integrate LTED package with a SpecC environment to address the equivalence checking between different abstractions of SpecC as a system level language.

Acknowledgement

This work was supported in part by Semiconductor Technology Academic Research Center (STARC).

References

1. Alizadeh, B., Fujita, M.: LTED: A Canonical and Compact Hybrid Word-Boolean Representation as a Formal Model for Hardware/Software Co-designs. In: CFV 2007. The fourth Workshop on Constraints in Formal Verification, pp. 15–29 (2007)
2. Vasudevan, S., Viswanath, V., Abraham, J., Tu, J.: Automatic Decomposition for Sequential Equivalence Checking of System Level and RTL Descriptions. In: MemoCode 2006. Proceedings of Formal Methods and Models for Co-Design, pp. 71–80 (2006)

3. Feng, X., Hu, A.: Early Cutpoint Insertion for High-Level Software vs. RTL Formal Combinational Equivalence Verification. In: DAC 2006. Proceedings of 43th Design Automation Conference, pp. 1063–1068 (2006)
4. Matsumoto, T., Saito, H., Fujita, M.: Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In: ISQED 2006. Proceedings of 7th International Symposium on Quality Electronic Design, pp. 370–375 (2006)
5. Koelbl, A., Lu, Y., Mathur, A.: Embedded tutorial: Formal Equivalence Checking Between System-level Models and RTL. In: Proceedings of ICCAD 2005, pp. 965–971 (2005)
6. Kroening, D., Clarke, E., Yorav, K.: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In: DAC 2003. Proceedings of 40th Design Automation Conference, pp. 368–371 (2003)
7. Karfa, C., Mandal, C., Sarkar, D., Pentakota, S.R., Reade, C.: A Formal Verification Method of Scheduling in High-level Synthesis. In: ISQED 2006. Proceedings of 7th International Symposium on Quality Electronic Design, pp. 71–78 (2006)
8. Fallah, F., Devadas, S., Keutzer, K.: Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In: DAC 1998. Proceedings of 35th Design Automation Conference, pp. 528–533 (1998)
9. Alizadeh, B., Fujita, M.: A Hybrid Approach for Equivalence Checking Between System Level and RTL Descriptions. In: IWLS 2007. 16th International Workshop on Logic and Synthesis, pp. 298–304 (2007)
10. Horeth, S., Drechsler, R.: Formal Verification of Word-Level Specifications. In: DATE 1999. Proceedings of Design Automation and Test in Europe, pp. 52–58 (1999)
11. Alizadeh, B., Navabi, Z.: Word Level Symbolic Simulation in Processor Verification. IEE Proceedings Computers and Digital Techniques Journal 151(5), 356–366 (2004)
12. Grass, E., Tittelbach, K., Jagdhold, U., Troya, A., Lippert, G., Krueger, O., Lehmann, J., Maharatna, K., Fiebig, N., Dombrowski, K., Kraemer, R., Aehoenen, P.: On the Single Chip Implementation of a Hiperlan/2 and IEEE802.11a Capable Modem. IEEE Pers. Commun. 8, 48–57 (2001)

Proving Termination of Tree Manipulating Programs

Peter Habermehl[1], Radu Iosif[2], Adam Rogalewicz[2,3], and Tomáš Vojnar[3]

[1] LSV, ENS Cachan, CNRS, INRIA; 61 av. du Président Wilson, F-94230 Cachan, France
haberm@liafa.jussieu.fr
[2] VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières
{iosif,rogalewi}@imag.fr
[3] FIT BUT, Božetěchova 2, CZ-61266, Brno
vojnar@fit.vutbr.cz

Abstract. We consider the termination problem of programs manipulating tree-like dynamic data structures. Our approach is based on a counter-example guided abstraction refinement loop. We use abstract regular tree model-checking to infer invariants of the program. Then, we translate the program to a counter automaton (CA) which simulates it. If the CA can be shown to terminate using existing techniques, the program terminates. If not, we analyse the possible counterexample given by a CA termination checker and either conclude that the program does not terminate, or else refine the abstraction and repeat. We show that the spuriousness problem for lasso-shaped counterexamples is decidable in some non-trivial cases. We applied the method successfully on several interesting case studies.

1 Introduction

Verification of programs with dynamic linked data structures is a difficult task, among other reasons, due to the use of unbounded memory, and the intricate nature of pointer manipulations. Most of the approaches existing in this area concentrate on checking safety properties such as, e.g., absence of null pointer dereferences, preservation of shape invariants, etc. In this paper, we go further and tackle the universal termination problem of programs manipulating tree data structures. Namely, we are interested in proving that such a program terminates for any input tree out of a given set described as an infinite regular tree language over a finite alphabet.

We handle sequential, non-recursive programs working on trees with parent pointers and data values from a finite domain. The basic statements we consider are data assignments, non-destructive pointer assignments, and tree rotations. This is sufficient for verifying termination of many practical programs over tree-shaped data structures (e.g., AVL trees or red-black trees) used, in general, for storage and a fast retrieval of data. Moreover, many programs working on singly- and doubly-linked lists fit into our framework as well. We do not consider dynamic allocation in this version of the paper, but insertion/removal of leaf nodes, common in many practical tree manipulating programs, can be easily added, if not used in a loop.

We build on *Abstract Regular Tree Model Checking* (ARTMC) [5], a generic framework for symbolic verification of infinite-state systems, based on representing regular sets of configurations by finite tree automata, and program statements as operations

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 145–161, 2007.
© Springer-Verlag Berlin Heidelberg 2007

on tree automata. We represent a given program as a control flow graph whose nodes are annotated with (overapproximations of) sets of reachable configurations computed using ARTMC. From the annotated control flow graph, we build a counter automaton (CA) that simulates the program. The counters of the CA keep track of different measures within the working tree: the distances from the root to nodes pointed to by certain variables, the sizes of the subtrees below such nodes, and the numbers of nodes with a certain data value. Termination of the CA is analysed by existing tools, e.g., [8,12,23].

Our analysis uses a *Counter-example Guided Abstraction Refinement* (CEGAR) loop [10]. If the tool we use to prove termination of the CA succeeds, this implies that the program terminates on any input from the given set. Otherwise, the CA checker tool outputs a lasso-shaped counterexample. For the class of CA generated by our translation scheme, we prove that it is decidable whether there exists a non-terminating run of the CA over the given lasso[1].

However, even if we are given a real lasso in the generated CA, due to the abstraction involved in its construction, we still do not know whether this implies also non-termination of the program. We then map the lasso over the generated CA back into a lasso in the control of the program, and distinguish two cases. If (1) the program lasso does not contain tree rotations, termination of all computations along this path is decidable. Otherwise, (2) if the lasso contains tree rotations, we can decide termination under the additional assumption that there exists a CA (not necessarily known to us) that witnesses termination of the program (i.e., intuitively, in the case when the tree measures we use are strong enough to show termination). In both cases, if the program lasso is found to be spurious, we refine the abstraction and generate a new CA from which an entire family of counterexamples (including this particular one) is excluded.

The analysis loop is not guaranteed to terminate even if the given program terminates due to the fact that our problem is not recursively enumerable. However, experience with our implementation of a prototype tool shows that the method is successfully applicable to proving termination of various real-life programs.

All proofs and more details can be found in the full version [18] of the paper.

Contributions of the Paper: (1) We have developed a systematic translation of programs working on trees into counter automata; the translation is based on an adequate choice of measures that map parts of the memory structures into positive integers. (2) We provide a new CEGAR loop for refining the translation of programs into counter automata on demand. (3) We present new decidability results for the spuriousness problem of lasso-shaped counterexamples for both counter automata and programs with trees. (4) We have implemented our techniques on top of the existing framework of *Abstract Regular Tree Model Checking*; our tool can handle examples of tree manipulating programs that, to the best of our knowledge, are not handled by any existing tool.

Related Work. The area of research on automated verification of programs manipulating dynamic linked data structures is recently quite live. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have been proposed. They are based on, e.g., monadic second order logic

[1] If the analyser used returns a spurious lasso-shaped counterexample for the termination of the CA, we suggest choosing another tool.

[21], 3-valued predicate logic with transitive closure [24], separation logic [22,17], or finite automata [16,6].

With few exceptions, all existing verification methods for programs with recursive data structures tackle verification of safety properties. In [1,25], specialised ranking functions over the number of nodes reachable from pointer variables are used to verify termination of programs manipulating linked lists. Termination of programs manipulating lists has further been considered in [17,4] using constraints on the lengths of the list segments not having internal nodes pointed from outside. To the best of our knowledge, automated checking of termination of programs manipulating trees has so-far been considered in [20] only, where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic.

In the past several years, a number of industrial-scale software model checkers such as SLAM [2], BLAST [19], or MAGIC [9] were built using the CEGAR approach [10]. However, these tools consider verification of safety properties only. On what concerns termination, CEGAR was applied in [12,13], and implemented in the TERMINATOR [14] and ARMC [23] tools. Both of these tools are designed to prove termination of integer programs without recursive data structures.

Concerning termination of programs with recursive data structures, the available termination checkers for integer programs can be used *provided* that there is a suitable abstraction of such programs into programs over integers, i.e., counter automata. Such abstraction can be obtained by recording some numerical characteristics of the heap in the counters, while keeping the qualitative properties of the heap in the control of the CA. Indeed, this is the approach taken in [4] for checking termination of programs over singly-linked lists. The abstraction used in [4] is based on compacting each list segment into a single abstract node and recording its length in the counters of the generated CA. The number of abstract heap graphs that one obtains this way is finite (modulo the absence of garbage)—therefore they can be encoded in the control of the CA. The translation produces a CA that is *bisimilar* to the original program, and therefore any (positive or negative) result obtained by analysing the CA holds for the program.

However, in the case of programs over trees, one cannot use the idea of [4] to obtain a bisimilar CA since the number of branching nodes in a tree is unbounded. Therefore, the translation to CA that we propose here loses some information about the behaviour of the program, i.e., the semantics of the CA overapproximates the semantics of the original program. Then, if a spurious non-termination counterexample is detected over the generated CA, the translation is to be refined. This refinement is done by a specialised CEGAR loop that considers also structural information about the heaps. To the best of our knowledge, no such CEGAR loop was proposed before in the literature.

As said already above, the approach of [17] is similar to [4] in that it tracks the length of the list segments. However, it does not generate a CA simulating the original program. Instead, it first obtains invariants of the program (using separation logic) and then computes the so-called variance relations that say how the invariants change within each loop when the loop is fired once more. When the computed variance relations are well-founded, termination of the program is guaranteed. Unlike the approach of [4] (bisimulation preserving) and the approach we present here (based on CEGAR), the analysis of [17] fails if the initial abstraction is not precise enough.

The approach of [17] was recently generalised in [3] to a general framework that one can use to extend existing invariance analyses to variance analyses that can in turn be used for checking termination. Up to now, this framework has not been instantiated for programs with trees (by providing the appropriate measures and their abstract semantics). Moreover, it is not clear how the variance analysis framework fits with the CEGAR approach.

2 Preliminaries

Programs with Trees. We consider sequential, non-recursive C-like programs working over tree-shaped data structures with a finite set of pointer variables *PVar*. Each node in a tree contains a data value from a finite set *Data* and three next pointers, denoted `left`, `right`, and `up`.[2] For $x, y \in PVar$ and $d \in Data$, we allow the following statements: (1) assignments $x = \texttt{null}$, $x = y$, $x = y.\{\texttt{left}|\texttt{right}|\texttt{up}\}$, $x.\texttt{data} = d$, (2) conditional statements and loops based on the tests $x == \texttt{null}$, $x == y$, $x.\texttt{data} == d$, and (3) the standard left and right tree rotations [15] (cf. Figure 1). This syntax covers a large set of practical tree-manipulating procedures. For technical reasons, we require w.l.o.g. that *no statements take the control back to the initial line*.

Memory configurations of the considered programs can be represented as trees with nodes labelled by elements of the set $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$—a node is either null and represented by $\square$ or it contains a data value and a set of pointer variables pointing to it. Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree). Let $\mathcal{T}(\mathcal{C})$ be the set of all such trees and *Lab* the set of all program lines. A configuration of a program is a pair $\langle l, t \rangle \in Lab \times \mathcal{T}(\mathcal{C})$. For space reasons, the semantics of the program statements considered is given in [18].

Some program statements may influence the counters of the CA that we build to simulate programs in several different ways. For instance, after $x = x.\texttt{left}$, the distance of x from the root may increase by one, but it may also become undefined (which we represent by the special value -1) if $x.\texttt{left}$ is null. Similarly, a single rotation statement may change the distance of a node pointed by some variable from the root in several different ways according to where the node is located in a particular input tree. For technical reasons related to our abstraction refinement scheme, we need a one-to-one mapping between actions of a program and the counter manipulations simulating them. In order to ensure the existence of such a mapping, we decompose each program statement into several *instructions*. The semantics of a statement is the union of the semantics of its composing instructions, and exactly one instruction is always executable in each program configuration.

In particular, the assignments $x = \texttt{null}$ and $x = y$ are instructions. Conditional statements of the form $x == \texttt{null}$ and $x == y$ are decomposed into two instructions each, corresponding to their true and false branches. A conditional statement $x.\texttt{data} == d$ is decomposed into three instructions, corresponding to its true and false branches, and an error branch for the case $x == \texttt{null}$. Each statement $x = y.\texttt{left}$ is decomposed into instructions $\texttt{goLeftNull}(x, y)$ for the case when $y.\texttt{left} == \texttt{null}$, $\texttt{goLeftNonNull}(x, y)$ for the case $y.\texttt{left}! = \texttt{null}$, and $\texttt{goLeftErr}(x, y)$ for the case of a null pointer

[2] A generalisation of our approach to trees with another arity is straightforward.

dereference. The statements $x = y.\texttt{right}$ and $x = y.\texttt{up}$ are treated in a similar way. The statements $x.\texttt{data} = d$ are decomposed into a set of instructions $\texttt{changeData}(x,d',d)$ for all $d' \in Data$. A special instruction $\texttt{changeDataErr}(x)$ for the null pointer dereference is also introduced.

Finally, a left rotation on a node pointed by a variable $x \in PVar$ is decomposed into a set of instructions $\texttt{leftRotate}(x,X,Y,A,B)$ where X contains variables aliased to x, Y variables pointing to the right son of x, A variables pointing inside the left subtree of x, and B variables pointing into the right subtree of the right son of x (Figure 1). The instruction $\texttt{leftRotateErr}(x)$ is introduced for the case of a null dereference within the rotation. Right rotations are decomposed analogously.

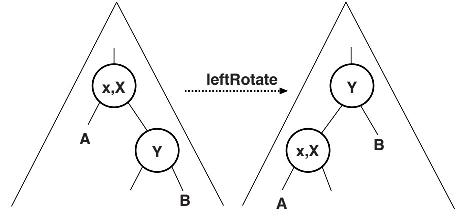


Fig. 1. $\texttt{leftRotate}(x,X,Y,A,B)$

Given a program P, we denote by $Instr$ the set of instructions that appear in P and by $\langle l,t\rangle \xrightarrow[P]{i} \langle l',t'\rangle$ the fact that P has a transition from $\langle l,t\rangle$ to $\langle l',t'\rangle$ caused by firing an instruction $i \in Instr$. By $i(t)$ we denote the effect of i on a tree $t \in \mathcal{T}(\mathcal{C})$. We denote by $\xrightarrow[P]{}$ the union $\bigcup_{i \in Instr} \xrightarrow[P]{i}$, and by $\xrightarrow[P]{*}$ the reflexive and transitive closure of $\xrightarrow[P]{}$. For $i \in Instr$ and $I \subseteq \mathcal{T}(\mathcal{C})$, let $post(i,I) = \{i(t) \mid t \in I\}$. We also generalise $post$ to sequences of instructions.

Counter Automata. For an arithmetic formula φ, let $FV(\varphi)$ denote the set of free variables of φ. For a set of variables X, let $\Phi(X)$ denote the set of arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$. If $\nu : X \to \mathbb{Z}$ is an assignment of $FV(\varphi) \subseteq X$, we denote by $\nu \models \varphi$ the fact that ν is a satisfying assignment of φ.

A counter automaton (CA) is a tuple $A = \langle X, Q, q_0, \varphi_0, \to\rangle$ where X is the set of counters, Q is a finite set of control locations, $q_0 \in Q$ is a designated initial location, φ_0 is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing the initial assignments of the counters, and $\to \in Q \times \Phi(X) \times Q$ is the set of transition rules.

A configuration of a CA is a pair $\langle q, \nu\rangle \in Q \times (X \to \mathbb{Z})$. The set of all configurations is denoted by $\mathfrak{C}$. The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined by $(q,\nu) \xrightarrow[A]{\varphi} (q',\nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of $FV(\varphi)$, where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\sigma \models \varphi$ and $\nu(x) = \nu'(x)$ for all variables x with $x' \notin FV(\varphi)$. We denote by $\xrightarrow[A]{}$ the union $\bigcup_{\varphi \in \Phi} \xrightarrow[A]{\varphi}$, and by $\xrightarrow[A]{*}$ the reflexive and transitive closure of $\xrightarrow[A]{}$. A *run* of A is a sequence of configurations $(q_0,\nu_0),(q_1,\nu_1),(q_2,\nu_2)\ldots$ such that $(q_i,\nu_i) \xrightarrow[A]{} (q_{i+1},\nu_{i+1})$ for each $i \geq 0$ and $\nu_0 \models \varphi_0$. We denote by $\mathfrak{R}_A$ the set of all configurations reachable by A, i.e., $\mathfrak{R}_A = \{(q,\nu) \mid (q_0,\nu_0) \xrightarrow[A]{*} (q,\nu) \text{ for some } \nu_0 \models \varphi_0\}$.

3 The Termination Analysis Loop

Our termination analysis procedure based on abstraction refinement is depicted in Fig. 2. We start with the control flow graph (CFG) of the given program and use ARTMC to

generate invariants for its control points. Then, the CFG annotated with the invariants (an abstract CFG, see Section 4) is converted into a CA[3], which is checked for termination using an existing tool (e.g., [23]). If the CA is proved to terminate, termination of the program is proved too. Otherwise, the termination analyser outputs a lasso-shaped counterexample. We check whether this counterexample is real in the CA—if not, we suggest the use of another CA termination checker (for brevity, we skip this in Fig. 2). If the counterexample is real on the CA, it is translated back into a sequence of program instructions and analysed for spuriousness on the program. If the counterexample is found to be real even there, the procedure reports non-termination. Otherwise, the program CFG is refined by splitting some of its nodes (actually, the sets of program configurations associated with certain control locations), and the loop is reiterated. Moreover, ARTMC may also be re-run to refine the invariants used (as briefly discussed in Section 6).

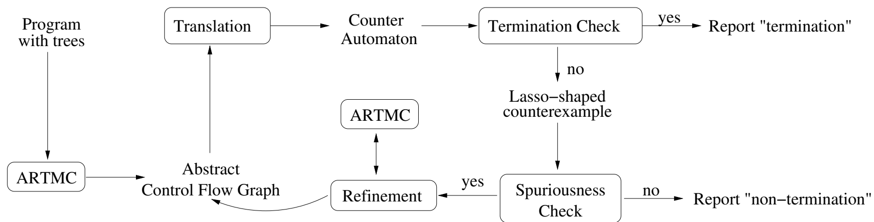


Fig. 2. The proposed abstract-check-refine loop

If our termination analysis stops with either a positive or a negative answer, the answer is exact. However, we do not guarantee termination for any of these cases. Indeed, this is the best we can achieve as the problem we handle is not recursively enumerable even when destructive updates (i.e., tree rotations) are not allowed. This can be proved by a reduction from the complement of the halting problem for 2-counter automata.

Theorem 1. *The problem whether a program with trees without destructive updates terminates on any input tree is not recursively enumerable.*

Therefore we do not further discuss termination guarantees for our analysis procedure in this paper, and postpone the research on potential partial guarantees, in some restricted cases, for the future. However, despite the theoretical limits, the experimental results reported in Section 7 indicate a practical usefulness of our approach.

ARTMC. We use *abstract regular tree model checking* to overapproximate the sets of configurations reachable at each line of a program (i.e., to compute *abstract invariants* for these lines) and also to check that the program is *free of basic memory inconsistencies* like null pointer dereferences. Due to space limitations, we only give a very brief overview of ARTMC here—more details can be found in [6,18]. The idea is to represent each program configuration as a tree over a finite alphabet, regular sets of such configurations by finite tree automata, and program instructions as operations on tree automata.

[3] The use of invariants in the abstract CFGs allows us to remove impossible transitions and therefore improves the accuracy of the translation to CA.

Starting from a regular set of initial configurations, these operations are then iteratively applied until a fixpoint is reached. In order to make the computation stop, the sets of reachable configurations (i.e., finite tree automata) are abstracted at each step. Several abstraction schemes based on collapsing states of the encountered tree automata may be used. For example, the *finite-height abstraction* collapses the automata states that accept exactly the same trees up to some height. All the abstractions are finite-range, guaranteeing termination of the abstract fixpoint computation, and can be automatically refined (e.g., in the mentioned case, by increasing the abstraction height).

For the needs of ARTMC, we encode configurations of the considered programs simply as trees over the alphabet $\mathcal{C} = Data \times 2^{PVar} \cup \{\Box\}$. Most of the instructions can be encoded as structure-preserving tree transducers. A transducer can check conditions like $x == y$ or $x.\texttt{data} == d$ by checking node labels. Transducers can also be used to move symbols representing the variables to nodes marked by some other variable ($x = y$), remove a symbol representing a variable from the tree ($x = \texttt{null}$), move it one level up or down ($x = y.\{\texttt{left}|\texttt{right}|\texttt{up}\}$), or change the data element in the node marked by some variable ($x.\texttt{data} = d$). The rotations are a bit more complex. They cannot be implemented as tree transducers. However, they can still be implemented as special operations on tree automata. First, a test of the mutual positioning of the variables in the tree required by their distribution in the sets X, Y, A, B is implemented as an intersection with a tree automaton that remembers which variables were seen, and in which branches. Then, we locate the automata states that accept the tree node representing the root of the rotation (cf. Figure 1), their children, and their right grandchildren. Finally, we reconnect these states in the automaton control structure in order to match the semantics of the tree rotations.

4 Abstraction of Programs with Trees into Counter Automata

In this section, we provide a translation from tree manipulating programs to counter automata such that existing techniques for proving termination of counter automata can be used to prove termination of the programs. Before describing the translation, we define the simulation notion that we will use to formalise correctness of the translation.

Let P be a program with a set of instructions $Instr$, an initial label $l_0 \in Lab$, a set of input trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$, and a set of reachable configurations $\mathcal{R}_P \subseteq Lab \times \mathcal{T}(\mathcal{C})$. Let us also have a counter automaton $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ with $\rightarrow \in Q \times \Phi(X) \times Q$, and a set of reachable configurations $\mathfrak{R}_A$. A function $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ is said to be a *measure* assigning counters integer values for a particular tree[4]. Let $\mathbf{M}(t) = \{M(x,t) \mid x \in X\}$.

Definition 1. *The program P is simulated by the counter automaton A w.r.t. $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ and $\theta : Instr \rightarrow \Phi$ iff there exists a relation $\sim \subseteq \mathcal{R}_P \times \mathfrak{R}_A$ such that (1) $\forall t_0 \in I_0 : \mathbf{M}(t_0) \models \varphi_0 \wedge \langle l_0, t_0 \rangle \sim \langle q_0, \mathbf{M}(t_0) \rangle$ and (2) $\forall (l_1, t_1), (l_2, t_2) \in \mathcal{R}_P \; \forall i \in Instr \; \forall (q_1, \nu_1) \in \mathfrak{R}_A : (l_1, t_1) \xrightarrow{i} (l_2, t_2) \wedge (l_1, t_1) \sim (q_1, \nu_1) \Rightarrow \exists (q_2, \nu_2) \in \mathfrak{R}_A : (q_1, \nu_1) \xrightarrow{\theta(i)} (q_2, \nu_2) \wedge (l_2, t_2) \sim (q_2, \nu_2)$.*

[4] Intuitively, certain counters will measure, e.g., the distance of a certain node from the root, the size of the subtree below it, etc.

The measure M ensures that the counters are initially correctly interpreted over the input trees, whereas θ ensures that the counters are updated in accordance with the manipulations done on the trees. Simulation in the sense of Definition 1 guarantees that if we prove termination of the CA, the program will terminate on any $t \in I_0$.

4.1 Abstract Control Flow Graphs

According to Figure 2, we construct the CA simulating a program in two steps: we first construct the so-called *abstract control flow graph* (ACFG) of a program, and then translate it into a CA. Initially, the ACFG of a program is computed from its CFG by decorating its nodes with ARTMC-overapproximated sets of configurations reachable at each line (we keep the initial set of trees exact exploiting the fact that w.l.o.g. there are no statements leading back to the initial line). These sets allow us to exclude impossible (not fireable) transitions from the ACFG and thus derive a more exact CA. Further, in subsequent refinement iterations, infeasible termination counterexamples are excluded by splitting these sets (if this appears to be insufficient, we re-run ARTMC to compute a better overapproximation of the reachable sets of configurations). Below, we first define the notion of ACFG, then we provide its translation to counter automata.

In what follows, let P be a program with instructions $Instr$, working on trees from $\mathcal{T}(\mathcal{C})$, and let $l_0 \in Lab$ be the initial line of P. The *control flow graph* (CFG) of P is a labelled graph $F = \langle Instr, Lab, l_0, \Rightarrow \rangle$ where $l \overset{i}{\Rightarrow} l'$ denotes the presence of an instruction i between control locations $l, l' \in Lab$. We further suppose that the input tree configurations for P are described by the user as a (regular) set of trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$. An *abstract control flow graph* (ACFG) for P is then a graph $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ where LI is a *finite* subset of $Lab \times 2^{\mathcal{T}(\mathcal{C})}$, $\langle l_0, I_0 \rangle \in LI$, and there is an edge $\langle l, I \rangle \overset{i}{\longmapsto} \langle l', I' \rangle$ iff $l \overset{i}{\Rightarrow} l'$ in the CFG of P and $post(i, I) \cap I' \neq \emptyset$.

Note that since we work with ACFGs annotated with regular sets of configurations and since we can implement the effect of each instruction on a regular set as an operation on tree automata, we can effectively check that $post(i, I) \cap I' \neq \emptyset$, which is needed for computing the edges of ACFGs. Note also that a location in P may correspond to more than one locations in G.

We say that G *covers the invariants of* P whose set of reachable states is $\mathcal{R}_P$ iff each tree $t \in \mathcal{T}(\mathcal{C})$ that is reachable at a program line $l \in Lab$ (i.e., $\langle l, t \rangle \in \mathcal{R}_P$), appears in some of the sets of program configurations associated with l in the locations of G. Formally, $\forall l \in Lab : \mathcal{R}_P \cap (\{l\} \times \mathcal{T}(\mathcal{C})) \subseteq \{l\} \times \bigcup_{\langle l, I \rangle \in LI} I$. The following lemma captures the relation between the semantics of a program and that of an ACFG.

Lemma 1. *Let P be a program with trees and G an ACFG that covers the invariants of P. Then, the semantics of G simulates that of P in the classical sense.*

4.2 Translation to Counter Automata

We now describe the construction of a CA $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ from an ACFG $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ of a program P such that $A_{rsc}(G)$ simulates P in the sense of Def. 1. We consider two sorts of counters, i.e., $X = X_{PVar} \cup X_{Data}$, where $X_{PVar} =$

$\{r_x \mid x \in PVar\} \cup \{s_x \mid x \in PVar\}$ and $X_{Data} = \{c_d \mid d \in Data\}$. The role of these counters is formalised via a measure $M_{rsc} : X \times \mathcal{T}(\mathcal{C}) \to \mathbb{Z}$ in [18]. Intuitively, $M_{rsc}(r_x,t)$ and $M_{rsc}(s_x,t)$ record the distance from the root of the node n pointed to by x and the size of the subtree below n, respectively, and $M_{rsc}(c_d,t)$ gives the number of nodes with data d in a tree $t \in \mathcal{T}(\mathcal{C})$.

We build $A_{rsc}(G)$ from G by simply replacing the instructions on edges of G by operations on counters. Formally, this is done by the translation function θ_{rsc} defined in Table 1. The mapping for the instructions $x = y$.`right` and `rightRotate`(x,X,Y,A,B) is skipped in Table 1 as it is analogous to that of x=y.`left` and `leftRotate`(x,X,Y,A,B), respectively. Also, for brevity, we skip the instructions leading to the error state Err. As a convention, if the future value of a counter is not explicitly defined, we consider that the future value stays the same as the current value. Moreover, in all formulae, we assume an implicit guard $-1 \leq r_x < TreeHeight \wedge -1 \leq s_x < TreeSize$ for each $x \in PVar$[5] and $0 \leq c_d \leq TreeSize \wedge \sum_{d \in Data} c_d = TreeSize$ for each $d \in Data$. $TreeHeight$ and $TreeSize$ are parameters restricting the range in which the other counters can change according to a given input tree. They are needed as a basis on which termination of the resulting automaton can be shown.

Next, we define $Q = LI$, $q_0 = \langle l_0, I_0 \rangle$, and $q \xrightarrow{\theta_{rsc}(i)} q'$ iff $q \overset{i}{\longmapsto} q'$ for all $i \in Instr$. The initial constraint φ_0 on the chosen counters can be automatically computed[6] from the regular set of input trees I_0 such that it satisfies requirement (1) of Definition 1. The following theorem shows the needed simulation relation between the counter automata we construct and the programs.

Theorem 2. *Given a program P and an ACFG G of P covering its invariants, the CA $A_{rsc}(G)$ simulates P in the sense of Definition 1 wrt. θ_{rsc} and M_{rsc}.*

The generated CA $A_{rsc}(G)$ has the property that each transition $q \xrightarrow{\varphi} q'$ can be mapped back into the program instruction from which it originates. This is because the instructions onto which we decompose each program statement are assigned different formulae, by the translation function θ_{rsc}, and there is at most one statement between each two control locations of the program. Formally, we capture this by a function $\xi : Q \times \Phi \times Q \to Instr$ such that $\forall q_1, q_2 \in Q, \varphi \in \Phi : q \xrightarrow{\varphi} q' \Rightarrow q \overset{\xi(q_1,\varphi,q_2)}{\longmapsto} q'$. We generalise θ_{rsc} and ξ to sequences of transitions, i.e., for a path π in A_{rsc}, $\xi(\pi)$ denotes the sequence of program instructions leading to π, and $\theta_{rsc}(\xi(\pi))$ denotes the sequence of counter operations on π obtained by projecting out the control locations from π.

5 Checking Spuriousness of Counterexamples

Since the CA A_{rsc} generated from a program P with trees is a simulation of P (cf. Theorem 2), proving termination of A_{rsc} suffices to prove termination of P. However, if A_{rsc}

[5] -1 corresponds to x being null.

[6] This can be done by computing the Parikh image of a context-free language $\mathcal{L}(I_0)$ corresponding to the regular tree language I_0. For each tree $t \in I_0$ there is a word in $\mathcal{L}(I_0)$ consisting of all nodes of t. We use special symbols to denote the position of a node in the tree relative to a given variable (under the variable, between it and the root) and the data values of nodes.

Table 1. The mapping θ_{rsc} from program instructions to counter manipulations

instruction i	counter manipulation $\theta_{rsc}(i)$
$\mathtt{if}(x == \mathtt{null})$	$r_x = -1$
$\mathtt{if}(x! = \mathtt{null})$	$r_x \geq 0$
$\mathtt{if}(x == y)$	$r_x = r_y \wedge s_x = s_y$
$\mathtt{if}(x! = y)$	$true$
$\mathtt{if}(x.\mathtt{data} == d)$	$r_x \geq 0 \wedge c_d \geq 1$
$\mathtt{if}(x.\mathtt{data}! = d)$	$r_x \geq 0 \wedge c_d < TreeSize$
$x = \mathtt{null}$	$r'_x = s'_x = -1$
$x = y$	$r'_x = r_y \wedge s'_x = s_y$
$\mathtt{goLeftNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goLeftNonNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 2 \wedge r'_x = r_y + 1 \wedge s'_x < s_y$
$\mathtt{goUpNull}(x,y)$	$r_y = 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goUpNonNull}(x,y)$	$r_y \geq 1 \wedge s_y \geq 1 \wedge r'_x = r_y - 1 \wedge s'_x > s_y$
$\mathtt{changeData}(x,d,d)$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_d > 0$
$\mathtt{changeData}(x,d_1,d_2), d_1 \neq d_2$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_{d_1} > 0 \wedge c'_{d_2} = c_{d_2} + 1 \wedge c'_{d_1} = c_{d_1} - 1$
$\mathtt{leftRotate}(x,X,Y,A,B)$	$gLeftRotate(x,X,Y,A,B) \wedge aLeftRotate(x,X,Y,A,B)$

$$\begin{array}{ll}
gLeftRotate(x,X,Y,A,B) = & aLeftRotate(x,X,Y,A,B) = \\
\quad r_x \geq 0 \ \wedge \ s_x \geq 2 \ \wedge & \\
\quad (\forall v \in X : r_v = r_x \ \wedge \ s_v = s_x) \ \wedge & \quad (\forall v \in X : r'_v = r_v + 1 \ \wedge \ s'_v < s_v) \ \wedge \\
\quad (\forall v, v' \in Y : r_v = r_x + 1 \ \wedge \ s_v < s_x \ \wedge & \quad (\forall v \in Y : r'_v = r_v - 1 \ \wedge \ s'_v > s_v) \ \wedge \\
\qquad\qquad r_v = r_{v'} \ \wedge \ s_v = s_{v'}) \ \wedge & \\
\quad (\forall v \in A : r_v \geq r_x + 1 \ \wedge \ s_v < s_x) \ \wedge & \quad (\forall v \in A : r'_v = r_v + 1 \ \wedge \ s'_v = s_v) \ \wedge \\
\quad (\forall v \in B : r_v \geq r_x + 2 \ \wedge \ s_v < s_x - 1) & \quad (\forall v \in B : r'_v = r_v - 1 \ \wedge \ s'_v = s_v)
\end{array}$$

is not proved to terminate by the termination checker of choice, there are three possibilities: (1) A_{rsc} terminates, but the chosen termination checker did not find a termination argument, (2) both A_{rsc} as well as P do not terminate, and (3) P terminates, but A_{rsc} does not, as a consequence of the abstraction used in its construction. In all cases, the CA termination checker outputs a counterexample consisting of a finite path (stem) that leads to a cycle, both paths forming a lasso. Formally, a lasso $S.L$ over the control structure of a CA A_{rsc} is said to be *spurious* iff there exists a non-terminating run of A_{rsc} along $S.L$, and for no $t \in I_0$ does P have an infinite run along the path $\xi(S).\xi(L)$.

The three cases are dealt with in the upcoming paragraphs.

Deciding termination of CA lassos. We first show that termination of a given control loop is decidable in a CA whose transition relations are conjunctions of difference constraints, i.e. formulae of the forms $x - y \leq c$, $x' - y \leq c$, $x - y' \leq c$, or $x' - y' \leq c$ where x' denotes the future value of the counter x and $c \in \mathbb{Z}$ is an arbitrary integer constant. For this type of CA, the composed transition relation of the given control loop is also expressible as a conjunction of difference constraints. Then, this relation can be encoded as a constraint graph G such that the control loop terminates iff G contains a negative cycle (for details see [18]). Using the results of [11,7], this fact can be encoded as a Presburger formula and hence decided. At the same time, it is clear that the CA

generated via the translation function θ_{rsc} fall into the described class of CA. In particular, the constraint that each counter is bounded from below by -1 and from above by the $TreeHeight$ or $TreeSize$ parameters is expressible using difference constraints.[7]

Theorem 3. *Let $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ be a counter automaton with transition relations given as difference constraints. Then, given a control loop in A, the problem whether there exists an infinite computation along the loop is decidable.*

Checking termination of program lassos. Due to the above result, we may henceforth assume that the lasso $S.L$ returned by the termination analyser has a real non-terminating run in the CA. The lasso is mapped back into a sequence of program instructions $\xi(S).\xi(L)$ forming a program lasso. Two cases may arise: either the lasso is real on the program or it is spurious.

Non-spurious program lassos. Since we do not consider dynamic allocation, the number of configurations that a program can reach from any input tree is finite. Consequently, if there is a tree t_ω from which the program will have an infinite run along a given lasso, then we can discover it by an exhaustive enumeration of trees. We handle the discovery of t_ω by evaluating the lasso for all trees of up to a certain (increasingly growing) height at the same time (by encoding them as regular tree language and using the implementation of program instructions over tree automata that we have). As we work with finite sets of trees, we are bound to visit the same set twice after a finite number of iterations, if there exists a non-terminating run along the lasso.

Spurious program lassos. We handle this case also by a symbolic iteration of a given program lasso $\sigma.\lambda$ starting with the initial set of trees. We compute iteratively the sets $post(\sigma.\lambda^k, I_0), k = 1, 2, \ldots$. In the case of lassos without destructive updates, this computation is shown to reach the empty set after a number of iterations that is bounded by a double exponential in the length of the lasso (cf. Section 5.1). In the case of lassos with destructive updates, we can guarantee termination of the iteration with the empty set provided there exists some CA A_u (albeit unknown) keeping track of the particular tree measures we consider here (formalised via the functions M_{rsc} and θ_{rsc} in Section 4.2) that simulates the given program and that terminates[8] (cf. Section 5.2). In the latter case, even though we cannot guarantee the discovery of A_u, we can at least ensure that the sequence $post(\sigma.\lambda^k, I_0)$, $k = 1, 2, \ldots$ terminates with the empty set. This gives us a basis for refining the current ACFG such that we get rid of the spurious lasso encountered, and we can go on in the search for a CA showing the termination of the given program.

5.1 Deciding Spuriousness of Lassos Without Destructive Updates

In this section, we show that the spuriousness problem for a given lasso in a program with trees is decidable, if the lasso does not contain destructive updating instructions, i.e., tree rotations. The argument for decidability is that, if there exists a non-terminating

[7] To encode conditions of the form $x \leq c$ we add a new variable z, initially set to zero, with the condition $z' = z$ appended to each transition, and rewrite the original condition as $x - z \leq c$.

[8] We can relax this condition by saying that A_u does not have any infinite run, not corresponding to a run of the program. For the sake of clarity, we have chosen the first stronger condition.

run along the loop, then there exists also a non-terminating run starting from a tree of size bounded by a constant depending on the program. Thus, there exists a tree within this bound that will be visited infinitely many often.[9]

Given a loop without destructive updates, we first build an abstraction of it by replacing the go{Left|Right|Up}Null(x,y) instructions by $x = \texttt{null}$, and by eliminating all changeData(x,d_1,d_2) instructions and the tests. Clearly, if the original loop has a non-terminating computation, then its abstraction will also have a non-terminating run starting with the same tree. The loop is then encoded as an iterative linear transformation which, for each pointer variable $x \in PVar$, has a counter p_x encoding the binary position of the pointer in the current tree using 0/1 as the left/right directions. Additionally, the most significant bit of the encoding is required to be always one, which allows for differentiating between, e.g., the position 001 encoded by $9 = (1001)_2$, and 0001 encoded by $17 = (10001)_2$. Null pointers are encoded by the value 0. The program instructions are translated into counter operations as follows:

$$x = \texttt{null} : p_x = 0 \qquad x = y : p_x = p_y \qquad \texttt{goLeftNonNull}(x,y) : p_x = 2\star p_y$$
$$\texttt{goRightNonNull}(x,y) : p_x = 2\star p_y + 1 \qquad \texttt{goUpNonNull}(x,y) : p_x = \tfrac{1}{2} p_y$$

where $2\star$ and $\frac{1}{2}$ denote the integer functions of multiplication ($x \mapsto 2x$) and division ($x \mapsto x/2$). Assuming that we have n pointer variables, each program instruction is modelled by a linear transformation of the form $\mathbf{p}' = A\mathbf{p} + B$ where A is an $n \times n$ matrix with at most one non-null element, which is either $1, 2$ or $\frac{1}{2}$, and B is an n-column vector with at most one 1 and the rest 0[10]. The composition of the instructions on the loop is also a linear transformation, except that A has at most one non-null element on each line, which is either I, or a composition of $2\star$'s and $\frac{1}{2}$'s.

Since A has at most one non-null element on each line, one can extract an $m \times m$ matrix A_0 for some $m \le n$ that has exactly one non-null element on each line and column. Our proof is based on the fact that there exists some constant k bounded by $O(3^m)$ such that ${A_0}^k$ is a diagonal matrix. Intuitively, this means that the position of each pointer at step $i + k$ is given by a linear function of the position of the pointer at i. Then A^i is an exponential function of i. As there is no dynamic allocation of nodes in the tree, the non-termination hypothesis implies that the positions of pointers have to stay in-between bounds. But this is only possible if the elements of the main diagonal of ${A_0}^k$ are either I or compositions of the same number of $2\star$ and $\frac{1}{2}$. Intuitively, this means that all pointers are confined to move inside bounded regions of the working tree.

Theorem 4. *Let P be a program over trees, $PVar$ and $Data$ be its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$, $I_0 \subseteq \mathcal{T}(\mathcal{C})$ be an initial set of trees, and $\sigma.\lambda$ be a lasso of P. Then, if P has an infinite run along the path $\sigma.\lambda^\omega$ for some $t_0 \in I_0$, then there exists a tree $t_{b0} \in \mathcal{T}(\mathcal{C})$ of height bounded by $(\|PVar\| + 1) \cdot \big(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\big)$ such that P, started with t_{b0}, has an infinite run along the same path.*

---

[9] Since there is no dynamic allocation, all trees visited starting with a tree of size k will also have size k. Hence each run of the program will either stop, or re-visit the same program configuration after a bounded number of steps.

[10] We interpret the matrix operations over the semiring of integer functions $\langle \mathbb{N} \to \mathbb{N}, +, \circ, 0, I \rangle$, where $\circ$ is functional composition and I is the identity function.

Decidability of spuriousness is an immediate consequence of this theorem. Also, there is a bound on the number of symbolic unfoldings of a spurious lasso starting with the initial set of trees.

Corollary 1. *Let P be a program over trees, $PVar$ and $Data$ its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$, and $I_0 \subseteq \mathcal{T}(\mathcal{C})$ an initial set of trees. Given a lasso $S.L$ in the CA $A_{rsc}(G)$ built from an ACFG G of P, let $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, if $\sigma.\lambda$ does not contain destructive updates, its spuriousness is decidable. Moreover, if the lasso is spurious, for all $k \geq |\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot(|\sigma|+|\lambda|\cdot 3^{\|PVar\|})}}$, we have $post(\sigma.\lambda^k, I_0) = \emptyset$.*

Despite the double exponential bound, experimental evidence (see Section 7) shows that the number of unfoldings necessary to eliminate a spurious lasso is fairly small.

5.2 Analysing Lassos with Destructive Updates

Theorem 5 stated below shows that spuriousness of a lasso that contains destructive updates, i.e., tree rotations, is decidable if there exists a *terminating* CA A_u, not necessarily known to us, simulating the program wrt. θ_{rsc} and M_{rsc}. That is, if there exists a termination argument for the program based on the tree measures we use, then we can prove spuriousness of the lasso by a symbolic iteration of the initial set.

Theorem 5. *Let P be a program with an ACFG G and let $S.L$ be a spurious lasso in $A_{rsc}(G)$. If there exists a CA A_u that simulates P wrt. θ_{rsc} and M_{rsc} and that terminates on all inputs, then there exists $k \in \mathbb{N}$ such that $post(\xi(S).\xi(L)^k, I_0) = \emptyset$.*

Indeed, imagine that for any $l \in \mathbb{N}$ there is an input tree $t_l \in I_0$ for which $\xi(S).\xi(L)^l$ is fireable. Then, the CA A_u, having a finite-control, and simulating P has to contain a lasso with a stem $S.L^{n_1}$ and a loop L^{n_2} for some $n_1, n_2 \in \mathbb{N}$ (i.e., a possibly partially unfold $S.L$). However, as the set of initial counter valuations of A_u must include the one of $A_{rsc}(G)$ (as that is the smallest possible wrt. M_{rsc}), this means that A_u has a non-terminating run also, which is a contradiction.

6 Abstraction Refinement

Let P be a program, $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ be an ACFG of P, and $A_{rsc}(G)$ be the CA obtained by the translation described in Section 4. Let $S.L$ be a spurious lasso, i.e., a path over which $A_{rsc}(G)$ has an infinite run, while P does not have an infinite run over the corresponding program path $\sigma.\lambda$ where $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, we produce a new ACFG $G_{S.L}$ of P such that $A_{rsc}(G_{S.L})$ will not exhibit any lasso-shaped path with a stem labelled with the sequence of counter operations $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^p$ and a loop labelled with $\theta_{rsc}(\lambda)^q$ for any $p, q \geq 0$. Provided that the spuriousness of the lasso $S.L$ is detected using either Corollary 1 or Theorem 5, we know that there exists $k > 0$ such that $post(\sigma.\lambda^l, I_0) = \emptyset$

for all $l \geq k$. To build the refined ACFG $G_{S.L}$, we use the sets $post(\sigma.\lambda^i, I_0)$, $0 \leq i < k$, computed in the spuriousness analysis of $S.L$ (cf. Section 5).

We refine G into $G_{S.L}$ by *splitting* some of its locations $\langle l_i, I_i \rangle \in LI$ into several locations of the form $\langle l_i, I_{ij} \rangle$, and by recomputing the edges according to the definition of ACFG (cf. Section 4.1). Intuitively, the sets I_{ij} form a partition of I_i such that I_{ij} will contain all trees from I_i that are visited in *at most* j iterations of the loop. As we prove in [18], since we keep apart the sets of trees for which the lasso may be iterated a different number of times, $A_{rsc}(G_{S.L})$ will not contain lassos of the form $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^p.\left(\theta_{rsc}(\lambda)^q\right)^\omega$ for any $p, q > 0$.

Due to the fact that our verification problem is not r.e. (cf. Theorem 1), the Abstract-Check-Refine loop might diverge. One situation in which this can happen is when, at each refinement step, a certain line invariant I_i is split such that one of the parts, say I_{ij}, is finite (e.g., it contains only trees up to some height). However, to exclude a spurious counterexample, it might be the case that I_i has to be split into infinite sets according to some more general property (e.g., the same number of red and black nodes).[11] In such situations, we use a heuristic *acceleration* method consisting in applying the finite height abstraction α of ARTMC (cf. Section 3) to split the line invariants. That is, apart from splitting I_i wrt. the sets $post(\sigma.\lambda^i, I_0)$, we split wrt. the sets $\alpha(post(\sigma.\lambda^i, I_0))$ too. In Section 7, we report on an example in which the accelerated refinement based on the finite height abstraction was successfully used to make the analysis converge.

Another problem that may occur is that the invariants computed by ARTMC may not be precise enough for proving termination of the given program. That is why after a predefined number of steps of refining ACFGs by splitting, we *repeat ARTMC with a more precise abstraction* (e.g., we increase the abstraction height). We re-run ARTMC on the underlying CFG of the last computed ACFG G and restrict the computed reachability sets to the sets appearing in the locations of G in order to preserve the effect of the refinement steps done so far. Due to the space restrictions, we provide a detailed description of these issues in [18].

7 Implementation and Experimental Results

To demonstrate the applicability of our approach, we tested it on several real procedures manipulating trees. We restricted the ARTMC tool from [6] to binary trees with parent pointers and added support for tree rotations, instead of using general purpose destructive updates. The absence of null pointer dereferences was verified fully automatically. Termination of the generated CA was checked using the ARMC tool [23].

We first considered the following set of case studies (for more details see [18]): (1) a non-recursive *depth-first tree traversal*, (2) a procedure for *searching a data value in a red-black tree* [15] (with the actual data abstracted away and all the comparisons done in a random way), and (3) the procedure that *rebalances red-black trees after inserting a new element* using tree rotations [15]. In the latter two cases, the set of input trees was a regular overapproximation of all red-black trees (we abstracted away the balancedness condition).

[11] A similar case is encountered in classical abstraction refinement for checking safety properties.

Table 2. Experimental Results

Example	T_{ARTMC}	$\|Q\|_{Inv}$	T_{CA}	N_{cnt}	N_{loc}	N_{tr}
Depth-first tree traversal	43s	67	10s	5	15	20
RB-search	2s	22	1s	3	8	11
RB-rebalance after insert	1m 9s	87	36s	7	44	66

The results of the experiments that we performed on a PC with a 1.4 GHz Intel Xeon processor are summarised in Table 2. The table contains the ARTMC running times (T_{ARTMC}), the number of states of the largest invariant generated by ARTMC ($|Q|_{Inv}$), the time spent by the ARMC tool to show termination (T_{CA}), and the number of counters (N_{cnt}), locations (N_{loc}), and transitions (N_{tr}) of the CA.

For the three above programs, it turned out to be possible to prove the termination without a need of refinement. In the third experiment, we could even remove checking of the condition of the red-black trees (a red node has only black sons), leading to smaller verification times, with less precise invariants, which were, however, still precise enough for the termination proof.

To test our refinement procedure, we applied it on another case study where the initial invariants were not sufficient to prove termination. In particular, we considered the procedure in Figure 3 that marks the elements of a singly-linked list as even or odd, depending on whether their distance to the end of the list is even or odd. As the procedure does not know the length of the list and cannot use back-pointers, it tries to mark the first element as even, and at the end of the list, it checks whether the last element was marked as odd. If this is true, the marking is correct, otherwise the marking has to be reversed.

```
bool odd = false;
if (list != null) then
  while (true) do
    x = list;
    while (x != null) do
      x.data = odd;
      odd = not(odd);
      x = x.next;
    od
    if (not(odd)) then break;
  od
```

Fig. 3. A procedure marking elements of a list as odd or even from the end of the list

For this procedure, even if one builds the CA starting with the exact line invariants, termination cannot be established. To establish termination, one has to separate configurations where the procedure is marking the list in a correct way from those where the marking is incorrect. Then, the outer loop of the procedure will not appear in the CA at all since, in fact, it can be fired at most twice: once when the initial guess is correct and twice otherwise. The challenge is to recognise this fact automatically.

We managed to verify termination of the procedure on an arbitrary input list after excluding 9 spurious lassos (in 2 cases, the refinement was accelerated by the use of the finite-height abstraction on the I_{ij} sets that resulted from splitting line invariants when excluding certain spurious lassos).

8 Conclusion

We addressed the problem of proving termination of a significant class of tree manipulating programs. We provide a counter-example guided abstraction refinement loop

based on the ARTMC framework and on exploiting the existing work on checking termination of counter automata. A number of results related to the decidability of the spuriousness problem of lasso-shaped termination counterexamples were given. Our method is not guaranteed to stop (as the problem is not r.e.), but when it stops, it provides a precise answer (both in the positive and negative case). The method was experimentally tested to be successful on several interesting practical programs.

Future work includes a more efficient implementation of our framework as well as its extension to more complex programs (like, e.g., programs with unbounded dynamic allocation and general destructive pointer updates). We would also like to further investigate cases in which the universal termination problem is r.e. (and hence allowing complete verification techniques).

Acknowledgement. The work was supported by the French Ministry of Research (ACI Securité Informatique), the Czech Grant Agency (projects 102/07/0322, 102/05/H050), the Czech-French Barrande project 2-06-27, and the Czech Ministry of Education by project MSM 0021630528 *Security-Oriented Research in Information Technology*.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape Analysis by Predicate Abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
2. Ball, T., Rajamani, S.K.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
3. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Analyses from Invariance Analyses. In: Proc. of POPL 2007, ACM Press, New York (2007)
4. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
5. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking. ENTCS 149, 37–48 (2006)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
7. Bozga, M., Iosif, R., Lakhnech, Y.: Flat Parametric Counter Automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, Springer, Heidelberg (2006)
8. Bradley, A.R., Manna, Z., Sipma, H.B.: The Polyranking Principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, Springer, Heidelberg (2005)
9. Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient Verification of Sequential and Concurrent C Programs. Formal Methods in System Design 25(2–3) (2004)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
11. Comon, H., Jurski, Y.: Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, Springer, Heidelberg (1998)
12. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, Springer, Heidelberg (2005)

13. Cook, B., Podelski, A., Rybalchenko, A.: Termination Proofs for Systems Code. In: Proc. of PLDI 2006, ACM Press, New York (2006)
14. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond Safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
16. Deshmukh, J.V., Emerson, E.A., Gupta, P.: Automatic Verification of Parameterized Data Structures. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
17. Distefano, D., Berdine, J., Cook, B., O'Hearn, P.W.: Automatic Termination Proofs for Programs with Shape-shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
18. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving Termination of Tree Manipulating Programs. Verimag, TR-2007 -1, http://www-verimag.imag.fr/index.php?page=techrep-list
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, Springer, Heidelberg (2003)
20. Loginov, A., Reps, T.W., Sagiv, M.: Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
21. Møller, A., Schwartzbach, M.I.: The Pointer Assertion Logic Engine. In: Proc. of PLDI 2001, ACM Press, New York (2001)
22. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002, IEEE Computer Society Press, Los Alamitos (2002)
23. Rybalchenko, A.: The ARMC tool. www.mpi-inf.mpg.de/~rybal/armc/
24. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. TOPLAS 24(3) (2002)
25. Yahav, E., Reps, T., Sagiv, M., Wilhelm, R.: Verifying Temporal Heap Properties Specified via Evolution Logic. In: Degano, P. (ed.) ESOP 2003 and ETAPS 2003. LNCS, vol. 2618, Springer, Heidelberg (2003)

Symbolic Fault Tree Analysis for Reactive Systems*

Marco Bozzano**, Alessandro Cimatti, and Francesco Tapparo

FBK-IRST, Via Sommarive 18, 38050 Trento, Italy
Tel.: +39 0461 314367; Fax: +39 0461 302040
bozzano@itc.it

Abstract. Fault tree analysis is a traditional and well-established technique for analyzing system design and robustness. Its purpose is to identify sets of basic events, called *cut sets*, which can cause a given *top level event*, e.g. a system malfunction, to occur. Generating fault trees is particularly critical in the case of reactive systems, as hazards can be the result of complex interactions involving the dynamics of the system and of the faults. Recently, there has been a growing interest in model-based fault tree analysis using formal methods, and in particular symbolic model checking techniques. In this paper we present a broad range of algorithmic strategies for efficient fault tree analysis, based on binary decision diagrams (BDDs). We describe different algorithms encompassing different directions (forward or backward) for reachability analysis, using dynamic cone of influence techniques to optimize the use of the finite state machine of the system, and dynamically pruning of the frontier states. We evaluate the relative performance of the different algorithms on a set of industrial-size test cases.

1 Introduction

The goal of safety analysis is to investigate the behavior of a system in degraded conditions, that is, when some parts of the system are not working properly, due to malfunctions. Safety analysis includes a set of activities, that have the goal of identifying and characterizing all possible hazards, and are performed in order to ensure that the system meets the safety requirements that are required for its deployment and use. Safety analysis activities are particularly critical in the case of reactive systems, because hazards can be the result of complex interactions involving the dynamics of the system [29].

Recently, there has been a growing interest in model-based safety analysis [1, 4, 6, 7, 8, 9, 15, 17, 18, 25] using formal methods, and in particular symbolic model checking techniques. Traditionally, safety analysis activities are performed manually, and rely on the skill of the safety engineers, hence they are an error-prone and time-consuming activity, and they may rapidly become impractical in case of large and complex systems. Safety analysis based on formal methods, on the other hand, aims at reducing the effort involved and increase the quality of the results, by focusing the effort on building formal models of the system [7, 8], rather than carrying out the analyses.

Fault Tree Analysis (FTA) [35] is one of the most popular safety analysis activities. It is a deductive, top-down method to analyze system design and robustness. It involves

* This work has been partly supported by the E.U.-sponsored project ISAAC, contract no. AST3-CT-2003-501848.

** Corresponding author.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 162–176, 2007.
© Springer-Verlag Berlin Heidelberg 2007

specifying a *top level event* (TLE hereafter) and a set of possible *basic events* (e.g., component faults); the goal is the identification of all possible *cut sets*, i.e. sets of basic events which cause the TLE to occur. Fault trees provide a convenient symbolic representation of the combination of events causing the top level event, and they are usually represented as a parallel or sequential combination of logical gates. To allow a quantitative evaluation, the probability of the events is also included in the fault tree.

In this paper, we focus on the problem of FTA for reactive systems, i.e. systems with infinite behavior over time. The problem is substantially harder than the traditional case, where the system is abstracted and modeled to be state-less and combinational. In fact, the presence of dynamics can influence the presence and the effect of failures.

The main contribution of the paper is the definition, implementation and evaluation of a broad range of algorithms and strategies for efficient fault tree analysis of reactive systems. The algorithms are based on (extended) reachability analysis of a model of the system, and apply to a general framework, where different dynamics for failure mode variables (e.g. persistent and sporadic faults) are possible, thus extending the work in [7]. We point out several distinguishing features. First, fault trees can be constructed both by forward and backward search; it is indeed well known that depending on the structure of the state space of the reactive system, dramatic differences in performance can result, depending on the search direction. Second, backward search is optimized with the use of *dynamic cone of influence* (DCOI), which consists in a lazy generation of restricted models to be used for computing the backward search steps. Third, we propose an optimization called *dynamic pruning*, applicable both to the forward and the backward reachability algorithms, that detects minimal cut sets during the search, and thus can limit the exploration and reduce the number of required iterations.

The algorithms leverage techniques borrowed from symbolic model checking, in particular Binary Decision Diagrams (BDDs for short), that enable the exploration of very large state spaces. The algorithms have been implemented and integrated within FSAP [7, 16], a platform aiming at supporting design and safety engineers in the development and in the safety assessment of complex, reactive systems. The platform automates the generation of artifacts that are typical of reliability analysis, for example failure mode and effect analysis tables, and fault trees. FSAP consists of a graphical interface and an engine based on the NuSMV model checker [11, 23]. NuSMV provides support for user-guided or random simulation, as well as standard model checking capabilities like property verification and counterexample trace generation.

We experimentally evaluate the different algorithms on a set of scalable benchmarks derived from industrial designs. The results show that the forward and backward search directions are indeed complementary, and the proposed optimizations result in a substantial performance improvement.

The paper is structured as follows. In Sect. 2 we give some background about model checking reactive systems; in Sect. 3 we discuss fault tree analysis; in Sect. 4 we describe the algorithms for fault tree generation; in Sect. 5 we outline the implementation in the FSAP platform; in Sect. 6 we present the experimental evaluation; finally, in Sect. 7 we discuss some related work, and in Sect. 8 we draw some conclusions and outline future work.

2 Background

2.1 Modeling Reactive Systems

We are interested in the analysis of reactive systems, whose behavior is potentially infinite over time, such as operating systems, physical plants, and controllers. Reactive systems are modeled as Kripke structures.

Definition 1 (Kripke Structure). *Let $\mathcal{P}$ be a set of propositions. A Kripke structure is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ where: $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states; $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation; $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$ is the labeling function.*

We require the transition relation to be total, i.e. for each state s there exists a successor state s' such that $\mathcal{R}(s, s')$. We notice that Kripke structures can model non-deterministic behavior, by allowing states to have multiple successors. The labeling function associates each state with information on which propositions hold in it.

An execution of the system is modeled as a trace (also called a behavior) in such a Kripke structure, obtained starting from a state $s \in \mathcal{I}$, and then repeatedly appending states reachable through $\mathcal{R}$.

Definition 2 (Trace). *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure. A trace for $\mathcal{M}$ is a sequence $\pi = s_0, s_1, \ldots, s_k$ such that $s_0 \in \mathcal{I}$ and $\mathcal{R}(s_{i-1}, s_i)$ for $1 \leq i \leq k$.*

A state is reachable if and only if there exists a trace containing it. Given the totality of $\mathcal{R}$, a trace can always be extended to have infinite length. We notice that systems are often presented using module composition; the resulting structure can be obtained by composing the sub-structures, but its state space may be exponential in the number of composed modules. In the following we confuse a system and the Kripke structure modeling it; we also assume that a Kripke structure $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is given.

2.2 Symbolic Model Checking

Model checking is a widely used formal verification technique. While testing and simulation may only verify a limited portion of the possible system behaviors, model checking provides a formal guarantee that some given specification is obeyed, i.e. all the traces of the system are within the acceptable traces of the specification. Given a system $\mathcal{M}$, represented as a Kripke structure, and a requirement ϕ, typically specified as a formula in some temporal logic, model checking analyzes whether $\mathcal{M}$ satisfies ϕ, written $\mathcal{M} \models \phi$. Model checking consists in exhaustively exploring every possible system behavior, to check automatically that the specifications are satisfied.

In its simpler form, referred to as *explicit state*, model checking is based on the expansion and storage of individual states. These techniques suffer from the so-called state explosion problem, i.e. they need to explore and store the states of the state transition graph. A major breakthrough was enabled by the introduction of *symbolic model checking* [21]. The idea is to manipulate *sets of* states and transitions, using a logical formalism to represent the characteristic functions of such sets. Since a small logical formula may admit a large number of models, this results in many practical cases in a very compact representation which can be effectively manipulated. Another key issue is

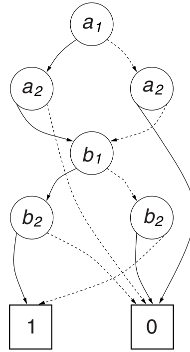


Fig. 1. A BDD for the formula $(a_1 \leftrightarrow a_2) \wedge (b_1 \leftrightarrow b_2)$

the use of an efficient machinery to carry out the manipulation. For this, we use Ordered Binary Decision Diagrams [10] (BDDs for short).

BDDs are a representation for boolean formulae, which is canonical once an order on the variables has been established. Fig. 1 depicts the BDD for the boolean formula $(a_1 \leftrightarrow a_2) \wedge (b_1 \leftrightarrow b_2)$, using the variable ordering a_1, a_2, b_1, b_2. Solid lines represent "then" arcs (the corresponding variable has to be considered positive), dashed lines represent "else" arcs (the corresponding variable has to be considered negative). Paths from the root to the node labeled with "1" represent the satisfying assignments of the represented boolean formula (e.g., $a_1 \leftarrow 1, a_2 \leftarrow 1, b_1 \leftarrow 0, b_2 \leftarrow 0$). Efficient BDD packages are available. Despite the worst-case complexity (e.g. certain classes of boolean functions are proved not to have a polynomial-size BDD representation for any order), in practice it is possible to represent Kripke structures effectively.

We now show how a Kripke structure can be symbolically represented. Without loss of generality, we assume that there exists a bijection between $\mathcal{S}$ and $2^{\mathcal{P}}$ (if the cardinality of $\mathcal{S}$ is not a power of two, standard constructions can be applied to extend the Kripke structure). Each state of the system assigns a valuation to each variable in $\mathcal{P}$. We say that a proposition p holds in s, written $s \models p$, if and only if $p \in \mathcal{L}(s)$. Similarly, it is possible to evaluate a boolean formula $\phi(\mathcal{P})$, written $s \models \phi(\mathcal{P})$. The representation of a Kripke structure with BDDs is as follows. For each proposition in $\mathcal{P}$ we introduce a BDD variable; we use $\underline{x}$ to denote the vector of such variables, that we call state variables, and we assume a fixed correspondence between the propositions in $\mathcal{P}$ and the variables in $\underline{x}$. We use $\underline{x}$ to denote the vector of variables representing the states of a given system. We write $\mathcal{I}(\underline{x})$ for the BDD (corresponding to the formula) representing the initial states. To represent the transitions, we introduce a set of "next" variables $\underline{x}'$, used for the state resulting after the transition. A transition from s to s' is then represented as a truth assignment to the current and next variables. We use $\mathcal{R}(\underline{x}, \underline{x}')$ for the formula representing the transition relation expressed in terms of those variables.

Operations over sets of states can be represented by means of boolean operators. For instance, intersection amounts to conjunction between the formulae representing the sets, union is represented by disjunction, complement is represented by negation,

and projection is realized with quantification. BDDs provide primitives to compute efficiently all these operations.

3 Fault Tree Analysis for Reactive Systems

Safety analysis is a fundamental step in the design of complex, critical systems. The idea is to analyze the behavior of the system in presence of *faults*, under the hypothesis that components may break down. Model-based safety analysis is carried out on formally specified models which take into account system behavior in the presence of faults. The first step is to identify a set of state variables, called *failure mode variables*, to denote the possible failures, and to identify a set of properties of interest. Intuitively, a failure mode variable is true in a state when the corresponding fault occurs (different failure mode variables are associated to different faults). Once this is done, different forms of analysis investigate the relationship between failures and the occurrence of specific events (called top level events), typically the violation of some of the properties. In the rest of this section, we assume that a set of failure mode variables $\mathcal{F} \subseteq \mathcal{P}$ is given.

Two traditional safety analysis activities are Failure Mode and Effects Analysis (FMEA), and Fault Tree Analysis (FTA). FMEA analyzes which properties are lost, under a specific failure mode configuration, i.e. a truth assignment to the variables in $\mathcal{F}$; the results of the analysis are summarized in a FMEA table. FTA progresses in the opposite direction: given a property, the set of causes has to be identified that can cause the property to be lost. In the rest of this paper we focus on FTA (although many of the techniques described here may be applied also to FMEA).

Traditionally, safety analysis is carried out on a combinational view of the system. Here we specifically focus on reactive systems. In this context, different models of failure may have different impact on the results. Moreover, the duration and temporal relationships between failures may be important, e.g. fault f_1 may be required to persist for a give number of steps, or to occur before another fault f_2. We refer the reader to [9] for a proposal to enrich the notion of minimal cut set along these lines.

Our framework is completely general: it encompasses both the case of *permanent* failure modes (*once failed, always failed*), and the case of *sporadic* or *transient* ones, that is, when faults are allowed to occur sporadically (e.g., a sensor showing an invalid reading for a limited period of time), or when repairing is possible.

Fault tree analysis [35] is an example of deductive analysis, which, given the specification of an undesired condition, usually referred to as a *top level event* (TLE), systematically builds all possible chains of one of more basic faults that contribute to the occurrence of the event. The result of the analysis is a *fault tree*, representing the logical interrelationships of the basic events that lead to the undesired state. In its simpler form [7] a fault tree can be represented with a two-layer logical structure, namely a top level disjunction of the combinations of basic faults causing the top level event. Each combination, which is called *cut set*, is in turn the conjunction of the corresponding basic faults. In general, logical structures with multiple layers can be used, for instance based on the hierarchy of the system model [3]. A cut set is formally defined as follows.

Definition 3 (Cut set). *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a system with a set of failure mode variables $\mathcal{F} \subseteq \mathcal{P}$, let $FC \subseteq \mathcal{F}$ be a fault configuration, and $TLE \in \mathcal{P}$. We say that FC*

is a cut set of TLE, written $cs(FC, TLE)$ *if there exists a trace* $s_0, s_1, \ldots, s_k$ *for* $\mathcal{M}$ *such that:* $i)$ $s_k \models TLE$; $ii)$ $\forall f \in \mathcal{F} \quad f \in FC \iff \exists i \in \{0, \ldots, k\}\ (s_i \models f)$.

Intuitively, a cut set corresponds to the set of failure mode variables that are active at some point along a trace witnessing the occurrence of the top level event. Typically, one is interested in isolating the fault configurations that are minimal in terms of failure mode variables, that is, those that represent simpler explanations, in terms of faults, for the occurrence of the top level event. Under the hypothesis of independent faults, a minimal configuration is a more probable explanation with respect to a configuration being a proper superset, hence it has a higher importance in reliability analysis. Minimal configurations are called *minimal cut sets* and are formally defined as follows.

Definition 4 (Minimal Cut Sets). *Let* $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ *be a system with a set of failure mode variables* $\mathcal{F} \subseteq \mathcal{P}$*, let* $F = 2^{\mathcal{F}}$ *be the set of all fault configurations, and* $TLE \in \mathcal{P}$*. We define the set of cut sets and minimal cut sets of TLE as follows:*

$$\begin{aligned} CS(TLE) &= \{FC \in F \mid cs(FC, TLE)\} \\ MCS(TLE) &= \{cs \in CS(TLE) \mid \forall cs' \in CS(TLE)\ (cs' \subseteq cs \Rightarrow cs' = cs)\} \end{aligned}$$

The notion of minimal cut set can be extended to the more general notion of *prime implicant* (see [14]), which is based on a different definition of minimality, involving both the activation and the absence of faults. We omit the details for lack of space. We also remark that, although not illustrated in this paper, the fault tree can be complemented with probabilistic information, as done, e.g., in the FSAP platform [16].

Based on the previous definitions, fault tree analysis can be described as the activity that, given a TLE, involves the computation of all the minimal cut sets (or prime implicants) and their arrangement in the form of a tree. Typically, each cut set is also associated with a trace witnessing the occurrence of the top level event.

4 Symbolic Fault Tree Analysis

We now review the different algorithms for fault tree generation, available in FSAP. We focus on the computation of the cut sets (standard model checking techniques can be used to generate the corresponding traces). We start in Sect. 4.1 with the standard, forward algorithm that we described in [7]. In this context, we extend the algorithm in order to support sporadic failure modes, which were not allowed in [7]. Then, we describe a novel backward algorithm in Sect. 4.2, and its optimization based on dynamic cone of influence in Sect. 4.3; finally, in Sect. 4.4 we introduce an optimization called *dynamic pruning*, which is applicable both to the forward and the backward algorithms.

In the following, we assume that a Kripke structure $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is given, and we use the following notations. First, $\underline{f}$ denotes the vector of failure variables. Given two vectors $\underline{v} = v_1 \ldots v_k$ and $\underline{w} = w_1 \ldots w_k$, the notation $\underline{v} = \underline{w}$ stands for $\bigwedge_{i=1}^{k} (v_i = w_i)$. We use $ITE(p, q, r)$ (if-then-else) to denote $((p \rightarrow q) \wedge (\neg p \rightarrow r))$. Finally, given a set of states Q, the image of Q is defined as $\{s' \mid \exists s \in Q.\ \mathcal{R}(s, s')\}$. It can be encoded symbolically as follows: $fwd_img(\mathcal{M}, Q(\underline{x})) = \exists \underline{x}.\ (Q(\underline{x}) \wedge \mathcal{R}(\underline{x}, \underline{x}'))$. Similarly, the preimage of Q is defined as $\{s \mid \exists s' \in Q.\ \mathcal{R}(s, s')\}$ and can be encoded as $bwd_img(\mathcal{M}, Q(\underline{x}')) = \exists \underline{x}'.\ (Q(\underline{x}') \wedge \mathcal{R}(\underline{x}, \underline{x}'))$.

```
function FTA-Forward (M, Tle)
1    M := Extend(M, R^o);
2    Reach := I ∩ (o = f);
3    Front := I ∩ (o = f);
4    while (Front ≠ ∅) do
5       temp := Reach;
6       Reach := Reach ∪
                 fwd_img(M, Front);
7       Front := Reach\temp;
8    end while;
9    CS := Proj(o, Reach ∩ Tle);
10   MCS := Minimize(CS);
11   return Map_{o→f}(MCS);
```

```
function FTA-Backward (M, Tle)
1    M := Extend(M, R^g);
2    Reach := Tle ∩ (g = f);
3    Front := Tle ∩ (g = f);
4    while (Front ≠ ∅) do
5       temp := Reach;
6       Reach := Reach ∪
                 bwd_img(M, Front);
7       Front := Reach\temp;
8    end while;
9    CS := Proj(g, Reach ∩ I);
10   MCS := Minimize(CS);
11   return Map_{g→f}(MCS);
```

Fig. 2. Forward and backward algorithms

4.1 Forward Fault Tree Analysis

The forward algorithm starts from the initial states of the system and accumulates, at each iteration, the forward image. In order to take into account sporadic failure modes (compare Def. 3, condition ii), at each iteration we need to "remember" if a failure mode has been activated. To this aim, for each failure mode variable $f_i \in \mathcal{F}$, we introduce an additional variable o_i (*once* f_i), which is true if and only if f_i has been true at some point in the past. This construction is traditionally referred to as *history variable*, and is formalized by the transition relation $\mathcal{R}^o$ given by the following condition:

$$\bigwedge_{f_i \in \mathcal{F}} ITE(o_i, o_i', o_i' \leftrightarrow f_i')$$

Let $Extend(\mathcal{M}, \mathcal{R}^o)$ be the Kripke structure obtained from $\mathcal{M}$ by replacing the transition relation $\mathcal{R}$ with the synchronous product between $\mathcal{R}$ and $\mathcal{R}^o$, in symbols $\mathcal{R}(\underline{x}, \underline{x}') \wedge \mathcal{R}^o(\underline{x}, \underline{x}')$ and modifying the labeling function $\mathcal{L}$ accordingly.

The pseudo-code of the algorithm is described in Fig. 2 (left). The inputs are $\mathcal{M}$ and Tle (the set of states satisfying the top level event). A variable $Reach$ is used to accumulate the reachable states, and a variable $Front$ to keep the *frontier*, i.e. the newly generated states (at each step, the image operator needs to be applied only to the latter set). Both variables are initialized with the initial states, and the history variables with the same value as the corresponding failure mode variables. The core of the algorithm (lines 4-8) computes the set of reachable states by applying the image operator to the frontier until a fixpoint is reached (i.e, the frontier is the empty set). The resulting set is intersected (line 9) with Tle, and projected over the history variables. Finally, the minimal cut sets are computed (line 10) and the result is mapped back from the history variables to the corresponding failure mode variables (line 11).

Note that all the primitives used in algorithm can be realized using BDD data structures, as explained in Sect. 2.2. For instance, set difference (line 7) can be defined as $Reach(\underline{x}) \wedge \neg temp(\underline{x})$, and projection (line 9) as $\exists \underline{y}.(Reach(\underline{x}) \wedge Tle(\underline{x}))$, where $\underline{y}$ is the set of variables in $\underline{x}$ and not in $\underline{o}$; the minimization routine (line 10) can be realized

```
function FTA-Backward-DCOI (M, Tle)
1   i := 0;
2   M := Extend(M, R^g);
3   Reach := Tle ∩ (g = f);
4   Front := Tle ∩ (g = f);
5   while (Front ≠ ∅) do
6      temp := Reach;
7      M^i = dcoi_get(M, Tle, i);
8      Reach := Reach ∪
               bwd_img(M^i, Front);
9      Front := Reach\temp;
10     i := i + 1;
11  end while;
12  CS := Proj(g, Reach ∩ I);
13  MCS := Minimize(CS);
14  return Map_{g→f}(MCS);
```

```
function FTA-Forward-Pruning (M, Tle)
1   CS := ∅;
2   M := Extend(M, R^o);
3   Reach := I ∩ (o = f);
4   Front := I ∩ (o = f);
5   while (Front ≠ ∅) do
6      CS := CS ∪ Proj(o, Reach ∩ Tle);
7      temp := Reach;
8      Reach := Reach ∪
               fwd_img(M, Front);
9      Front := Reach\temp;
10     Front := Front\Widen(CS);
11  end while;
12  MCS := Minimize(CS);
13  return Map_{o→f}(MCS);
```

Fig. 3. Backward algorithm, using DCOI; forward algorithm, with pruning

with standard BDD operations (we refer the reader to [14, 26] for details); finally, the mapping function (line 11) can be defined as $Map_{\underline{o}\rightarrow\underline{f}}(\phi(\underline{o})) = \exists\underline{o}.(\phi(\underline{o}) \wedge (\underline{o} = \underline{f}))$.

4.2 Backward Fault Tree Analysis

The backward algorithm performs reachability via the preimage operator bwd_img. This time, we need to "remember" if a failure mode has been activated at some step *in the future*. To this aim, for each $f_i \in \mathcal{F}$ we introduce an additional variable g_i (these variables are referred to as *guess* or *prophecy variables*, and they can be seen as the dual of history variables). Let $\mathcal{R}^g$ be the transition relation defined by the condition $\bigwedge_{f\in\mathcal{F}} ITE(g_i', g_i, g_i \leftrightarrow f_i)$ and $Extend(\mathcal{M}, \mathcal{R}^g)$ be defined as in Sect. 4.1.

The backward algorithm is presented in Fig. 2 (right). The core (lines 4-8) is similar to forward one. It starts from the set of states satisfying the top level event, with the additional constraints that the prophecy variables have been initialized with the same value as the corresponding failure mode variables, and it performs backward reachability until a fixpoint is reached. The resulting set is intersected (line 9) with the initial states, and projected over the prophecy variables. Finally, the minimal cut sets are computed (line 10) and the result is mapped back to the failure mode variables (line 11).

4.3 Backward Fault Tree Analysis with Dynamic Cone of Influence

The algorithm for backward fault tree analysis can be optimized by means of the following technique, referred to as Dynamic Cone of Influence reduction (DCOI). The idea is based on the fact that often models enjoy some local structure, so that the next value of a certain variable only depends on a subset of the whole state variables. Suppose that the top level event Tle only depends on a limited set of variables, say $\underline{x}^0 \subset \underline{x}$. Thus, when computing the preimage of Tle (line 6 in Fig. 2, right), it is possible to

consider only those parts of the Kripke structure that influence the next value of Tle. Such a restricted Kripke structure, referred to as $\mathcal{M}^0$, is typically much simpler than the whole $\mathcal{M}$, and preimages can be computed much more effectively. The resulting preimage may also depend on a restricted set of variables $\underline{x}^1 \subset \underline{x}$, and at the second step the corresponding $\mathcal{M}^1$ is used to compute the preimage. The process is iterated until a fix point is reached; in the worst case, the whole machine is taken into account, but it is possible that convergence is reached before the whole $\mathcal{M}$ is constructed.

The structure of the algorithm (see Fig. 3, left) is the same as the standard backward algorithm. However, at each step i, a different Kripke structure is used, instead of the global $\mathcal{M}$ (lines 7 and 8); the $dcoi_get$ primitive encapsulates the process of lazily constructing the Kripke structure necessary for the i-th preimage computation.

4.4 Dynamic Pruning

All the algorithms previously discussed can be optimized by using dynamic pruning. We describe below the extension of the forward algorithm (the other ones can be extended similarly). The idea is that, at each iteration (lines 4-8 in Fig. 2), it is safe to discard a state, provided we know that it will not contribute to the set of *minimal* cut sets. In particular, this is true whenever we know that the failure modes being active in that state are a superset of a fault configuration that has already been proved to be a cut set. The implementation (see Fig. 3, right) is as follows. At each iteration (line 6) a partial set of cut sets CS is computed. Based on this set, all the states on the frontier, whose active failure modes are a superset of any fault configuration in CS, are pruned (line 10). The primitive $Widen$ is defined as $Widen(CS) = \{s \mid \exists cs \in CS\ (cs \subseteq s)\}$. Intuitively, it collects all the states that include any element in CS as a proper subset (the definition can be extended to the more general case of *prime implicants*). Typically, the use of dynamic pruning may result in a significant reduction of the search space.

5 Implementation in the FSAP Platform

In this section we discuss the implementation of the algorithms described in Sect. 4 in the FSAP platform [7, 16]. As advocated in [8], it is important to have a complete decoupling between the system model and the fault model. For this reason, the FSAP platform relies on the notions of *nominal system model* and *extended system model*. The nominal model formalizes the behavior of the system when it is working as expected, whereas the extended model defines the behavior of the system in presence of faults.

The decoupling between the two models is achieved in the FSAP platform by generating the extended model automatically via a so-called *model extension step*. Model extension takes as input a system and a specification of the faults to be added, and automatically generates the corresponding extended system. It can be formalized as follows. Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be the nominal system model. A fault is defined by the proposition $p \in \mathcal{P}$ to which it must be attached to, and by its type, specifying the "faulty behavior" of p in the extended system (e.g., p can non-deterministically assume a random value, or be stuck at a particular value). FSAP introduces a new proposition p^{FM}, the *failure mode variable*, modeling the possible occurrence of the fault, and two further

propositions p^{Failed} and p^{Ext}, with the following intuitive meaning. The proposition p^{Failed} models the behavior of p when a fault has occurred. For instance, the following condition (where $\mathcal{S}'$ is the set of states of the extended system) defines a so-called *inverted* failure mode (that is, p^{Failed} holds if and only if p does not hold):

$$\forall s \in \mathcal{S}' \quad (s \models p^{Failed} \iff s \not\models p) \tag{1}$$

The proposition p^{Ext} models the extended behavior of p, that is, it behaves as the original p when no fault is active, whereas it behaves as p^{Failed} in presence of a fault:

$$\forall s \in \mathcal{S}' \quad s \not\models p^{FM} \Rightarrow (s \models p^{Ext} \iff s \models p) \tag{2}$$

$$\forall s \in \mathcal{S}' \quad s \models p^{FM} \Rightarrow (s \models p^{Ext} \iff s \models p^{Failed}) \tag{3}$$

The extended system $\mathcal{M}^{Ext} = \langle \mathcal{S}', \mathcal{I}', \mathcal{R}', \mathcal{L}' \rangle$ can be easily defined in terms of the nominal system by adding the new propositions, modifying the definition of the (initial) states and of the transition relation, and imposing the conditions (1) (for an inverted failure mode), (2) and (3). We omit the details for the sake of simplicity. Finally, system extension with respect to a *set* of propositions can be defined in a straightforward manner, by iterating system extension over single propositions.

The system model resulting from the extension step is used in FSAP to carry out the analyses. The algorithms described in Sect. 4 are implemented in FSAP on top of the NuSMV tool [11, 23], using BDDs as explained in Sect. 2.2 (we refer to Sect. 8 for a discussion on alternative algorithms). The FSAP platform can be used to compute both the minimal cut sets and the prime implicants of a given top level event, and the associated traces. FSAP can also compute the probability of the top level event, on the basis of the probabilities of the basic faults, under the hypothesis of independent faults.

6 Experimental Evaluation

In this section we describe the experimental evaluation we carried out. Five algorithms have been evaluated: forward algorithm (FWD in the following), forward algorithm with dynamic pruning (FWD-PRUN), backward algorithm (BWD), backward algorithm with DCOI (DCOI), and backward algorithm with DCOI and dynamic pruning (DCOI-PRUN). Two different test-cases have been used. Both models are of industrial size and have been developed inside the ISAAC project[1]. We remark that the models are covered by a non-disclosure agreement, hence they are only briefly described.

The first model (referred to as "TDS model") is a model of a subsystem of an aircraft. It consists of a mechanical and a pneumatic line, driving a set of utilities, and being controlled by a central unit. Faults are attached to different components of the two lines. We ran different experiments by limiting, at model level, the faults that can be active at any time: in the simplest experiment only 2 faults out of 34 are active. The second test-case (referred to as "Cassini model") is a model of the propulsion system of the Cassini-Huygens spacecraft. The propulsion system is composed of two engines fed by redundant propellant/gas circuit lines. Each line contains several valves and pyrovalves

[1] http://www.isaac-fp6.org

(a pyrovalve being a pyrotechnically actuated valve, whose status – open or close– can be changed at most once). Faults are attached to the engines, the propellant/gas tanks, and the (pyro)valves. We built several variants of the Cassini model by modifying (increasing the redundancy of) the (pyro)valve layer located between the propellant tanks and the engines. The property used to generate the fault tree is related to a correct input pressure in (at least one of the) engines in presence of a correct output pressure from the gas and the propellant tanks.

We ran each experiment with a different invocation of the model checker. For each model, we list the number of minimal cut sets (column **MCS**). For each run, we report the total usage of time (column **T**, in seconds – in parentheses the fraction of time spent for compiling the model) and memory (column **M**, in Mb), and the number of iterations needed to reach the fixpoint (column **K**). Compilation includes parsing, encoding of the variables, and – for all the algorithms except DCOI and DCOI-PRUN – building of the Kripke structure into BDD (the low compilation time for DCOI and DCOI-PRUN is due to the fact that building of the Kripke structure is delayed). The experiments have been run on a 2-processor machine equipped with Intel Xeon 3.00GHz, with 4Gb of memory, running Linux RedHat Enterprise 4 (only one processor was allowed to run for each experiment). The time limit was set to 1 hour and the memory limit to 1 Gb. A '↑' in the time or memory columns stands for a time-out or memory-out, respectively.

The algorithms have been implemented in NuSMV 2.4.1, and run with the following options. In both cases, we used static variable ordering. The motivation is to allow for a fair comparison between the relative performances of the different algorithms, as dynamic re-ordering could affect the performances in an unpredictable manner, depending on how the re-ordering is performed inside the BDD package. We notice that, in general, a good variable ordering is crucial to achieve the best performances, and typically dynamic re-ordering over-performs static ordering. For the TDS model, given the high complexity of the model, we used an off-line pre-computed variable ordering, which has been passed to NuSMV at command line (option `-i`). For the Cassini model, we used a pre-defined static ordering available in NuSMV (option `-vars_order lexicographic`). Furthermore, for the Cassini model we disabled conjunctive partitioning (option `-mono`), in order to purify the results of the DCOI and DCOI-PRUN algorithms from the effect of a better of worse partitioning choice (for the TDS model, this option was not necessary, as it will be evident from the experimental data).

The experimental results are reported in Figs. 4 and 5. The first comment is that there is great variance of performance in the different algorithms. Proceeding forwards appears to be extremely effective in the TDS model, while for the Cassini model, backward search is very effective. This can be partly justified by the different structure of the two models: complex but mostly flat for the TDS model; deeply layered (including different layers of valves connected in series), with the level of layering increasing with the complexity of the model instance, for the Cassini model. In general, as in model checking, the nature of the state spaces may dramatically vary depending on the search direction, and it is not predictable in advance. We claim that, for this reason, it is very important to have available different styles of search.

Second, we notice that the proposed optimizations are always effective, or unnoticeable. DCOI results in dramatic savings for the Cassini model (in fact, the experiments

NR_FAIL	MCS	FWD			FWD-PRUN			BWD			DCOI			DCOI-PRUN		
		T	M	K	T	M	K	T	M	K	T	M	K	T	M	K
2	2	58.7 (8.2)	30	62	9.3 (8.2)	26	11	↑	-	-	↑	-	-	↑	-	-
3	3	102.9 (8.5)	47	67	10.0 (8.5)	26	14	↑	-	-	↑	-	-	↑	-	-
4	4	259.0 (8.6)	67	69	12.1 (8.6)	27	14	↑	-	-	↑	-	-	↑	-	-
5	5	616.3 (8.9)	138	69	15.7 (8.9)	26	14	↑	-	-	↑	-	-	↑	-	-
6	6	521.6 (9.4)	96	69	13.8 (9.4)	27	14	↑	-	-	↑	-	-	↑	-	-
7	7	609.0 (9.5)	131	69	16.2 (9.5)	27	14	↑	-	-	↑	-	-	↑	-	-
8	8	656.3 (9.6)	124	69	16.5 (9.6)	27	14	↑	-	-	↑	-	-	↑	-	-
9	8	1141.9 (10.0)	187	69	16.7 (10.0)	27	14	↑	-	-	↑	-	-	↑	-	-
10	8	2371.5 (10.1)	318	69	19.7 (10.1)	27	14	↑	-	-	↑	-	-	↑	-	-
11	8	↑	-	-	16.2 (10.2)	27	14	↑	-	-	↑	-	-	↑	-	-
12	8	↑	-	-	17.4 (10.4)	27	14	↑	-	-	↑	-	-	↑	-	-
13	8	↑	-	-	20.9 (10.7)	28	14	↑	-	-	↑	-	-	↑	-	-
14	8	↑	-	-	32.4 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
15	8	↑	-	-	25.2 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
16	8	↑	-	-	26.3 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
17	8	↑	-	-	25.0 (11.7)	28	14	↑	-	-	↑	-	-	↑	-	-
18	8	↑	-	-	26.2 (12.0)	29	14	↑	-	-	↑	-	-	↑	-	-
19	8	↑	-	-	27.1 (12.3)	29	14	↑	-	-	↑	-	-	↑	-	-
20	8	↑	-	-	30.5 (12.5)	29	14	↑	-	-	↑	-	-	↑	-	-
34	12	↑	-	-	1219.0 (17.8)	129	14	↑	-	-	↑	-	-	↑	-	-

Fig. 4. Experimental results for the TDS model

MODEL	MCS	FWD			FWD-PRUN			BWD			DCOI			DCOI-PRUN		
		T	M	K	T	M	K	T	M	K	T	M	K	T	M	K
v-2222	329	6.7 (3.2)	15	5	9.0 (3.2)	16	5	37.6 (3.2)	14	5	1.5 (0.1)	13	4	1.5 (0.1)	13	4
v-3222	401	29.0 (3.6)	19	6	14.0 (3.6)	19	6	↑	-	-	1.8 (0.1)	14	5	1.8 (0.1)	15	5
v-3322	489	↑	-	-	29.8 (4.4)	26	6	↑	-	-	2.6 (0.1)	17	5	2.7 (0.1)	17	5
v-3332	599	↑	-	-	578.3 (4.7)	30	6	↑	-	-	3.4 (0.1)	17	5	3.5 (0.1)	17	5
v-3333	734	↑	-	-	↑	-	-	↑	-	-	4.2 (0.1)	17	5	4.3 (0.1)	17	5
v-4333	869	↑	-	-	↑	-	-	↑	-	-	5.5 (0.1)	18	6	6.1 (0.1)	18	6
v-4433	1029	↑	-	-	↑	-	-	↑	-	-	7.3 (0.1)	18	6	7.4 (0.1)	18	6
v-4443	1221	↑	-	-	↑	-	-	↑	-	-	9.4 (0.2)	19	6	10.1 (0.2)	19	6
v-4444	1449	↑	-	-	↑	-	-	↑	-	-	11.9 (0.2)	19	6	12.6 (0.2)	19	6
v-5444	1667	↑	-	-	↑	-	-	↑	-	-	17.8 (0.2)	20	7	17.7 (0.2)	20	7
v-5544	1941	↑	-	-	↑	-	-	↑	-	-	25.0 (0.2)	21	7	28.3 (0.2)	24	7
v-5554	-	↑	-	-	↑	-	-	↑	-	-	↑	-	-	↑	-	-

Fig. 5. Experimental results for the Cassini model

show that DCOI is the crucial factor that makes backward search a winning strategy over forward search). Dynamic pruning has a minor impact in the case of backward search on the Cassini model, but is an enabling factor for TDS, where it substantially

reduces the number of iterations needed to reach convergence. In general, dynamic pruning is more effective when lower-order cut sets are found earlier in the search (that is, they are witnessed by shorter traces), hence its effectiveness is highly model-dependent.

7 Related Work

A large amount of work has been done in the area of probabilistic safety assessment (PSA) and in particular on *dynamic reliability* [29]. Dynamic reliability is concerned with extending the classical event or fault tree approaches to PSA by taking into consideration the mutual interactions between the hardware components of a plant and the physical evolution of its process variables [20]. For different approaches to dynamic reliability see, e.g., [2, 12, 20, 24, 30]. These approaches are mostly concerned with the evaluation (as opposed to generation) of a given fault tree. Concerning fault tree evaluation, we also mention DIFTree [19], a methodology for the analysis of dynamic fault trees, implemented in the Galileo tool [31]. The methodology is able to identify independent sub-trees, translate them into suitable models, analyze them and integrate the results of the evaluation. Different techniques can be used for the evaluation, e.g., BDD-based techniques, Markov techniques or Monte Carlo simulation. Concerning fault tree validation, we mention [28, 32], both concerned with automatically proving the consistency of fault trees using model checking techniques; [32] presents a fault tree semantics based on Clocked CTL (CCTL) and uses timed automata for system specification, whereas [28] presents a fault tree semantics based on the Duration Calculus with Liveness (DCL) and uses Phase Automata as an operational model.

The FSAP platform has been developed within ESACS[2] (Enhanced Safety Assessment for Complex Systems) and ISAAC[1] (Improvement of Safety Activities on Aeronautical Complex systems), two European-Union-sponsored projects involving various research centers and industries from the avionics sector. For a more detailed description of the project goals we refer to [6, 8, 9]. Within the project, the same methodology has been also implemented in other platforms, see e.g. [1, 4, 15, 25]. Regarding model-based safety analysis, we mention [17, 18], sharing some similarities with the ISAAC approach. In particular, the integration of the traditional development activities with the safety analysis activities, based on a formal model of the system, and the clear separation between the nominal model and the fault model, are ideas that have been pioneered by ESACS [8]. The authors propose to integrate this approach into the traditional "V" safety assessment process. Finally, we mention [22, 33, 34], sharing with ISAAC the application field (i.e., avionics), and the use of NuSMV as a target verification language.

Finally, the routines used to extract the set of minimal cut sets are based on classical procedures for *minimization* of boolean functions, specifically on the implicit-search procedures described in [13, 14, 26, 27], based on Binary Decision Diagrams [10].

8 Conclusions

In this paper we have presented a broad range of algorithmic strategies for efficient fault tree analysis of reactive systems. In particular, we have described algorithms

[2] `http://www.esacs.org`

encompassing different directions for reachability analysis, and some useful optimizations. The experimental evaluation showed the complementarity of the search directions and confirmed the impact of the optimizations on the overall performance.

In the future, we intend to investigate the following research directions. First, we intend to explore the automatic combination of forward and backward search. Second, we will explore an alternative symbolic implementation, using SAT-based bounded model checking techniques [5]. As opposed to BDDs, that work by saturating sets of states, these techniques are typically used to find single traces of bounded length. The challenge is to make these techniques complete; a possible solution could be the generalization of induction techniques. Furthermore, we also want to investigate a "hybrid" approach combining BDD-based and SAT-based techniques into the same routine.

Acknowledgments. We wish to thank Antonella Cavallo from Alenia Aeronautica and Massimo Cifaldi from AleniaSIA for allowing us to use the TDS model.

References

1. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing Safe, Reliable Systems using Scade. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, Springer, Heidelberg (2006)
2. Aldemir, T.: Computer-assisted Markov Failure Modeling of Process Control Systems. IEEE Transactions on Reliability R-36, 133–144 (1987)
3. Banach, R., Bozzano, M.: Retrenchment, and the Generation of Fault Trees for Static, Dynamic and Cyclic Systems. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, Springer, Heidelberg (2006)
4. Bieber, P., Castel, C., Seguin, C.: Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In: Proceedings of Dependable Computing EDCC-4: 4th European Dependable Computing Conference, Toulouse, France, October 23-25, 2002. LNCS, vol. 2485, pp. 19–31. Springer, Heidelberg (2002)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, Springer, Heidelberg (1999)
6. Bozzano, M., Cavallo, A., Cifaldi, M., Valacca, L., Villafiorita, A.: Improving Safety Assessment of Complex Systems: An industrial case study. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, Springer, Heidelberg (2003)
7. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA Safety Analysis Platform. Software Tools for Technology Transfer 9(1), 5–24 (2007)
8. Bozzano, M., et al.: ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In: Proc. ESREL 2003, Balkema Publisher (2003)
9. Bozzano, M., et al.: ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In: Proc. ERTS 2006 (2006)
10. Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
11. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. Software Tools for Technology Transfer 2(4), 410–425 (2000)
12. Cojazzi, G., Izquierdo, J.M., Meléndez, E., Perea, M.S.: The Reliability and Safety Assessment of Protection Systems by the Use of Dynamic Event Trees. The DYLAM-TRETA Package. In: Proc. XVIII Annual Meeting Spanish Nucl. Soc. (1992)

13. Coudert, O., Madre, J.C.: Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In: Proc. DAC 1992, IEEE Computer Society Press, Los Alamitos (1992)
14. Coudert, O., Madre, J.C.: Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In: Proc. RAMS 1993 (1993)
15. Deneux, J., Åkerlund, O.: A Common Framework for Design and Safety Analyses using Formal Methods. In: Proc. PSAM7/ESREL 2004 (2004)
16. The FSAP platform. http://sra.itc.it/tools/FSAP
17. Joshi, A., Heimdahl, M.P.E.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, Springer, Heidelberg (2005)
18. Joshi, A., Miller, S.P., Whalen, M., Heimdahl, M.P.E.: A Proposal for Model-Based Safety Analysis. In: Proc. DASC 2005 (2005)
19. Manian, R., Dugan, J.B., Coppit, D., Sullivan, K.J.: Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In: Proc. HASE 1998, IEEE Computer Society Press, Los Alamitos (1998)
20. Marseguerra, M., Zio, E., Devooght, J., Labeau, P.E.: A concept paper on dynamic reliability via Monte Carlo simulation. Math. and Comp. in Simulation 47, 371–382 (1998)
21. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publ., Dordrecht (1993)
22. Miller, S.P., Tribble, A.C., Heimdahl, M.P.E.: Proving the Shalls. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, Springer, Heidelberg (2003)
23. The NuSMV model checker. http://nusmv.itc.it
24. Papazoglou, I.A.: Markovian Reliability Analysis of Dynamic Systems. In: Reliability and Safety Assessment of Dynamic Process Systems, pp. 24–43. Springer, Heidelberg (1994)
25. Peikenkamp, T., Böede, E., Brückner, I., Spenke, H., Bretschneider, M., Holberg, H.-J.: Model-based Safety Analysis of a Flap Control System. In: Proc. INCOSE 2004 (2004)
26. Rauzy, A.: New Algorithms for Fault Trees Analysis. Reliability Engineering and System Safety 40(3), 203–211 (1993)
27. Rauzy, A., Dutuit, Y.: Exact and Truncated Computations of Prime Implicants of Coherent and Non-Coherent Fault Trees within Aralia. Reliability Engineering and System Safety 58(2), 127–144 (1997)
28. Schäfer, A.: Combining Real-Time Model-Checking and Fault Tree Analysis. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, Springer, Heidelberg (2003)
29. Siu, N.O.: Risk Assessment for Dynamic Systems: An Overview. Reliability Engineering and System Safety 43, 43–74 (1994)
30. Smidts, C., Devooght, J.: Probabilistic Reactor Dynamics II. A Monte-Carlo Study of a Fast Reactor Transient. Nuclear Science and Engineering 111(3), 241–256 (1992)
31. Sullivan, K.J., Dugan, J.B., Coppit, D.: The Galileo Fault Tree Analysis Tool. In: Proc. FTCS 1999, IEEE Computer Society Press, Los Alamitos (1999)
32. Thums, A., Schellhorn, G.: Model Checking FTA. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, Springer, Heidelberg (2003)
33. Tribble, A.C., Lempia, D.L., Miller, S.P.: Software Safety Analysis of a Flight Guidance System. In: Proc. DASC 2002 (2002)
34. Tribble, A.C., Miller, S.P.: Software Safety Analysis of a Flight Management System Vertical Navigation Function - A Status Report. In: Proc. DASC 2003 (2003)
35. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission (1981)

Computing Game Values for Crash Games

Thomas Gawlitza and Helmut Seidl

TU München, Institut für Informatik, I2
85748 München, Germany
{gawlitza,seidl}@in.tum.de

Abstract. We consider crash games which are a generalization of parity games in which the play value of a play is an integer, $-\infty$ or ∞. In particular, the play value of a finite play is given as the sum of the payoffs of the moves of the play. Accordingly, one player aims at maximizing the play value whereas the other player aims at minimizing this value. We show that the game value of such a crash game at position v, i.e., the least upper bounds to the minimal play value that can be enforced by the maximum player in a play starting at v, can be characterized by a hierarchical system of simple integer equations. Moreover, we present a practical algorithm for solving such systems. The run-time of our algorithm (w.r.t. the uniform cost measure) is independent of the sizes of occurring numbers. Our method is based on a strategy improvement algorithm. The efficiency of our algorithm is comparable to the efficiency of the discrete strategy improvement algorithm developed by Vöge and Jurdzinski for the simpler Boolean case of parity games [19].

1 Introduction

Crash games are a generalization of parity games where game positions have non-negative ranks and additionally, each possible move of a player comes with a payoff in $\mathbb{Z}$. A play is played by two opponents, the $\vee$-player and the $\wedge$-player. The $\vee$-player wants to maximize the play value while the $\wedge$-player wants to minimize it. The play value of a finite play is determined as the sum of payoffs of moves chosen in the play. The play value of an infinite play, on the other hand, is determined similarly as for parity games: If the least rank of an infinitely often visited position is *odd*, then the $\vee$-player wins, i.e., the play value is ∞. Accordingly, if the least rank of an infinitely often visited position is *even*, then the $\wedge$-player wins, i.e., the play value is $-\infty$.

Thus, crash games are *payoff* games. The notable difference to *mean-payoff* games, for instance, is the fact that the goal for crash games is not to maximize (resp. minimize) the *mean* payoff during a play but the *total* payoff. Similar to *mean-payoff parity games* [3], play values are not only determined by the payoff function but also by a rank function. Also similar to *mean-payoff parity games*, winning strategies are no longer necessarily *positional* (also called *memoryless*) [9,10]. Instead already for quite simple crash games, unbounded memory is required. Another class of games related to crash games are the *longest shortest paths games* from [13]. In contrast to our games, the max player in longest shortest path games is bound to use a *positional* strategy which is not the case in our setting. Also, longest shortest path games do not consider ranks for positions in the game graph.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 177–191, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In this paper we present basic techniques for dealing with crash games. In particular, we show that computing the game values of a crash game can be reduced to solving hierarchical systems of equations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, \infty\}$. The occurring equations are *simple*, i.e., they only use the operations maximum, minimum as well as addition restricted to at most one non-constant argument. Since the lattice has infinite strictly ascending and descending chains, extra insights are necessary to solve hierarchical systems of such equations. Our main technical contribution therefore is to provide a fast practical algorithm for solving these systems.

In hierarchical systems least and greatest fixpoint variables alternate. Such systems naturally occur when model-checking finite-state systems w.r.t. temporal logics formulas (see, e.g., [1]). While classically, only two-valued logics are considered, more general complete lattices have attracted attention recently [4,18]. The approaches in [4,18] are restricted to finite lattices. In [17] the complete lattice of the non-negative integers extended with ∞ is considered. This lattice is of infinite height and hierarchical systems over this lattice allow to analyze quantitative aspects of the behavior of finite-state systems [17]. Opposed to that paper, we here allow also negative integers. Our algorithm for solving hierarchical systems is based on *strategy improvement*. Strategy improvement has been introduced by Howard for solving stochastic control problems [12,16]. For the two-valued logic case, *strategy improvement algorithms* has been suggested for model-checking for the modal μ-calculus as well as for computing game values of parity games [11,15,19].

Our strategy improvement algorithm works directly on the hierarchical system. Thereby a strategy is a function which selects for every expression $e_1 \vee e_2$ ("$\vee$" denotes the maximum-operator) one of the subexpressions e_1, e_2. Thus, a strategy describes which side of a maximum-expression should be used and characterizes a hierarchical system in which maximum-operators do not occur any more. In general, strategy improvement algorithms try to find optimal strategies. Therefore they compute a valuation for a given strategy which, if the strategy is not yet optimal, gives hints on how to improve the strategy locally. In our case the valuation is given as the canonical solution of the system which is described by the strategy.

We have not found a technique to apply this idea to general integer systems directly. Instead, we first consider the case of integer systems where all solutions are guaranteed to be finite. In this case, we can *instrument* the underlying lattice in such a way that the resulting system has exactly one solution — from which the canonical solution of the original system can be read off. The lattice obtained by our instrumentation is closely related to the progress measures proposed by Jurdzinski for computing the winning positions in parity games [14]. Our technique is more general as it also allows to deal with integers instead of booleans. The interesting idea of Jurdzinski (cast in our terminology) is to instrument the Boolean lattice just to replace all greatest fixpoints by unique fixpoints. By this, computing canonical solutions over the Boolean lattice is reduced to computing *least* solutions over the instrumented lattice. A similar idea can also be applied in the integer case — given that the canonical solution is finite. The resulting algorithm, however, will not be *uniform*, i.e., its run-time (w.r.t. the uniform cost measure where, e.g., arithmetic operations are counted for $\mathcal{O}(1)$) may depend on the sizes of occurring numbers. Instead, our instrumentation allows us to construct a uniform

algorithm for computing canonical solutions (given that they are finite) through a generalization of the strategy iteration techniques in [5,8].

Using any method for computing finite canonical solutions as a subroutine, we solve the unrestricted case in two stages. First, we design an algorithm which can deal with positive infinities in the equation system. Repeatedly applying this algorithm then allows us to deal with systems whose canonical solutions may both contain $-\infty$ and ∞.

The rest of the paper is organized as follows. In section 2 we introduce crash games and give simple examples. In section 3, we introduce hierarchical systems of simple equations over $\mathcal{Z}$, which we use as a tool for solving crash games. The corresponding reduction will be discussed in section 4. In section 5, we present our key technical idea for computing canonical solutions. There, we are first restricted to hierarchical systems with a finite canonical solution. In section 6, we apply the developed technique to derive a method for computing canonical solutions without this restriction. We conclude with section 7.

2 Crash Games

In this section we introduce crash games. Such games are played by a $\vee$-player and a $\wedge$-player. They model the situation where two opponents want to maximize (resp. minimize) their total sums of investments and rewards. With each play we therefore associate a play value form the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, \infty\}$, where $-\infty$ and ∞ denote the least and the greatest element, respectively.

The crash game G itself is given by a finite labeled graph whose nodes are equipped with non-negative ranks and whose edges carry payoffs in $\mathbb{Z}$. We assume that every node has at least one out-going edge — besides a distinguished sink node $\mathbf{0}$ indicating the end of finite plays. Each non-sink node either is a $\vee$- or a $\wedge$-node. At a $\vee$-node, the $\vee$-player may choose one of the out-going edges; likewise at a $\wedge$-node, the $\wedge$-player has a choice. The value of a play reaching 0 is the sum of the payoffs of edges chosen during the play. For infinite plays, the play values $-\infty$ or ∞ are determined similarly to the play values of plays in a parity game. Formally, we define a *crash game* as a tuple $G = (V_\vee, V_\wedge, E, c, r)$ where

1. $V_\vee$ and $V_\wedge$ are disjoint finite sets of *positions* that belong to the $\vee$-player and the $\wedge$-player, respectively. The set of all *positions* is the set $V = V_\vee \cup V_\wedge \cup \{\mathbf{0}\}$ where $\mathbf{0} \notin V_\vee \cup V_\wedge$.
2. $E \subseteq V^2$ is the set of *moves* where $\{v\}E \neq \emptyset$ for all $v \in V_\vee \cup V_\wedge$ and $\{\mathbf{0}\}E = \emptyset$.[1] This means that every position besides the position $\mathbf{0}$ must have a successor. The position $\mathbf{0}$ must not have a successor.
3. $c : E \to \mathbb{Z}$ is the *payoff function* which associates a *payoff* to each move.
4. $r : V_\vee \cup V_\wedge \to \mathbb{N}$ is the *rank function* which associates a natural number to each position from $V_\vee \cup V_\wedge$.

A *finite play* over G is a finite word $\pi = v_1 \cdots v_k$ with $v_k = \mathbf{0}$ and $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k-1$. The *play value* $val_G(\pi)$ of the finite play π is the sum

[1] For $E \subseteq V^2$ and $V' \subseteq V$, $V'E$ denotes the set $\{v_2 \in V \mid \exists v_1 \in V' \text{ such that } (v_1, v_2) \in E\}$.

$\sum_{i=1}^{k-1} c((v_i, v_{i+1}))$. Accordingly, an *infinite* play over G is an infinite word $\pi = v_1 v_2 \cdots$ with $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$. Assume that m denotes the natural number $\min\{r(v) \in V_\vee \cup V_\wedge \mid v \text{ occurs infinitely often in } \pi\}$. The *play value* $val_G(\pi)$ then is ∞ if m is odd and $-\infty$ otherwise. By $Play_G$ we denote the set of all plays over G and by $Play_G(v)$ the set of all plays starting at $v \in V$, i.e., $Play_G(v) = Play_G \cap \{v\} \cdot (V^\omega \cup V^*)$.

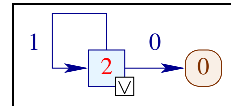
For a finite play $\pi = v_1 \cdots v_k$ (resp. infinite play $\pi = v_1 v_2 \cdots$) the set of *prefixes of* π is the set $\{v_1 \cdots v_i \mid i = 0, \ldots, k\}$ (resp. $\{v_1 \cdots v_i \mid i \in \mathbb{N}_0\}$) which we denote by $Prefix(\pi)$. The set of all prefixes of plays over G ending in a $\vee$-position (resp. $\wedge$-position) is denoted by $Prefix_\vee(G)$ (resp. $Prefix_\wedge(G)$). For a play prefix $\pi = v_1 \cdots v_k$ we write $c(\pi)$ for the sum $\sum_{i=1}^{k-1} c((v_i, v_{i+1}))$. A $\vee$-strategy (resp. $\wedge$-strategy) is a function $f : Prefix_\vee(G) \to V$ (resp. $f : Prefix_\wedge(G) \to V$) where, for every play prefix π, $\pi \cdot f(\pi)$ is again a play prefix. A $\vee$-strategy (resp. $\wedge$-strategy) f is *positional* iff $f(\pi v) = f(\pi' v)$ for all play prefixes $\pi v, \pi' v$ ending in the same $\vee$-position v (resp. $\wedge$-position v). We write $F_\vee(G)$ for the set of all $\vee$-strategies and $F_\wedge(G)$ for the set of all $\wedge$-strategies. The play π is *consistent* with the $\vee$-strategy f (resp. $\wedge$-strategy f) iff for every finite prefix $\pi' v$ of π, $f(\pi') = v$ whenever $\pi' \in Prefix_\vee(G)$ (resp. $\pi' \in Prefix_\wedge(G)$). For a set P of plays, we write $P|_f$ for the set of plays from P that are consistent with f. For a position v, we define the *game value* $\langle\langle v \rangle\rangle_G$ by

$$\langle\langle v \rangle\rangle_G = \bigvee_{f_\vee \in F_\vee(G)} \bigwedge \{val_G(Play_G(v)|_{f_\vee})\}$$

where, for $X \subseteq \mathcal{Z}$, $\bigvee X$ (resp. $\bigwedge X$) denotes the least upper bound (resp. greatest lower bound) of X. Thus, $\langle\langle v \rangle\rangle_G$ is the least upper bound to all play values the $\vee$-player can enforce. These definitions are analogous to the definitions in [18] for multi-valued model checking games. For infinite plays, we inherit the winning condition from parity games as considered, e.g., in [6,19]. For the two-valued case (as well as for the finite-valued case in [18]), however, there exist optimal strategies for each player which are positional. As shown in the following example, this does not hold for crash games.

Example 1

Consider the game $G = (V_\vee, V_\wedge, E, c, r)$ with $V_\vee = \{v\}$, $V_\wedge = \emptyset$, $E = \{(v, v), (v, \mathbf{0})\}$, $c((v, v)) = 1$, $c((v, \mathbf{0})) = 0$ and $r(v) = 2$ (cf. the fig.). The game value for v is ∞. This value, though, cannot be realized by any individual play. Instead there is, for every $z \in \mathbb{Z}$, a $\vee$-strategy f_z such that $val_G(\pi) = z$ for the single play $\pi \in Play_G(v)|_f$. For $z > 0$, this strategy, though, is not *positional*. □



Note that for the two-valued case, the algorithms and constructions heavily rely on the fact that there are optimal positional strategies for both players. For a crash game $G = (V_\vee, V_\wedge, E, c, r)$ we only have the remarkable property that the choice only depends on the current payoff and position. I.e., for a given $z \in \mathbb{Z}$ with $z \leq \langle\langle v \rangle\rangle_G$, there exists a $\vee$-strategy f_z with $\bigwedge \{val_G(Play_G(v)|_f\ )\} \geq z$ and the property that $f_z(\pi v) = f_z(\pi' v)$ whenever $c(\pi v) = c(\pi' v)$.

3 Hierarchical Systems of Simple Integer Equations

In the next section, we will reduce the problem of computing game values of crash games to the problem of solving hierarchical systems of simple integer equations. An integer equation $\mathbf{x} = e$ is called *simple* iff the right-hand side e is of the form

$$e \ ::= \ c \mid \mathbf{x} \mid e + a \mid e_1 \wedge e_2 \mid e_1 \vee e_2$$

where $\mathbf{x}$ is a variable, e, e_1, e_2 are expressions, and $a, c \in \mathbb{Z}$. Note that we restrict addition such that the second argument is always a constant. These second arguments are called *addition constants* whereas other constants c are called *basic constants*. The operator $+a$ has highest precedence, followed by $\wedge$ and finally $\vee$ which has lowest precedence. A *system* $\mathcal{E}$ of simple integer equations is a finite sequence $\mathbf{x}_1 = e_1, \ldots, \mathbf{x}_n = e_n$ of simple integer equations where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are pairwise distinct variables. Let us denote the set of variables $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ occurring in $\mathcal{E}$ by $\mathbf{X}$. The system $\mathcal{E}$ is called *conjunctive* (*disjunctive*) iff no right-hand side contains the maximum-operator "$\vee$" (minimum-operator "$\wedge$"). For a *variable assignment* $\mu : \mathbf{X} \to \mathcal{Z}$ an expression e is mapped to a value $[\![e]\!]\mu \in \mathcal{Z}$ as follows:

$$\begin{array}{llll} [\![c]\!]\mu & = c & [\![\mathbf{x}]\!]\mu \quad = \mu(\mathbf{x}) & [\![a + e]\!]\mu = a + [\![e]\!]\mu \\ [\![e_1 \wedge e_2]\!]\mu & = [\![e_1]\!]\mu \wedge [\![e_2]\!]\mu & [\![e_1 \vee e_2]\!]\mu = [\![e_1]\!]\mu \vee [\![e_2]\!]\mu & \end{array}$$

where $\mathbf{x}$ is a variable, e, e_1, e_2 are expressions, and $a, c \in \mathbb{Z}$. Here, we extend the operation "+" to $\pm\infty$ by: $x + (-\infty) = (-\infty) + x = -\infty$ for all x and $x + \infty = \infty + x = \infty$ for all $x > -\infty$. Thus, "+" distributes over $\vee$ and $\wedge$. Assume that $\mathcal{E}$ denotes the system $\mathbf{x}_1 = e_1, \ldots, \mathbf{x}_n = e_n$. As usual, a *solution* of $\mathcal{E}$ is a variable assignment μ which satisfies all equations of $\mathcal{E}$, i.e. $\mu(\mathbf{x}_i) = [\![e_i]\!]\mu$ for $i = 1, \ldots, n$. We also use the term *fixpoint* instead of solution. We call a variable assignment μ a *pre-solution* of $\mathcal{E}$ iff $\mu(\mathbf{x}_i) \leq [\![e_i]\!]\mu$ for $i = 1, \ldots, n$ and a *post-solution* of $\mathcal{E}$ iff $\mu(\mathbf{x}_i) \geq [\![e_i]\!]\mu$ for $i = 1, \ldots, n$. Note that the function $[\![e]\!] : (\mathbf{X} \to \mathcal{Z}) \to \mathcal{Z}$ is monotonic for every expression e.

A *hierarchical* system $\mathcal{H} = (\mathcal{E}, r)$ of simple integer equations consists of a system $\mathcal{E}$ of simple integer equations and a rank function r mapping the variables $\mathbf{x}_i$ of $\mathcal{E}$ to natural numbers $r(\mathbf{x}_i) \in \{1, \ldots, d\}, d \in \mathbb{N}$. For variables with odd (resp. even) rank, we are interested in greatest (resp. least) solutions. Further, the variables of greater ranks are assumed to live within the scopes of the variables with smaller ranks. We call the resulting solution *canonical*. In order to define the canonical solution formally, let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and $\mathbf{X}_{\bowtie j}$ denote the set of variables $\mathbf{x}_i$ with $r(\mathbf{x}_i) \bowtie j$ where $\bowtie \in \{=, <, >, \leq, \geq\}$. Then the equations $\mathbf{x}_i = e_i$ of $\mathcal{E}$ with $\mathbf{x}_i \in \mathbf{X}_{\geq j}$ define a monotonic mapping $[\![\mathcal{E}, r]\!]_j$ from the set $\mathbf{X}_{<j} \to \mathcal{Z}$ of variable assignments with domain $\mathbf{X}_{<j}$ into the set $\mathbf{X}_{\geq j} \to \mathcal{Z}$ of variable assignments with domain $\mathbf{X}_{\geq j}$. Assume that j is even, i.e., corresponds to a least solution. Given the mapping $[\![\mathcal{E}, r]\!]_{j+1}$, the mapping $[\![\mathcal{E}, r]\!]_j$ is given by:

$$[\![\mathcal{E}, r]\!]_j \ \rho = \mu + [\![\mathcal{E}, r]\!]_{j+1}(\rho + \mu)$$

where $\rho : \mathbf{X}_{<j} \to \mathcal{Z}$ is a variable assignment and $\mu : \mathbf{X}_{=j} \to \mathcal{Z}$ is the *least* variable assignment such that

$$\mu(\mathbf{x}_i) = [\![e_i]\!](\rho + \mu + [\![\mathcal{E}, r]\!]_{j+1}(\rho + \mu))$$

for all $\mathbf{x}_i \in \mathbf{X}_{=j}$. Here, the operator "+" denotes combination of two variable assignments with disjoint domains. The case where j is odd, i.e., corresponds to a greatest solution is analogous. Finally, the canonical solution μ^* is given by $[\![\mathcal{E}, r]\!]_1$ applied to the empty variable assignment $\{\}$. The next example illustrates how one can compute the canonical solution by a transfinite fixpoint iteration.

Example 2. Consider the system of equations

$$\mathbf{x}_1 = 5 + \mathbf{x}_2 \wedge 7 \qquad \mathbf{x}_2 = \mathbf{x}_3 \qquad \mathbf{x}_3 = -5 + \mathbf{x}_1 \vee 1$$

where $r(\mathbf{x}_1) = r(\mathbf{x}_2) = 1$ and $r(\mathbf{x}_3) = 2$. Thus $\mathbf{x}_3$ lives within the scope of $\mathbf{x}_1, \mathbf{x}_2$.

<table>
<tr><td></td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td></tr>
<tr><td>$\mathbf{x}_1$</td><td colspan="2">∞</td><td colspan="2">7</td><td colspan="2">7</td><td colspan="2">7</td></tr>
<tr><td>$\mathbf{x}_2$</td><td colspan="2">∞</td><td colspan="2">∞</td><td colspan="2">∞</td><td colspan="2">2</td></tr>
<tr><td rowspan="2">$\mathbf{x}_3$</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>$-\infty$</td><td>∞</td><td>$-\infty$</td><td>∞</td><td>$-\infty$</td><td>2</td><td>$-\infty$</td><td>2</td></tr>
</table>

The fixpoint iteration is illustrated in the table at the right-hand side. The column labeled with i corresponds to the i-th outer iteration step. The inner iterations are illustrated by the tables in the row for the variables $\mathbf{x}_3$. As for the outer iteration, the column labeled with i contains the value after the i-th inner iteration step. Since we are interested in greatest solutions for the variables $\mathbf{x}_1$ and $\mathbf{x}_2$, the outer iteration starts with the value ∞ for these variables. Then, the inner iteration for $\mathbf{x}_3$ starts with $-\infty$ and reaches a fixpoint after one iteration step. Then, the outer iteration goes on with the new values for $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$. Finally, we get the canonical solution after three outer iteration steps. □

Note that in general transfinite fixpoint iterations are necessary for computing canonical solutions. Related systems over non-negative integers have been considered in [17]. Zero-one-valued systems using minimum and maximum only are also known as *Boolean* fixpoint equations and can be used for checking validity of propositional μ-calculus formulas interpreted over finite labeled transition systems or for computing the winning positions of parity games [1].

4 Computing Game Values

Instead of determining game values of crash games directly, we reduce this problem to solving hierarchical systems of simple integer equations. Although there is a one-to-one correspondence, we are here interested in the reduction from crash games to hierarchical systems only. Let $G = (V_\vee, V_\wedge, E, c, r)$ denote a crash game. We construct a corresponding system $\mathcal{E}_G$ of simple integer equations which uses variables from the set $\mathbf{X} = \{\mathbf{x}_v \mid v \in V_\vee \cup V_\wedge\}$ as follows. For each position $v \in V_\vee$, we add the equation

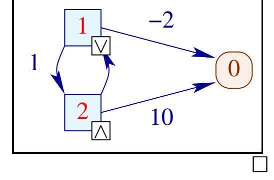
$$\mathbf{x}_v = \bigvee_{v' \in \{v\}E} ([v'] + c(v, v'))$$

and, likewise, for each position $v \in V_\wedge$, we add the equation

$$\mathbf{x}_v = \bigwedge_{v' \in \{v\}E} ([v'] + c(v, v'))$$

where $[v]$ denotes the variable $\mathbf{x}_v$ if $v \in V_\vee \cup V_\wedge$ and $[\mathbf{0}] = 0$. Then the *hierarchical* system $\mathcal{H}_G$ of simple integer equations which corresponds to the crash game G is the pair $(\mathcal{E}_G, r_G)$ where $r_G(\mathbf{x}_v) = r(v)$ for $v \in V_\vee \cup V_\wedge$.

Example 3



The Fig. on the right shows a crash game with two positions, say, 1 and 2 of the respective ranks. Then the corresponding system of integer equations is given by

$$\mathbf{x}_1 = \mathbf{x}_2 + 1 \vee -2 \qquad \mathbf{x}_2 = \mathbf{x}_1 \wedge 10$$

where the rank of $\mathbf{x}_i$ equals i. □

Theorem 1. *For a crash game $G = (V_\vee, V_\wedge, E, c, r)$, let μ^* denote the canonical solution of $\mathcal{H}_G$. Then $\langle\!\langle v \rangle\!\rangle_G = \mu^*(\mathbf{x}_v)$ for all $v \in V_\vee \cup V_\wedge$. Furthermore $\mathcal{H}_G$ can be constructed in time $\mathcal{O}(|V_\vee \cup V_\wedge| + |E|)$. Vice-versa, given a hierarchical system of equations $\mathcal{H}$ we can compute a crash game $G = (V_\vee, V_\wedge, E, c, r)$ in time $\mathcal{O}(|\mathcal{E}|)$ whose game values corresponds to the values of the canonical solution of $\mathcal{H}$.* □

Note that theorem 1 also holds if we define $\langle\!\langle v \rangle\!\rangle_G$ as $\bigwedge_{f_\wedge \in F_\wedge(G)} \bigvee \{val_G(\pi) \mid \pi \in Play_{G,v}|_{f_\wedge}\}$. Thus, we get the following duality theorem as a corollary:

Theorem 2. *Let $G = (V_\vee, V_\wedge, E, c, r)$ be a crash game and $v \in V_\vee \cup V_\wedge \cup \{\mathbf{0}\}$. Then $\langle\!\langle v \rangle\!\rangle_G = \bigwedge_{f_\wedge \in F_\wedge(G)} \bigvee \{val_G(Play_G(v)|_{f_\wedge})\}$.* □

5 Solving Hierarchical Systems

In this section, we present our strategy improvement algorithm for computing canonical solutions. Assume that $\mathcal{H} = (\mathcal{E}, r)$ is a hierarchical system of simple equations where the range of r is contained in the set $\{1, \dots, d\}$. Instead of solving the original system over $\mathcal{Z}$, we consider a corresponding system over an instrumented lattice. In case that all solutions of this system are finite, the instrumentation will assure that the canonical solution is the only solution. The instrumentation technique here is a generalization of the instrumentation used in [8] to determine *least solutions* of systems of integer equations. We instrument $\mathcal{Z}$ by introducing one extra component from $\mathbb{N}$ for every $j = 1, \dots, d$. Thus, we consider the instrumented lattice $\mathcal{D}_d = \mathbb{D}_d \cup \{-\infty, \infty\}$ with $\mathbb{D}_d = \mathbb{Z} \times \mathbb{N}^d$ where $-\infty$ is the least and ∞ is the greatest element and the ordering on $\mathbb{D}_d$ (the *finite* elements of $\mathcal{D}_d$) is given by:

$$(a, j_1, \dots, j_d) < (a', j'_1, \dots, j'_d)$$

iff $a < a'$ or $a = a'$ and there exists some $1 \le k \le d$ with the following properties:

1. $j_i = j'_i$ for all $i < k$;
2. $j_k > j'_k$ whenever k is even;
3. $j_k < j'_k$ whenever k is odd.

Note that values get larger w.r.t. this ordering when components corresponding to greatest fixpoints are increased or components corresponding to least fixpoints are decreased. Addition on the finite elements of $\mathcal{D}_d$ operates on all components simultaneously, i.e.:

$$(a, j_1, \dots, j_d) + (a', j'_1, \dots, j'_d) = (a + a', j_1 + j'_1, \dots, j_d + j'_d)$$

Note that $+$ distributes over $\vee$ and $\wedge$. The evaluation of an expression e over $\mathcal{D}_d$ will be denoted by $[\![e]\!]^\sharp$. As for expressions over $\mathcal{Z}$, $[\![e]\!]^\sharp$ is monotonic.

A slightly different choice is made in [8] where an extra operator *inc* is introduced for incrementing the extra component. Accordingly, our *lifting* transformation differs from the one chosen in [8]. In order to *lift* an equation $\mathbf{x}_i = e_i$ to $\mathcal{D}_d$, we replace every finite constant $c \in \mathbb{Z}$ occurring in e_i with $(c, 0, \ldots, 0)$. Moreover, we replace every equation $\mathbf{x}_i = e_i$ with $\mathbf{x}_i = e_i + \mathbf{1}_{r(\mathbf{x}_i)}$ where $\mathbf{1}_k$ is the $(d+1)$-tuple consisting of 0 everywhere besides the $(k+1)$-th component where it equals 1. We denote the lifted system of simple integer equations by $\mathcal{E}^\sharp$. To simplify notations we define $\beta : \mathcal{D}_d \to \mathcal{Z}$ by

$$\beta(-\infty) = -\infty \qquad \beta(\infty) = \infty \qquad \beta(z, j_1, \ldots, j_d) = z.$$

The following theorem states that we can recover the canonical solution of the original system from the canonical solution of the corresponding lifted system.

Theorem 3. *Assume that $(\mathcal{E}, r)$ is a hierarchical system. Let $\mathcal{E}^\sharp$ be the lifted system corresponding to $\mathcal{E}$ and let $\mu^\sharp$ be the canonical solution of the hierarchical system $(\mathcal{E}^\sharp, r)$. Then $\beta \circ \mu^\sharp$ is the canonical solution of $(\mathcal{E}, r)$.* □

Our key observation is that *finite* solutions of lifted systems are unique. Here, a variable assignment μ is called *finite* iff $-\infty < \mu(\mathbf{x}_i) < \infty$ for all variables $\mathbf{x}_i$. For an equation system $\mathcal{E}$, let $a_\mathcal{E}$ denote the sum of the smallest basic constant together with all negative addition constants. Moreover, let $b_\mathcal{E}$ denote the sum of the largest basic constant together with all positive addition constants. We have:

Theorem 4. *Assume that $\mathcal{E}^\sharp$ is the system of lifted equations corresponding to the hierarchical system $(\mathcal{E}, r)$ with n variables where n_k variables are of rank k. Then:*

1. *$\mathcal{E}^\sharp$ has at most one finite solution.*
2. *If $-\infty < \mu(\mathbf{x}_i) < \infty$ for a solution μ of $\mathcal{E}^\sharp$ and a variable $\mathbf{x}_i$, then $\mu(\mathbf{x}_i) = (a, j_1, \ldots, j_d)$ for some $a \in [a_\mathcal{E}, b_\mathcal{E}]$ and $j_1, \ldots, j_d \in \mathbb{N}$ with $0 \leq j_k \leq n_k$.*
3. *If $\mathcal{E}^\sharp$ is conjunctive, the greatest solution of $\mathcal{E}^\sharp$ can be computed in time $\mathcal{O}(d \cdot n \cdot |\mathcal{E}|)$.*

Proof. To simplify the proof, here we additionally allow the constants $-\infty$ and ∞ to occur as basic constants. In order to prove assertion 1, we first consider the case of a lifted system which consists in exactly one equation. W.l.o.g. consider the equation

$$\mathbf{x} = \mathbf{x} + a \wedge b \vee c$$

where $(0, \ldots, 0) \neq a \in \mathbb{D}_d$ and $b, c \in \mathcal{D}_d$. Assume that μ is a finite solution of the above system. We show that μ is given by

$$\mu(\mathbf{x}) \;=\; \begin{cases} c & \text{if } a < (0, \ldots, 0) \\ b \vee c & \text{if } a > (0, \ldots, 0). \end{cases}$$

Note that necessarily $c \leq \mu(\mathbf{x}) \leq b \vee c$. If $b \leq c$, the statement follows immediately. Assume therefore that $b > c$. First assume that $a > (0, \ldots, 0)$ and $c \leq \mu(\mathbf{x}) < b \vee c = b$. Then $\mu(\mathbf{x}) = [\![\mathbf{x} + a]\!]^\sharp \mu$ — which is impossible for finite values. Now assume that $a < (0, \ldots, 0)$ and $c < \mu(\mathbf{x}) \leq b \vee c = b$. Then $\mu(\mathbf{x}) = [\![\mathbf{x} + a]\!]^\sharp \mu$ — which is again impossible for finite values.

Now we consider the general case. We show assertion 1 and 2 simultaneously. Therefore, we first introduce the following notations. Let $\mathcal{E}^\sharp$ denote a system of equations over $\mathcal{D}_d$. We call a sequence π of expressions $e_1, \ldots, e_k$ a *path* iff for $i = 1, \ldots, k-1$ either e_i is a variable $\mathbf{x}_j$ and $\mathbf{x}_j = e_{i+1}$ is an equation of $\mathcal{E}^\sharp$ or e_{i+1} is an immediate subexpression of e_i. The path π is *short* iff all expressions e_i that are variables are distinct. In order to define the weight of a system of simple integer equations, we set $w(e) = a$, if e denotes an expression $e' + a$, $w(e) = c$, if e denotes an expression c, and, $w(e) = 0$ otherwise. Thereby e, e' denote expressions, a denotes an addition constant and c a basic constant. Then, the sum $\sum_{i=1,\ldots,k} w(e_i)$ is called the *weight* of the path. Let P denote the set of all short paths ending with a *finite* basic constant. We define $w_{max}(\mathcal{E}^\sharp)$ (resp. $w_{min}(\mathcal{E}^\sharp)$) as the maximal (resp. minimal) weight of paths in P. Furthermore, for $j = 1, \ldots, d$, we define $w_j(\mathcal{E}^\sharp)$ to be the maximum of the $j+1$-th component of the weights of paths in P. We call $[w_{min}, w_{max}]$ and w_j for $j = 1, \ldots, d$ the weights of $\mathcal{E}^\sharp$.

Let $\mathcal{E}^\sharp$ be the lifted system $\mathbf{x}_1 = e_1, \ldots, \mathbf{x}_n = e_n$. Assume that μ is a finite solution of $\mathcal{E}^\sharp$. We show by induction on the number of variables occurring in right-hand sides that the following holds:

1. μ is the only finite solution of $\mathcal{E}^\sharp$;
2. $\mu(\mathbf{x}) \in [w_{min}(\mathcal{E}^\sharp), w_{max}(\mathcal{E}^\sharp)]$ for every variable $\mathbf{x}$ and
3. the $(j+1)$-th component of $\mu(\mathbf{x})$ is less than or equal to $w_j(\mathcal{E}^\sharp)$.

The statement is obviously fulfilled if there are no variables in right-hand sides. For the induction step let $\mathbf{x}_i$ be a variable that occurs in a right-hand side of $\mathcal{E}^\sharp$, and consider the equation $\mathbf{x}_i = e_i$ of $\mathcal{E}^\sharp$.

If e_i does not contain $\mathbf{x}_i$, we can substitute e_i everywhere in the remaining equations for $\mathbf{x}_i$ to obtain a system $\mathcal{E}^{\sharp\prime}$ with the same set of solutions and the same weights. Since $\mathbf{x}_i$ does not occur in right-hand sides any more, the result follows by the induction hypothesis.

Otherwise, we first have to replace the equation $\mathbf{x}_i = e_i$ by an equation $\mathbf{x}_i = e$ s.t. (1) e does not contain the variable $\mathbf{x}_i$ and (2) the systems $\mathbf{x}_i = e_i\sigma$ and $\mathbf{x}_i = e\sigma$ have the same set of finite solutions for every substitution σ mapping variables other than $\mathbf{x}_i$ to finite values. Then replacing $\mathbf{x}_i = e_i$ by $\mathbf{x}_i = e$ will preserve the set of finite solutions. A suitable expression e is constructed as follows. By using distributivity, we rewrite the equation $\mathbf{x}_i = e_i$ into

$$\mathbf{x}_i = \mathbf{x}_i + a \wedge e_1' \vee e_2'$$

where $\mathbf{x}_i + a \wedge e_1' \vee e_2'$ is in disjunctive normal form. Given e' as $e_1' \vee e_2'$ if $a > (0, \ldots, 0)$ and as e_2' if $a < (0, \ldots, 0)$, we get, from our considerations for a single equation, that the systems $\mathbf{x}_i = e_i\sigma$ and $\mathbf{x}_i = e'\sigma$ (which consist of a single equation) have the same set of finite solutions for every substitution σ mapping variables other than $\mathbf{x}_i$ to finite values. Since $e_1' \vee e_2'$ and e_2' are in disjunctive normal form and have one occurrence of the variable $\mathbf{x}_i$ less than e, this process can be repeated to eliminate all occurrences of the variable $\mathbf{x}_i$. Doing so, an expression e with the desired properties can be obtained. Thus, we can replace the equation $\mathbf{x}_i = e_i$ with $\mathbf{x}_i = e$ and substitute every occurrence of $\mathbf{x}_i$ in right-hand sides with e to obtain a system $\mathcal{E}^{\sharp\prime}$ of equations which has the same

set of finite solutions as $\mathcal{E}^\sharp$. Furthermore the weights of $\mathcal{E}^{\sharp\prime}$ are less than or equal to the weights of $\mathcal{E}^\sharp$. Since $\mathbf{x}_i$ does not occur in a right-hand side of $\mathcal{E}^{\sharp\prime}$, we can apply the induction hypothesis to finish the proof.

The above implies assertion 1. Since

$$[w_{min}(\mathcal{E}^\sharp), w_{max}(\mathcal{E}^\sharp)] \subseteq [(a_\mathcal{E}, 0, n_2, \ldots), (b_\mathcal{E}, n_1, 0, \ldots)]$$

and $w_j(\mathcal{E}^\sharp) \leq n_j$ for $j = 1, \ldots, d$ assertion 2 follows for finite solutions. Non-finite solutions can be reduced to the finite case by removing all infinite variables. The third assertion holds, since, by similar arguments as in [7], n rounds of Round-Robin iteration suffice to compute μ. Since elements in $\mathbb{D}_d$ are $(d+1)$-tuples, addition and comparison has uniform complexity $\mathcal{O}(d)$. □

In particular, theorem 4 implies that finite values in solutions are *bounded*:

Corollary 1. *Assume that μ^* denotes the canonical solution of a hierarchical system $(\mathcal{E}, r)$ over $\mathcal{Z}$. Then (1) $\mu^*(\mathbf{x}_i) = -\infty$ whenever $\mu^*(\mathbf{x}_i) < a_\mathcal{E}$ and (2) $\mu^*(\mathbf{x}) = \infty$ whenever $\mu^*(\mathbf{x}_i) > b_\mathcal{E}$.* □

Note that this corollary has important consequences for crash games. It implies that every finite game value lies in the interval $[a_\mathcal{E}, b_\mathcal{E}]$.

Now assume that the canonical solution μ^* of the hierarchical system $(\mathcal{E}, r)$ over $\mathcal{Z}$ and hence also the canonical solution $\mu^\sharp$ of the corresponding lifted hierarchical system $(\mathcal{E}^\sharp, r)$ is finite and thus by theorem 4 the only finite solution. By theorem 3 our problem of computing μ^* reduces to the computation of $\mu^\sharp$. Assume that $\mathcal{E}^\sharp$ consists of the equations $\mathbf{x}_i = e_i, i = 1, \ldots, n$, and let $a^\sharp_\mathcal{E}$ and $b^\sharp_\mathcal{E}$ denote the corresponding lifted constants:

$$a^\sharp_\mathcal{E} = (a_\mathcal{E}, 0, n_2, 0, n_4 \ldots) \qquad b^\sharp_\mathcal{E} = (b_\mathcal{E}, n_1, 0, n_3, 0, \ldots)$$

where n_k is the number of variables of rank k. In order to compute $\mu^\sharp$, we replace each equation $\mathbf{x}_i = e_i$ of $\mathcal{E}^\sharp$ with $\mathbf{x}_i = e_i \wedge b^\sharp_\mathcal{E} \vee a^\sharp_\mathcal{E}$. For simplicity, we denote the resulting system again by $\mathcal{E}^\sharp$. Since $\mathcal{E}^\sharp$ does not have non-finite solutions any more, by theorem 4, $\mu^\sharp$ is now the *only* solution of $\mathcal{E}^\sharp$. In order to compute $\mu^\sharp$ we propose *strategy iteration*.

Let $M_\vee(\mathcal{E}^\sharp)$ denote the set of all $\vee$-expressions in $\mathcal{E}^\sharp$. A *strategy* π is a function mapping every expression $e_1 \vee e_2$ in $M_\vee(\mathcal{E}^\sharp)$ to one of the subexpressions e_1, e_2. Given a strategy π together with an expression e, we write $e\pi$ for the expression obtained by recursively replacing every $\vee$-expression in $\mathcal{E}^\sharp$ by the respective subexpression selected by π. Formally, $e\pi$ is given as:

$$\begin{array}{lll} c\pi = c & (e_1 \wedge e_2)\pi = e_1\pi \wedge e_2\pi & \mathbf{x}_i\pi = \mathbf{x}_i \\ (e + a)\pi = e\pi + a\pi & (e_1 \vee e_2)\pi = (\pi(e_1 \vee e_2))\pi & \end{array}$$

Accordingly, the system $\mathcal{E}^\sharp(\pi)$ of equations extracted from $\mathcal{E}^\sharp$ via the strategy π is the system $\mathbf{x}_i = e_i\pi$, $i = 1, \ldots, n$, assuming that $\mathcal{E}^\sharp$ is the system $\mathbf{x}_i = e_i$, $i = 1, \ldots, n$. $\mathcal{E}^\sharp(\pi)$ is a conjunctive system.

Assume that $\mu^\sharp_\pi$ denotes the greatest solution of $\mathcal{E}^\sharp(\pi)$ for a strategy π. By monotonicity, $\mu^\sharp_\pi \leq \mu^\sharp$ for all strategies π. Given a strategy π and the greatest solution $\mu^\sharp_\pi$ of

Algorithm 1. Strategy Improvement Algorithm

```
/* The system E♯ has only finite solutions. */
μ ← variable assignment which maps every variable to −∞;
while (μ is not a solution of E♯) {
    π ← P(μ);
    μ ← greatest solution of E♯(π);
}
return μ;
```

$\mathcal{E}^\sharp(\pi)$ our goal is to determine an improved strategy π' such that the greatest solution $\mu^\sharp_{\pi'}$ of $\mathcal{E}^\sharp(\pi')$ is nearer to $\mu^\sharp$ than $\mu^\sharp_\pi$, i.e. $\mu^\sharp_\pi < \mu^\sharp_{\pi'} \leq \mu^\sharp$. Thereby we pursue the policy to modify π at all expressions $e = e_1 \vee e_2$ where $[\![\pi(e)]\!]^\sharp \mu^\sharp_\pi \neq [\![e]\!]^\sharp \mu^\sharp_\pi$ simultaneously. Therefore, we define the strategy $P(\mu)$ induced by a variable assignment μ by:

$$P(\mu)(e_1 \vee e_2) = \begin{cases} e_1 & \text{if } [\![e_1]\!]^\sharp \mu \geq [\![e_2]\!]^\sharp \mu \\ e_2 & \text{if } [\![e_1]\!]^\sharp \mu < [\![e_2]\!]^\sharp \mu \end{cases}$$

The following lemma implies that we can consider $P(\mu^\sharp_\pi)$ as an improvement of π.

Lemma 1. *Let $\mu^\sharp$ denote the only solution of the system $\mathcal{E}^\sharp$. Let $\mu < \mu^\sharp$ be a pre-solution of $\mathcal{E}^\sharp$. Let μ' denote the greatest solution of $\mathcal{E}^\sharp(P(\mu))$. Then $\mu < \mu'$.*

Proof. Since $\mu^\sharp$ is the only solution of $\mathcal{E}^\sharp$, μ is no solution of $\mathcal{E}^\sharp$. By the definition of P, μ is also a pre-solution of $\mathcal{E}^\sharp(P(\mu))$ and no solution of $\mathcal{E}^\sharp(P(\mu))$. Since μ is a pre-solution, Knaster-Tarski's fixpoint theorem implies that $\mu \leq \mu'$. Moreover, $\mu \neq \mu'$, since μ is no solution. □

According to lemma 1, we can compute $\mu^\sharp$ using alg. 1. For the correctness of alg. 1 consider the following argumentation. Obviously, alg. 1 returns the unique solution $\mu^\sharp$ of $\mathcal{E}^\sharp$ whenever it terminates. Let $\pi_1, \pi_2, \ldots$ denote the sequence of occurring strategies. Since the program variable μ is always the greatest solution of $\mathcal{E}^\sharp(\pi)$ for some strategy π, μ is always a pre-solution of $\mathcal{E}^\sharp$ and $\mu \leq \mu^\sharp$. Therefore, by lemma 1, the greatest solutions $\mu^\sharp_{\pi_i}$ of $\mathcal{E}^\sharp(\pi_i)$ form a strictly increasing sequence. In particular no strategy occurs twice in the sequence $\pi_1, \pi_2, \ldots$. Since the total number of strategies is bounded, the algorithm eventually terminates. For a precise characterization of the run-time, let $\Pi(m)$ denote the maximal number of updates of strategies necessary for systems with m $\vee$-expressions. We have:

Theorem 5. *Assume that $(\mathcal{E}, r)$ is hierarchical system with n variables and m $\vee$-expressions where the canonical solution μ^* of $(\mathcal{E}, r)$ is finite. Then μ^* can be computed by strategy iteration in time $\mathcal{O}(d \cdot n \cdot |\mathcal{E}| \cdot \Pi(m + n))$.* □

The following example illustrates a run of alg. 1.

Example 4. Consider the system $(\mathcal{E}, r)$ given by:

$$\mathbf{x}_1 = \mathbf{x}_2 + -1 \wedge 10 \qquad \mathbf{x}_2 = \mathbf{x}_1 \wedge \mathbf{x}_2 + 1 \vee 0$$

where $r(\mathbf{x}_1) = 1$ and $r(\mathbf{x}_2) = 2$. The canonical solution maps $\mathbf{x}_1$ to -1 and $\mathbf{x}_2$ to 0. The corresponding lifted system $\mathcal{E}^\sharp$ is given by

$$\begin{aligned}\mathbf{x}_1 &= ((\mathbf{x}_2+(-1,0,0) \wedge (10,0,0))+(0,1,0)) \wedge (11,1,0) \vee (-1,0,1)\\ \mathbf{x}_2 &= ((\mathbf{x}_1 \wedge \mathbf{x}_2+(1,0,0) \vee (0,0,0))+(0,0,1)) \wedge (11,1,0) \vee (-1,0,1)\end{aligned}$$

where we already have added the safe lower and upper bounds. After the first iteration, the value of the program variable μ is the variable assignment μ_0 mapping every variable to the lower bound $(-1,0,1)$. The resulting strategy $P(\mu_0)$ gives as the system:

$$\mathbf{x}_1 = (-1,0,1) \qquad \mathbf{x}_2 = (0,0,0)+(0,0,1) \wedge (11,1,0)$$

with the obvious greatest solution. Accordingly, the next improvement results in:

$$\begin{aligned}\mathbf{x}_1 &= ((\mathbf{x}_2+(-1,0,0) \wedge (10,0,0))+(0,1,0) \wedge (11,0,0)\\ \mathbf{x}_2 &= (0,0,0)+(0,0,1) \wedge (11,1,0)\end{aligned}$$

giving us the unique finite solution of $\mathcal{E}^\sharp$ that corresponds to the canonical solution. □

The efficiency of strategy iteration crucially depends on the size of the factor $\Pi(m)$. In practical implementations this factor seems to be surprisingly small. Interestingly, though, it is still open whether (or: under which circumstances) the trivial upper bound of 2^m for $\Pi(m)$ can be significantly improved [19,2].

6 General Canonical Solutions

In this section we show how the restriction to finite canonical solutions can be lifted. The idea is to successively identify variables which are $-\infty$ or ∞ in the canonical solution and to remove these from the system until the remaining system has a finite canonical solution. Let $\mathcal{B} = \{0 < 1\}$ denote the Boolean lattice, and consider the mappings $\alpha_{-\infty} : \mathcal{Z} \to \mathcal{B}$ and $\alpha_\infty : \mathcal{Z} \to \mathcal{B}$ defined by:

$$\begin{array}{lll}\alpha_{-\infty}(-\infty) = 0 & \alpha_{-\infty}(z) = 1 & \text{if } z > -\infty\\ \alpha_\infty(\infty) \quad = 1 & \alpha_\infty(z) \quad = 0 & \text{if } z < \infty\end{array}$$

The mapping $\alpha_{-\infty}$ commutes with arbitrary least upper bounds whereas the mapping α_∞ commutes with arbitrary greatest lower bounds. Additionally, we have:

$$\begin{array}{ll}\alpha_{-\infty}(x+a) = \alpha_{-\infty}(x) & \alpha_\infty(x+a) = \alpha_\infty(x)\\ \alpha_{-\infty}(x \vee y) = \alpha_{-\infty}(x) \vee \alpha_{-\infty}(y) & \alpha_\infty(x \vee y) = \alpha_\infty(x) \vee \alpha_\infty(y)\\ \alpha_{-\infty}(x \wedge y) = \alpha_{-\infty}(x) \wedge \alpha_{-\infty}(y) & \alpha_\infty(x \wedge y) = \alpha_\infty(x) \wedge \alpha_\infty(y)\end{array}$$

where $x, y \in \mathcal{Z}$ and $a \in \mathbb{Z}$. Thus, the mappings $\alpha_{-\infty}$ and α_∞ are homomorphisms mapping the operations "$+\,a$", "$\wedge$", and "$\vee$" on $\mathcal{Z}$ to logical connectivities. Using these abstractions we define, for a system $\mathcal{E}$ of equations over $\mathcal{Z}$, the system $\mathcal{E}_{-\infty}$ of equations over $\mathcal{B}$ as the system obtained from $\mathcal{E}$ by applying $\alpha_{-\infty}$ to all constants and substituting the operators accordingly. Analogously, we define the system $\mathcal{E}_\infty$ of equations over $\mathcal{B}$ using the abstraction α_∞. The following lemma enables us to identify some variables that are $-\infty$ or ∞ in the canonical solution.

Lemma 2. *Assume $(\mathcal{E}, r)$ is a hierarchical equation system over $\mathcal{Z}$ with canonical solution μ^*. Let $\mu^*_{-\infty}$ and μ^*_{∞} denote the canonical solutions of $(\mathcal{E}_{-\infty}, r)$ and $(\mathcal{E}_{\infty}, r)$, respectively. Then (1) $\mu^*(\mathbf{x}_i) = -\infty$ whenever $\mu^*_{-\infty}(\mathbf{x}_i) = 0$ and (2) $\mu^*(\mathbf{x}_i) = \infty$ whenever $\mu^*_{\infty}(\mathbf{x}_i) = 1$.* □

In order to deal with the second source of infinities we introduce the following notions. For $z \in \mathcal{Z}$, we write $(\mathcal{E}^{\vee z}, r)$ for the system obtained from $(\mathcal{E}, r)$ by replacing $\mathbf{x}_i = e_i$ with $\mathbf{x}_i = e_i \vee z$ for every variable $\mathbf{x}_i$ with odd rank. Accordingly, we write $(\mathcal{E}^{\wedge z}, r)$ for the system obtained from $(\mathcal{E}, r)$ by replacing $\mathbf{x}_i = e_i$ with $\mathbf{x}_i = e_i \wedge z$ for every variable $\mathbf{x}_i$ with even rank. We have:

Lemma 3. *Consider a hierarchical system $(\mathcal{E}, r)$ of integer equations. Assume that μ^* denotes the canonical solution of $(\mathcal{E}, r)$, and that $\mu_{a_\mathcal{E}}$ and $\mu_{b_\mathcal{E}}$ denote the canonical solutions of $(\mathcal{E}^{\vee a_\mathcal{E} - 1}, r)$ and $(\mathcal{E}^{\wedge b_\mathcal{E} + 1}, r)$, respectively. Then:*

1. *$\mu^* \leq \mu_{a_\mathcal{E}}$ and $\mu_{b_\mathcal{E}} \leq \mu^*$;*
2. *$\mu_{a_\mathcal{E}} = \mu^*$ whenever $\mu_{a_\mathcal{E}}(\mathbf{x}_i) \geq a$ for all variables $\mathbf{x}_i$ of odd rank;*
3. *$\mu_{b_\mathcal{E}} = \mu^*$ whenever $\mu_{b_\mathcal{E}}(\mathbf{x}_i) \leq b$ for all variables $\mathbf{x}_i$ $\mathbf{x}_i$ of even rank.* □

In order to compute the solution of a hierarchical system of simple integer equations, we proceed in two steps. First, we only consider systems with canonical solutions μ^* which never return $-\infty$, i.e. $\mu^*(\mathbf{x}_i) > -\infty$ for every variable $\mathbf{x}_i$.

Theorem 6. *Assume that $(\mathcal{E}, r)$ is a hierarchical system of simple integer equations with n variables and m occurrences of "$\vee$" where the canonical solution μ^* never returns $-\infty$. Then μ^* can be computed in time $\mathcal{O}(d \cdot n^2 \cdot |\mathcal{E}| \cdot \Pi(m + n))$.*

Proof. Assume that $\mathcal{E}$ consists of the equations $\mathbf{x}_i = e_i$ for $i = 1, \ldots, n$. Let μ_∞ denote the canonical solution of $(\mathcal{E}_\infty, r)$. In the first stage, we remove all variables $\mathbf{x}_i$ with $\mu_\infty(\mathbf{x}_i) = 1$. That means that we obtain a system $\mathcal{E}'$ from $\mathcal{E}$ by removing all equations $\mathbf{x}_i = e_i$ s.t. $\mu_\infty(\mathbf{x}_i) = 1$ and replacing all occurrences of these variables in right-hand sides by the constant ∞. The hierarchical system $(\mathcal{E}', r)$ is equivalent to $(\mathcal{E}, r)$ and moreover $({\mathcal{E}'}_\infty, r)$ is equivalent to $(\mathcal{E}_\infty, r)$ for the remaining variables.

For the second stage, assume w.l.o.g. that for all variables $\mathbf{x}_i$ of $(\mathcal{E}, r)$, $\mu_\infty(\mathbf{x}_i) = 0$. Now let $b := b_\mathcal{E}$ denote the upper bound of $\mathcal{E}$ for finite values. Then by corollary 1 $\mu^*(\mathbf{x}_i) \leq b$, if $\mu^*(\mathbf{x}_i) \in \mathbb{Z}$. Since the canonical solution μ_b of $(\mathcal{E}^{\wedge b+1}, r)$ is finite, it can be computed by strategy iteration according to theorem 5.

If $\mu_b(\mathbf{x}_i) \leq b$ for all variables $\mathbf{x}_i$ we are done, since lemma 3 implies that $\mu^* = \mu_b$. Otherwise, $\mu_b(\mathbf{x}_i) > b$ for some variable $\mathbf{x}_i$ which (using corollary 1) implies that $\mu^*(\mathbf{x}_i) = \infty$. Then we can again remove the equation for $\mathbf{x}_i$ and replace all occurrences of the variable $\mathbf{x}_i$ in right-hand sides of the remaining equations by ∞ to obtain an equivalent system.

Repeating this two-stage procedure for the resulting system with fewer variables may identify more variables of value ∞ until either all variables are found to obtain values ∞ or have values at most b. Then we have found the canonical solution μ^*. Summarizing, we apply the sub-routine for finite canonical systems at most n times for computing the abstractions and another n times for the bounded versions of the integer system — giving us the complexity statement. □

In the second step, we also lift the restriction that canonical solutions should never return $-\infty$. The key idea is now to repeatedly use the abstraction $\alpha_{-\infty}$ together with the algorithm from theorem 6.

Theorem 7. *Assume that* $(\mathcal{E}, r)$ *is a hierarchical system of simple integer equations with* n *variables and* m *occurrences of the maximum operator. Then the canonical solution can be computed in time* $\mathcal{O}(d \cdot n^3 \cdot |\mathcal{E}| \cdot \Pi(m + n))$. □

Proof. Assume that $\mathcal{E}$ consists of the equations $\mathbf{x}_i = e_i$ for $i = 1, \dots, n$, and that μ^* is the canonical solution of $(\mathcal{E}, r)$. Let $\mu_{-\infty}$ denote the canonical solution of $(\mathcal{E}_{-\infty}, r)$. In the first stage, we remove all variables $\mathbf{x}_i$ with $\mu_{-\infty}(\mathbf{x}_i) = 0$. In the second stage therefore, $\mu_{-\infty}(\mathbf{x}_i) = 1$ for all variables $\mathbf{x}_i$. Let further $a = a_{\mathcal{E}}$ denote the lower bound of finite values in the canonical solution of $(\mathcal{E}, r)$. Let $\mu_{a_{\mathcal{E}}}$ denote the canonical solution of $(\mathcal{E}^{\vee a_{\mathcal{E}}-1}, r)$. By construction, $\mu_{a_{\mathcal{E}}}$ never returns $-\infty$. Thus we can apply theorem 6 to compute $\mu_{a_{\mathcal{E}}}$. Analogous as in the proof for theorem 6, we compute $\mu_{a_{\mathcal{E}}}$. This will provide us either with some new variables $\mathbf{x}_i$ with $\mu^*(\mathbf{x}_i) = -\infty$ or verify that $\mu_{a_{\mathcal{E}}}$ already equals μ^*. This two-stage procedure will be repeated until we have found the canonical solution.

Thus, similar to the proof of theorem 6 we repeatedly remove variables for which $\mu_{-\infty}$ returns 0 followed by the computation of the canonical solution of a hierarchical system whose canonical solution never returns $-\infty$. Since by theorem 6, the complexity of the corresponding sub-routine is $\mathcal{O}(d \cdot n^2 \cdot |\mathcal{E}| \cdot \Pi(m'))$ for $m' = m + n$, the assertion follows. □

Thus, using theorem 1 we have deduced our main theorem for crash games:

Theorem 8. *Assume that* $G = (V_\vee, V_\wedge, E, c, r)$ *is a crash game. Let* $V = V_\vee \cup V_\wedge$*. The values* $\langle\!\langle v \rangle\!\rangle_G$*,* $v \in V$ *can be computed by a strategy improvement algorithm in time* $\mathcal{O}(d \cdot |V|^3 \cdot |G| \cdot \Pi(|G|))$*, where* $|G| = |V| + |E|$ *and* d *denotes the maximal rank of a position occurring in* G. □

7 Conclusion

In this paper, we have introduced the concept of crash games where each player aims at optimizing the total payoff of a play. Although crash games do not admit optimal positional strategies, we succeeded in characterizing their game values by canonical solutions of hierarchical systems of simple integer equations.

We then have shown how the canonical solution of a hierarchical system of simple integer equations can be computed. For that, we used an *instrumentation* of the underlying lattice to obtain a lifted system of equations where finite solutions are *unique*. We exploited this insight to design a strategy iteration algorithm for hierarchical systems with *finite* canonical solutions. This algorithm is quite natural and comparable in its efficiency with the discrete strategy iteration algorithm for the Boolean case [19]. We then showed how general hierarchical systems of simple integer equations can be solved by iteratedly solving systems with finite canonical solutions only.

Using our algorithm for hierarchical systems, we are thus able to determine the game values of crash games. Further investigations are needed for automatically designing strategies with guaranteed payoffs.

References

1. Arnold, A., Niwinski, D.: Rudiments of μ-Calculus. In: Studies in Logic and The Foundations of Computer Science, vol. 146, North-Holland, Amsterdam (2001)
2. Bjorklund, H., Sandberg, S., Vorobyov, S.: Complexity of Model Checking by Iterative Improvement: the Pseudo-Boolean Framework. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 381–394. Springer, Heidelberg (2004)
3. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS, pp. 178–187. IEEE Computer Society, Los Alamitos (2005)
4. Chechik, M., Devereux, B., Easterbrook, S.M., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. 12(4), 371–408 (2003)
5. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005)
6. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS, pp. 368–377. IEEE Computer Society Press, Los Alamitos (1991)
7. Gawlitza, T., Reineke, J., Seidl, H., Wilhelm, R.: Polynomial Exact Interval Analysis Revisited. Technical report, TU München (2006)
8. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 300–315. Springer, Heidelberg (2007)
9. Gimbert, H., Zielonka, W.: When can you play positionally? In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 686–697. Springer, Heidelberg (2004)
10. Gimbert, H., Zielonka, W.: Games where you can play optimally without any memory. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 428–442. Springer, Heidelberg (2005)
11. Hoffman, A.J., Karp, R.M.: On Nonterminating Stochastic Games. Management Sci. 12, 359–370 (1966)
12. Howard, R.: Dynamic Programming and Markov Processes. Wiley, New York (1960)
13. Jrklund, H., Nilsson, O., Svensson, O., Vorobyov, S.: The Controlled Linear Programming Problem. Technical report, DIMACS (2004)
14. Jurdziński, M.: Small Progress Measures for Solving Parity Games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
15. Puri, A.: Theory of Hybrid and Discrete Systems. PhD thesis, University of California, Berkeley (1995)
16. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (1994)
17. Seidl, H.: A Modal μ Calculus for Durational Transition Systems. In: IEEE Conf. on Logic in Computer Science (LICS), pp. 128–137 (1996)
18. Shoham, S., Grumberg, O.: Multi-valued model checking games. In: Peled, D.A., Tsay, Y.K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 354–369. Springer, Heidelberg (2005)
19. Vöge, J., Jurdzinski, M.: A Discrete Strategy Improvement Algorithm for Solving Parity Games. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)

Timed Control with Observation Based and Stuttering Invariant Strategies

Franck Cassez[1,⋆,⋆⋆], Alexandre David[2], Kim G. Larsen[2], Didier Lime[1,⋆], and Jean-François Raskin[3,⋆⋆⋆]

[1] IRCCyN, CNRS, Nantes, France
{franck.cassez,didier.lime}@irccyn.ec-nantes.fr
[2] CISS, CS, Aalborg University, Denmark
{adavid,kgl}@cs.aau.dk
[3] Computer Science Department, Université Libre de Bruxelles (U.L.B.), Belgium
jraskin@ulb.ac.be

Abstract. In this paper we consider the problem of controller synthesis for timed games under imperfect information. Novel to our approach is the requirements to strategies: they should be based on a finite collection of *observations* and must be *stuttering invariant* in the sense that repeated identical observations will not change the strategy. We provide a constructive transformation to equivalent finite games with perfect information, giving decidability as well as allowing for an efficient on-the-fly forward algorithm. We report on application of an initial experimental implementation.

1 Introduction

Timed automata introduced by Alur and Dill [2] is by now a well-established formalism for representing the behaviour of real-time systems. Since their definition several contributions have been made towards the theoretical and algorithmic characterization of this formalism. In particular, industrial mature tools supporting model-checking for timed automata now exist [7,6].

More recently the problem of controller synthesis for timed automata based models have been considered: i.e. given a timed game automaton modelling the possible moves of an environment as well as the possible moves of a control program, the problem consists in synthesizing a strategy for the control program in order that a given control objective is met no matter how the environment behaves [15].

Controller synthesis and time-optimal controller synthesis for timed games was shown decidable in [4] and [3]. First steps towards efficient synthesis algorithms were taken in [1,16]. In [9] a truly on-the-fly algorithm based on a mixture of

⋆ Work supported by the French National Research Agency ANR-06-SETI-DOTS.
⋆⋆ Author supported by the Fonds National de la Recherche Scientifique, Belgium.
⋆⋆⋆ Author supported by the Belgian FNRS grant 2.4530.02 of the FRFC project "Centre Fédéré en Vérification" and by the project "MoVES", an Interuniversity Attraction Poles Programme of the Belgian Federal Government.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 192–206, 2007.
© Springer-Verlag Berlin Heidelberg 2007

forward search and backwards propagation was proposed and later used as the basis for an efficient implemention in the tool UPPAAL TIGA [5].

In all of the papers cited above it has been assumed that the controller has perfect information about the plant: at any time, the controller knows precisely in what state the environment is. In general however — e.g. due to limited sensors — a controller will only have imperfect (or partial) information about the state of the environment. In the discrete case it is well known how to handle partial observability, however for the timed case it has been shown in [8] that the controller synthesis problem under partial observability is in general undecidable. Fixing the resources of the controller (i.e. a maximum number of clocks and maximum allowed constants in guards) regains decidability [8], a result which also follows from the quotient and model construction results of [12,13].

In this paper we also deal with the problem of controller synthesis for timed games under imperfect information following the approach of [10,17]. That is, the imperfect information is given in terms of (a finite number of possible) *observations* to be made on the system configurations, providing the sole basis for the strategy of the controller. However, in contrast to [10,17], which is essentially turn-based in the untimed setting, we will here consider a more general framework, where in each step the controller and environment are competing. In particular, the strategy of the controller is supposed to be *stuttering invariant*, i.e. the strategy will not be affected by a sequence of environment or time steps unless changes in the observations occur.

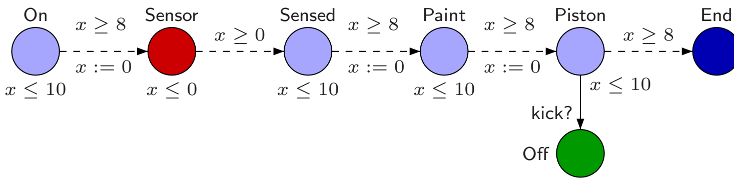


Fig. 1. Timed Game with Imperfect Information

To illustrate the concepts of imperfect information and stuttering invariance consider the timed game automaton in Figure 1 modelling a production system for *painting* a box moving on a conveyor belt. The various locations indicate the position of the box in the system: in Sensor a sensor is assumed to reveal the presence of the box, in Sensed the box is moving along the belt towards the painting area, in Paint the actual painting of the box takes place, in Piston the box may be kick?'ed off the belt leading to Off; if the box is not kicked off it ends in End. All phases are assumed to last between 8 and 10 seconds, except for the phase Sensor, which is instantaneous. The uncontrollability of this timing uncertainty is indicated by the dashed transitions between phases. The controller should now issue a *single* kick?'command at the appropriate moment in order to guarantee that the box will — regardless of the above timing uncertainy — be

kicked off the belt. However the controller has *imperfect information* of the position of the box in the system. In particular, the controller cannot directly observe whether the box is in the Sensed, Paint or in the Piston phase nor can the value of the clock x be observed. Still equipping the controller with its own clock y –which it may reset and test (against a finite number of predicates) – it might be possible to synthesize a control strategy despite having only partial information: in fact it may be deduced that the box will definitely be in the Piston area within 20-24 seconds after being sensed. In contrast, an increased timing uncertainty where a phase may last between 6 and 10 seconds will make a single-kick? strategy impossible.

The main contributions of this paper are: (i) we show how a variant of the subset construction of [10,17] allows us to transform a timed game H with imperfect information into an equivalent game $G(H)$ of perfect information; (ii) we show that $G(H)$ can be effectively and symbolically computed and this implies that the control problem under imperfect information is decidable; this allows us to apply the efficient on-the-fly forward algorithm from [9] and (iii) we report on application of an initial experimental implementation of this algorithm and a number of heuristics for minimizing the explored state-space as well as the size of the finally synthesized strategy.

The detailed proofs can be found in the extended version available from the authors web pages.

2 Timed Games and Observation Based Strategies

In this section, we define the timed game structures, the notion of strategies that we are interested in, and useful vocabulary for the rest of the paper. We denote $\mathbb{R}_{\geq 0}$ the set of non-negative reals and $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$ and A^B the set of mappings from B to A.

Timed game structures (TGS) will be defined using a timed automaton like notation. The semantics of the notation will be defined by a two-player labeled timed transition system (2-LTTS), and the games will be played on this 2-LTTS.

Definition 1 (2-LTSS). *A 2-player labeled timed transition system (2-LTTS) is a tuple $(S, s_0, \Sigma_1, \Sigma_2, \rightarrow)$ where S is a (infinite) set of states, s_0 is the initial state, Σ_1 and Σ_2 are the two disjoint alphabets of actions for Player 1 and Player 2 respectively, and $\rightarrow \subseteq S \times \Sigma_1 \cup \Sigma_2 \cup \mathbb{R}_{>0} \times S$ is the transition relation.*

Given a state $s \in S$, we define $\mathsf{enable}(s)$ as the set of $\sigma \in \Sigma_1 \cup \Sigma_2 \cup \mathbb{R}_{>0}$ such that there exists s' and $(s, \sigma, s') \in \rightarrow$.

Let X be a finite set of real-valued variables called clocks. Let M be a natural number. We note $\mathcal{C}(X, M)$ the set of constraints φ generated by the grammar: $\varphi ::= x \sim k \mid x - y \sim k \mid \varphi \wedge \varphi$ where $k \in \mathbb{Z} \cap [0, M]$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X, M)$ is the subset of $\mathcal{C}(X, M)$ generated by the following grammar: $\varphi ::= \top \mid k_1 \leq x < k_2 \mid \varphi \wedge \varphi$ where $k, k_1, k_2 \in \mathbb{Z} \cap [0, M]$, $k_1 < k_2$, $x \in X$, and $\top$ is the boolean constant *true*. In the sequel, we will restrict our attention to bounded timed automata where clock values are all bounded by a

natural number M; this does not reduce the expressive power of timed automata. Given a natural number M, an *M-valuation* of the variables in X is a mapping $X \mapsto \mathbb{R}_{\geq 0} \cap [0, M]$. We also use the notation $[X \to [0, M]]$ for valuations and $\mathbf{0}$ for the valuation that assigns 0 to each clock. For $Y \subseteq X$, we denote by $v[Y]$ the valuation assigning 0 (*resp.* $v(x)$) for any $x \in Y$ (*resp.* $x \in X \setminus Y$). Let $t \in \mathbb{R}_{\geq 0}$, v be an M-valuation for the set of clocks X, if for all $x \in X$, $v(x) + t \leq M$ then $v + t$ is the M-valuation defined by $(v + t)(x) = v(x) + t$ for all $x \in X$. For $g \in \mathcal{C}(X, M)$ and $v \in (\mathbb{R}_{\geq 0} \cap [0, M])^X$, we write $v \models g$ if v satisfies g and $[\![g]\!]$ denotes the set of valuations $\{v \in (\mathbb{R}_{\geq 0} \cap [0, M])^X \mid v \models g\}$. An *$M$-zone* Z is a subset of $(\mathbb{R}_{\geq 0} \cap [0, M])^X$ s.t. $Z = [\![g]\!]$ for some $g \in \mathcal{C}(X, M)$.

Definition 2 (Timed Game Structure). *Let M be a natural number, an M-*timed game structure *(M-TGS) is a tuple $H = (L, \iota, X, \delta, \Sigma_1, \Sigma_2, \mathsf{inv}, \mathcal{P})$ where:*

- *L is a finite set of locations,*
- *$\iota \in L$ is the initial location,*
- *X is a finite set of real-valued clocks,*
- *Σ_1, Σ_2 are two disjoint alphabets of actions, Σ_1 is the set of actions of Player 1 and Σ_2 the set of actions of Player 2,*
- *$\delta \subseteq (L \times \mathcal{B}(X, M) \times \Sigma_1 \times 2^X \times L) \cup (L \times \mathcal{C}(X, M) \times \Sigma_2 \times 2^X \times L)$ is partitioned into transitions*[1] *of Player 1 and transitions of Player 2.*
- $\mathsf{inv} : L \to \mathcal{B}(X, M)$ *associates to each location its* invariant.
- *$\mathcal{P}$ is a finite set of pairs (K, φ) where $K \subseteq L$ and $\varphi \in \mathcal{B}(X, M)$, called* observable predicates.

In the definition above, each observable predicate $(K, \varphi) \in \mathcal{P}$ is a predicate over the state space of the TGS, i.e. the set $L \times [X \to [0, M]]$. For $l \in L$ and v an M-valuation of the clocks in X, we write $(l, v) \models (K, \varphi)$ iff $l \in K$ and $v \models \varphi$. Two pairs (l_1, v_1), (l_2, v_2) that satisfy the same observable predicates from $\mathcal{P}$ have the same *observation* (they can not be distinguished by our controllers). So, an *observation* is a function $o : \mathcal{P} \to \{0, 1\}$, or equivalently, a class of equivalent states w.r.t $\mathcal{P}$. We note $\mathcal{O}$ the set of functions $[\mathcal{P} \to \{0, 1\}]$, it is called the set of *observations* of the system. With each TGS H with set of observable predicates $\mathcal{P}$, we associate the function γ that maps observations to classes of equivalent states, i.e. $\gamma : \mathcal{O} \to 2^{L \times [X \to [0, M]]}$, and it is defined as follows:

$$\gamma(o) = \left\{ (l, v) \mid \bigwedge_{(K,\varphi) \;\mid\; o(K,\varphi)=1} (l, v) \models (K, \varphi) \wedge \bigwedge_{(K,\varphi) \;\mid\; o(K,\varphi)=0} (l, v) \not\models (K, \varphi) \right\}$$

Note that the set of observations $\mathcal{O}$ defines a partition of the state space of the M-TGS, i.e. $\bigcup_{o \in \mathcal{O}} \gamma(o) = L \times [X \to [0, M]]$, and for all $o_1, o_2 \in \mathcal{O}$, if $o_1 \neq o_2$, then $\gamma(o_1) \cap \gamma(o_2) = \emptyset$. Given (l, v) we write $\gamma^{-1}(l, v)$ for the observation o of

[1] Note that we impose that guards of Player 1's transitions are left closed. This ensures that, when a guard becomes true for an action owned by Player 1, there is always a first instant where it becomes true.

state (l, v), *i.e.* $\gamma^{-1}(l, v)$ is the function $o : \mathcal{P} \to \{0, 1\}$ s.t. $\forall p \in \mathcal{P}, o(p) = 1 \iff (l, v) \models p$.

We associate with any M-TGS H a semantics in the form of a (infinite state) 2-LTTS. The state space of the 2-LTTS will be composed of elements of the form (l, v) where l is a location of the TGS and v is a valuation of the clocks. In order to avoid deadlocks in the 2-LTTS, we require that our TGS are *deadlock-free*[2], that is, for every state (l, v) such that $v \models \mathsf{inv}(l)$, there exists $\sigma \in \Sigma_2 \cup \mathbb{R}_{>0}$, such that either there is a transition $(l, g, \sigma, Y, l') \in \delta$ such that $v \models g$ and $v[Y] \models \mathsf{inv}(l')$, or for all t', $0 \le t' \le \sigma$, $v + t' \models \mathsf{inv}(l)$.

Definition 3 (Semantics of a TGS). *The semantics of an M-TGS $H = (L, \iota, X, \delta, \Sigma_1, \Sigma_2, \mathsf{inv}, \mathcal{P})$ is a 2-LTTS $S_H = (S, s_0, \Sigma_1, \Sigma_2, \to)$ where:*

- $S = \{(l, v) \mid l \in L \wedge v \in (\mathbb{R}_{\ge 0} \cap [0, M])^X \wedge v \models \mathsf{inv}(l)\}$;
- $s_0 = (\iota, \mathbf{0})$;
- *the transition relation is composed of*
  - (i) *discrete transitions. For all $(l_1, v_1), (l_2, v_2) \in S$, for all $\sigma \in \Sigma_1 \cup \Sigma_2$, $((l_1, v_1), \sigma, (l_2, v_2)) \in\to$ iff there exists a transition $(\ell, g, a, Y, \ell') \in \delta$ such that $\ell = l_1$, $\ell' = l_2$, $v_1 \models g$, and $v_2 = v_1[Y]$;*
  - (ii) *time transitions. For all $(l_1, v_1), (l_2, v_2) \in S$, for all $t \in \mathbb{R}_{>0}$, there is a transition $((l_1, v_1), t, (l_2, v_2)) \in\to$ iff $l_1 = l_2$, $v_2 = v_1 + t$, and for all t', $0 \le t' < t$, $(l_1, v_1 + t') \in S$ and $\gamma^{-1}(l_1, v_1 + t') = \gamma^{-1}(l_1, v_1)$.*

Remark 1. This semantics has the following important property: changes of observations can occur only during a discrete transition, or at the last point of a time delay. This is consistent with our definition of observations using constraints in $\mathcal{B}(X, M)$: the form of the constraints implies that either for all $t \ge 0$, $(l, v) \xrightarrow{t} (l, v + t)$, and $\gamma^{-1}(l, v + t) = \gamma^{-1}(l, v)$, or there is a first instant $t_0 > 0$ s.t. $(l, v) \xrightarrow{t_0} (l, v + t_0)$ and $\gamma^{-1}(l, v + t_0) \ne \gamma^{-1}(l, v)$, and for all $0 \le t' < t_0$, $\gamma^{-1}(l, v + t') = \gamma^{-1}(l, v)$.

The 2-LTTS of a TGS has no deadlock because a TGS is deadlock-free. This also implies that any state of the 2-LTTS is the source of an infinite path. As a TGS is bounded, these infinite paths contain infinitely many discrete steps and in the sequel we will consider only these type of infinite paths.

Playing with Observation Based Stuttering Invariant Strategies. In the remainder of this section, we will define the rules of the timed games that we want to consider. We start by an informal presentation and then turn to the formalization.

Player 1 and Player 2 play on the underlying 2-LTTS of a TGS as follows. Player 1 has to play according to *observation based stuttering invariant strategies* (OBSI strategies for short). Initially and whenever the current observation of the system state changes, Player 1 either proposes an action $\sigma_1 \in \Sigma_1$, or the special

[2] And more precisely, either time can elapse or Player 2 can do a discrete action from any state: thus Player 1 cannot block the game by refusing to take its actions.

action delay. When Player 1 proposes σ_1, this intuitively means that he wants to play the action σ_1 whenever this action is enabled in the system. When Player 1 proposes delay, this means that he does not want to play discrete actions until the next change of observation, he is simply waiting for the next observation. Thus, in the two cases, Player 1 sticks to his choice until the observation of the system changes: in this sense he is playing with an observation based stuttering invariant strategy. Once Player 1 has committed to a choice, Player 2 decides of the evolution of the system until the next observation but respects the following rules:

1. if the choice of Player 1 is a discrete action $\sigma_1 \in \Sigma_1$ then Player 2 can choose to play, as long as the observation does not change, either (i) a discrete actions in $\Sigma_2 \cup \{\sigma_1\}$ or (ii) let time elapse as long as σ_1 is not enabled. This entails that σ_1 is urgent,
2. if the choice of Player 1 is the special action delay then Player 2 can choose to play, as long as the observation does not change, any of its discrete actions in Σ_2 or let time pass,
3. the turn is back to Player 1 as soon as the next observation is reached.

Plays. In the following, we define *plays* of a game where choices of Player 1 are explicitly mentioned. A *play* in H is an infinite sequence of transitions in S_H, $\rho^H = (l_0, v_0)c_0\sigma_0(l_1, v_1)c_1\sigma_1 \ldots (l_n, v_n)c_n\sigma_n \ldots$, such that for all $i \geq 0$, $(l_i, v_i) \xrightarrow{\sigma} (l_{i+1}, v_{i+1})$ and

- either $\sigma_i \in \{c_i\} \cup \Sigma_2$, or
- $\sigma_i \in \mathbb{R}^X_{>0}$ and $\forall 0 \leq t < \sigma_i, c_i \notin \mathsf{enable}(l_i, v_i + t)$ (time elapses only when the choice of Player 1 is not enabled).[3]
- if σ_i and σ_{i+1} are in $\mathbb{R}^X_{>0}$ then $\gamma^{-1}(l_i, v_i) \neq \gamma^{-1}(l_{i+1}, v_{i+1})$.

We write $\mathsf{Play}((l, v), H)$ for the set of plays in H that start at state (l, v). We write $\mathsf{Play}(H)$ for the *initial* plays that start at the initial state of H, that is the set $\mathsf{Play}((\iota, \mathbf{0}), H)$.

Prefixes, Strategies, and Outcomes. A *prefix* of H is a prefix of a play in H that ends in a state of H. We note $\mathsf{Pref}((l, v), H)$ for the set of prefixes of plays in H that starts in (l, v), i.e. plays in $\mathsf{Play}((l, v), H)$. We note $\mathsf{Pref}(H)$, for prefixes of initial plays in H, i.e. prefixes of plays in $\mathsf{Play}(H)$. Let $\rho^H = (l_0, v_0)c_0\sigma_0 \cdots (l_n, v_n)c_n\sigma_n \cdots$ be a play or a prefix of a play, $\rho^H(n)$ denotes the prefix up to (l_n, v_n). In the sequel, we measure the *length of a prefix* by counting the number of states that appear in the prefix. For example, $\rho^H(n)$ has a length equal to $n + 1$. A *strategy* for Player 1 in H is a function $\lambda^H : \mathsf{Pref}(H) \to \Sigma_1 \cup \{\mathsf{delay}\}$. The *outcome* of a strategy λ^H in H is the set of plays $\rho^H = (l_0, v_0)c_0\sigma_0(l_1, v_1)c_1\sigma_1 \ldots (l_n, v_n)c_n\sigma_n \ldots$ such that $l_0 = \iota$, $v_0 = \mathbf{0}$ and for all $i \geq 0$, $c_i = \lambda^H(\rho^H(i))$. We note $\mathsf{Outcome}^H(\lambda^H)$ this set of plays.

[3] Remember that delay is never enabled and if Player 1 wants to let time elapse he plays delay.

Consistent Plays, Choice Points and OBSI Strategies in H***.*** We are interested in strategies for Player 1 where the choice of action can only change if the observation of the state of the system changes. Such a strategy is called an *observation based stuttering invariant strategy* as presented before. When Player 1 plays such a strategy, the resulting plays have the property of being *consistent*. This notion is defined as follows. A play $\rho^H = (l_0, v_0)c_0\sigma_0 \cdots (l_n, v_n)c_n\sigma_n \cdots$ is *consistent* iff for all $i \geq 0$: $\gamma^{-1}(\ell_{i+1}, v_{i+1}) = \gamma^{-1}(\ell_i, v_i) \implies c_{i+1} = c_i$. We note $\mathsf{Play}^{co}(H)$ the set of consistent plays of H, and $\mathsf{Pref}^{co}(H)$ the set of prefixes of consistent plays of H. Let $\rho^H = (l_0, v_0)c_0\sigma_0 \cdots (l_{n-1}, v_{n-1})c_{n-1}\sigma_{n-1}(l_n, v_n) \in \mathsf{Pref}^{co}(H)$. ρ^H is a *choice point* if either $n = 0$, or $n > 0$ and $\gamma^{-1}(l_{n-1}, v_{n-1}) \neq \gamma^{-1}(l_n, v_n)$. Note that we have that $\mathsf{ChoicePoint}(H) \subseteq \mathsf{Pref}^{co}(H) \subseteq \mathsf{Pref}(H)$ and $\mathsf{Play}^{co}(H) \subseteq \mathsf{Play}(H)$.

Let $\rho^H = (l_0, v_0)c_0\sigma_0 \cdots (l_n, v_n)c_n\sigma_n \cdots$ be a consistent play in H. Let $I = \{m \mid \rho^H(m) \in \mathsf{ChoicePoint}(H)\}$. The *stuttering free observation* $\mathsf{Obs}(\rho^H)$ of ρ^H is the sequence in $(\mathcal{O}.\Sigma_1)^\omega$ defined by:

– if $I = \{n_0, n_1, \cdots, n_k\}$ is finite,

$$\mathsf{Obs}(\rho^H) = \gamma^{-1}(l_{n_0}, v_{n_0})c_{n_0} \cdots \gamma^{-1}(l_{n}, v_{n})c_{n} (\gamma^{-1}(l_{n}, v_{n})c_{n})^\omega$$

– if $I = \{n_0, n_1, \cdots, n_k, \cdots\}$ is infinite,

$$\mathsf{Obs}(\rho^H) = \gamma^{-1}(l_{n_0}, v_{n_0})c_{n_0}\gamma^{-1}(l_{n_1}, v_{n_1})c_{n_1} \cdots \gamma^{-1}(l_{n}, v_{n})c_{n} \cdots$$

We call it "stuttering free" as, for all $i \in I$, $\gamma^{-1}(l_{n}, v_{n}) \neq \gamma^{-1}(l_{n+1}, v_{n+1})$ except when I is finite, but in this case, only the last observation is repeated infinitely. Let $\rho^H \in \mathsf{ChoicePoint}(H)$, let $I = \{n_0, n_1, \cdots, n_k\}$ be the set of indices n_i such that $\rho^H(n_i) \in \mathsf{ChoicePoint}(H)$. The (finite) observation of ρ^H, noted $\mathsf{Obs}^*(\rho^H)$, is $\gamma^{-1}(l_{n_0}, v_{n_0})c_{n_0} \cdots \gamma^{-1}(l_{n}, v_{n})c_{n}\ \gamma^{-1}(l_{n}, v_{n})c_{n}$. We say that a strategy λ^H is an *observation based stuttering invariant* (OBSI) strategy if the following property holds: for all $\rho_1^H, \rho_2^H \in \mathsf{Pref}^{co}(H)$, let n_1 be the maximal value such that $\rho_1^H(n_1) \in \mathsf{ChoicePoint}(H)$, let n_2 be the maximal value such that $\rho_2^H(n_2) \in \mathsf{ChoicePoint}(H)$, if $\mathsf{Obs}^*(\rho_1^H(n_1)) = \mathsf{Obs}^*(\rho_2^H(n_2))$ then $\lambda^H(\rho_1^H) = \lambda^H(\rho_2^H)$.

Winning Conditions and Winning Strategies. Let $\rho^H \in \mathsf{Play}(H)$ s.t. $\mathsf{Obs}(\rho^H) = o_0c_0o_1c_1 \ldots o_nc_n \ldots$. The projection $\mathsf{Obs}(\rho^H) \downarrow \mathcal{O}$ over $\mathcal{O}$ of $\mathsf{Obs}(\rho^H)$ is the sequence $o_0o_1 \ldots o_n \ldots$. A *winning condition* $\mathcal{W}$ is a stuttering closed[4] subset of $\mathcal{O}^\omega$. A strategy λ^H for Player 1 is *winning* in H for $\mathcal{W}$, if and only if, $\forall \rho \in \mathsf{Outcome}^H(\lambda^H) \cdot \mathsf{Obs}(\rho) \downarrow \mathcal{O} \in \mathcal{W}$.

To conclude this section, we define the control problem OBSI-CP we are interested in: let H be a TGS with observations $\mathcal{O}$, $\mathcal{W}$ be a stuttering closed subset of $\mathcal{O}^\omega$,

is there an OBSI winning strategy in H *for* $\mathcal{W}$*?* (OBSI-CP)

[4] A language is stutter closed, if for any word w in the language, the word w' obtained from w by either adding a stuttering step (repeating a letter), or erasing a stuttering step, is also in the language.

In case there is such a strategy, we would like to synthesize one. The problem of constructing a winning strategy is called the *synthesis* problem.

3 Subset Construction for Timed Games

In this section, we show how to transform a timed game of imperfect information into an equivalent game of perfect information. Let $H = (L, \iota, X, \delta, \Sigma_1, \Sigma_2, \mathsf{inv}, \mathcal{P})$ be an M-TGS and let $\mathcal{S}_H = (S, s_0, \Sigma_1, \Sigma_2, \rightarrow)$ be its semantics. In this section we assume $\mathsf{delay} \in \Sigma_1$ but H has no transition labeled delay.

Useful functions. Let $\sigma \in \Sigma_1$. We define the relation $\xrightarrow{\sigma}_{nobs}$ by: $(l, v) \xrightarrow{\sigma}_{nobs} (l', v')$ if there is a prefix $\rho = (\ell_0, v_0)c_0\sigma_0(\ell_1, v_1)c_1\sigma_1 \cdots (l_k, v_k)c_k\sigma_k(\ell_{k+1}, v_{k+1})$ in $\mathsf{Pref}((l, v), H)$ with $(\ell_0, v_0) = (l, v)$, $(\ell_{k+1}, v_{k+1}) = (l', v')$, $\forall 0 \leq i \leq k, c_i = \sigma$, $\gamma^{-1}(\ell_i, v_i) = \gamma^{-1}(\ell_0, v_0)$, and $\gamma^{-1}(\ell_{k+1}, v_{k+1}) \neq \gamma^{-1}(\ell_0, v_0)$. Notice that because of the definition of time transitions in Definition 3, if $\sigma_i \in \mathbb{R}_{>0}$ and $0 \leq i < k$ then $\gamma^{-1}(\ell_i, v_i) = \gamma^{-1}(\ell_{i+1}, v_{i+1})$ and if $\sigma_i \in \mathbb{R}_{>0}$ and $i = k$, $\gamma^{-1}(\ell_i, v_i) = \gamma^{-1}(\ell_i, v_i + t)$ for all $0 \leq t < \sigma_i$, $\gamma^{-1}(\ell_i, v_i) \neq \gamma^{-1}(\ell_i, v_i + t)$ and $(\ell_{i+1}, v_{i+1}) = (\ell_i, v_i + \sigma_i)$ (*i.e.* σ_i is the first instant at which the observation changes). By the constraints imposed by $\mathcal{B}(X, M)$ this first instant always exists. We define the function $\mathsf{Next}_\sigma(l, v)$ by:

$$\mathsf{Next}_\sigma(l, v) = \{(l', v') \mid (l, v) \xrightarrow{\sigma}_{nobs} (l', v')\} \tag{1}$$

This function computes the first next states after (l, v) which have an observation different from $\gamma^{-1}(l, v)$ when Player 1 continuously plays σ. Next is extended to sets of states as usual.

We also define the function $\mathsf{Sink}_\sigma(\cdot) : L \times \mathbb{R}^X_{\geq 0} \rightarrow L \times \mathbb{R}^X_{\geq 0}$ for $\sigma \in \Sigma_1$: $(l', v') \in \mathsf{Sink}_\sigma(l, v)$ iff there is an (infinite) play[5] $\rho = (\ell_0, v_0)c_0\sigma_0(\ell_1, v_1)c_1\sigma_1 \cdots (l_k, v_k) c_k\sigma_k(\ell_{k+1}, v_{k+1}) \cdots$ in $\mathcal{S}_H$ such that: $(\ell_0, v_0) = (l, v)$, $(\ell_{k+1}, v_{k+1}) = (l,' v')$, $\forall 0 \leq i, c_i = \sigma$, and $\forall 0 \leq i, \gamma^{-1}(l_i, v_i) = \gamma^{-1}(l_0, v_0)$.

Non-Deterministic Game (of Perfect Information). The games of perfect information that we consider here are (untimed) *non-deterministic games*, and they are defined as follows: in each state s Player 1 chooses an action σ and Player 2 chooses the next state among the σ-successors of s.

Definition 4 (Non-Deterministic Game). *We define a* non-deterministic game (NDG) *to be a tuple $G = (S, \mu, \Sigma_1, \Delta, \mathcal{O}, \Gamma)$ where:*

- *$S = S_0 \cup S_1$ is a set of states;*
- *$\mu \in S_0$ is the initial state of the game;*
- *Σ_1 is a finite alphabet modeling choices for Player 1;*
- *$\Delta \subseteq S_0 \times \Sigma_1 \times S$ is the transition relation;*

[5] With an infinite number of discrete transitions because of the boundedness assumption. If needed we can add the requirement that this path is non-zeno if we want to rule out zeno-runs.

- *$\mathcal{O}$ is a finite set of observations;*
- *$\Gamma : \mathcal{O} \to 2^S \setminus \emptyset$ maps an observation to the set of states it represents, we assume Γ partitions S ($\Gamma^{-1}(s) = o \iff s \in \Gamma(o)$).*

Definition 5 (Plays in NDG). *A* play *in G from s_0 is either an infinite sequence $s_0 a_0 s_1 a_1 \ldots s_n a_n \ldots$ such that for all $i \geq 0$, $s_i \in S_0$, $a_i \in \Sigma_1$, and $(s_i, a_i, s_{n+1}) \in \Delta$ or a finite sequence $s_0 a_0 s_1 a_1 \ldots s_n$ such that all i, $0 \leq i < n$, $s_i \in S_0$, $(s_i, a_i, s_{i+1}) \in \Delta$, and $s_n \in S_1$. We note $\mathsf{Play}(s_0, G)$ the set of plays starting in s_0 in G and let $\mathsf{Play}(G) = \mathsf{Play}(\mu, G)$. The observation of an infinite play $\rho^G = s_0 a_0 s_1 a_1 \ldots s_n a_n \ldots$ is defined by $\mathsf{Obs}(\rho^G) = \Gamma^{-1}(s_0) a_0 \Gamma^{-1}(s_1) a_1 \ldots \Gamma^{-1}(s_n) a_n \ldots$. If $\rho^G = s_0 a_0 s_1 a_1 s_{n-1} a_{n-1} \ldots s_n a_n s_{n+1}$ is finite then $\mathsf{Obs}(\rho^G) = \Gamma^{-1}(s_0) a_0 \Gamma^{-1}(s_1) a_1 \ldots \Gamma^{-1}(s_n) a_n (\Gamma^{-1}(s_{n+1}) a_n)^\omega$.*

A *prefix* in G is a finite sequence $s_0 a_0 s_1 a_1 \ldots s_n$ ending in $s_n \in S_0$, such that for all i, $0 \leq i < n$, $(s_i, a_i, s_{i+1}) \in \Delta$. We let $\mathsf{Pref}(G)$ be the set of prefixes of G. The observation of a prefix is $\mathsf{Obs}(\rho^G) = \Gamma^{-1}(s_0) a_0 \Gamma^{-1}(s_1) a_1 \ldots a_n \Gamma^{-1}(s_n)$. For any $\rho^G \in \mathsf{Play}(G)$, $\rho^G(n) = s_0 a_0 \cdots s_n$ is the prefix up to state s_n and we let $|\rho^G| = n$ to be the length of ρ^G.

Remark 2. S_0 is the set of states where Player 1 has to make a choice of action. In S_1-states, Player 1 does not have any choice. A prefix ends in an S_0-state. Finite sequences ending in S_1-states are not prefixes but finite plays.

Strategies and Winning Conditions for NDG. A *strategy* in G is a function[6] $\lambda^G : \mathsf{Pref}(G) \to \Sigma_1$. The *outcome*, $\mathsf{Outcome}^G(\lambda^G)$, of a strategy λ^G is the set of (finite or infinite) plays $\rho^G = s_0 a_0 \cdots s_n a_n \cdots$ s.t. $s_0 = \mu$, $\forall i \geq 0, a_i = \lambda^G(\rho^G(i))$. Let ρ^G be a play of G. We let $\mathsf{Obs}(\rho^G) \downarrow \mathcal{O}$ be the projection of $\mathsf{Obs}(\rho^G)$ on $\mathcal{O}$. A *winning condition* $\mathcal{W}$ for G is a subset of $\mathcal{O}^\omega$. A strategy λ^G is *winning* for $\mathcal{W}$ in G iff $\forall \rho^G \in \mathsf{Outcome}^G(\lambda^G)$, $\mathsf{Obs}(\rho^G) \in \mathcal{W}$.

Remark 3. Strategies in NDG are based on the history of the game since the beginning: in this sense, this is a perfect information game.

Knowledge Based Subset Construction

Definition 6. *Given a game $H = (L, \iota, X, \delta, \Sigma_1, \Sigma_2, \mathsf{inv}, \mathcal{O}, \gamma)$, we construct a NDG $G(H) = (S, \mu, \Sigma_1, \Delta, \mathcal{O}, \Gamma)$ as follows:*

- *let $\mathcal{V} = \{W \in 2^{L \times \mathbb{R}_{\geq 0}} \setminus \emptyset \mid \gamma^{-1}(l, v) = \gamma^{-1}(l', v')$ for all $(l, v), (l', v') \in W\}$,*
- *$S = \mathcal{V} \times \{0, 1\}$, and we note S_0 the set $\mathcal{V} \times \{0\}$ and S_1 the set $\mathcal{V} \times \{1\}$,*
- *$\mu = (\{(\iota, \mathbf{0})\}, 0)$,*
- *$\Delta \subseteq S \times \Sigma_1 \times S$ is the smallest relation that satisfies: $((V_1, i), \sigma, (V_2, j)) \in \Delta$ if*
  - *$i = 0$. A consequence is that if $i = 1$ (a state in S_1) there are no outgoing transitions.*

[6] Notice that S_1-states have no outgoing transitions and we do not need to define a strategy for these states.

- $j = 0$ *and* $V_2 = \mathsf{Next}_\sigma(V_1) \cap o$ *for some* $o \in \mathcal{O}$ *such that* $\mathsf{Next}_\sigma(V_1) \cap o \neq \emptyset$, or
- $j = 1$ *if* $\mathsf{Sink}_\sigma(s) \neq \emptyset$ *for some* $s \in V_1$ *and* $V_2 = \cup_{s \in V_1} \mathsf{Sink}_\sigma(s)$,

– $\Gamma^{-1} : S \to \mathcal{O}$, *and* $\Gamma^{-1}((W, i)) = \gamma^{-1}(v)$ *for* $v \in W$. *Note that this is well-defined as* W *is a set of states of* H *that all share the same observation.*

Notice that the game $G(H)$ is non-deterministic as there may be many transitions labeled by σ and leaving a state s to many different states with different observations. $G(H)$ is total for S_0 states: $\forall (V, 0) \in S_0$, $\forall \sigma \in \Sigma_1$, $\sigma \in \mathsf{Enabled}(V)$, because either there is an infinite path from some $s \in V$ with the same observation or there is an infinite path with a new observation (remember that time can elapse or Player 2 can do a discrete action from any state in H). Although non-deterministic $G(H)$ enjoys a weaker notion of determinism formalized by the following proposition:

Proposition 1. *Let* (V, i) *be a state of* $G(H)$, $\sigma \in \Sigma_1$ *and* $o \in \mathcal{O}$. *There is at most one* (V', j) *with* $V' \subseteq \gamma(o)$ *s.t.* $((V, i), \sigma, (V', j)) \in \Delta$.

Note also that if $((V, 0), \sigma, (V', 0)) \in \Delta$ then $\Gamma^{-1}((V, 0)) \neq \Gamma^{-1}((V', 0))$. We can relate the consistent plays in H and plays in G. For that, we define the function $\mathsf{Abs} : \mathsf{Play}^{co}(H) \to \mathsf{Play}(G)$ as follows.

Definition 7 (Abs for Plays of H). *Let* $\rho^H = (l_0, v_0) c_0 \sigma_0 (l_1, v_1) \ldots (l_m, v_m) c_m \sigma_m \ldots$ *be in* $\mathsf{Play}^{co}(H)$. *Let* $I = \{j \in \mathbb{N} \mid \rho^H(j) \in \mathsf{ChoicePoint}(H)\}$. *Then* $\mathsf{Abs}(\rho^H)$ *is defined by:*

– *if* I *is a finite set, let* $I = \{j_0, j_1, \cdots, j_n\}$. *Define* $\mathsf{Abs}(\rho^H) = s_0 a_0 s_1 a_1 \ldots s_n a_n s_{n+1}$ *by induction as follows:*
  1. $s_0 = (\{(l_0, v_0)\}, 0)$, $a_0 = c_{j_0}$ *(and* $j_0 = 0$*)*,
  2. *and for all* i, $0 < i \leq n$, *if* $s_{i-1} = (V, 0)$ *then let* $V' = \mathsf{Next}_{a_{i-1}}(V) \cap \gamma^{-1}(v_{j_i})$, $s_i = (V', 0)$ *and* $a_i = c_{j_i}$.
  3. *As* ρ^H *has a finite number of choice points, it must be the case that* $\forall k \geq j_n$, $\gamma^{-1}(l_k, v_k) = \gamma^{-1}(l_{j_n}, v_{j_n})$; *moreover, because* ρ^H *is consistent,* $\forall k \geq j_n, c_k = c_{j_n}$. *If* $s_n = (V, 0)$ *we let* $V' = \cup_{v \in V} \mathsf{Sink}_{c_{j_n}}(v)$. V' *must be non empty and we define* $s_{n+1} = (V', 1)$.

– *if* I *is an infinite set, let* $I = \{j_0, j_1, \cdots, j_n, \cdots\}$. *We define* $\mathsf{Abs}(\rho^H) = s_0 a_0 s_1 a_1 \ldots s_n a_n s_{n+1} \cdots$ *by induction as follows:*
  1. $s_0 = (\{(l_0, v_0)\}, 0)$, $a_0 = c_{j_0}$ *(and* $j_0 = 0$*)*,
  2. *and for all* $i \geq 1$, *if* $s_{i-1} = (V, 0)$ *then let* $V' = \mathsf{Next}_{a_{i-1}}(V) \cap \gamma^{-1}(v_{j_i})$, $s_i = (V', 0)$ *and* $a_i = c_{j_i}$.

Definition 8 (Abs for consistent prefixes). *Let* $\rho^H = (l_0, v_0) \ldots (l_{n-1}, v_{n-1}) c_{n-1} \sigma_{n-1} (l_n, v_n) \in \mathsf{Pref}^{co}(H)$ *and,* $I = \{j_0, j_1, \cdots, j_m\}$ *be the set of choice points of* ρ^H. *Then* $\mathsf{Abs}(\rho^H) = s_0 a_0 \ldots s_{m-1} a_{m-1} s_m$ *with:*

– $s_0 = (\{(l_0, v_0)\}, 0)$,
– *and for all* i, $0 < i \leq m$, *if* $s_{i-1} = (V_{i-1}, 0)$ *then* $s_i = (V_i, 0)$ *where* $V_i = \mathsf{Next}_{a_{i-1}}(V_{i-1}) \cap o_i$,
– $\forall 0 \leq i < m$, $a_i = c_{j_i}$.

It can be checked that $\mathsf{Abs}(\rho^H)$ is well-defined for consistent plays as well for prefixes. The following theorem states the correctness of our construction:

Theorem 1. *Let Φ be a stuttering closed subset of $\mathcal{O}^\omega$. Player 1 has a winning strategy in $G(H)$ for Φ iff Player 1 has an observation based stuttering invariant winning strategy in H for Φ.*

4 Symbolic Algorithms

In this section, we show that the game $G(H)$ is a finite game and design an efficient symbolic algorithm for reachability and safety objectives.

Given a set of states S represented as a finite union of zones, and an action $c \in \Sigma_1$ we can compute the sets $\bigcup_{s \in S} \mathsf{Next}_c(s)$ and $\bigcup_{s \in S} \mathsf{Sink}_c(s)$ as finite unions of zones. Since the clocks are bounded in our M-TGS, only a finite number of zones are computed during the computation of those operators. As a consequence, the game $G(H)$ is finite.

To implement efficiently the computations, we use *Difference Bound Matrices* (DBMs), which allow efficient realisation of most set operations (inclusion, interesection, future, reset...) [11,14].

Since $G(H)$ is finite, we can apply standard control algorithms to compute the set of winning states. In particular, for reachability or safety objectives, we can use the efficient on-the-fly forward algorithm of [9] that has been implemented in Uppaal-Tiga [5].

The algorithm for finite games given in [9] can easily be tailored to solve NDGs of type $G = (\mathcal{S}, s_0, \Sigma_1, \Delta, \mathcal{O}, \Gamma)$ with reachability objective Goal s.t. $\mathsf{Goal} \in \mathcal{O}$. Then, to obtain an algorithm for games of imperfect information, we replace the transition relation of $G(H)$ in this algorithm with the definition (see Definition 6) of the transition relation of $G(H)$ using the Next and Sink operators. This way we obtain the algorithm OTFPOR for TGS which is given Figure 2.

An important feature added to the algorithm is the propagation of *losing* state-sets, that is state-sets for which the reachability objective can be directly concluded not to be achievable. For reachability games, a state-set W may declared to be losing provided it is not among the Goal sets and is a deadlock. Safety games are dual to reachability games in the sense that if the algorithm concludes that the initial state is not losing it is possible to extract a strategy to avoid losing states.

5 Example and Experiments

In this section we report on an application of a prototype implementation of the OTFPOR algorithm. Similar to the Box Painting Production System (BPPS) from the Introduction, we want to control a system consisting of a moving belt and an ejection piston at the end of the belt. However, a complicating feature compared with BPPS is that the system can receive both *light* and *heavy* boxes. When receiving a light box, its speed is high and the box takes between 4 to

Initialization:

$Passed \leftarrow \{\{s_0\}\}$;
$Waiting \leftarrow \{(\{s_0\}, \alpha, W') \mid \alpha \in \Sigma_1,\ o \in \mathcal{O},\ \ W' = \mathsf{Next}_\alpha(\{s_0\}) \cap o \wedge W' \neq \emptyset\}$;
$Win[\{s_0\}] \leftarrow (\{s_0\} \subseteq \gamma(\mathsf{Goal})\ ?\ 1 : 0)$;
$Losing[\{s_0\}] \leftarrow (\{s_0\} \not\subseteq \gamma(\mathsf{Goal}) \wedge (Waiting = \emptyset \vee \forall \alpha \in \Sigma_1, \mathsf{Sink}_\alpha(s_0) \neq \emptyset)\ ?\ 1 : 0)$;
$Depend[\{s_0\}] \leftarrow \emptyset$;

Main:

while $((Waiting \neq \emptyset) \wedge Win[\{s_0\}] \neq 1 \wedge Losing[\{s_0\}] \neq 1))$ **do**
  $e = (W, \alpha, W') \leftarrow pop(Waiting)$;
  if $s' \notin Passed$ **then**
    $Passed \leftarrow Passed \cup \{W'\}$;
    $Depend[W'] \leftarrow \{(W, \alpha, W')\}$;
    $Win[W'] \leftarrow (W' \subseteq \gamma(\mathsf{Goal})\ ?\ 1 : 0)$;
    $Losing[W'] \leftarrow (W' \not\subseteq \gamma(\mathsf{Goal}) \wedge \mathsf{Sink}_\alpha(W') \neq \emptyset\ ?\ 1 : 0)$;
    if $(Losing[W'] \neq 1)$ **then (* if losing it is a deadlock state *)**
      $NewTrans \leftarrow \{(W', \alpha, W'') \mid \alpha \in \Sigma,\ o \in \mathcal{O},\ \ W' = \mathsf{Next}_\alpha(W) \cap o \wedge W' \neq \emptyset\}$;
      if $NewTrans = \emptyset \wedge Win[W'] = 0$ **then** $Losing[W'] \leftarrow 1$;
    if $(Win[W'] \vee Losing[W'])$ **then** $Waiting \leftarrow Waiting \cup \{e\}$;
    $Waiting \leftarrow Waiting \cup NewTrans$;
  else (* reevaluate *)
    $Win^* \leftarrow \bigvee_{c \in \mathsf{Enabled}(W)} \bigwedge_{W \longrightarrow W''} Win[W'']$;
    if Win^* **then**
      $Waiting \leftarrow Waiting \cup Depend[W]$; $Win[W] \leftarrow 1$;
    $Losing^* \leftarrow \bigwedge_{c \in \mathsf{Enabled}(W)} \bigvee_{W \longrightarrow W''} Losing[W'']$;
    if $Losing^*$ **then**
      $Waiting \leftarrow Waiting \cup Depend[W]$; $Losing[W] \leftarrow 1$;
    if $(Win[W'] = 0 \wedge Losing[W'] = 0)$ **then** $Depend[W'] \leftarrow Depend[W'] \cup \{e\}$;
  endif
endwhile

Fig. 2. OTFPOR: **O**n-**T**he-**F**ly Algorithm for **P**artially **O**bservable **R**eachability

6 seconds to reach the zone where it can be Eject?'ed with a piston. When receiving a heavy box, the speed of the belt is slower and the box takes between 9 and 11 seconds to reach the zone where it can be Eject?'ed by the piston. The controller should command the piston so that the order to Eject? a box is given only when the box is in the right region. The system is modeled by the timed game automaton of Figure 3. The initial location of the automaton is l_0. The system receives boxes when it is in location l_0, if it receives a heavy box then it branches to l_1, if it receives a light box then it branches to l_4. The only event that is shared with the controller is the Eject? event. This event should be issued when the control of the automaton is in l_3 or l_6 (which has the effect of ejecting the box at the end of the belt), in all other locations, if this event is received then the control of the automaton evolves to location l_7 (the bad location that we want to avoid); those transitions are not depicted in the figure. The control objective is to avoid entering location l_7.

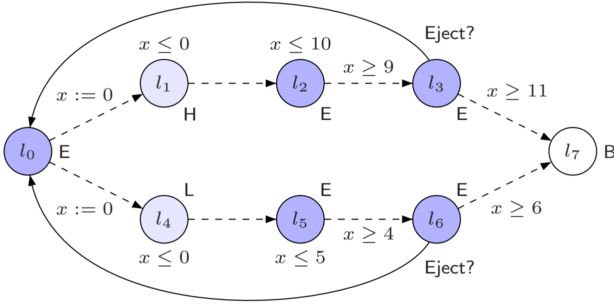


Fig. 3. Timed Game for Sorting Bricks. Edges to l_7 with action Eject? are omitted.

To control the system, the controller can use a clock y that it can reset at any moment. The controller can also issue the order Eject! or propose to play delay, which allows for the time to elapse. The controller has an imperfect information about the state of the system and the value of the clock y. The controller gets information throughout the following observations: E: the control of the automaton is in location l_0, l_2, l_3, l_5, or l_6; H: the control of the automaton is in location l_1; L: the control of the automaton is in location l_4; B: the control of the automaton is in location l_7; $0 \leq y < M$: the value of clock y is in the interval $[0, M[$, M being a parameter. The observations E, H, L, and B are mutually exclusive and cover all the states of the automaton but they can be combined with the observation $0 \leq y < M$ on the clock but also with its complement $y \geq M$. So formally, the set of observations that the controller can receive at any time is $\mathcal{O} = \{(\mathsf{E}, 0 \leq y < M), (\mathsf{E}, y \geq M), (\mathsf{H}, 0 \leq y < M), (\mathsf{H}, y \geq M), (\mathsf{L}, 0 \leq y < M), (\mathsf{L}, y \geq M), (\mathsf{B}, 0 \leq y < M), (\mathsf{B}, y \geq M)\}$. The set of actions that the controller can choose from is $\Sigma_c = \{\mathsf{Reset}_y, \mathsf{Eject!}, \mathsf{delay}\}$.

We modelled this example in our prototype and checked for controllability of the safety property $A\Box\neg\mathsf{B}$. Controllability as well as the synthesized strategy heavily depend on the choice of the parameter M, i.e. the granularity at which the clock y may be set and tested. Table 1 gives the experimental results for $M \in \{1, 0.5, 0.25, 0.2\}$[7]. It turns out that the system is not controllabe for $M = 1$: a granularity of 1 is simply too coarse to determine (up to that granularity) with certainty when, say, a light box will be in l_6 and should be Eject?'ed. The differences between the guards and invariants in l_5, l_6 and l_7 are simply too small. As can be seen from Table 1 the finer granularities yield controllability.

We report on the number of explored state-sets (*state-set*) and the number of state-sets that are part of the strategy (*strat*). To get an impression of the complexity of the problem of controller synthesis under partial observability we note that the the model in Figure 3 has 115 reachable symbolic states when viewed as a regular timed automaton. Table 1 reports on experiments exploiting an additional inclusion checking option in various ways: (*notfi*) without on-the-fly

[7] Fractional values of M are dealt with by multiplying all constants in the timed game automaton with $\frac{1}{M}$.

Table 1. Number of state-sets and size of strategy obtained for different heuristics for observations $0 \leq y < M$ of the clock y with $M \in \{1, 0.5, 0.25, 0.2\}$. The case $M = 1$ is not controllable.

	notfi				otfi			
	state-set	strat	+post	+filter	state-set	strat	+post	+filter
$[0, 0.2[$	110953	36169	1244	70	52615	16372	841	176
$[0, 0.25[$	72829	23750	1014	60	35050	11016	697	146
$[0, 0.5[$	20527	6706	561	41	10586	3460	407	88
$[0, 1[$	2284	-	-	-	1651	-	-	-

inclusion checking, and (*otfi*) with on-the-fly inclusion checking. In addition, we apply the post-processing step of inclusion checking (*+post*) on the strategy and a filtering of the strategy (*+filter*) on top to output only the reachable state-sets under the identified strategy. The results show that on-the-fly inclusion checking gives a substantial reduction in the number of explored state-sets – and is hence substantially faster. Both (*notfi*) and (*otfi*) shows that post processing and filtering reduces the size of the control strategy with a factor of approximately 100. It can also be seen that the size of the final strategy grows when granularity is refined; this is to be expected as the strategies synthesized can be seen to involve counting modulo the given granularity. More suprising is the observation that the final strategies in (*notfi+post+filter*) are uniformly smaller than the final strategies in (*otfi*): *not* performing on-the-fly inclusion checking explores more state-sets, thus having the potential for a better reduction overall.

6 Conclusions and Future Works

During the last five years a number of very promissing algorithmic techniques has been introduced for controller synthesis in general and controller synthesis for timed systems in particular. The contribution of this paper to the decidability and algorithmic support for timed controller synthesis under imperfect information is an important new step within this line of research. Future research includes more experimental investigation as well as search for additional techniques for minimizing the size of the produced strategies (e.g. using minimization wrt. (bi)simulation or alternating simulation). For safety objectives, we need methods to insure that the synthesized strategies do not obtain their objective simply by introducing zeno behaviour. Finally, a rewrite of the prototype as extension of Uppaal-Tiga is planned.

References

1. Altisen, K., Tripakis, S.: Tools for controller synthesis of timed systems. In: Proc. 2nd Work. on Real-Time Tools (RT-TOOLS'02), Proc. published as Technical Report 2002-025, Uppsala University, Sweden (2002)
2. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

3. Asarin, E., Maler, O.: As Soon as Possible: Time Optimal Control for Timed Automata. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 19–30. Springer, Heidelberg (1999)
4. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller Synthesis for Timed Automata. In: Proc. IFAC Symp. on System Structure & Control, pp. 469–474. Elsevier Science, Amsterdam (1998)
5. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., Lime, D.: Uppaal-tiga: Time for playing games! In: Damm, W., Herrmanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
7. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society Press, Los Alamitos (2006)
8. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
9. Cassez, F., David, A., Fleury, E., Larsen, K., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
10. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games with imperfect information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
11. Dill, D.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) Workshop on Automatic Verification Methods for Finite-State Systems. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
12. Laroussinie, F., Larsen, K.G.: CMC: A tool for compositional model-checking of real-time systems. In: Budkowski, S., Cavalli, A.R., Najm, E. (eds.) Proc. of IFIP TC6 WG6.1 Joint Int. Conf. FORTE'XI and PSTV'XVIII. IFIP Conf. Proc, vol. 135, pp. 439–456. Kluwer Academic Publishers, Dordrecht (1998)
13. Laroussinie, F., Larsen, K.G., Weise, C.: From timed automata to logic - and back. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 529–539. Springer, Heidelberg (1995)
14. Larsen, K., Pettersson, P., Yi, W.: Model-checking for real-time systems. Fundamentals of Computation Theory, 62–88 (1995)
15. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
16. Tripakis, S., Altisen, K.: On-the-Fly Controller Synthesis for Discrete and Timed Systems. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 233–252. Springer, Heidelberg (1999)
17. Wulf, M.D., Doyen, L., Raskin, J.-F.: A lattice theory for solving games of imperfect information. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 153–168. Springer, Heidelberg (2006)

Deciding Simulations on Probabilistic Automata*

Lijun Zhang and Holger Hermanns

Department of Computer Science, Saarland University, Saarbrücken, Germany

Abstract. Probabilistic automata are a central model for concurrent systems exhibiting random phenomena. This paper presents, in a uniform setting, efficient decision algorithms for strong simulation on probabilistic automata, but with subtly different results. The algorithm for strong probabilistic simulation is shown to be of polynomial complexity via a reduction to LP problem, while the algorithm for strong simulation has complexity $\mathcal{O}(m^2 n)$. The former relation allows for convex combinations of transitions in the definition and is thus less discriminative than the latter. As a byproduct, we obtain minimisation algorithms with respect to strong simulation equivalences and – for Markov decision processes – also to strong bisimulation equivalences. When extending these algorithms to the continuous-time setting, we retain same complexities for both strong simulation and strong probabilistic simulations.

1 Introduction

Randomization has been employed widely for performance and dependability models, and consequently the study of verification techniques of probabilistic systems has drawn a lot of attention in recent years. In this paper, we consider probabilistic automata (PAs) in the style of Segala & Lynch [13], which extend transition systems with probabilistic selection. They constitute a natural model of concurrent computation involving random phenomena. In a nutshell, a labelled transition in some PA leads to a probability distribution over the set of states, rather than a single state. The resulting model thus exhibits both nondeterministic choice (as in labelled transition systems) and probabilistic choice (as in Markov chains). A special case of PAs is formed by Markov decision processes (MDPs) which in their standard form do not have nondeterminism between the equally-labelled transitions [11].

Similar to the transition system setting, strong bisimulation and strong simulation relations [9, 10, 13] have been proposed as means to compare the stepwise behaviour of states in PAs. Intuitively, state s is simulated by another state s', formally $s \precsim s'$ ("s' simulates s"), if state s' can mimic all stepwise behaviours of s; the converse, i. e., $s' \precsim s$ is not necessarily guaranteed, so state s' may perform steps that cannot be matched by s. In the non-probabilistic setting, $s \precsim s'$

* This work is supported by the NWO-DFG bilateral project VOSS and by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 207–222, 2007.
© Springer-Verlag Berlin Heidelberg 2007

requires every successor of s via some action α to be related to a corresponding successor of s' reachable via the same action α. For PAs, the above condition is lifted to distributions: It is required that every successor distribution of s via action α, called α-successor distribution, has a corresponding α-successor distribution at s'. The correspondence of distributions is naturally defined with the concept of weight functions [9].

In the context of model checking, strong simulation relations can be used to combat the infamous state space explosion problem, owed to the preservation of PCTL-safety formulas [13]. The kernel of strong simulation, i. e., strong simulation equivalence, preserves both safe and live fragments of PCTL. Therefore, one can perform model checking on the quotient induced by these equivalences, if interested in safety or liveness properties. Since strong simulation equivalence is strictly coarser than strong bisimulation, the induced quotient automaton is also smaller.

All statements in the above paragraph stay perfectly valid if considering "strong *probabilistic* simulation" instead of "strong simulation". The former [13] is a relaxation of the latter in the sense that it enables convex combinations of multiple distributions belonging to equally labelled transitions. More concretely, assume that a state s has no α-successor distribution which can be related to an α-successor distribution of s', yet there exists such a so-called *α-combined transition*, a convex combination of several α-successor distributions. Strong probabilistic simulation accounts for this and is thus coarser than strong simulation, but still preserves the same class of PCTL-properties as strong simulation does. Since it is coarser, the induced simulation quotient is potentially again smaller.

Cattani and Segala [6] have presented decision algorithms for strong (probabilistic) *bi*simulation for PAs. They reduced the decision problems to linear programming (LP) problems. In this paper, we will focus on decision algorithms for strong (probabilistic) simulation and strong (probabilistic) simulation equivalence for PAs. We will also extend the notion of strong (probabilistic) simulation to the continuous-time setting and study corresponding decision algorithms.

To compute the coarsest strong simulation for PAs, Baier *et. al.* [2] presented an algorithm which reduces the query whether a state strongly simulates another to a maximum flow problem. Their algorithm has complexity $\mathcal{O}((mn^6 + m^2n^3)/\log n)$ for PAs[1], where n denotes the number of states and m denotes the number of transitions. For Markov chains, we have presented an algorithm [15] with complexity $\mathcal{O}(m^2n)$. This algorithm is also based on maximum flows, however it exploits the similarity of successive network flows across iterations. In the present paper, we extend that algorithm to the PA case and retain its complexity of $\mathcal{O}(m^2n)$. Especially in the very common case, where the state fanout of a model is bounded by a constant k (and hence $m \leq kn$), our strong simulation algorithm has complexity $\mathcal{O}(n^2)$. The computational complexity of strong probabilistic simulation has not been tackled yet. We show that it can be determined by solving LP problems. As a byproduct of the decision algorithms

[1] Note that the m used here is slight different from the m as we use it. A detailed comparison is provided later, in Remark 1 of Section 4.

for strong simulation preorders, we obtain one for strong simulation equivalences $\precsim \cap \precsim^{-1}$. For the special case of MDPs [11], which arise from PAs by disallowing nondeterministic choices of equally labelled transitions and sub-stochastic distributions, strong simulation equivalence and strong bisimulation coincide [1]. We thus obtain a decision algorithm for computing strong bisimulations for MDPs.

Further, we consider continuous-time probabilistic automata (CPAs), the continuous-time counterpart of PAs. We give decision algorithms for strong simulation and strong probabilistic simulation on CPAs. For both of them, we show that the decision algorithm have the same complexity as the corresponding one for PAs.

In summary, our paper makes the following contributions: It presents novel decision algorithms for strong (probabilistic) simulation on discrete-time and continuous-time probabilistic automata (PAs, CPAs), and does so in a uniform way. As special cases, discrete-time and continuous-time Markov decision processes are considered, where in particular, our results yield an efficient algorithm to compute strong *bi*simulations.

Organisation of this paper. In Section 2 we recall necessary definitions of models and relations we consider. Section 3 recalls how to effectively compute strong simulation relations on fully probabilistic systems. Then, we extend the algorithm in Section 4 to deal with PAs, and show that strong probabilistic simulation can be computed by solving LP problems. We will also discuss strong (probabilistic) simulation equivalences. The algorithms are extended to CPAs in Section 5. Section 6 concludes the paper.

2 Preliminaries

This section introduces the basic models and simulation relations we consider.

Models. Let AP be a fixed, finite set of atomic propositions. Let X, Y be finite sets. For $f : X \to \mathbb{R}$, let $f(A)$ denote $\sum_{x \in A} f(x)$ for all $A \subseteq X$. If $f : X \times Y \to \mathbb{R}$ is a two-dimensional function, let $f(x, A)$ denote $\sum_{y \in A} f(x, y)$ for all $x \in X$ and $A \subseteq Y$, and $f(A, y)$ denote $\sum_{x \in A} f(x, y)$ for all $y \in Y$ and $A \subseteq X$. For a finite set S, a distribution μ on S is a function $\mu : S \to [0, 1]$ satisfying the condition $\mu(S) \leq 1$. The support of μ is defined by $Supp(\mu) = \{s \mid \mu(s) > 0\}$, and the size of μ is defined by $|\mu| = |Supp(\mu)|$. The distribution μ is called stochastic if $\mu(S) = 1$, absorbing if $\mu(S) = 0$, and sub-stochastic otherwise. We use an auxiliary state (not a *real* state) $\perp \notin S$ and set $\mu(\perp) = 1 - \mu(S)$. Note that $\mu(\perp) > 0$ if μ is not stochastic. Further, let $S_\perp$ denote the set $S \cup \{\perp\}$, and let $Supp_\perp(\mu) = Supp(\mu) \cup \{\perp\}$ if $\mu(\perp) > 0$. We let $Dist(S)$ denote the set of distributions over the set S. We recall the definition of probabilistic automata [13]:

Definition 1. *A probabilistic automaton (PA) is a tuple $\mathcal{M} = (S, Act, \mathbf{P}, L)$ where S is a finite set of states, Act is a finite set of actions, $\mathbf{P} \subseteq S \times Act \times Dist(S)$ is a finite set, called the probabilistic transition matrix, and $L : S \to 2^{AP}$ is a labeling function.*

For $(s, \alpha, \mu) \in \mathbf{P}$, we use $s \xrightarrow{\alpha} \mu$ as a shorthand notation, and call μ an α-successor distribution of s. Let $Act(s) = \{\alpha \mid \exists \mu : s \xrightarrow{\alpha} \mu\}$ denote the set of actions enabled at s. For $s \in S$ and $\alpha \in Act(s)$, let $Steps_\alpha(s) = \{\mu \in Dist(S) \mid s \xrightarrow{\alpha} \mu\}$ and $Steps(s) = \bigcup_{\alpha \in Act(s)} Steps_\alpha(s)$. We introduce the notion of *fanout* for $\mathcal{M}$. The fanout of a state s is defined by $fan(s) = \sum_{\alpha \in Act(s)} \sum_{\mu \in Steps\ (s)} |\mu|$. Intuitively, $fan(s)$ denote the total sum of size of outgoing distributions of state s. The fanout of $\mathcal{M}$ is defined by $\max_{s \in S} fan(s)$.

A *Markov decision process* (MDP) [11] arises from the PA $\mathcal{M}$ such that for $s \in S$ and $\alpha \in Act$, there is at most one α-successor distribution μ of s which must be stochastic. $\mathcal{M}$ is a *fully probabilistic system* (FPS) if for $s \in S$, there is at most one transition $s \xrightarrow{\alpha} \mu$. A discrete-time Markov chain (DTMC) is a FPS where all distributions are either stochastic or absorbing. For ease of notation, we give a simpler definition for FPSs by dropping the set of actions:

Definition 2. *An FPS is a tuple $\mathcal{D} = (S, \mathbf{P}, L)$ where S, L as defined for PAs, and $\mathbf{P} : S \times S \to [0, 1]$ is the probabilistic transition matrix such that $\mathbf{P}(s, \cdot) \in Dist(S)$ for all $s \in S$.*

Strong simulation relations. Strong simulation requires that every α-successor distribution of one state have a corresponding α-successor distribution of the other state. The correspondence of distributions is naturally defined with the concept of *weight functions* [9].

Definition 3. *Let $\mu, \mu' \in Dist(S)$ and $R \subseteq S \times S$. A weight function for (μ, μ') with respect to R, denoted by $\mu \sqsubseteq_R \mu'$, is a function $\Delta : S_\perp \times S_\perp \to [0, 1]$ such that $\Delta(s, s') > 0$ implies $s\ R\ s'$ or $s = \perp$, $\mu(s) = \Delta(s, S_\perp)$ for $s \in S_\perp$ and $\mu'(s') = \Delta(S_\perp, s')$ for $s' \in S_\perp$.*

Now we recall the definition of strong simulation for PAs [9, 13]:

Definition 4. *Let $\mathcal{M} = (S, Act, \mathbf{P}, L)$ be a PA. $R \subseteq S \times S$ is a* strong simulation *on $\mathcal{M}$ iff for all s_1, s_2 with $s_1\ R\ s_2$: $L(s_1) = L(s_2)$ and if $s_1 \xrightarrow{\alpha} \mu_1$ then there exists a transition $s_2 \xrightarrow{\alpha} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$. We write $s_1 \precsim_{\mathcal{M}} s_2$ iff there exists a strong simulation R on $\mathcal{M}$ such that $s_1\ R\ s_2$.*

We say also that s_2 strongly simulates s_1 in $\mathcal{M}$ iff $s_1 \precsim_{\mathcal{M}} s_2$. Obviously $\precsim_{\mathcal{M}}$ is the coarsest strong simulation relation for $\mathcal{M}$.

Simulation up to R. For an arbitrary relation R on the state space S of $\mathcal{M}$ with $s_1\ R\ s_2$, we say that s_2 simulates s_1 strongly up to R, denoted $s_1 \precsim_R s_2$, if $L(s_1) = L(s_2)$ and if $s_1 \xrightarrow{\alpha} \mu_1$ then there exists a transition $s_2 \xrightarrow{\alpha} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$. Otherwise we write $s_1 \not\precsim_R s_2$. Note that $s_1 \precsim_R s_2$ does not imply $s_1 \precsim_{\mathcal{M}} s_2$ unless R is a strong simulation, since only the first step is considered for $\precsim_R$.

Strong Probabilistic Simulation Relations. We recall first the notion of combined transition [13], a convex combination of several equally labelled transitions:

Definition 5. *Let $\mathcal{M} = (S, Act, \mathbf{P}, L)$ be a PA. Assume that $Steps_\alpha(s) = \{\mu_1, \ldots, \mu_k\}$ where $k = |Steps_\alpha(s)|$. The tuple (s, α, μ) is a combined transition, denoted by $s \xrightarrow{\alpha}_C \mu$, iff there exist constants $c_1, \ldots, c_k \in [0,1]$ with $\sum_{i=1}^{k} c_i = 1$ such that $\mu(s) = \sum_{i=1}^{k} c_i \mu_i(s)$ for all $s \in S$.*

Strong probabilistic simulation is insensitive to combined transitions [13], thus, it is a relaxation of strong simulation. It is coarser than strong simulation, but still preserves the same class of PCTL-properties as strong simulation does. The key difference to Definition 4 is the use of $\xrightarrow{\alpha}_C$ instead of $\xrightarrow{\alpha}$:

Definition 6. *Let $\mathcal{M} = (S, Act, \mathbf{P}, L)$ be a PA. $R \subseteq S \times S$ is a* strong probabilistic simulation *on $\mathcal{M}$ iff for all s_1, s_2 with $s_1 \; R \; s_2$: $L(s_1) = L(s_2)$ and if $s_1 \xrightarrow{\alpha} \mu_1$ then there exists a combined transition $s_2 \xrightarrow{\alpha}_C \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$. We write $s_1 \precsim^p_{\mathcal{M}} s_2$ iff there exists a strong probabilistic simulation R on $\mathcal{M}$ such that $s_1 \; R \; s_2$.*

Similar to strong simulation, $\precsim^p_{\mathcal{M}}$ is the coarsest strong probabilistic simulation relation for $\mathcal{M}$. The definition of simulation up to R for strong simulation ($\precsim_R$) carries over directly to strong probabilistic simulation, denoted by ($\precsim^p_R$). Since MDPs can be considered as special PAs, we obtain the notion of strong simulation and strong probabilistic simulation for MDPs. Moreover, strong simulation and strong probabilistic simulation trivially coincide for MDPs as, by definition, for each state there is at most one successor distribution per action. Note that the above statements are also true for FPSs.

3 Algorithms for Fully Probabilistic Systems

In this section, we briefly review the algorithm to decide strong simulation preorder for FPSs. For more detail, we refer to [15], where the maximum flow problem and the preflow algorithm for computing maximum flow are also repeated, which are key components of the decision algorithms to be presented. As DTMCs are special FPSs, the algorithm applies directly.

Firstly, we discuss the decisive part of the algorithm: The check whether s_2 strongly simulates s_1 up to a relation R, i. e., $s_1 \precsim_R s_2$. As the condition $L(s_1) = L(s_2)$ is easy to check, we need to check whether $\mathbf{P}(s_1, \cdot) \sqsubseteq_R \mathbf{P}(s_2, \cdot)$ holds. This is reduced to a maximum flow computation on the network $\mathcal{N}(\mathbf{P}(s_1, \cdot), \mathbf{P}(s_2, \cdot), R)$ constructed out of $\mathbf{P}(s_1, \cdot)$, $\mathbf{P}(s_2, \cdot)$ and R. This network is constructed via a graph containing a copy $\overline{t} \in \overline{S_\perp}$ of each state $t \in S_\perp$ where $\overline{S_\perp} = \{\overline{t} \mid t \in S_\perp\}$ defined as follows: Let $\nearrow$ (the source) and $\searrow$ (the sink) be two additional vertices not contained in $S_\perp \cup \overline{S_\perp}$. For functions $\mu, \mu' : S \to \mathbb{R}_{\geq 0}$ and a relation $R \subseteq S \times S$ we define the network $\mathcal{N}(\mu, \mu', R) = (V, E, u)$ with the set of vertices $V = \{\nearrow, \searrow\} \cup Supp_\perp(\mu) \cup \overline{Supp_\perp(\mu')}$ and the set of edges E defined by $E = \{(s, \overline{t}) \mid (s,t) \in R \vee s = \perp\} \cup \{(\nearrow, s), (\overline{t}, \searrow)\}$ where $s \in Supp_\perp(\mu)$ and $t \in Supp_\perp(\mu')$. The capacity function u is defined as follows: $u(\nearrow, s) = \mu(s)$ for all $s \in S_\perp$, $u(\overline{t}, \searrow) = \mu'(t)$ for all $t \in S_\perp$, $u(s, \overline{t}) = \infty$ for all $(s,t) \in E$ and $u(v, w) = 0$ otherwise. This network is a bipartite network, where the vertices can

be partitioned into two subsets $V_1 := Supp_\perp(\mu) \cup \{\searrow\}$ and $V_2 := \overline{Supp_\perp(\mu')} \cup \{\nearrow\}$ such that all edges have one endpoint in V_1 and another in V_2. For two states s_1, s_2 of an FPS, we let $\mathcal{N}(s_1, s_2, R)$ denote the network $\mathcal{N}(\mathbf{P}(s_1, \cdot), \mathbf{P}(s_2, \cdot), R)$. The following lemma [2] expresses the crucial relationship between maximum flows and weight functions:

Lemma 1. *Let S be a finite set of states and R be a relation on S. Let $\mu, \mu' \in Dist(S)$. Then, $\mu \sqsubseteq_R \mu'$ iff the maximum flow in $\mathcal{N}(\mu, \mu', R)$ is 1.*

Thus we can decide $s_1 \precsim_R s_2$ by computing the maximum flow in $\mathcal{N}(s_1, s_2, R)$. A key observation we made in [15] is that the networks $\mathcal{N}(s_1, s_2, \cdot)$ constructed later in successive iterations are very similar: They differ from iteration to iteration only by deletion of some edges induced by the successive clean up of R. The algorithm, which we shall repeat later, exploits this fact by leveraging preflow rather than re-starting maximum flow computation from scratch each time. Formally, we look at the network $\mathcal{N}(s_1, s_2, R_{init})$ where $R_{init} = \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$. Let $D_1, \ldots, D_k$ be pairwise disjoint subsets of R_{init}, which correspond to the pairs deleted from R_{init} in iteration i. Let $\mathcal{N}(s_1, s_2, R_i)$ denote $\mathcal{N}(s_1, s_2, R_{init})$ if $i = 1$, and $\mathcal{N}(s_1, s_2, R_{i-1} \setminus D_{i-1})$ if $1 < i \leq k+1$. Let f_i denote the maximum flow of the network $\mathcal{N}(s_1, s_2, R_i)$ for $i = 1, \ldots, k+1$. We address the problem of checking $|f_i| = 1$ for all $i = 1, \ldots, k+1$. Very similar to the *parametric maximum algorithm* [7, p. 34], the algorithm $\textsc{Smf}_{(s_1,s_2)}$ (*sequence of maximum flows*) for the pair (s_1, s_2) consists of initialising the preflow $f_{(s_1,s_2)}$ and the distance function $d_{(s_1,s_2)}$ as for the preflow algorithm, setting $i = 0$, and repeating the following steps at most k times:

$\textsc{Smf}_{(s_1,s_2)}$

1. Increase i by 1. If $i = 1$ go to step 2. Otherwise, for all pairs $(u_1, u_2) \in D_{i-1}$, set $f_{(s_1,s_2)}(u_1, \overline{u_2}) = 0$ and replace the flow $f_{(s_1,s_2)}(\overline{u_2}, \searrow)$ by $f_{(s_1,s_2)}(\overline{u_2}, \searrow) - f_{(s_1,s_2)}(u_1, \overline{u_2})$. Set $\mathcal{N}(s_1, s_2, R_i) = \mathcal{N}(s_1, s_2, R_{i-1} \setminus D_{i-1})$. Let $f_{(s_1,s_2)}$ and $d_{(s_1,s_2)}$ be the resulting flow and final valid distance function.
2. Apply the preflow algorithm to calculate the maximum flow for $\mathcal{N}(s_1, s_2, R_i)$ with preflow $f_{(s_1,s_2)}$ and distance function $d_{(s_1,s_2)}$.
3. If $|f_{(s_1,s_2)}| < 1$ return false for all $j \geq i$. Otherwise, return true and continue with step 1.

To understand this algorithm, assume $i > 1$. At step (1.), before we remove the edges D_{i-1} from the network $\mathcal{N}(s_1, s_2, R_{i-1})$, we modify the flow $f_{(s_1,s_2)}$, which is the maximum flow of the network $\mathcal{N}(s_1, s_2, R_{i-1})$, by

- setting $f_{(s_1,s_2)}(u_1, \overline{u_2}) = 0$ for all deleted edges $(u_1, u_2) \in D_{i-1}$, and
- modifying $f_{(s_1,s_2)}(\overline{u_2}, \searrow)$ such that the flow $f_{(s_1,s_2)}$ becomes consistent with the flow conservation rule.

The excess $e(v)$ is increased if there exists $(v, w) \in D_{i-1}$ such that $f_{(s_1,s_2)}(v, w) > 0$, and unchanged otherwise. Hence, the modified flow is a preflow. The distance function $d_{(s_1,s_2)}$ keeps valid, since by removing the set of edges D_{i-1}, no new

SimRel($\mathcal{D}$)

```
1  R, R_new ← {(s1, s2) ∈ S × S | L(s1) = L(s2)}
2  l ← 0          // auxiliary variable to count the number of iterations.
3  for ((s1, s2) ∈ R)
4      Construct the initial network N(s1, s2, R_init) := N(s1, s2, R)
5      Initialise the flow function f_(s1,s2) and the distance function d_(s1,s2)
6      Listener_(s1,s2) ← {(u1, u2) | u1 ∈ pre(s1) ∧ u2 ∈ pre(s2) ∧ L(u1) = L(u2)}
7  do
8      l++
9      D ← R\R_new and R ← R_new and R_new ← ∅
10     for ((s1, s2) ∈ D)
11         for ((u1, u2) ∈ Listener_(s1,s2))
12             D_l^(u1,u2) ← D_l^(u1,u2) ∪ {(s1, s2)}
13     for ((s1, s2) ∈ R)
14         if (SMF_(s1,s2) returns true on the set D_l^(s1,s2))
15             R_new ← R_new ∪ {(s1, s2)}.
16 until(R_new = R)
17 return R
```

Fig. 1. Efficient algorithm for deciding strong simulation for FPSs

residual edges are induced. This guarantees that, at step (2.), the *preflow algorithm* finds a maximum flow over the network $\mathcal{N}(s_1, s_2, R_i)$. If $|f_{(s_1,s_2)}| < 1$ at some iteration i, then $|f_{(s_1,s_2)}| < 1$ for all iterations $j \geq i$ because more edges will be deleted in subsequent iterations. Therefore, at step (3.), the algorithm returns true and continues with step (1.) if $|f_{(s_1,s_2)}| = 1$, otherwise, returns false for all subsequent iterations. Let $post(s)$ denote $Supp(\mathbf{P}(s,\cdot))$, i. e., the set of successor states of s. The complexity of the algorithm [15] is given by:

Lemma 2. *Let $D_1, \ldots, D_k$ be pairwise disjoint subsets of $R_{init} \cap post(s_1) \times post(s_2)$. Let f_i denote the flow constructed at the end of step (2.) in iteration i. Assume that $|post(s_1)| \leq |post(s_2)|$. The algorithm* SMF$_{(s_1,s_2)}$ *correctly computes maximum flow f_i for $\mathcal{N}(s_1, s_2, R_i)$ where $i = 1, \ldots, k+1$, and runs in time $\mathcal{O}(|post(s_1)||post(s_2)|^2)$.*

The algorithm SimRel for deciding strong simulation for FPSs is depicted in Fig. 1. It takes the model $\mathcal{D}$ as a parameter. To calculate the strong simulation relation for $\mathcal{D}$, the algorithm starts with the trivial relation $R_{init} = \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$ (line 1). The variable l (line 2) denotes the number of iterations of the until-loop, and the set D (line 9) contains edges removed from R. For every pair $(s_1, s_2) \in R_{init}$, the network $\mathcal{N}(s_1, s_2, R_{init})$ (line 4), the flow function $f_{(s_1,s_2)}$ and the distance function $d_{(s_1,s_2)}$ are initialised as for the preflow algorithm (line 5). At line 6 a set

$$\mathbf{Listener}_{(s_1,s_2)} = \{(u_1, u_2) \mid u_1 \in pre(s_1) \wedge u_2 \in pre(s_2) \wedge L(u_1) = L(u_2)\}$$

is saved, where $pre(s) = \{t \in S \mid \mathbf{P}(t,s) > 0\}$ is the set of predecessors of s. The set $\mathbf{Listener}_{(s_1,s_2)}$ contains all pairs (u_1, u_2) such that the network $\mathcal{N}(u_1, u_2, R)$ contains the edge $(s_1, \overline{s_2})$. In lines 10–12, the pair (s_1, s_2) is inserted into the set $D_l^{(u_1,u_2)}$ if $(s_1, s_2) \in D$ and $(u_1, u_2) \in \mathbf{Listener}_{(s_1,s_2)}$. $D_l^{(u_1,u_2)}$ contains edges which should be removed to update the network for (u_1, u_2) in iteration l. At line 14, the algorithm $\text{SMF}_{(s_1,s_2)}$ constructs the maximum flow for the set $D_l^{(s_1,s_2)}$. Note that l corresponds to i in SMF. The initialisation of SMF corresponds to lines 4–5. In the first iteration (in which $D_1^{(s_1,s_2)} = \emptyset$), $\text{SMF}_{(s_1,s_2)}$ skips the computations in step (1.) and proceeds directly to step (2.), in which the maximum flow f_1 for $\mathcal{N}(s_1, s_2, R_{init})$ is constructed. In iteration $l > 1$, $\text{SMF}_{(s_1,s_2)}$ takes the set $D_l^{(s_1,s_2)}$, updates the flow f_{l-1} and the network, and constructs the maximum flow f_l for the network $\mathcal{N}(s_1, s_2, R_l)$. If $\text{SMF}_{(s_1,s_2)}$ returns true, (s_1, s_2) is inserted into R_{new} and survives this iteration. Otherwise, s_2 cannot strongly simulate s_1 up to the current relation R, hence the pair (s_1, s_2) is removed. This proceeds until there is no such pair left (line 16), i. e., $R_{new} = R$. Invariantly throughout the loop it holds that R is coarser than $\precsim_{\mathcal{D}}$. Hence, we obtain the strong simulation preorder $\precsim_{\mathcal{D}} = R$, once the algorithm terminates (line 17). For a given FPS, let m, n denote the number of transitions and states. Note that the fanout for FPSs is given by $\max_{s \in S} |\mathbf{P}(s, \cdot)|$. We recall the decisive complexity result from [15]:

Lemma 3. $\text{SimRel}(\mathcal{D})$ *runs in time* $\mathcal{O}(m^2 n)$ *and in space* $\mathcal{O}(m^2)$*. If the fanout is bounded by a constant, it has complexity* $\mathcal{O}(n^2)$*, both in time and space.*

4 Algorithms for Probabilistic Automata

In this section we present algorithms for deciding strong (probabilistic) simulations for PAs. First, we extend the algorithm SimRel to deal with strong simulation for PAs. For strong probabilistic simulation, we show that the algorithm can be reduced to LP problems. Finally, we discuss the strong (probabilistic) simulation equivalence for PAs.

Strong Simulations. We aim to extend SimRel in Fig. 1 to determine the strong simulation on PAs instead of FPSs. Assume that $L(s_1) = L(s_2)$. We consider line 14, which checks the condition $\mathbf{P}(s_1, \cdot) \sqsubseteq_R \mathbf{P}(s_2, \cdot)$ using SMF. By Definition 4 of strong simulation for PAs, we should check the condition

$$\forall s_1 \xrightarrow{\alpha} \mu_1.\ \exists s_2 \xrightarrow{\alpha} \mu_2 \text{ with } \mu_1 \sqsubseteq_R \mu_2 \tag{1}$$

instead. Recall the condition $\mu_1 \sqsubseteq_R \mu_2$ is true iff the maximum flow of the network $\mathcal{N}(\mu_1, \mu_2, R)$ has value one. For notational convenience, this network is denoted by $\mathcal{N}(s_1, \alpha, \mu_1, s_2, \mu_2, R)$. We say that the check $\mu_1 \sqsubseteq_R \mu_2$ is successful if the corresponding maximum flow has value one, and unsuccessful otherwise.

Our goal is to carry out a sequence of checks on similar networks (obtained by successive clean up of R) using only a single call to a slightly adaption of the algorithm SMF. For any states s_1, s_2, action α, consider the two transitions

$s_1 \xrightarrow{\alpha} \mu_1$ and $s_2 \xrightarrow{\alpha} \mu_2$. Let $k(\mu_1, \mu_2)$ denote the number of successful checks of $\mu_1 \sqsubseteq_R \mu_2$. We modify the algorithm SMF slightly such that the $k(\mu_1, \mu_2)$ successful checks for (s_1, s_2), plus at most one unsuccessful check, can be performed by only a single call to SMF. To enable that, we take α, μ_1, μ_2 as additional parameters for SMF. Now, for $(s_1, s_2) \in R$ with $s_1 \xrightarrow{\alpha} \mu_1$ and $s_2 \xrightarrow{\alpha} \mu_2$, the network $\mathcal{N}(s_1, \alpha, \mu_1, s_2, \mu_2, R)$ is constructed instead of $\mathcal{N}(s_1, s_2, R)$. Other parts of SMF remain unchanged. Denote the modified version by SMF'$_{(s_1,\alpha,\mu_1,s_2,\mu_2)}$. Let $R_{init} = \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$. As complexity of SMF' we get:

Lemma 4. *Let $(s_1, s_2) \in R_{init}$. Consider the two transitions $s_1 \xrightarrow{\alpha} \mu_1$ and $s_2 \xrightarrow{\alpha} \mu_2$. Let $D_1, \ldots, D_k$ be pairwise disjoint subsets of $R_{init} \cap Supp(\mu_1) \times Supp(\mu_2)$. Let f_i denote the flow constructed at the end of step (2.) in iteration i of* SMF'*. Assume that $|\mu_1| \le |\mu_2|$. The algorithm* SMF'$_{(s_1,\alpha,\mu_1,s_2,\mu_2)}$ *correctly computes maximum flow f_i for $\mathcal{N}(s_1, \alpha, \mu_1, s_2, \mu_2, R_i)$ where $i = 1, \ldots, k+1$, and runs in time $\mathcal{O}(|\mu_1||\mu_2|^2)$.*

The algorithm SIMREL for deciding strong simulation for PAs is presented in Fig. 2. We use similar notations as Baier *et. al.* in [2]. During the initialisation (lines 2–8), for $(s_1, s_2) \in R$ and $s_1 \xrightarrow{\alpha} \mu_1$, the set $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ is initialised to $Steps_\alpha(s_2)$ (line 4). Intuitively, $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ contains all potential candidates of α-successor distributions of s_2 which could be used to establish the condition $\mu_1 \sqsubseteq_R \mu_2$ for the relation R considered. The set $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ is represented as a list. We use the operation **head**$(\cdot)$ to get the first element of the list, and use the operation **tail**$(\cdot)$ to remove the first element of the list. The operation **empty**$(\cdot)$ checks whether the list is empty. At line 5, the first element of $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ is assigned to μ_2. The network $\mathcal{N}(s_1, \alpha, \mu_1, s_2, \mu_2, R_{init})$, preflow and distance function for it are initialised (lines 6–7) as for SIMREL($\mathcal{D}$). Similarly, the set **Listener**$_{(s_1,s_2)}$ for (s_1, s_2) is introduced which contains tuples $(u_1, \alpha, \mu_1, u_2, \mu_2)$ such that the network $\mathcal{N}(u_1, \alpha, \mu_1, u_2, \mu_2, R_{init})$ contains the edge $(s_1, \overline{s_2})$.

In the main iteration, the sets D, R, R_{new} and $D_l^{(u_1,\alpha,\mu_1,u_2,\mu_2)}$ are updated (lines 10–13) in a similar way as SIMREL($\mathcal{D}$) for FPSs. Lines 14–30 check condition 1 by exploiting the modified algorithm SMF'. For the moment we fix the pair $(s_1, s_2) \in R$. For $s_1 \xrightarrow{\alpha} \mu_1$, a boolean variable $match_\alpha$ is introduced, which is initialised to **false** (line 16), and has value **true** iff SMF'$_{(s_1,\alpha,\mu_1,s_2,\mu_2)}$ returns true on the set $D_l^{(s_1,\alpha,\mu_1,s_2,\mu_2)}$ (lines 19–20). In this case, we break the while loop (line 21), and continue to check the next successor distribution of s_1. If SMF'$_{(s_1,\mu_1,s_2,\mu_2,\alpha)}$ returns **false**, we remove the first element of $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ (line 22), and take the next candidate of μ_2 (line 26) if the set $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ is not empty (line 23). If it is empty, we can not find an α-successor distribution related to μ_1, so the variable $match_\alpha$ remains false. In this case the pair (s_1, s_2) does not survive this iteration, and will be dropped out later at line 30. Assume now that the set $Sim_{(s_1,\alpha,\mu_1)}(s_2)$ is not empty. In this case the set $D_l^{(s_1,\alpha,\mu_1,s_2,\mu_2)}$ is then reset to $\emptyset$ (line 27), and the network, preflow, distance function are initialised (lines 28–29) for the new candidate μ_2. Then, we start from the beginning of the while loop, and check if the new candidate μ_2 satisfies the condition $\mu_1 \sqsubseteq_R \mu_2$. Note that at line 30, the condition is true if and only

```
SimRel(M)
 1  R, R_new ← {(s1, s2) ∈ S × S | L(s1) = L(s2)}
 2  for ((s1, s2) ∈ R)
 3     for (α ∈ Act(s1), μ1 ∈ Steps_α(s1))
 4          Sim_(s1,α,μ1)(s2) ← Steps_α(s2)
 5          μ2 ← head(Sim_(s1,α,μ1)(s2))
 6          Construct the initial network N(s1, α, μ1, s2, μ2, R)
 7          Initialise the flow and distance functions for N(s1, α, μ1, s2, μ2, R)
 8     Listener_(s1,s2) ← {(u1, α, μ1, u2, μ2) |
                    u1 -α-> μ1 ∧ u2 -α-> μ2 ∧ μ1(s1) > 0 ∧ μ2(s2) > 0 ∧ L(u1) = L(u2)}
 9  do
10     D ← R\R_new and R ← R_new and R_new ← ∅
11     for ((s1, s2) ∈ D)
12          for ((u1, α, μ1, u2, μ2) ∈ Listener_(s1,s2))
13            D_l^(u1,α,μ1,u2,μ2) ← D_l^(u1,α,μ1,u2,μ2) ∪ {(s1, s2)}
14     for ((s1, s2) ∈ R)
15          for (α ∈ Act(s1), μ1 ∈ Steps_α(s1))
16            match_α ← false
17            while (!empty(Sim_(s1,α,μ1)(s2)))
18                 μ2 ← head(Sim_(s1,α,μ1)(s2))
19                 if (Smf'_(s1,α,μ1,s2,μ2) returns true on the set D_l^(s1,α,μ1,s2,μ2))
20                    match_α ← true
21                    break
22                 tail(Sim_(s1,α,μ1)(s2))
23                 if (empty(Sim_(s1,α,μ1)(s2)))
24                    match_α ← false
25                    break
26                 μ2 ← head(Sim_(s1,α,μ1)(s2))
27                 D_l^(s1,α,μ1,s2,μ2) ← ∅
28                 Construct the initial network N(s1, α, μ1, s2, μ2, R)
29                 Initialise the flow and distance functions for N(s1, α, μ1, s2, μ2, R)
30          if (⋀_{α∈Act(s1)} match_α) R_new ← R_new ∪ {(s1, s2)}
31  until R_new = R
32  return R
```

Fig. 2. Efficient algorithm for deciding strong simulation for PAs

if $match_\alpha$ is true for all $\alpha \in Act(s_1)$. In this case condition 1 is satisfied and we insert the pair (s_1, s_2) to R_{new} (line 30). Similar to the algorithm for FPSs, the invariant throughout the loop is that R is coarser than $\precsim_\mathcal{M}$. Hence, we obtain the preorder $\precsim_\mathcal{M} = R$, once the algorithm terminates (line 32).

Let n denote the number of states, and $m = \sum_{s\in S}\sum_{\alpha\in Act(s)}\sum_{\mu\in Steps\ \ (s)}|\mu|$ denote the number of transitions. We give the complexity of the algorithm:

Lemma 5. Simrel($\mathcal{M}$) *runs in time* $\mathcal{O}(m^2 n)$ *and in space* $\mathcal{O}(m^2)$*. If the fanout of* $\mathcal{M}$ *is bounded by a constant, it has complexity* $\mathcal{O}(n^2)$*, both in time and space.*

Remark 1. For a PA $\mathcal{M} = (S, Act, \mathbf{P}, L)$, let $m_b = \sum_{s\in S}\sum_{\alpha\in Act(s)}|Steps_\alpha(s)|$. The algorithm for deciding strong simulation introduced by Baier *et. al.* has

time complexity $\mathcal{O}((m_b n^6 + m_b^2 n^3)/\log n)$, and space complexity $\mathcal{O}(m_b^2)$. Note that m_b and m are related by $m_b \leq m \leq n m_b$. The left equality is established if $|\mu| = 1$ for all distributions, and the right equality is established if $|\mu| = n$ for all distributions. Note that for sparse models, we have $m = k m_b$ for some $k \in \mathbb{N}$.

Strong Probabilistic Simulations. We now consider finding an algorithm for deciding strong probabilistic simulation. We show that it can be computed by solving LP problems which are decidable in polynomial time [12]. Recall that strong probabilistic simulation is a relaxation of strong simulation in the sense that it enables combined transitions, which are convex combinations of multiple distributions belonging to equally labelled transitions. Again, the most important part is to check the condition $s_1 \precsim_R^p s_2$. By Definition 6, it suffices to check $L(s_1) = L(s_2)$ and the condition:

$$\forall s_1 \xrightarrow{\alpha} \mu_1.\ \exists s_2 \xrightarrow{\alpha}_C \mu_2 \text{ with } \mu_1 \sqsubseteq_R \mu_2 \tag{2}$$

Since the combined transition involves the quantification of the constants c_i ranging over reals, the maximum flow approach for checking $\mu_1 \sqsubseteq_R \mu_2$ cannot be applied directly to check $s_1 \precsim_R^p s_2$. The following lemma shows that condition 2 can be checked by solving LP problems.

Lemma 6. *For a given PA, $s_1 \precsim_R^p s_2$ iff $L(s_1) = L(s_2)$ and for each transition $s_1 \xrightarrow{\alpha} \mu$, the following LP has a solution:*

$$\sum_{i=1}^{k} c_i = 1 \tag{3}$$

$$0 \leq c_i \leq 1\ \forall\ i = 1, \ldots, k \tag{4}$$

$$0 \leq f_{(s,t)} \leq 1 \qquad \forall (s,t) \in R \tag{5}$$

$$\mu(s) = \sum_{t \text{ with } (s,t)\in R} f_{(s,t)} \qquad \forall s \in S \tag{6}$$

$$\sum_{s \text{ with } (s,t)\in R} f_{(s,t)} = \sum_{i=1}^{k} c_i \mu_i(t) \qquad \forall t \in S \tag{7}$$

where $k = |Steps_\alpha(s_2)|$ and $Steps_\alpha(s_2) = \{\mu_1, \ldots, \mu_k\}$.

We introduce a predicate $LP(s_1, \alpha, \mu, s_2)$ which is true iff the above LP problem has a solution. Intuitively, the variables c_i correspond to the constants for the combined transition. Constraints 3 and 4 correspond to the requirements of these constants in Definition 5. For every pair $(s,t) \in R$, a variable $f_{(s,t)}$ ranging over $[0,1]$ (Equation 5) is introduced, whose value corresponds to the value of the weight function for (s,t). Equations 6 and 7 establish the weight function conditions of the strong probabilistic simulation. Any solution of the LP problem induces $c_1, \ldots, c_k$ from which we can construct the desired combined transitions $\mu_c = \sum_{i=1}^{k} c_i \mu_i$ satisfying $\mu \sqsubseteq_R \mu_c$. Assuming that $L(s_1) = L(s_2)$, then, $s_1 \precsim_R^p s_2$ iff the conjunction $\bigwedge_{\alpha \in Act(s_1)} LP(s_1, \alpha, \mu, s_2)$ is true.

SIMREL$^p(\mathcal{M})$
1 $R, R_{new} \leftarrow \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$
2 **do**
3 $R \leftarrow R_{new}$ and $R_{new} \leftarrow \emptyset$
4 **for** $((s_1, s_2) \in R)$
5 **for** $(\alpha \in Act(s_1), \mu_1 \in Steps_\alpha(s_1))$
6 $match_\alpha \leftarrow LP(s_1, \alpha, \mu_1, s_2)$
7 **if** $(\bigwedge_{\alpha \in Act(s_1)} match_\alpha)$ $R_{new} \leftarrow R_{new} \cup \{(s_1, s_2)\}$
8 **until** $R_{new} = R$
9 **return** R

Fig. 3. Algorithm for deciding strong probabilistic simulation for PAs

The algorithm, denoted by SIMREL$^p(\mathcal{M})$, is depicted in Fig. 3. It takes the skeleton of algorithm SIMREL$(\mathcal{M})$. The key difference is that we incorporate the predicate $LP(s_1, \alpha, \mu, s_2)$ in line 6. The correctness of the algorithm SIMREL$^p(\mathcal{M})$ is thus similar to the one of SIMREL$(\mathcal{M})$. We discuss the complexity. The number of variables in the LP problem in Lemma 6 is $k + |R|$, and the number of constraints is $1 + k + |R| + 2|S| \in \mathcal{O}(|R|)$. In every iteration of SIMREL$^p(\mathcal{M})$, for $(s_1, s_2) \in R$ and $s_1 \xrightarrow{\alpha} \mu_1$, this core is queried one time. The number of iterations is bounded by $|R_{init}| \in \mathcal{O}(n^2)$. Therefore, in the worst case, one has to solve $n^2 \sum_{s \in S} \sum_{\alpha \in Act(s)} \sum_{\mu \in Steps\ \ (s)} 1 \leq n^2 m$ many such LP problems and each of them has maximal $\mathcal{O}(n^2)$ constraints.

Strong Simulation Equivalence. For DTMCs, strong simulation equivalence and strong bisimulation coincide [4, Proposition 3.5]. This is not true for PAs, where strong (probabilistic) simulation equivalence is strictly coarser than strong (probabilistic) bisimulation. Since it is coarser, the induced strong (probabilistic) simulation quotient is potentially again smaller. This is not surprising, as PAs subsume labelled transition systems, in which strong simulation equivalence is strictly coarser than strong bisimulation. The presented decision algorithm can be used to obtain strong simulation equivalence $\precsim_{\mathcal{M}} \cap \precsim_{\mathcal{M}}^{-1}$ with complexity $\mathcal{O}(m^2 n)$, and to obtain strong probabilistic simulation equivalence $\precsim_{\mathcal{M}}^p \cap (\precsim_{\mathcal{M}}^p)^{-1}$ via solving LP problems.

This is directly useful for model checking. We discuss briefly which classes of properties are preserved by strong (probabilistic) simulation equivalence. We assume acquaintance with the logic PCTL, particularly the safe and live fragments of PCTL. For more detail, we refer to [4]. Strong (probabilistic) simulation is known to preserve all safe fragments of PCTL [13], which can be lifted to the quotient automaton:

Lemma 7. *For PAs, the strong (probabilistic) simulation quotient automaton preserves both safe and live fragments of PCTL formulas.*

Recall that for MDPs, strong simulation and strong probabilistic simulation trivially coincide. More surprisingly, strong simulation equivalence and strong

bisimulation also coincide on MDPs [1, Theorem 3.4.15]. Therefore, as a byproduct of algorithm SIMREL($\mathcal{M}$), we obtain a decision algorithm for strong bisimulation of MDPs: strong bisimulation for an MDP $\mathcal{M}$ can be obtained by $\precsim_{\mathcal{M}} \cap \precsim_{\mathcal{M}}^{-1}$, with complexity $\mathcal{O}(m^2 n)$.

5 Algorithms for Continuous-Time Probabilistic Automata

In this section we introduce a generalisation of PAs where the transitions are described by rates instead of probabilities. Then, we extend the decision algorithms to continuous-time models. For that purpose we let $r : S \to \mathbb{R}_{\geq 0}$ denote the rate function, and let $Rate(S)$ denote the set of all rate functions.

Definition 7. *A continuous-time PA (CPA) is a tuple* $(S, Act, \mathbf{R}, L)$ *where* S, Act, L *as defined for PAs, and* $\mathbf{R} \subseteq S \times Act \times Rate(S)$ *a finite set, called the rate matrix.*

We write $s \xrightarrow{\alpha} r$ if $(s, \alpha, r) \in \mathbf{R}$. The model continuous-time Markov decision processes (CTMDPs) [3, 11] can be considered as special CPAs where for $s \in S$ and $\alpha \in Act$, there exists at most one rate function $r \in Rate(S)$ such that $s \xrightarrow{\alpha} r$. The model CTMDPs considered in paper [14] essentially agrees with our CPAs. Note that CPAs generalise PAs in a similar way as CTMDPs generalise MDPs.

Strong Simulations. For a rate function r with $r(S) > 0$, we let $\mu(r) \in Dist(S)$ denote the induced distribution defined by: $\mu(r)(s)$ equals $r(s)/r(S)$ if $r(S) > 0$ and 0 otherwise. Now we introduce the notion of strong simulation for CPAs, which can considered as an extension of the definition for CTMCs [4]:

Definition 8. *Let* $\mathcal{M} = (S, Act, \mathbf{R}, L)$ *be a CPA.* $R \subseteq S \times S$ *is a* strong simulation *on* $\mathcal{M}$ *iff for all* s_1, s_2 *with* $s_1 \; R \; s_2$*:* $L(s_1) = L(s_2)$ *and if* $s_1 \xrightarrow{\alpha} r_1$ *then there exists a transition* $s_2 \xrightarrow{\alpha} r_2$ *with* $\mu(r_1) \sqsubseteq_R \mu(r_2)$ *and* $r_1(S) \leq r_2(S)$*. We write* $s_1 \precsim_{\mathcal{M}} s_2$ *iff there exists a strong simulation* R *on* $\mathcal{M}$ *such that* $s_1 \; R \; s_2$*.*

The additional rate condition $r_1(S) \leq r_2(S)$ indicates that r_2 is faster than r_1. The decision algorithm for strong simulation can be adapted from algorithm SIMREL in Fig. 2 easily: We replace every occurrence of μ_1 and μ_2 by r_1 and r_2, respectively. Other notations are extended with respect to rate functions in an obvious way. To guarantee the additional rate condition, we rule out transitions that violate it by replacing line 4 by: $Sim_{(s_1,\alpha,r_1)}(s_2) \leftarrow \{r_2 \in Steps_\alpha(s_2) \mid r_1(S) \leq r_2(S)\}$. Obviously, the so obtained algorithm for CPAs has the same complexity $\mathcal{O}(m^2 n)$ as for PAs.

Strong Probabilistic Simulations. We extend the notion of strong probabilistic simulation to CPAs. Firstly, we introduce combined transitions for CPAs:

Definition 9. *Let* $\mathcal{M} = (S, Act, \mathbf{R}, L)$ *be a CPA. Assume that* $\{r_1, \ldots, r_k\} \subseteq Steps_\alpha(s)$ *where* $r_i(S) = r_j(S)$ *for* $i, j \in \{1, \ldots, k\}$*. The tuple* (s, α, r) *is a*

combined transition, denoted by $s \xrightarrow{\alpha}_C r$, *iff there exist constants* $c_1, \ldots, c_k \in [0,1]$ *with* $\sum_{i=1}^{k} c_i = 1$ *such that* $r(s) = \sum_{i=1}^{k} c_i r_i(s)$ *for all* $s \in S$.

In the above definition, unlike for the PA case, only α-successor rate functions with same exit rate can be combined together. This restriction makes the combined transition also exponential distributed. Without this restriction, the combined transition is not exponential distributed any more, precisely, it is hyper-exponential distributed. To see this we consider $S = \{s_1, s_2, s_3\}$, and $s_1 \xrightarrow{\alpha} r_1$ and $s_1 \xrightarrow{\alpha} r_2$. The rate function r_1 is defined by $r_1(s_2) = 2, r_1(s_3) = 8$, and r_2 is defined by $r_2(s_2) = 12, r_2(s_3) = 6$. Taking r_1 and r_2 with equal probability 0.5, we would get the combined transition $r = 0.5r_1 + 0.5r_2$ satisfying: $r(s_2) = 7$ and $r(s_3) = 7$. If r is exponential distributed, we would expect that the probability of reaching state s_2 within time t should be $\frac{7}{14} * (1 - \exp^{-14t})$. However, the combined transition is hyper-exponential distributed: the probability of reaching state s_2 within time t under r is given by:

$$0.5 * \frac{2}{10} * (1 - \exp^{-10t}) + 0.5 * \frac{12}{18} * (1 - \exp^{-18t})$$

Similarly, the probability of reaching state s_3 within time t is given by: $0.5 * \frac{8}{10} * (1 - \exp^{-10t}) + 0.5 * \frac{6}{18} * (1 - \exp^{-18t})$.

Similar to PAs, strong probabilistic simulation is insensitive to combined transitions, is thus a relaxation of strong simulation:

Definition 10. *Let* $\mathcal{M} = (S, Act, \mathbf{R}, L)$ *be a CPA.* $R \subseteq S \times S$ *is a* strong probabilistic simulation *on* $\mathcal{M}$ *iff for all* s_1, s_2 *with* $s_1 \; R \; s_2$: $L(s_1) = L(s_2)$ *and if* $s_1 \xrightarrow{\alpha} r_1$ *then there exists a combined transition* $s_2 \xrightarrow{\alpha}_C r_2$ *with* $\mu(r_1) \sqsubseteq_R \mu(r_2)$ *and* $r_1(S) \leq r_2(S)$. *We write* $s_1 \precsim_{\mathcal{M}}^{p} s_2$ *iff there exists a strong simulation* R *on* $\mathcal{M}$ *such that* $s_1 \; R \; s_2$.

The notation of simulation up to R for strong probabilistic simulation can be defined in a similar way as for PAs. To check the condition $s_1 \precsim_R^p s_2$ for the CPA $\mathcal{M}$ we resort to a reduction to LP problems:

Lemma 8. *For a given CPA,* $s_1 \precsim_R^p s_2$ *iff* $L(s_1) = L(s_2)$ *and for each transition* $s_1 \xrightarrow{\alpha} r$ *there exists* $\{r_1, \ldots, r_k\} \subseteq Steps_\alpha(s_2)$ *with* $r_i(S) = r_j(S)$ *and* $r_i(S) \geq r(S)$ *for* $i, j \in \{1, \ldots, k\}$ *such that the following LP has a solution, which consists of constraints 3, 4, 5 of Lemma 6, and additionally:*

$$r(s) = r(S) \sum_{t \text{ with } (s,t) \in R} f_{(s,t)} \qquad \forall s \in S \tag{8}$$

$$r_1(S) \sum_{s \text{ with } (s,t) \in R} f_{(s,t)} = \sum_{i=1}^{k} c_i r_i(t) \qquad \forall t \in S \tag{9}$$

$$r(S) \leq r_1(S) \tag{10}$$

Similar to Lemma 6, for every $E \in \{r^*(S) \mid s_2 \xrightarrow{\alpha} r^*\}$ with $E > r_1(S)$ we introduce the predicate $LP'(s_1, \alpha, r, s_2, E)$ which is true iff the above LP problem has a solution. In comparison to the LP problem in Lemma 6, Equation 10

establishes the rate condition. Recall that $\mu(r)(s) = r(s)/r(S)$. Equation 8 corresponds to Equation 6 where $r(S)$ is multiplied on both sides. Now we consider Equation 9. Let $r_c(t) = \sum_{i=1}^{k} c_i r_i(t)$ be the combined transition that simulates r. Note $\mu(r_c)(t) = \sum_{i=1}^{k} c_i r_i(t) / \sum_{i=1}^{k} c_i r_i(S)$. Since we have $r_i(S) = r_j(S)$ for all $i, j \in \{1, \ldots, k\}$, the denominator is simplified to $r_1(S)$. Hence, Equation 9 corresponds to Equation 7 where $r_1(S)$ is multiplied on both side. Note that the denominator could not be simplified without the restriction in the combined transition, i. e., only rate function with the same exit rate can be combined. Equation 9 would not be a linear constraint any more for this case. Obviously, the LP problem has a solution iff there is such a combined transition r_c satisfying the conditions $\mu(r) \sqsubseteq_R \mu(r_c)$ and $r(S) \leq r_c(S)$. Assuming that $L(s_1) = L(s_2)$, then, $s_1 \precsim_R^p s_2$ iff the conjunction $\bigwedge_{\alpha \in Act(s_1)} LP'(s_1, \alpha, r, s_2, E)$ is true for an $E \in \{r^*(S) \mid s_2 \xrightarrow{\alpha} r^* \wedge r^*(S) \geq r_1(S)\}$. The decision algorithm can be obtained by replacing the predicate $LP(s_1, \alpha, \mu, s_2)$ by $LP'(s_1, \alpha, r, s_2, E)$ of algorithm $\text{SimRel}^p(\mathcal{M})$ in Fig. 3 where E ranges over $\{r^*(S) \mid s_2 \xrightarrow{\alpha} r^* \wedge r^*(S) \geq r_1(S)\}$. As complexity we have to solve $n^2 m$ LP problems and each of them has maximal $\mathcal{O}(n^2)$ constraints.

Strong Simulation Equivalence. Similar to PAs, as a byproduct we obtain a decision algorithm for strong (probabilistic) simulation equivalence for CPAs. We discuss also which classes of properties are preserved by strong (probabilistic) simulation equivalence. We assume acquaintance with the logic CSL, particularly the safe and live fragments of CSL. For more detail, we refer to [4]. Now we give the continuous-time counterpart of Lemma 7 for CPAs:

Lemma 9. *For CPAs, the strong (probabilistic) simulation preserves all safe fragments of CSL formulas. Moreover, the strong (probabilistic) simulation quotient automaton preserves both safe and live fragments of CSL formulas.*

For CTMDPs, strong simulation and strong probabilistic simulation also trivially coincide, as for MDPs. Another similar result is that the strong simulation quotient and the strong bisimulation for CTMDPs also coincide.

6 Conclusion

We presented algorithms for computing simulation preorders for PAs. We achieved an algorithm with complexity $\mathcal{O}(m^2 n)$ for strong simulation. For strong probabilistic simulation, we have shown that the preorder can be determined by solving LP problems. We extended the algorithms to CPAs with same complexities for both strong simulation and strong probabilistic simulation. As further work, we would like to extend our results to weak (probabilistic) simulations for PAs and CPAs.

Acknowledgement. We thank Martin Neuhäußer for pointing out an error in the definition of combined transitions for CPAs in an earlier version of this paper.

References

1. Baier, C.: On Algorithmic Verification Methods for Probabilistic Systems, Habilitations-schrift zur Erlangung der venia legendi der Fakultät für Mathematik and Informatik, Universität Mannheim (1998)
2. Baier, C., Engelen, B., Majster-Cederbaum, M.E.: Deciding bisimilarity and similarity for probabilistic processes. J. Comput. Syst. Sci. 60(1), 187–231 (2000)
3. Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time markov decision processes. Theor. Comput. Sci. 345(1), 2–26 (2005)
4. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for markov chains. Inf. Comput 200(2), 149–214 (2005)
5. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004)
6. Cattani, S., Segala, R.: Decision algorithms for probabilistic bisimulation. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 371–385. Springer, Heidelberg (2002)
7. Gallo, G., Grigoriadis, M.D., Tarjan, R.E.: A fast parametric maximum flow algorithm and applications. SIAM J. Comput. 18(1), 30–55 (1989)
8. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM 35(4), 921–940 (1988)
9. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. LICS, pp. 266–277 (1991)
10. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94(1), 1–28 (1991)
11. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Chichester (1994)
12. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
13. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nord. J. Comput. 2(2), 250–273 (1995)
14. Wolovick, N., Johr, S.: A characterization of meaningful schedulers for continuous-time markov decision processes. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 352–367. Springer, Heidelberg (2006)
15. Zhang, L., Hermanns, H., Eisenbrand, F., Jansen, D.N.: Flow faster: Efficient decision algorithms for probabilistic simulations. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 155–169. Springer, Heidelberg (2007)

Mechanizing the Powerset Construction for Restricted Classes of ω-Automata*

Christian Dax[1], Jochen Eisinger[2], and Felix Klaedtke[1]

[1] ETH Zurich, Switzerland
[2] Albert-Ludwigs-Universität Freiburg, Germany

Abstract. Automata over infinite words provide a powerful framework to solve various decision problems. However, the mechanized reasoning with restricted classes of automata over infinite words is often simpler and more efficient. For instance, weak deterministic Büchi automata (WDBAs) can be handled algorithmically almost as efficient as deterministic automata over finite words. In this paper, we show how and when the standard powerset construction for automata over finite words can be used to determinize automata over infinite words. An instance is the class of automata that accept WDBA-recognizable languages. Furthermore, we present applications of this new determinization construction. Namely, we apply it to improve the automata-based approach for the mixed first-order linear arithmetic over the reals and the integers, and we utilize it to accelerate finite state model checking. We report on experimental results for these two applications.

1 Introduction

Automata over infinite objects have emerged as a powerful tool for specification and verification of nonterminating programs [23,32], and for implementation of decision procedures for logical theories [2,4,9,18]. For instance, the automata-theoretic approach to model checking is easy to understand, automatic, and thus attractive to practitioners. However, its effectiveness is often sensitive to the automaton model and the sizes of the automata.

In [5], it is remarked that many specifications in model checking describe languages that can be recognized by restricted classes of automata. Reasoning about or with restricted classes of automata over infinite words is often simpler and more efficient. A prominent example are weak deterministic Büchi automata (WDBAs), which can be handled algorithmically almost as efficient as deterministic automata over finite words. For instance, in contrast to Büchi automata, WDBAs have a canonical minimal form, which can be obtained efficiently [25]. WDBAs can be used to represent and manipulate sets definable in the mixed first-order logic over the reals and the integers with addition and the ordering, i.e., $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, <)$ [4]. Such an automata-based representation of $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, <)$-definable sets has applications in infinite-state model checking (see, e.g., [3,9]).

* This work was supported by the German Research Council (DFG) and the Swiss National Science Foundation (SNF).

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 223–236, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Further, languages that describe temporal properties like safety and guarantee properties and boolean combinations thereof, so-called obligation properties, can be recognized by WDBAs (see [6]).

However, it is not obvious how we can benefit from the algorithms for WDBAs if a given automaton is, e.g., a nondeterministic Muller automaton that accepts a WDBA-recognizable language. In [19], Kupferman et al. observed that the standard powerset construction for automata over finite words can be used to obtain an equivalent WDBA from a given automaton when it accepts a WDBA-recognizable language. However, no concrete algorithm is given. In particular, the crucial point how to efficiently determine the accepting states of the WDBA is not addressed.

In this paper, we provide an efficient algorithm to determine the accepting states of the WDBA obtained by the standard powerset construction for automata over finite words. Furthermore, we give a sufficient condition for automata for which we can use the powerset construction to obtain equivalent deterministic Büchi automata. For such automata, we provide a general determinization construction. We also present a method to check whether this new determinization construction can be applied. Finally, we propose how to use the new constructions in relevant applications. We evaluate our approaches experimentally.

One of the applications is the construction of automata-based representations for sets definable in $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, <)$. Our new construction handles quantifiers more efficiently than previously proposed constructions as, e.g., in [4]. Another application for our determinization constructions discussed in this paper is finite state model checking. Whenever the specification is an obligation property, we suggest to construct the minimal WDBA. The advantage of using the minimal WDBA is that it contains no redundant states and no nondeterminism that might lead to a more expensive verification process. In [29], Sebastiani and Tonetta suggest an approach with a similar flavor to optimize the verification process. Instead of constructing the minimal WDBA, they apply heuristics to reduce nondeterminism in the transition function of the Büchi automaton for the specification. For both applications, our evaluations show an improvement in the state of the art in the respective area.

We proceed as follows. In §2, we recall background. In §3, we show how and when we can use the powerset construction for automata over infinite words. In §4, we give applications and experimental results of the new determinization constructions. Finally, in §5, we draw conclusions.

2 Background

We assume that the reader is familiar with the basics of automata theory. The purpose of this section is to recall background in this area, and fix some of the notation and terminology that we use in the remainder of the text.

Let Σ be an alphabet. We denote the set of all finite words over Σ by Σ^*. We define $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$, where ε is the empty word. Σ^ω is the set of all infinite words over Σ. We often write a word $w \in \Sigma^*$ of length $\ell \geq 0$ as $w_0 \dots w_{\ell-1}$

and $\alpha \in \Sigma^\omega$ as $\alpha_0\alpha_1\ldots$, where w_i and α_i denote the ith letter of w and α, respectively. We denote the infinite repetition of a finite word $u \in \Sigma^+$ by u^ω.

A *transition system* (TS) T is a tuple $(Q, \Sigma, \delta, q_\mathrm{I})$, where Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function, and $q_\mathrm{I} \in Q$ is the initial state. We extend δ to the function $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ defined as $\hat{\delta}(q, \varepsilon) := \{q\}$ and $\hat{\delta}(q, bu) := \bigcup_{p\in\delta(q,b)} \hat{\delta}(p, u)$, where $q \in Q$, $b \in \Sigma$, and $u \in \Sigma^*$. T is *deterministic* if $|\delta(p, b)| = 1$, for all $p \in Q$ and $b \in \Sigma$. In this case, we write $\delta(p, b) = q$ and $\hat{\delta}(p, w) = q$ instead of $\delta(p, b) = \{q\}$ and $\hat{\delta}(p, w) = \{q\}$, respectively.

For $L \subseteq \Sigma^\omega$, we define the congruence relation $\approx_L \subseteq \Sigma^* \times \Sigma^*$ as $u \approx_L v$ iff $u\alpha \in L \Leftrightarrow v\alpha \in L$, for all $\alpha \in \Sigma^\omega$. If $\approx_L$ has finite index, we define the deterministic TS $\mathcal{C}_L$ as $\mathcal{C}_L := (\{[v] : v \in \Sigma^*\}, \Sigma, \delta, [\varepsilon])$ with $\delta([v], b) := [vb]$, where $[u]$ denotes the equivalence class of $u \in \Sigma^*$, i.e., $[u] := \{v \in \Sigma^* : v \approx_L u\}$. Note that δ is well-defined.

In the following, let $T = (Q, \Sigma, \delta, q_\mathrm{I})$ be a TS. A state $q \in Q$ is *reachable* from $p \in Q$ if there is a word $w \in \Sigma^*$ such that $q \in \hat{\delta}(p, w)$. In the remainder of the text, we assume that every state in a TS is reachable from its initial state. A *strongly connected component* (SCC) of T is a set $S \subseteq Q$ such that every $p \in S$ is reachable from every $q \in S$ and S is maximal. A *loop* in T is a word $q_0 \ldots q_n \in Q^*$ with $n \geq 1$, $q_0 = q_n$, and for all $i \in \{0, \ldots, n-1\}$, there is a letter $b \in \Sigma$ such that $q_{i+1} \in \delta(q_i, b)$. A *run* of T on $\alpha \in \Sigma^\omega$ is a word $\varrho \in Q^\omega$ such that $\varrho_0 = q_\mathrm{I}$ and $\varrho_{i+1} \in \delta(\varrho_i, \alpha_i)$, for all $i \geq 0$. $\mathrm{Inf}(\varrho)$ is the set of states that occur infinitely often in ϱ.

An *automaton* $\mathcal{A}$ is a tuple (T, C), where T is a TS and C is an acceptance condition. In the following, we mainly use the Büchi and co-Büchi conditions, which are defined as follows.

– $S \subseteq Q$ satisfies the *Büchi condition* $C \subseteq Q$ if $S \cap C \neq \emptyset$.
– $S \subseteq Q$ satisfies the *co-Büchi condition* $C \subseteq Q$ if $S \cap C = \emptyset$.

Due to space limitations, we do not give the definition of the other common acceptance conditions like Muller, Rabin, and Streett condition. Instead, we refer the reader to [31]. A run ϱ is *accepting* if $\mathrm{Inf}(\varrho)$ satisfies the acceptance condition C; it is *rejecting*, otherwise. We define $L(\mathcal{A}) := \{\alpha \in \Sigma^\omega :$ there is an accepting run of $\mathcal{A}$'s TS on $\alpha\}$.

We type an automaton $\mathcal{A} = (T, C)$ according to its acceptance condition C. For instance, if C is the Büchi condition, $\mathcal{A}$ is a *Büchi automaton* (BA) and if C is the co-Büchi condition, we call $\mathcal{A}$ a *co-Büchi automaton* (co-BA). If T is *deterministic*, $\mathcal{A}$ is a *deterministic* BA (DBA) or *deterministic co-*BA (co-DBAs), respectively. A BA (T, C) is *weak* if $S \cap C = \emptyset$ or $S \subseteq C$, for every SCC $S \subseteq Q$. We use the initialisms WBA for "weak Büchi automaton" and WDBA for "weak deterministic Büchi automaton."

WDBA denotes the class of languages L for which there is a WDBA $\mathcal{A}$ with $L(\mathcal{A}) = L$. The classes of languages DBA and coDBA are defined as expected. There are different characterizations of these classes of languages and the relation between the classes has been investigated intensively. For example, it holds that $\mathsf{DBA} \cap \mathsf{coDBA} = \mathsf{WDBA}$. For details, we refer the reader to [6].

3 Determinization with the Powerset Construction

In this section, we investigate when and how we can use the powerset construction to determinize automata over infinite words. The *powerset transition system* of a TS $T = (Q, \Sigma, \delta, q_{\mathrm{I}})$ is $\mathcal{P}(T) := (\mathcal{P}(Q), \Sigma, \eta, \{q_{\mathrm{I}}\})$ with $\eta(R, b) := \bigcup_{q \in R} \delta(q, b)$, for $R \subseteq Q$ and $b \in \Sigma$. Let CONG be the class of languages L for which the DBA $(\mathcal{C}_L, E)$ accepts L, for some set E.

Lemma 1. *Let $\mathcal{A} = (T, C)$ be an automaton. If $L(\mathcal{A}) \in \mathsf{CONG}$ then there is a set F such that the* DBA *$(\mathcal{P}(T), F)$ accepts $L(\mathcal{A})$.*

Proof. Assume that $T = (Q, \Sigma, \delta, q_{\mathrm{I}})$ and that the DBA $(\mathcal{C}_{L(\mathcal{A})}, E)$ accepts $L(\mathcal{A})$. Define $F := \{P \subseteq Q : \hat{\delta}(q_{\mathrm{I}}, u) = P \text{ and } [u] \in E, \text{ for some } u \in \Sigma^*\}$. For $\alpha \in \Sigma^\omega$, let ϱ be the run of $\mathcal{C}_{L(\mathcal{A})}$ and ϱ' be the run of $\mathcal{P}(T)$. We show that $\varrho_i \in E$ iff $\varrho'_i \in F$, for all $i \geq 0$. Let $v := \alpha_0 \dots \alpha_{i-1}$. Note that $\varrho_i = [v]$. The direction from left to right holds by the definition of F. For the other direction, assume that $\varrho'_i \in F$, i.e., there is a word $u \in \Sigma^*$ with $\hat{\delta}(q_{\mathrm{I}}, u) = \varrho'_i$ and $[u] \in E$. Since $\varrho'_i = \hat{\delta}(q_{\mathrm{I}}, u) = \hat{\delta}(q_{\mathrm{I}}, v)$, we have that $u \approx_{L(\mathcal{A})} v$ and hence, $[u] = [v] = \varrho_i$. □

Note that Lemma 1 establishes the existence of the Büchi acceptance condition F for the TS $\mathcal{P}(T)$. It is left open how to algorithmically determine the set F. A naive algorithm checks whether it holds that the DBA $(\mathcal{P}(T), F)$ accepts $L(\mathcal{A})$, for each $F \subseteq \mathcal{P}(Q)$. In §3.1 and §3.2, we present more sophisticated algorithms to determine such a set F. For certain language classes, our algorithms have an exponentially better worst-case complexity than the sketched naive algorithm. With such algorithms at hand, we obtain new automata constructions for determinizing automata whenever they accept languages in CONG or subclasses thereof. We give concrete applications of these constructions in §4. Before we present the algorithms and their applications, we look in more detail at the languages in CONG and at the automata that accept languages in CONG.

First, we remark that the converse direction of Lemma 1 does not hold in general. To see this, let L be the language $\{\alpha \in \{0,1\}^\omega : 1 \text{ occurs infinitely often in } \alpha\}$. Since $\approx_L$ has only one equivalence class, it is straightforward to see that $L \notin \mathsf{CONG}$. However, there is a DBA $\mathcal{A} = (T, C)$ that accepts L and since T is deterministic, there is obviously a set F such that the DBA $(\mathcal{P}(T), F)$ accepts L. Second, we observe that $\mathsf{CONG} \subsetneq \mathsf{DBA}$. By definition, every language in CONG can be accepted by some DBA. As we have seen above, the DBA $\mathcal{A}$ accepts a language not in CONG.

Further, note that for a language $L \in \mathsf{WDBA}$, there is some DBA $(\mathcal{C}_{L(\mathcal{A})}, E)$ that accepts L [26]. Hence, CONG subsumes important classes of ω-regular languages. For instance, the ω-regular languages that describe boolean combinations of safety and guarantee properties are in CONG (see, e.g., [6]). Moreover, CONG contains the languages that are definable in the mixed first-order logic over the integers and the reals with addition and the ordering [4]. Unfortunately, checking whether an automaton accepts a language in CONG is PSPACE-hard. This can be shown by a similar argumentation as in the proof of Theorem 4.2 in [20].

Finally, note that for a language $L \subseteq \Sigma^\omega$, the minimal number of states of a deterministic automaton $\mathcal{A}$ with $L(\mathcal{A}) = L$ is at least the index of the congruence relation $\approx_L$. In the case where $L \in \mathsf{CONG}$, the minimal number of states of a deterministic automaton $\mathcal{A}$ that accepts L is the index of $\approx_L$. From Lemma 1, it follows that for $\mathcal{A}$'s TS T there exists a set F of states such that the DBA (T, F) accepts L. Note that the powerset transition system of T is isomorphic to T when we remove the states that are not reachable from its initial state. Similarly, as remarked in the paragraph after Lemma 1, it is left open how to determine the set F of accepting states algorithmically from the automaton $\mathcal{A}$. The algorithms presented in the following subsections can be used to solve this problem for converting the acceptance condition to a Büchi acceptance condition.

3.1 Determinization of Automata with Languages in WDBA

We first consider the special case, where we assume that the automaton $\mathcal{A}$ accepts a language in WDBA. Assume that $\mathcal{A}$ is the automaton (T, C) with $T = (Q, \Sigma, \delta, q_\mathrm{I})$ and that $\mathcal{P}(T) = (\mathcal{P}(Q), \Sigma, \eta, \{q_\mathrm{I}\})$. Before we present the automata construction to determinize $\mathcal{A}$, we make the following observations. From [26], we know that some DBA $(\mathcal{C}_{L(\mathcal{A})}, E)$ accepts $L(\mathcal{A})$. It follows from Lemma 1 that for some $F \subseteq \mathcal{P}(Q)$, the DBA $(\mathcal{P}(T), F)$ accepts $L(\mathcal{A})$. According to Theorem 5.2 in [4], $(\mathcal{P}(T), F)$ is inherently weak, i.e., there is no SCC S of $\mathcal{P}(T)$ with an accepting and a rejecting loop. Here, we call a loop $Q_0 \dots Q_n \in \mathcal{P}(Q)^+$ *accepting* if $Q_i \in F$, for some $i \in \{0, \dots, n-1\}$, and *rejecting*, otherwise.

Lemma 2. *Let $R \in \mathcal{P}(Q)$, $u \in \Sigma^*$ such that $\hat{\eta}(\{q_\mathrm{I}\}, u) = R$, and $w \in \Sigma^+$ such that $\hat{\eta}(R, w) = R$. It holds that $uw^\omega \in L(\mathcal{A})$ iff all loops of the* SCC *that contains R are accepting.*

Proof. ($\Rightarrow$) If $uw^\omega \in L(\mathcal{A})$ then $uw^\omega \in L(\mathcal{P}(T), F)$. Since $(\mathcal{P}(T), F)$ is inherently weak and R occurs infinitely often in the run of $\mathcal{P}(T)$ on uw^ω, all loops of the SCC that contains R are accepting.

($\Leftarrow$) If all loops of the SCC that contains R are accepting then $uw^\omega \in L(\mathcal{P}(T), F)$. Since $L(\mathcal{P}(T), F) = L(\mathcal{A})$, we have that $uw^\omega \in L(\mathcal{A})$. □

The determinization of $\mathcal{A}$ comprises two steps.[1] First, we construct $\mathcal{P}(T)$. Second, we use the algorithm in Figure 1 to compute a set $F' \subseteq \mathcal{P}(Q)$, where F' is the union of the SCCs for which the algorithm returns "accepting." In the algorithm the words u and w can be found, e.g., by a breadth-first search. Note that $uw^\omega \in L(\mathcal{A})$ is equivalent to $\{uw^\omega\} \cap L(\mathcal{A}) = \emptyset$. We can construct an automaton that accepts $\{uw^\omega\} \cap L(\mathcal{A})$ and check its emptiness according to $\mathcal{A}$'s acceptance condition. See [7,14,17], for several efficient emptiness checks with respect to the automaton's acceptance condition.

[1] In [19], it is stated that for a BA $\mathcal{B} = (U, G)$ that accepts a language in WDBA, the Büchi condition for $\mathcal{P}(U)$ can be chosen as $\{P : P \cap G \neq \emptyset\}$. A counterexample for this claim is the TS $(\{r, s, t\}, \{0\}, \delta, r)$ with $\delta(r, 0) = \{r, s\}$ and $\delta(s, 0) = \delta(t, 0) = \{t\}$ and the Büchi condition $\{s\}$.

1: **if** S has no loop **then return** rejecting
2: Let R be some state in S.
3: Let $u \in \Sigma^*$ a word such that $\hat{\eta}(\{q_\mathrm{I}\}, u) = R$.
4: Let $w \in \Sigma^+$ a word such that $\hat{\eta}(R, w) = R$.
5: **if** $uw^\omega \in L(\mathcal{A})$ **then return** accepting **else return** rejecting

Fig. 1. Algorithm to determine whether an SCC S of $\mathcal{P}(T)$ is accepting or rejecting

The correctness of this construction can be seen as follows. Note that for an SCC S without a loop it is irrelevant whether its states belong to F' or not. The language of the automaton is not altered, since these states can only occur at most once in a run. We make them rejecting. Otherwise, let S be an SCC with at least one loop. From Lemma 2, it follows that the algorithm in Figure 1 returns "accepting" for S iff all loops of S are accepting. It follows that $L(\mathcal{P}(T), F) = L(\mathcal{P}(T), F')$.

We remark that the constructed automaton is weak. Further, the construction is parametric in the type of the acceptance condition of the automaton $\mathcal{A}$. We obtain translations to WDBAs for automata with acceptance conditions such as parity, Rabin, Streett, and Muller.

In summary, the construction described in this subsection establishes the following theorem.

Theorem 3. *Let $\mathcal{A}$ be an automaton with n states. If $L(\mathcal{A}) \in \mathsf{WDBA}$ then we can construct a* WDBA *with at most 2^n states that accepts $L(\mathcal{A})$.*

3.2 The General Case

In this subsection, we consider the general case, where we are given an automaton $\mathcal{A}$ with $L(\mathcal{A}) \in \mathsf{CONG}$. We do not require that $\mathcal{A}$ accepts a language in WDBA as in the previous subsection. From Lemma 1, we know that there is a set F such that the DBA $(\mathcal{P}(T), F)$ accepts the language of $\mathcal{A}$, where T is the TS of $\mathcal{A}$. So, as in §3.1, we are left with the problem to determine algorithmically a set F' such that the DBA $(\mathcal{P}(T), F')$ accepts $L(\mathcal{P}(T), F)$. In fact, the algorithm that we present in the following solves a more general problem. The input of the algorithm consists of an automaton $\mathcal{B}$ and a deterministic TS U. The algorithm requires that there is at least one set F such that the DBA (U, F) accepts $L(\mathcal{B})$. It outputs a set F' such that the DBA (U, F') accepts $L(\mathcal{B})$. Assume that $U = (P, \Sigma, \eta, p_\mathrm{I})$.

Observe that we can consider each SCC of U separately, i.e., for each SCC S, we can compute a set $F_S \subseteq P$ without taking into account the states of U in the other SCCs of U. Note that such a set F_S is not uniquely determined and there might be dependencies on the states in S that we have to take care of. The algorithm in Figure 2 returns such a set F_S, for an SCC S of U. F' is then the union of the sets F_S, for all SCCs S of U.

Due to space limitations we only sketch the algorithm. We iteratively investigate loops π in the SCC S from which we gain additional information about

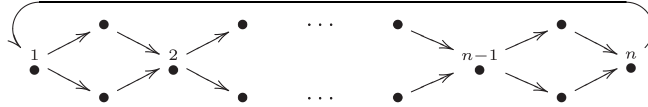
1: $R \leftarrow \emptyset$
2: $A \leftarrow \emptyset$
3: Let G be the graph (V, E) with $V := S$ and $E := \{(p, q) : \eta(p, b) = q$, for some $b \in \Sigma\}$.
4: **while** there is a loop $\pi = v_0 \dots v_\ell$ in G with $\ell \leq |S|$ and $v_0 \in V \setminus R$ and there is no $X \in A$ such that $X \subseteq \{v_0, \dots, v_{\ell-1}\}$ **do**
5: Let $u \in \Sigma^*$ be a word with $\hat{\eta}(q_{\mathrm{I}}, u) = v_0$.
6: Let $w \in \Sigma^+$ be a word of length ℓ with $\eta(v_i, w_i) = v_{i+1}$, for all $0 \leq i < \ell$.
7: **if** $uw^\omega \notin L(\mathcal{B})$ **then**
8: $R \leftarrow R \cup \{v_0, \dots, v_\ell\}$
9: Update A, i.e., remove the v_is in every $X \in A$.
10: **else**
11: $A \leftarrow A \cup \big\{\{v_i : 0 \leq i \leq \ell \text{ and } v_i \notin R\}\big\}$
12: **end if**
13: **while** there is a vertex $v \in V$ with $\{v\} \in A$ **do**
14: Delete vertex v from G.
15: Update A, i.e., remove $X \in A$ whenever $v \in X$.
16: **end while**
17: **end while**
18: **return** $S \setminus R$

Fig. 2. Algorithm to determine the set of accepting states for an SCC S of T'

which of the states in S have to be accepting and which have to be rejecting. For a loop $\pi = p_0 \dots p_\ell$, there is a word $w \in \Sigma^+$ that visits the states in π in the same order. Moreover, there is a word $u \in \Sigma^*$ with $\hat{\eta}(q_{\mathrm{I}}, u) = p_0$. We check if $uw^\omega \in L(\mathcal{B})$. If this is not the case, we know that the states $p_0, \dots, p_{\ell-1}$ must not be in F_S. If $uw^\omega \in L(\mathcal{B})$, we know that at least one of the states $p_0, \dots, p_{\ell-1}$ has to be in F_S. The algorithm maintains a set R, where R contains the states that must not be in F_S, and it maintains a set A of sets of states, where $X \in A$ means that at least one of the states in X has to be in F_S. Initially, R and A are empty. If we derive the fact that a state $p \in S$ has to be rejecting, we put p in R and delete p in every $X \in A$. If A contains a singleton $\{q\}$, we know that the state $q \in S$ has to be accepting and we remove the sets X from A that contain q.

The algorithm also maintains a graph G. Intuitively speaking, G together with the set A describe the loops of the SCC S that we still need to investigate. Initially, G is the transition graph of the SCC S. Note that we need not to investigate loops in G that visit a state for which we already know that it has to be in F_S. Thus, as soon as we conclude that a state p is accepting, we delete p in G (and all its in-going and out-going edges). That means, that no loop in the updated graph will visit p. Further, a loop π has to visit at least one state for which we do not know whether it is accepting or rejecting. Without loss of generality, we assume that π_0 is such a state. Moreover, we can restrict ourselves to loops π for which the set of visited states is not a superset of any $X \in A$. The reason for this is that at least one state in X has to be accepting and thus, $xy^\omega \in L(\mathcal{B})$, where $x \in \Sigma^*$ is a word from p_{I} to the state π_0 and $y \in \Sigma^+$ is a word corresponding to the loop π. Therefore, we do not obtain any new information by investigating π. Finally, note that it suffices to check loops of length at most $|S| + 1$.

The algorithm in Figure 2 terminates since it only checks finitely many loops. In the worst case, it checks exponentially many loops: Assume that the given deterministic TS U has the graph



and state 1 is the initial state. This graph has 2^{n-1} loops of length $2n$ that start in state 1. If the infinite repetition of the words corresponding to these loops are in $L(\mathcal{B})$, the algorithm checks exponentially many loops. We remark that from smaller loops we can obtain more information. In particular, from a self-loop we immediately see if the state in the self-loop has to be accepting or rejecting. So, a heuristic is to check loops ordered increasingly by their lengths.

Finally, note that the algorithm in Figure 2 can be easily adapted such that we can use it to obtain a set $F' \subseteq P$ for the co-Büchi condition, i.e., that the co-DBA (U, F') accepts $L(\mathcal{B})$.

3.3 Remarks on the Precondition of the Algorithm

In this subsection, we want to comment on the requirement of the algorithm in §3.2, i.e., the existence of a set F such that the DBA (U, F) accepts $L(\mathcal{B})$. If we do not know whether such a set F exists, we can proceed as follows. We use the algorithm presented in §3.2 to obtain a set F' of states of the TS U and check whether the DBA (U, F') accepts $L(\mathcal{B})$. Note that this check can be done by checking $L(U, F') \subseteq L(\mathcal{B})$ and $L(\mathcal{B}) \subseteq L(U, F')$, or equivalently, $(\Sigma^\omega \setminus L(U, F')) \cap L(\mathcal{B}) = \emptyset$ and $(\Sigma^\omega \setminus L(\mathcal{B})) \cap L(U, F') = \emptyset$, respectively. The first check can be done in polynomial time. Note that DBAs can be complemented in polynomial time [22]. However, the second check is expensive, since we have to complement $\mathcal{B}$ (e.g., by using the construction in [21] when $\mathcal{B}$ is a BA), which can lead to an exponential blowup.

Note that the decision problem of determining the existence of a set of states F such that the DBA (U, F) accepts $L(\mathcal{B})$, for an automaton $\mathcal{B}$ and a deterministic TS U is PSPACE-complete. The hardness follows by reducing the universality problem for BAs to it. The decision problem is in PSPACE, since we can guess a set F and check in PSPACE that it is indeed the case that the DBA (U, F) accepts $L(\mathcal{B})$.

4 Applications

In this section, we give applications of the determinization construction presented in §3.1 for languages in WDBA.

4.1 Projection of Definable Sets in Linear Arithmetic

In [4], Boigelot, Jodogne, and Wolper show that WDBAs can be used to decide the mixed first-order logic over the reals and the integers with addition and the

ordering, i.e., FO($\mathbb{R}, \mathbb{Z}, +, <$). The elements of the domain are represented by infinite words. For a given formula, one constructs recursively over the formula structure an automaton. This automaton accepts precisely the infinite words that represent the real numbers that satisfy the formula. Automata constructions handle the logical connectives and quantifiers. With the automata construction presented in §3.1, we can handle the quantifiers more efficiently.

Handling Quantifiers. Since WDBAs are closed under complement, it suffices to consider existential quantifiers. Assume that the WDBA $\mathcal{A}_\varphi$ accepts the words that represent the satisfying assignments for the formula φ. We want to construct a WDBA for the formula $\exists x\varphi$. From $\mathcal{A}_\varphi$, we first construct a WBA $\mathcal{B}$ that—intuitively speaking—guesses the digits for x.

In [4], Boigelot, Jodogne, and Wolper utilize the breakpoint construction [27, 21] to obtain a WDBA $\mathcal{A}_{\exists x\varphi}$ from the WBA $\mathcal{B}$. They turn $\mathcal{B}$ into an equivalent co-BA and apply the breakpoint construction to it. From the resulting co-DBA, they obtain the desired WDBA $\mathcal{A}_{\exists x\varphi}$. The last construction step is possible, since $\mathcal{B}$ accepts a language in WDBA.

Instead of using the breakpoint construction, we can apply the powerset construction to turn the WBA $\mathcal{B}$ into an equivalent WDBA $\mathcal{A}'_{\exists x\varphi}$ (see §3). Since WDBAs have a canonical form, minimization of $\mathcal{A}_{\exists x\varphi}$ and $\mathcal{A}'_{\exists x\varphi}$ result in WDBAs that are isomorphic [25].

Using the powerset construction has the following advantages over the breakpoint construction. Theoretically, we do not have to take a detour by switching the acceptance condition. We stay in the framework of weak Büchi automata. Practically, the advantages are: (1) The powerset construction builds automata that usually have fewer states than the automata obtained by the breakpoint construction. The worst case of the powerset construction is slightly better than the worst case of the breakpoint construction. (2) The powerset construction is easier to implement. For instance, the breakpoint construction builds an automaton, where the states are pairs of sets of states of a given co-BA; in the powerset construction, we only have to deal with sets of states.

Experimental Evaluation. We implemented both constructions in our tool LIRA [2] and evaluated them. The savings in terms of number of states range from 15% to 20%. Since the number of generated states is directly linked to the runtime required to construct the automata and it takes less time to minimize smaller automata, the savings in terms of runtime are slightly better, i.e., the improvement ranges from 20% to 25%.

4.2 Model Checking Finite State Systems

In model checking we want to establish automatically whether a system M satisfies a property φ. A practical relevant subclass of this problem is where M is a finite state system and the property φ is given as a formula in (propositional) linear time temporal logic (LTL). This model checking problem can be solved

Table 1. Characterization of LTL formulas found in the literature

	# of formulas	safety	guarantee	obligation
`eh`	12	3 (25%)	1 (8%)	4 (33%)
`sb`	27	8 (30%)	9 (33%)	15 (56%)
`patterns`	55	36 (65%)	1 (2%)	40 (73%)

algorithmically by using automata-theoretic methods [32]: M and $\neg\varphi$ are translated to BAs $\mathcal{A}_M$ and $\mathcal{A}_{\neg\varphi}$, where $\mathcal{A}_M$ accepts the traces of the system M and $\mathcal{A}_{\neg\varphi}$ accepts the traces that violate the property φ. It holds that M satisfies φ iff $L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$. The emptiness of the intersection of the languages can be checked by building the product automaton of $\mathcal{A}_M$ and $\mathcal{A}_{\neg\varphi}$ on the fly [13]. For instance, the model checker SPIN [15] is based on this automata-theoretic approach.

Instead of using the BA $\mathcal{A}_{\neg\varphi}$ for checking $L(\mathcal{A}_M) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$, we suggest to use the minimal WDBA $\mathcal{B}$ for $\neg\varphi$ whenever φ describes a language in WDBA. The intention of using the minimal WDBA is to accelerate the emptiness check of the product automaton. First, note that in practice $\mathcal{A}_{\neg\varphi}$ is much smaller than $\mathcal{A}_M$. Hence, an (even theoretically expensive) additional computation on $\mathcal{A}_{\neg\varphi}$ that accelerates the emptiness check can result in an overall speed-up. Intuitively, the algorithm of the emptiness check has to resolve the nondeterminism of $\mathcal{A}_{\neg\varphi}$ during the on-the-fly traversal of the product automaton of $\mathcal{A}_M$ and $\mathcal{A}_{\neg\varphi}$. Using the minimized deterministic version of $\mathcal{A}_{\neg\varphi}$ means solving this task in an optimal way. Note that the BA $\mathcal{A}_{\neg\varphi}$ might contain states that are redundant, i.e., states from which we accept the same language. Minimizing a WDBA merges states that are redundant.

Before we evaluate the suggested method, we survey on specifications that describe languages in WDBA and give details of how to construct the minimal WDBA $\mathcal{B}$.

Obligation Formulas. In [6], the properties that describe languages in WDBA are called *obligation* properties. These properties are boolean combinations of *safety* and *guarantee* properties. Intuitively, a safety property states that some bad thing never happens. A guarantee property is the negation of a safety property. Our survey of commonly used LTL formulas show that about half of them describe obligation properties. We checked 12 "hand selected formulas, including many that are in common use" [10], 27 "common formulae and formulae found in the literature" [30], and 55 formula patterns [8], which regularly occur in verification tasks. In the following, we refer to these formula suites as `eh`, `sb`, and `patterns`, respectively. Table 1 shows how many of these formulas describe safety, guarantee, and obligation properties. Note that safety and guarantee properties are also obligation properties.

WDBA Construction. For deciding whether an LTL formula describes an obligation, safety, or guarantee property, we implemented a prototype tool that takes an LTL formula as input and characterizes the described property. Moreover, if the LTL formula describes an obligation property, our tool outputs the minimal WDBA for the language described by the LTL formula.

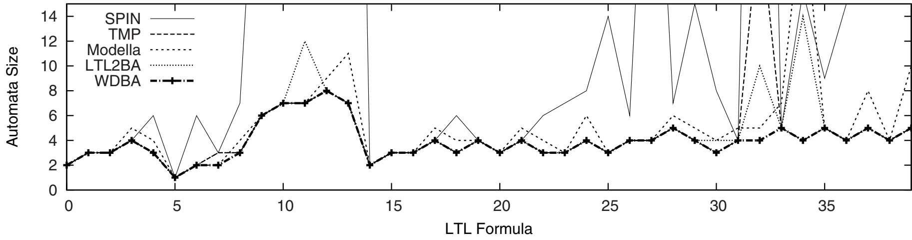


Fig. 3. Automata sizes for LTL formulas

Our tool works as follows. It first constructs BAs $\mathcal{A}$ and $\mathcal{B}$ for the given LTL formula φ and its negation, respectively. Based on the powerset construction and the algorithm in §3.1, we build from $\mathcal{A}$ a WDBA $\mathcal{A}'$. We use the algorithm described in §3.3 to check whether φ describes an obligation formula, i.e., whether it holds $(\Sigma^\omega \setminus L(\mathcal{A})) \cap L(\mathcal{A}') = \emptyset$ and $(\Sigma^\omega \setminus L(\mathcal{A}')) \cap L(\mathcal{A}) = \emptyset$. Since complementing the BA $\mathcal{A}$ is expensive, we use $\mathcal{B}$ instead. Note that complementation of WDBAs is simple: we just need to swap accepting and rejecting states.

- If the check is negative, i.e., $\mathcal{A}'$ does not accept the same language as $\mathcal{A}$, φ is not an obligation formula, and our tool stops.
- Otherwise, φ is an obligation formula. In this case, we minimize $\mathcal{A}'$ by applying the algorithm in [25] and output the resulting minimal WDBA $\mathcal{A}''$. Moreover, we check whether φ describes a safety or guarantee property. Our check is based on the following fact: The minimal WDBA $\mathcal{A}''$ describes a safety property iff $\mathcal{A}''$ has at most one rejecting state q and q is a sink state [24]. The dual statement holds for guarantee properties, since guarantee properties are negated safety properties.

Experimental Evaluation. We conducted two different kinds of experiments.[2] For both experiments we used a computer with an Intel Pentium 4 processor with 3 GHz and with 4 GBytes of main memory.

In the first experiment, we used different translators from LTL to BAs to compare the constructed automata with the minimal WDBAs. Namely, we used the tools TMP [10,11], LTL2BA [12], MODELLA [29], and the translator that is included in the model checker SPIN. Moreover, we used our prototype implementation that outputs the minimal WDBA whenever the input LTL formula describes a language in WDBA.

As test cases we used the 40 negated LTL formulas in `patterns` that describe obligation properties. Figure 3 summarizes the sizes of the BAs that are produced by the different tools. Although in theory, the minimal WDBA can be exponentially larger than an equivalent BA, we never observed such a blow-up on our test cases. Surprisingly, in all cases the size of the minimal WDBA is equal or even

[2] The experimental data is publicly available on the web page `http://www.inf.ethz.ch/personal/daxc/atva07/`.

Table 2. Running times (in minutes) and memory usage (in MBytes) of the model checker SPIN

	bobdb (56,56)		elevetor2 (14)		giop (3)		signarch (2)	
	time	memory	time	memory	time	memory	time	memory
SPIN	14m04	2865	–	> 3 GBytes	–	> 3 GBytes	17m57	2003
TMP	13m53	2865	7m19	2235	0m04	378	14m25	2003
LTL2BA	14m04	2865	7m16	2107	0m15	488	14m23	2003
MODELLA	14m04	2865	6m41	2162	–	> 3 GBytes	14m09	2003
WDBA	8m05	2112	6m31	2034	0m06	350	5m17	778

smaller than the smallest BA constructed by one of the other tools. We want to remark that the constructed BAs are nondeterministic in almost all cases, even in the cases where they have the same number of states as the corresponding minimal WDBAs. For each of the given LTL formulas, the construction of the minimal WDBA only took a few seconds.

In our second experiment, we measured the impact of the constructed BAs in finite state model checking. We used models from the database BEEM [28], which contains numerous finite state systems. For example, it contains the systems `bobdb` and `elevator2`: `bobdb` models an audio/video power controller and `elevator2` models an elevator controller. Additionally, we used the system model described in [16], which we name `giop` and the system model described in [1], which we name `signarch`.

Table 2 lists the running times and the memory usage of some of our test cases. Most of the models have parameters, which can be instantiated to concrete values, e.g., the model `elevator2` is parameterized by the number of floors. In the table, the numbers in the parentheses after the model names are the used values for the parameters of the models. Due to space limitations, we do not list all the concrete values for the parameters that we used in our tests. For all test cases, using the minimal WDBA accelerated the emptiness checks and reduced the memory usage. For the test case `signarch`, we obtained a speed-up of a factor of almost 3. The memory usage was smaller by more than a factor of 2. For the test case `bobdb`, SPIN, TMP, LTL2BA, and MODELLA produced almost identical BAs for the given LTL formula. So, it is not surprising that the consumed memory and the running times are similar for this test case. Further, we remark that the model `giop` does not satisfy the given property. With the BAs generated by SPIN and MODELLA, we were not able to find a counterexample.

5 Conclusion

We have presented novel automata constructions for determinizing restricted classes of automata over infinite words. We have applied and evaluated the constructions in the automata-based approach for $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, <)$. Moreover, based on the new determinization constructions, we have presented and evaluated a new method for model checking obligation properties. In both application

areas, our experimental evaluations demonstrate that the new constructions lead to faster running times and reduced memory usage. Further improvements are possible by tailoring the emptiness check in SPIN for WDBAs. Our experiments also revealed that many specifications that occur in practice describe obligation properties that can be represented by small WDBAs.

As future work, we want to use co-DBAs and minimal WDBAs for optimizing the SAT encoding of the specifications in bounded model checking. We believe that, similar as for explicit model checkers like SPIN, the use of deterministic automata accelerates the SAT solving. Moreover, we want to investigate and evaluate the presented determinization constructions for runtime verification.

Acknowledgements. We thank the reviewers for their detailed comments to improve this paper.

References

1. Basin, D., Kuruma, H., Miyazaki, K., Takaragi, K., Wolff, B.: Verifying a signature architecture: a comparative case study. Formal Aspects of Computing 19, 63–91 (2007)
2. Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: LIRA: Handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)
3. Boigelot, B., Bronne, L., Rassart, S.: An improved reachability analysis method for strongly linear hybrid systems (extended abstract). In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 167–178. Springer, Heidelberg (1997)
4. Boigelot, B., Jodogne, S., Wolper, P.: An effective decision procedure for linear arithmetic over the integers and reals. ACM Trans. Comput. Log. 6, 614–633 (2005)
5. Cerná, I., Pelánek, R.: Relating hierarchy of temporal properties to model checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003)
6. Chang, E., Manna, Z., Pnueli, A.: The safety-progress classification, in Logic and Algebra of Specifications. In: Bauer, F., Brauer, W., Schwichtenberg, H. (eds.) NATO Advanced Science Institutes Series, pp. 143–202. Springer, Heidelberg (1991)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8, 244–263 (1986)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE 1999, pp. 411–420 (1999), See also http://patterns.projects.cis.ksu.edu/
9. Eisinger, J., Klaedtke, F.: Don't care words with an application to the automata-based approach for real addition. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 67–80. Springer, Heidelberg (2006)
10. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–168. Springer, Heidelberg (2000)
11. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM J. Comput. 34, 1159–1175 (2005)

12. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
13. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: 15th IFIP WG6.1 Int. Symp. on Protocol Specification, Testing and Verification. IFIP Conf. Proc. vol. 38, pp. 3–18 (1995)
14. Henzinger, M.R., Telle, J.A.: Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In: Scandinavian Workshop on Algorithm Theory, pp. 16–27 (1996)
15. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
16. Kamel, M., Leue, S.: Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. Int. J. Softw. Tools Technol. Transf. 2, 394–409 (2000)
17. King, V., Kupferman, O., Vardi, M.Y.: On the complexity of parity word automata. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 276–286. Springer, Heidelberg (2001)
18. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. 13, 571–586 (2002)
19. Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. Int. J. Found. Comput. Sci. 17, 869–884 (2006)
20. Kupferman, O., Vardi, M.: Freedom, weakness, and determinism: From linear-time to branching-time. In: LICS 1998, pp. 81–92 (1998)
21. Kupferman, O., Vardi, M.: Weak alternating automata are not that weak, ACM Trans. Comput. Log. 2, pp. 408–429 (2001)
22. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. 35, 59–71 (1987)
23. Kurshan, R.P: Computer Aided Verification of Coordinating Processes. Princeton University Press (1994)
24. Landweber, L.H.: Decision problems for ω-automata. Math. Syst. Theory 3, 376–384 (1969)
25. Löding, C.: Efficient minimization of deterministic weak ω-automata. Inform. Process. Lett. 79, 105–109 (2001)
26. Maler, O., Staiger, L.: On syntactic congruences for omega-languages. Theoret. Comput. Sci. 181, 93–112 (1997)
27. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoret. Comput. Sci. 32, 321–330 (1984)
28. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007), http://anna.fi.muni.cz/models/
29. Sebastiani, R., Tonetta, S.: More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 126–140. Springer, Heidelberg (2003)
30. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
31. Thomas, W.: Automata over infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. ch. 4, vol. B, pp. 133–192. Elsevier, Amsterdam (1990)
32. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS 1986, pp. 322–331 (1986)

Verifying Heap-Manipulating Programs in an SMT Framework*

Zvonimir Rakamarić[2], Roberto Bruttomesso[1], Alan J. Hu[2], and Alessandro Cimatti[1]

[1] ITC-IRST, Povo, Trento, Italy
{bruttomesso,cimatti}@itc.it
[2] Department of Computer Science, University of British Columbia, Canada
{zrakamar,ajh}@cs.ubc.ca

Abstract. Automated software verification has made great progress recently, and a key enabler of this progress has been the advances in efficient, automated decision procedures suitable for verification (Boolean satisfiability solvers and satisfiability-modulo-theories (SMT) solvers). Verifying general software, however, requires reasoning about unbounded, linked, heap-allocated data structures, which in turn motivates the need for a logical theory for such structures that includes unbounded reachability. So far, none of the available SMT solvers supports such a theory. In this paper, we present our integration of a decision procedure that supports unbounded heap reachability into an available SMT solver. Using the extended SMT solver, we can efficiently verify examples of heap-manipulating programs that we could not verify before.

1 Introduction

Automated software verification has made great progress recently, with several successful tools developed in both industry and academia. A key enabling technology for this success has been the advances in automated decision procedures — the software verification tools almost all rely on some form of automatic logical reasoning engine. Some rely on SAT (Boolean satisfiability) or BDDs (binary decision diagrams) to maintain bit-accurate precision (e.g., [16,22,2]), whereas others use SMT solvers (satisfiability modulo theories — decision procedures for combinations of decidable theories) in order to capitalize on the natural abstractions present in software verification, such as integer and real linear arithmetic, arrays, and uninterpreted functions (e.g., [4,20,18,5]).

To be broadly applicable, however, software verification tools must be able to verify programs with dynamic memory allocation, i.e., that manipulate potentially unbounded, heap-allocated, linked data structures via pointers. Although verification of such *heap-manipulating programs* (HMPs) is obviously undecidable in general, careful crafting can produce a logic that is expressive enough to verify important properties of programs, yet is still decidable. In particular, a crucial feature for such logics is the ability to specify unbounded reachability (e.g., from node x, is it possible to reach node y by

* Supported by (1) a research grant from the Natural Sciences and Engineering Research Council of Canada, (2) a University of British Columbia Graduate Fellowship, (3) ORCHID, a project sponsored by Provincia Autonoma di Trento, and (4) a research grant from Intel.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 237–252, 2007.
© Springer-Verlag Berlin Heidelberg 2007

following pointers) and related concepts such as betweenness. Slightly more expressive logics, however, are undecidable [21].

Logics for HMP verification have long been a topic of research. Even Nelson's seminal work on software verification with SMT solvers supported a theory of unbounded S-expressions, although without reachability [35,37], and soon thereafter, Nelson proposed a first-order axiomatization that approximated unbounded reachability [36]. The past few years, however, have seen a blossoming of research in this area, with numerous proposed logics and decision procedures for HMPs, with varying degrees of expressiveness and efficiency, e.g., [3,6,9,15,21,24,27,28,29,33,34,39,40,41]. Research progress has been great, with verification examples that were beyond the reach of methods just a few years ago now being verified in seconds. However, the research on HMP verification has focused almost exclusively on the heap-verification aspects, while mainstream software verification research has largely ignored HMP verification — an understandable division, given the difficulty of both problems.

With the logics and decision procedures for HMPs maturing, the time is right to integrate them back into a general SMT solver, to enable verification of more general software. We want to verify software, including software that manipulates heaps, not just software that *only* manipulates heaps! A few researchers have started in this direction. For example, Lahiri and Qadeer have expressed an incomplete axiomatization of unbounded reachability as universally quantified axioms in the Simplify first-order prover [17], allowing verification of heap and non-heap properties and their interactions, but with a substantial performance penalty [27]. Beyer et al. [7] take a different approach, making calls to a specialized HMP verification system (the TVLA system [30]) to handle the heap aspects of the verification from within their non-heap-aware software verification tool. They report excellent performance, but such a loose combination doesn't allow verification of general interactions between heap properties and other program properties. In very recent follow-on work [8], they add a "strengthening" operator to propagate additional information between the heap and non-heap theories, but still not all interactions are captured. Similarly, Charlton and Huth [14] propose a software model checker in which separate analysis plugins (such as for heaps and for other theories) can cooperate, but the communication is ad hoc, so there are no guarantees that all interactions between theories are propagated. Closest to our work is extremely recent work by Lahiri and Qadeer [28]: Instead of their previous first-order axiomatization, they present a decision procedure based on a complete set of rewrite rules, inspired by our previous work [9]. However, they prototype an implementation of the rewrite rules by using the same trick of modeling rewrite rules as universally-quantified first-order axioms inside the theorem prover, as before. Practical implementation of their decision procedure into an SMT solver has not yet been done. The obviously promising next step is a tight integration of an efficient decision procedure for an HMP logic directly into a modern SMT solver, making all of the theories, and their interactions, efficiently available for the verification task. So far, however, nobody has actually done such an integration.

In this paper, we present the theory, methodology, and results of such an integration. In particular, we integrate our recent, efficient decision procedure for an HMP logic that supports unbounded reachability [39] into the established SMT solver

```
1:  procedure INIT-ADD-FLAG(head, val)
2:      assume  reach(next, head, t) ∧ reach(next, head, nil) ∧ ¬t = nil ∧ oldSum =
            data_int(sum, t) ∧ oldFlag = data_bool(flag, t)
3:      curr := head;
4:      while ¬curr = nil do
5:          if ¬(curr→flag) then
6:              curr→sum := curr→sum + val;
7:              curr→flag := true;
8:          end if
9:          curr := curr→next;
10:     end while
11:     assert  reach(next, head, t) ∧ reach(next, head, nil) ∧ ¬t = nil ∧ data_bool(flag, t) ∧
            (oldFlag ∨ data_int(sum, t) = oldSum + val)
12: end procedure
```

Fig. 1. HMP (Heap-Manipulating Program) Example. The procedure INIT-ADD-FLAG adds the integer variable *val* to integer field *sum* of every node whose boolean field *flag* is false in an acyclic singly-linked list. Also, boolean field *flag* of those nodes is set to true. We denote an integer data field named *sum* of a node x by $\mathsf{data_int}(sum, x)$, a boolean data field named *flag* of a node x by $\mathsf{data_bool}(flag, x)$, and the node pointed to by a pointer field named *next* of node x by $\mathsf{next}(next, x)$. Subformulas of the form $\mathsf{reach}(next, x, y)$ express that node y is reachable from node x by following a sequence of any number of *next* pointer fields. We will formally define these predicates in Sect. 3. The fact that nil is reachable from *head* enforces the acyclicity assumption. Variables *oldSum* and *oldFlag* are used to store values of fields *sum* and *flag* of node t before the procedure starts, respectively. In the **assume** and **assert** statements, variable t represents an arbitrary node (Skolem constant). Since our framework doesn't support quantification, we use the trick of introducing Skolem constants to represent universally quantified variables.

MATHSAT [12].[1] Our results indicate that the integration was fairly straightforward (as was hypothesized in [39] and thanks to the design of MATHSAT [10,11]), the performance overhead of the integration was reasonable, and the integration enabled verification of many example HMPs that we could not verify before.

2 Motivating HMP Example

In our framework, the *heap* consists of an unbounded number of heap *nodes*. HMPs can have program variables that are pointer variables (pointers) and data variables of different types. Similarly, heap nodes can have any number of pointer fields (i.e. links to other nodes) and data fields of different types.

We'll motivate the work presented in this paper with an illustrative HMP example given in Fig. 1. The procedure INIT-ADD-FLAG adds the value of the integer variable *val* to integer field *sum* of every node whose boolean field *flag* is false in the non-empty acyclic singly-linked input list *head*. Furthermore, boolean field *flag* of those nodes is set to true. Necessary assumptions are formalized by the **assume** statement on line 2 of the program. The body of the procedure is simple; it traverses the list, finds

[1] The extended MATHSAT is available at `http://mathsat.itc.it/`.

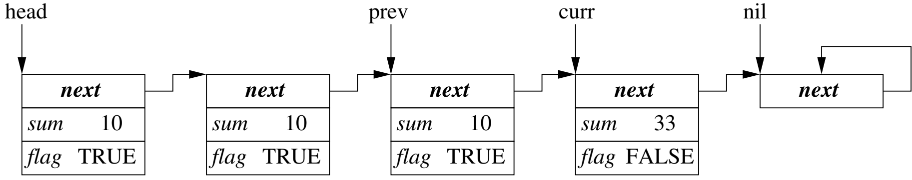


Fig. 2. Heap Structure Example. In this example, each list node has a pointer field *next*, an integer data field *sum*, and a boolean data field *flag*. We model nil as just a node where $\mathsf{next}(f, \mathsf{nil}) = \mathsf{nil}$ for all pointer fields f.

nodes whose field $flag$ is false, and on line 6 adds *val* to the data field *sum* at each iteration. Also, it assigns field $flag$ to true on line 7. The specification is expressed by the **assert** statement on line 11, and indicates that whenever line 11 is reached, *head* points to an acyclic singly-linked list with field *sum* of all nodes whose $flag$ field was false incremented by *val*. The verification problem we are solving can be stated as follows: given an HMP, determine whether it is the case that all executions that satisfy all **assume** statements also satisfy all **assert** statements. Note that even this simple example is beyond the capability of typical software model-checking tools: it is infinite-state due to both the unbounded integers as well as the unbounded heap. To verify such programs, we employ abstraction, using an SMT framework extended with a suitable logical theory described in the next section.

3 Logic for Verifying Heap-Manipulating Programs

Before we define our logic, we'll intuitively illustrate basic concepts on the example of a heap structure shown in Fig. 2. In this heap structure, *head, prev, curr*, and *nil* are pointer variables, *next* is a pointer field used to link nodes in the acyclic list, *sum* is an integer data field, and *flag* is a boolean data field. The node to which we get by following the *next* pointer field from the node pointed to by *head* is denoted in our syntax with $\mathsf{next}(next, head)$. The data field *flag* of the node pointed to by *prev* is accessed with $\mathsf{data_bool}(flag, prev)$. The node pointed to by *curr* is reachable from the node pointed to by *head* by following *next* pointer fields, and that concept of unbounded reachability in our syntax is written as $\mathsf{reach}(next, head, curr)$.

The syntax of our logic is presented in Fig. 3. It is a quantifier-free fragment of first-order logic that contains two equational theories:

1. Theory of data fields with the signature $\{=, \mathsf{data}, \mathsf{update_dfield}\}$. The theory of data fields can be easily translated into the theory of uninterpreted functions as described in Sect. 4.3. For the simplicity of presentation, in this section we give a single untyped theory of data fields. However, without the loss of generality, we can extend this to a family of theories of data fields whose signatures are parameterized using the respective data types. Currently, we support only boolean and integer data fields with the signatures $\{=, \mathsf{data_bool}, \mathsf{update_dfield_bool}\}$ and $\{=, \mathsf{data_int}, \mathsf{update_dfield_int}\}$, but that can easily be extended to other data types supported by the SMT solver (e.g. reals).

$$
\begin{array}{rcll}
c & \in & \mathit{Constants} & \\
x & \in & \mathit{DataVariables} & v \in \mathit{PointerVariables} \\
d, d' & \in & \mathit{DataFields} & f, f' \in \mathit{PointerFields} \\
\mathit{NodeTerm} & ::= & v \mid \mathsf{next}(f, \mathit{NodeTerm}) & \\
\mathit{DataTerm} & ::= & c \mid x \mid \mathsf{data}(d, \mathit{NodeTerm}) & \\
\mathit{Atom} & ::= & \mathit{NodeTerm} = \mathit{NodeTerm} \mid \mathit{DataTerm} = \mathit{DataTerm} \mid & \\
 & & \mathsf{reach}(f, \mathit{NodeTerm}, \mathit{NodeTerm}) \mid & \\
 & & \mathsf{between}(f, \mathit{NodeTerm}, \mathit{NodeTerm}, \mathit{NodeTerm}) & \\
\mathit{Literal} & ::= & \mathit{Atom} \mid \neg \mathit{Atom} \mid & \\
 & & \mathsf{update_pfield}(f, \mathit{NodeTerm}, \mathit{NodeTerm}, f') \mid & \\
 & & \mathsf{update_dfield}(d, \mathit{NodeTerm}, \mathit{DataTerm}, d') & \\
\mathit{Formula} & ::= & \mathit{Literal} \mid \mathit{Formula} \wedge \mathit{Formula} \mid \mathit{Formula} \vee \mathit{Formula} &
\end{array}
$$

Fig. 3. Syntax of the Logic. For brevity, we show the logic with untyped data fields.

2. Theory of unbounded reachability, which is defined below, with the signature $\{=$, next, reach, $\mathsf{between}$, $\mathsf{update_pfield}\}$.

Clearly, the signatures (other than equality) of these two theories are disjoint, and are also disjoint from the signatures of the various theories MathSAT currently supports, such as difference logic, linear arithmetic over reals, and linear arithmetic over integers.

3.1 Theory of Unbounded Reachability

The theory of unbounded reachability over heap nodes presented here is essentially the same as in [39], except that reasoning about data fields is now moved into the theory of data fields and handled by the SMT solver (see Sect. 4.3). The theory assumes a finite set of pointer variables *PointerVariables*, which model program variables that point to nodes in the heap, and a finite set of *pointer function* symbols *PointerFields*, which model pointer fields from a heap node to another heap node. Literals of the form $x = y$, $\neg x = y$, $\mathsf{reach}(f, x, y)$, and $\neg\mathsf{reach}(f, x, y)$ (where x and y are *NodeTerm*) are called *equality*, *disequality*, *reachability*, and *unreachability* literals, respectively. Literals of the form $\mathsf{between}(f, x, y, z)$ or its negation are called *between* literals.

The structures over which the semantics of the theory are defined are called *heap structures*. Formally, a heap structure $H = (N, \Theta)$ consists of a set of *nodes* N and an interpretation function Θ. The interpretation function Θ interprets each symbol σ in $\mathit{PointerVariables} \cup \mathit{PointerFields}$, so that:

– Each pointer variable symbol $\sigma \in \mathit{PointerVariables}$ is interpreted as a node $\Theta(\sigma) \in N$.
– Each pointer function symbol $\sigma \in \mathit{PointerFields}$ is interpreted as a mapping from nodes to nodes $\Theta(\sigma) \in N \to N$.

The interpretation function Θ extends to interpret any term, atom, or literal of the theory in a straightforward, inductive way. The interpretation of a node term $\tau \in \mathit{PointerVariables}$ is defined above, otherwise, τ has the form $\mathsf{next}(f, \tau')$ for some node

term τ', and the interpretation is $\Theta(\tau) = \Theta(f)(\Theta(\tau'))$. Atoms are interpreted by Θ as boolean values:

- An equality atom $\tau_1 = \tau_2$ is interpreted as true iff $\Theta(\tau_1) = \Theta(\tau_2)$.
- A reachability atom $\mathsf{reach}(f, \tau_1, \tau_2)$ is interpreted as true iff there exists some $n \geq 0$ such that $\Theta(f)^n(\Theta(\tau_1)) = \Theta(\tau_2)$.[2]
- A between atom $\mathsf{between}(f, \tau_1, \tau_2, \tau_3)$ is interpreted as true iff there exist $n_0, m_0 \geq 0$ such that $\Theta(\tau_2) = \Theta(f)^{n_0}(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^{m_0}(\Theta(\tau_1))$, $n_0 \leq m_0$, and for all n, m such that $\Theta(\tau_2) = \Theta(f)^n(\Theta(\tau_1))$, $\Theta(\tau_3) = \Theta(f)^m(\Theta(\tau_1))$, we have $n_0 \leq n$ and $m_0 \leq m$.

The interpretation of a pointer field update literal $\mathsf{update_pfield}(f, \tau_1, \tau_2, f')$ is defined using the well-known update operator[3] as true iff

$$\Theta(f') = \mathsf{update}(\Theta(f), \Theta(\tau_1), \Theta(\tau_2)).$$

Finally, the interpretation of a literal that is of the form $\neg\phi$ where ϕ is an atom is simply defined as $\Theta(\neg\phi) = \neg\Theta(\phi)$.

In previous work [9,39], we described a saturation-based decision procedure for the theory of unbounded reachability. The decision procedure is based on the exhaustive application of a set of inference rules and, as we showed on a number of experiments, is very efficient. Furthermore, we presented some theoretical results behind our logic and decision procedure [38]: our decision procedure is sound and always terminates, and the decision procedure is complete for the fragment of the logic without updates. The experiments showed that in practice completeness was not an issue, as we could verify all examples that we could specify.

3.2 Example

Returning to our example from Fig. 2, we'll illustrate the semantics of our logic extended with the boolean and integer data field types on this heap structure with the interpretation of a few representative literals:

- $\mathsf{reach}(next, head, curr)$ is interpreted as true because the node pointed to by *curr* is reachable from the node pointed to by *head* following *next* pointer fields.
- $\mathsf{reach}(next, head, \mathsf{nil})$ is interpreted as true because the node nil is reachable from the node pointed to by *head* following *next* pointer fields. The fact that nil is reachable from *head* enforces the acyclicity assumption.
- $\mathsf{next}(next, curr) = \mathsf{nil}$ is true because the node to which we get by following one *next* pointer field from *curr* is nil.
- $\mathsf{data_bool}(flag, prev) \leftrightarrow \mathsf{true}$ is interpreted as true because the boolean field *flag* of the node pointed to by *prev* is set to true.
- $\mathsf{data_int}(sum, prev) = 10$ is interpreted as true because the integer field *sum* of the node pointed to by *prev* is set to 10.

[2] Here, function exponentiation represents iterative application: for a function g and an element x in its domain, $g^0(x) = x$, and $g^n(x) = g(g^{n-1}(x))$ for all $n \geq 1$.

[3] If g is a function, a is an element in g's domain, and b is an element in g's codomain, then $\mathsf{update}(g, a, b)$ is defined to be the function $\lambda x.(\text{if } x = a \text{ then } b \text{ else } g(x))$.

- between$(next, head, prev, curr)$ is true because node $prev$ is between $head$ and $curr$.
- between$(next, head, \mathsf{nil}, curr)$ is interpreted as false because node nil is not between nodes $head$ and $curr$.

4 Theory Integration into MATHSAT

In this section, we briefly recall some recent results concerning theory combination in SMT, and we disclose some details about the integration of the theory of unbounded reachability into MATHSAT.

4.1 Efficient and Flexible Nelson-Oppen in SMT

Many verification tasks require the specification of properties at a level of expressiveness that is better captured by a logic that is the result of the combination (or union) of simpler theories T_1 and T_2, defined over signatures Σ_1 and Σ_2, respectively. In many situations, decision procedures `Dec(`T_i`)` for T_i, $i = 1,2$, are already available to be used.

Nelson and Oppen [37] showed that given two equational theories T_1 and T_2, it is possible to derive a procedure `Dec(`$T_1 \cup T_2$`)` for deciding quantifier-free formulae over $T_1 \cup T_2$, provided that:

- T_1 and T_2 are signature-disjoint (i.e. $\Sigma_1 \cap \Sigma_2 = \emptyset$);
- T_1 and T_2 are *stably infinite*[4].

A theory is stably infinite if for every satisfiable quantifier-free formula ϕ, there exists an interpretation satisfying ϕ whose domain is infinite. Many theories of interest are stably infinite, including the theory of integers and the theory of unbounded reachability from Sect. 3.1:

Theorem 1. *The theory of unbounded reachability (Sect. 3.1) is stably infinite.*

Proof. Let Ψ be a satisfiable quantifier-free formula, and let $H = (N, \Theta)$ be a heap structure satisfying Ψ. We'll show that one can always construct an infinite heap structure $H' = (N', \Theta')$ satisfying Ψ. Fig. 4 gives an example of how this is done. Basically, adding to the heap structure H an infinite number of nodes that point to themselves (and not changing the existing nodes) creates an infinite heap structure H' satisfying Ψ.

The heap structure H' is formally defined as follows. First, we fix an infinite set of nodes N_{Inf} disjoint from N. Then, we define $N' = N \cup N_{Inf}$, and interpretation Θ' as follows:

Interpretation function Θ' interprets each symbol $\sigma \in PointerVariables$ so that

$$\Theta'(\sigma) = \Theta(\sigma)$$

Every pointer function symbol $f \in PointerFields$ is interpreted so that

$$f^{\Theta'}(\tau) = \begin{cases} f^{\Theta}(\tau) & \text{if } \tau \in N \\ \tau & \text{otherwise} \end{cases}$$

[4] This restriction has been relaxed in the recent work by Krstić et al. [25].

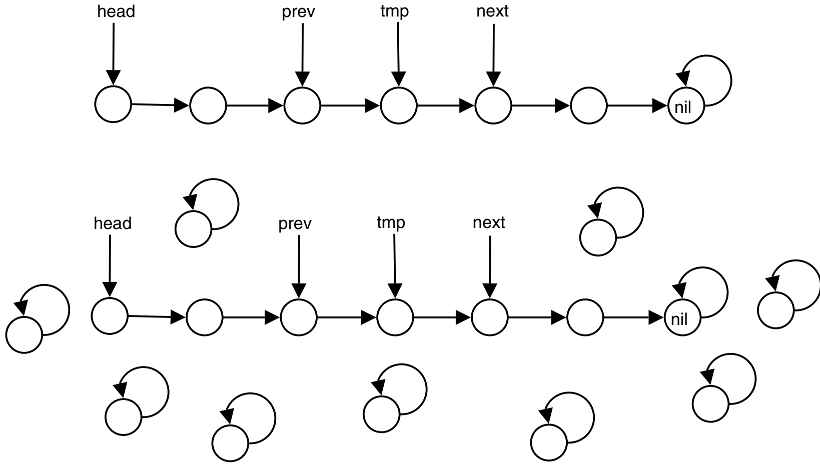


Fig. 4. An example of a heap structure H (top), and a constructed infinite heap structure H' (bottom) which satisfies every quantifier-free formula Ψ that is satisfied by H

Since H is a heap structure satisfying Ψ, the formula Ψ cannot syntactically include any of the nodes in N_{Inf}. Furthermore, for each type of atom, the additional nodes in N_{Inf} cannot change the truth values of those atoms in Ψ, since the new nodes are disconnected from the existing structure, which is unchanged. Therefore, H' also satisfies Ψ, and its domain is infinite. □

The Nelson-Oppen combination schema can be summarized as follows (for a more accurate survey the reader is referred to [32]). The input quantifier-free formula ϕ on $T_1 \cup T_2$ is initially *purified* into an equisatisfiable formula $\phi_1 \wedge \phi_2$ such that ϕ_i belongs to T_i, for $i = 1,2$. This can be easily achieved with the introduction of a set of fresh variables. The procedure is then based on an exhaustive communication between `Dec(`T_1`)` and `Dec(`T_2`)` by means of *interface* equalities, i.e. equalities between variables in $vars(\phi_1) \cap vars(\phi_2)$. Roughly speaking, the exchanging of interface equalities is sufficient for `Dec(`T_1`)` and `Dec(`T_2`)` to achieve an *agreement* on a common model, if such a model exists. This communication has to be implemented around `Dec(`T_1`)` and `Dec(`T_2`)` in order to obtain a correct `Dec(`$T_1 \cup T_2$`)`.

The Nelson-Oppen method is not limited to only two theories. In fact, if T_1 and T_2 are stably infinite, their union $T_1 \cup T_2$ is stably infinite as well. If we are given a decidable stably infinite T_3 over Σ_3 and $(\Sigma_1 \cup \Sigma_2) \cap \Sigma_3 = \emptyset$, than we can apply Nelson-Oppen and obtain a `Dec(`$T_1 \cup T_2 \cup T_3$`)`.

The introduction of a combination framework into an SMT schema can be naively done by considering `Dec(`$T_1 \cup T_2$`)` as a single *theory-solver*, by straightforwardly adapting a DPLL-like `Bool+Dec(`T`)` schema into a `Bool+Dec(`$T_1 \cup T_2$`)` setting.

Delayed Theory Combination (DTC) [10,11] is an alternative approach specifically studied for SMT solvers, based on the observation that it is possible to lift to the boolean level the communication of interface equalities between the theory-solvers, by exploiting

the boolean engine on top of them. The new framework, Bool+Dec(T_1)+Dec(T_2), can be easily achieved as follows.

Given a purified formula $\phi_1 \wedge \phi_2$, the atom set $E = \{x_1 = x_2 \mid x_1, x_2 \in vars(\phi_1) \cap vars(\phi_2)\}$ is first generated. E is nothing but the set of interface equalities that the two theory-solvers, Dec(T_1) and Dec(T_2), *might* need to exchange at any point in time. Any set of theory-atoms Γ assigned to a truth value by the SAT-solver during the search is divided into $\Gamma_1' = \Gamma_1 \cup \Gamma_E$ and $\Gamma_2' = \Gamma_2 \cup \Gamma_E$, where Γ_i are atoms belonging to T_i, for $i = 1,2$, while Γ_E is a set of atoms in E. The set Γ_i' is fed to the corresponding solver Dec(T_i) to be checked for consistency.

Intuitively, the communication in Dec($T_1 \cup T_2$), required for the correctness of the Nelson-Oppen procedure, is now emulated by the introduction of interface equalities that are shared by the two theories. In spite of the (potentially) quadratic number of new atoms generated in E, it is easily possible to control the model enumeration in the SAT-solver, as shown in [13], in order to avoid an enlargement of the search space.

The implementation of a Bool+Dec(T_1)+Dec(T_2) schema presents several advantages with respect to a standard Bool+Dec($T_1 \cup T_2$):

- There is no need to build a Nelson-Oppen "box" Dec($T_1 \cup T_2$) around Dec(T_1) and Dec(T_2), because the integration is implicitly handled at the boolean level and not at the solver level.
- Mixed-conflict generation is automatic.
- Disjunction in case of non-convex theories is automatically handled at the boolean level, while in Nelson-Oppen it must be handled inside Dec($T_1 \cup T_2$). This results in a better efficiency, because of the mechanisms of backjumping and learning implemented in state-of-the-art SAT-solvers.
- The theory-solvers do not need deduction capabilities. In contrast, this is a requirement in Nelson-Oppen. This feature greatly simplified the integration, since our pre-existing decision procedure for the heap logic did not implement deduction.

4.2 Handling Uninterpreted Functions Via Ackermann's Expansion

Ackermann's expansion [1] is a technique by means of which it is possible to translate a quantifier-free formula over $T \cup EUF$ into an equisatisfiable formula ϕ' over T only, where EUF is the well-known theory of Uninterpreted Functions with Equality.

Since function symbols are uninterpreted, the only requirement for satisfiability is *functional consistency*, i.e. the implication $(\bigwedge_{i=1}^{n} t_i = s_i) \rightarrow f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ must hold for every function symbol f of arity n, where t_i and s_i are terms.

In Ackermann's expansion, in order to fulfill the above condition, every distinct function application $f(t_1, \ldots, t_n)$ in ϕ is replaced with a fresh variable $v_{f(t_1,\ldots,t_n)}$. For each function symbol f of arity n, the obtained formula is then augmented with a set of axioms of the kind $(\bigwedge_{i=1}^{n} t_i = s_i) \rightarrow v_{f(t_1,\ldots,t_n)} = v_{f(s_1,\ldots,s_n)}$, for every pair of distinct fresh variables. It is easy to prove that the resulting formula ϕ' no longer contains any UF symbol and it is equisatisfiable to the original ϕ.

The same transformation can be used to remove uninterpreted predicate symbols, using fresh boolean variables and the logical connective $\leftrightarrow$ to equate them in the axiom instantiations.

4.3 Theory Integration

We have integrated the unbounded reachability decision procedure from Sect. 3.1 as a theory-solver `Dec(`HMP`)` into MATHSAT, resulting in a framework for the verification of HMPs supporting boolean and integer data fields, but potentially also any other data type already handled by MATHSAT.

The rationale behind our combination is to separate the "heap reachability" part of the formula from the reasoning about "data", in order to achieve a modular SMT($HMP \cup T$) decision procedure, where T is the theory for a generic data type. In particular, in the current implementation, we provide in the input language a binary predicate $\mathsf{data_bool}(d,h)$, and a binary function $\mathsf{data_int}(d,h)$ that can be used to select a boolean or an integer stored in $d \in DataField$ of $h \in NodeTerm$. Notice that both constructs are uninterpreted, and they merely represent a modular solution to bridge the data and the heap part.

For boolean data, we can exploit the SAT-solver in MATHSAT to decide subformulae expressed on boolean data, by the Ackermann's expansion of the $\mathsf{data_bool}(.,.)$ predicate. The interaction between the integer solver `Dec(`LIA`)` (or in general, the non-boolean) reasoning and `Dec(`HMP`)` can be dealt with in two different ways, either using a `Bool+Dec(`HMP`)+Dec(`LIA`)+Dec(`EUF`)` schema, or a `Bool+Dec(`HMP`)+Dec(`LIA`)` schema, after the Ackermannization of $\mathsf{data_int}(.,.)$ symbols.

Update operations on data $\mathsf{update_dfield}(d,t,v,d')$ may be eagerly replaced with a set of axioms $\{d'(t) \approx v\} \cup \{s \neq t \rightarrow d'(s) \approx d(s) \mid s \in NT\}$, where $\approx$ is the equality $=$ for integer data and $\leftrightarrow$ for boolean data, and NT is the set of $NodeTerm$s that appear in the formula. This solution is far from being optimal, but it worked well in practice for our experiments, where only a few updates were required.

`Dec(`HMP`)`, as any other theory-solver, also benefits of the EUF-layer of MATHSAT. Our experiments show that in many cases this layer is sufficient to determine the unsatisfiability of a query.

Example 1. We are given the following quantifier-free unsatisfiable SMT($HMP \cup LIA$) formula ϕ:

$$(\mathsf{data_int}(d,h_1) + \mathsf{data_int}(d,h_2) = 1) \wedge (h_1 = h_2)$$

Using Delayed Theory Combination: We first purify ϕ into ϕ' with the introduction of two new fresh variables v_1 and v_2, obtaining ϕ':

$$(v_1 = \mathsf{data_int}(d,h_1)) \wedge (v_2 = \mathsf{data_int}(d,h_2)) \wedge (v_1 + v_2 = 1) \wedge (h_1 = h_2).$$

The interface equality $v_1 = v_2$ is also generated. The atoms are assigned to the theories as follows:

$$\begin{array}{ll} HMP & \{h_1 = h_2\} \\ LIA & \{v_1 + v_2 = 1, v_1 = v_2\} \\ EUF & \{v_1 = \mathsf{data_int}(d,h_1), v_2 = \mathsf{data_int}(d,h_2), h_1 = h_2, v_1 = v_2\}. \end{array}$$

The SAT-solver assigns every atom in ϕ' to true. The contradiction is derived because `Dec(`LIA`)` immediately implies $v_1 \neq v_2$, which falsifies the functional consistency in `Dec(`EUF`)`.

Using Ackermann's Expansion: The original formula is expanded into ϕ':

$$(h_1 = h_2) \wedge (v_1 + v_2 = 1) \wedge (h_1 = h_2 \rightarrow v_1 = v_2).$$

Again, $\texttt{Dec}(LIA)$ implies $v_1 \neq v_2$ that contradicts $h_1 = h_2 \wedge (h_1 = h_2 \rightarrow v_1 = v_2)$.

5 Experimental Results

We ran MATHSAT extended with the unbounded reachability theory on a number of HMP verification queries. The queries are from a simple predicate abstraction [19]-based model checker that we are using to verify HMPs. This tool is a straightforward implementation of the software model checking algorithm with predicate abstraction [4], and is described in previous work [9,39]. The experiments were executed on a 2.6 GHz Pentium 4 machine.

The first question is how much overhead the greater complexity of an integrated SMT solver imposes. Table 1 gives a performance comparison with the previous results from [39], using the standalone decision procedure for the unbounded reachability logic. The examples have either no data fields or only boolean data fields, so the previous work could handle them. The safety properties we checked (when applicable) of the HMPs are:

- *no leaks* (NL) – all nodes reachable from the head of the list at the beginning of the program are also reachable at the end of the program.
- *insertion* (IN) – a distinguished node that is to be inserted into a list is actually reachable from the head of the list, i.e. the insertion "worked".
- *acyclic* (AC) – the final list is acyclic, i.e. nil is reachable from the head of the list.
- *cyclic* (CY) – list is a cyclic singly-linked list, i.e. the head of the list is reachable from its successor.
- *doubly-linked* (DL) – the final list is a doubly-linked list.
- *cyclic doubly-linked* (CD) – the final list is a cyclic doubly-linked list.
- *sorted* (SO) – list is a sorted linked list, i.e. each node's data field is less than or equal to its successor's.
- *data* (DT) – data fields of selected (possibly all) nodes in a list are set to a value.
- *remove elements* (RE) – for examples that remove node(s), this states that the node(s) was (were) actually removed.

The comparison shows that the integration isn't a serious overhead. Although MATHSAT, with the integrated unbounded reachability theory, is a more heavyweight tool than the pure unbounded reachability decision procedure we were using previously, the performance penalty is reasonable.

The next question is whether the integration allows effectively verifying example HMPs that could not be handled previously, such as the example in Fig. 1 from Sect. 2.

Without the integration into an SMT solver, we handled integer data fields by bit-blasting them into a fixed number of boolean data fields that represented integers of a certain bit width. We used 1-bit integers in most examples (except for SEARCH-AND-SET where we used 2-bit integers) because the number of states (and therefore the

Table 1. Performance Comparison Against Previous Work [39]. The column "property" specifies the verified property; "preds" is the number of predicates required for verification; "DP calls" is the number of decision procedure queries; "old time" is the total execution time from [39]; "new time" is the total execution time using MATHSAT. Our technical report [38] provides pseudocode and lists the required predicates for these examples. Some of the examples have been taken from related work, while the last three are from Linux kernel list container.

program	property	preds	DP calls	old time (s)	new time (s)
LIST-REVERSE	NL	8	184	0.2	0.2
LIST-ADD	NL∧AC∧IN	8	66	0.1	0.1
ND-INSERT	NL∧AC∧IN	13	259	0.5	0.6
ND-REMOVE	NL∧AC∧RE	12	386	0.9	1.2
ZIP [23]	NL∧AC	22	9153	17.3	27.3
SORTED-ZIP	NL∧AC∧SO∧IN	22	14251	22.8	46.2
SORTED-INSERT [27]	NL∧AC∧SO∧IN	20	5990	13.8	25.3
BUBBLE-SORT [3]	NL∧AC	18	3444	11.1	16.5
BUBBLE-SORT [3]	NL∧AC∧SO	24	31446	114.9	209.0
REMOVE-ELEMENTS	NL∧CY∧RE	17	3124	8.8	14.9
REMOVE-SEGMENT [31]	CY	15	944	2.2	10.0
SEARCH-AND-SET	NL∧CY∧DT	16	4892	5.3	10.8
SET-UNION [36]	NL∧CY∧DT∧IN	21	374	1.4	2.2
CREATE-INSERT	NL∧AC∧IN	24	3020	14.8	15.6
CREATE-INSERT-DATA	NL∧AC∧IN	27	8710	39.7	47.3
CREATE-FREE	NL∧AC∧IN∧RE	31	52079	457.4	489.2
INIT-LIST	NL∧AC∧DT	9	81	0.1	0.1
INIT-LIST-VAR	NL∧AC∧DT	11	244	0.2	0.4
INIT-CYCLIC	NL∧CY∧DT	11	200	0.2	0.4
SORTED-INSERT-DNODES	NL∧AC∧SO∧IN	25	7918	77.9	108.1
REMOVE-DOUBLY	NL∧DL∧RE	34	3238	24.3	33.0
REMOVE-CYCLIC-DOUBLY [27]	NL∧CD∧RE	27	1695	15.6	15.7
LINUX-LIST-ADD	NL∧CD∧IN	25	1240	6.4	8.9
LINUX-LIST-ADD-TAIL	NL∧CD∧IN	27	1638	7.3	10.0
LINUX-LIST-DEL	NL∧CD∧RE	29	2057	24.7	25.2

number of decision procedure queries) grows exponentially with integer bit width. Furthermore, for HMP examples that use addition and multiplication, we would also have had to implement n-bit integer addition and multiplication, which would add even more complexity to the verification problem. We didn't even attempt to verify such examples in our previous work.

With the integration into MATHSAT, a rich set of other theories is available to the verifier. Table 2 shows performance using MATHSAT on the HMP examples that contain (unbounded) integer data fields. In the verification of these examples, we are using a combination of multiple theories, including unbounded reachability, uninterpreted functions, and linear arithmetic. Some examples are the same as before, but with integers expanded from 1 or 2 bits to true integers. There is some slow-down for verification with unbounded integers, but the runtimes are quite comparable to the corresponding

Table 2. Performance on Examples with Integer Data Fields. These examples could not be verified without the SMT integration. Some examples are the same as in Table 1, except with integer data fields; other examples, marked with *, are completely new. Pseudocode and the required predicates for these examples can be downloaded from `http://www.cs.ubc.ca/~zrakamar/software/hmp-examples.tar.gz`.

program	property	preds	DP calls	time (s)
SORTED-ZIP	NL∧AC∧SO∧IN	22	5758	53.9
SORTED-INSERT	NL∧AC∧SO∧IN	20	2972	40.4
BUBBLE-SORT	NL∧AC	17	2348	16.9
BUBBLE-SORT	NL∧AC∧SO	23	17427	371.3
REMOVE-ELEMENTS	NL∧CY∧RE	17	3124	16.4
REMOVE-SEGMENT	CY	15	944	10.3
SEARCH-AND-SET	NL∧CY∧DT	16	5120	13.7
SET-UNION	NL∧CY∧DT∧IN	22	766	5.8
CREATE-INSERT-DATA	NL∧AC∧IN	27	8710	53.6
INIT-LIST	NL∧AC∧DT	9	81	0.1
INIT-LIST-VAR	NL∧AC∧DT	11	244	0.4
INIT-CYCLIC	NL∧CY∧DT	11	200	0.4
SORTED-INSERT-DNODES	NL∧AC∧SO∧IN	25	3636	175.7
LAZY-SIMPLE [7]*	AC∧DT	21	9290	33.4
LAZY-SIMPLE-BACKW [7]*	AC∧DT	15	1127	2.2
INIT-INCREMENT*	AC∧DT	11	354	1.6
INIT-ADD*	AC∧DT	11	354	1.8
INIT-ADD-FLAG*	AC∧DT	12	499	1.4
INIT-MULT*	AC∧DT	11	354	1.8

versions in Table 1. Several additional examples use arithmetic operators on the unbounded integers and have no analogue in Table 1. Overall, we see that we can efficiently verify many examples using the combined theories.

6 Conclusions and Future Work

The paper describes integration of the unbounded reachability theory described in our previous work into MATHSAT, a general purpose SMT solver. Integrating the theory into MATHSAT — easily accomplished through its theory combination framework — provides access to the rich set of theories it supports. Using a combination of different theories of the extended MATHSAT, we verified HMP examples we couldn't handle before. Comparing running times to our previous work shows that the much greater expressiveness comes with only a minor performance penalty. We believe this integration of an HMP-verification logic into a general SMT solver will be broadly applicable to many software verification tools, allowing them to be easily extended to handle both heap-related and other software verification properties.

The primary direction for future work is to improve our predicate abstraction framework to make better use of the capabilities of the combined SMT prover. Our simple predicate abstraction engine eagerly enumerates a huge number of small queries to the

SMT solver and is therefore not benefiting from the solver's powerful search algorithm. Using techniques similar to the *AllSAT* approach to predicate abstraction [26] should substantially improve performance.

References

1. Ackermann, W.: Solvable Cases of the Decision Problem. In: Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam (1954)
2. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)
3. Balaban, I., Pnueli, A., Zuck, L.: Shape analysis by predicate abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
4. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI. Conf. on Programming Language Design and Implementation, pp. 203–213 (2001)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
6. Benedikt, M., Reps, T., Sagiv, M.: A decidable logic for describing linked data structures. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, Springer, Heidelberg (1999)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
9. Bingham, J., Rakamarić, Z.: A logic and decision procedure for predicate abstraction of heap-manipulating programs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 207–221. Springer, Heidelberg (2005)
10. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P.V., Ranise, S., Sebastiani, R.: Efficient satisfiability modulo theories via delayed theory combination. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 335–349. Springer, Heidelberg (2005)
11. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P.V., Ranise, S., Sebastiani, R.: Efficient theory combination via boolean search. Information and Computation 204, 1493–1525 (2006)
12. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P.V., Schulz, S., Sebastiani, R.: The MathSAT 3 system. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 315–321. Springer, Heidelberg (2005)
13. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 527–541. Springer, Heidelberg (2006)
14. Charlton, N., Huth, M.: Hector: Software model checking with cooperating analysis plugins. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 168–172. Springer, Heidelberg (2007)
15. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)

16. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design 25(2-3), 105–127 (2004)
17. Detlefs, D., Nelson, G., Saxe, J.: Simplify: A theorem prover for program checking, Technical Report HPL-2003-148, HP Labs, Palo Alto, CA (2003)
18. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI. Conf. on Programming Language Design and Implementation, pp. 234–245 (2002)
19. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. Symp. on Principles of Programming Languages, pp. 58–70 (2002)
21. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive closure logics. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 160–174. Springer, Heidelberg (2004)
22. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: ICCD. Intl. Conf. on Computer Design, pp. 297–308 (2005)
23. Jensen, J.L., Jørgensen, M.E., Klarlund, N., Schwartzbach, M.I.: Automatic verification of pointer programs using monadic second-order logic. In: PLDI. Conf. on Programming Language Design and Implementation, pp. 226–236 (1997)
24. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, Springer, Heidelberg (2001)
25. Krstić, S., Goel, A., Grundy, J., Tinelli, C.: Combined satisfiability modulo parametric theories. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 602–617. Springer, Heidelberg (2007)
26. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 413–426. Springer, Heidelberg (2006)
27. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: POPL. Symp. on Principles of Programming Languages, pp. 115–126 (2006)
28. Lahiri, S.K., Qadeer, S.: A decision procedure for well-founded reachability, Microsoft Research Tech Report MSR-TR-2007-43 (2007)
29. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, Springer, Heidelberg (2005)
30. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)
31. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
32. Manna, Z., Zarba, C.G.: Combining decision procedures. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 381–422. Springer, Heidelberg (2003)
33. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
34. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI. Conf. on Programming Language Design and Implementation, pp. 221–231 (2001)
35. Nelson, G.: Techniques for program verification. PhD thesis, Stanford University (1979)

36. Nelson, G.: Verifying reachability invariants of linked structures. In: POPL. Symp. on Principles of Programming Languages, pp. 38–47 (1983)
37. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
38. Rakamarić, Z., Bingham, J., Hu, A.: A better logic and decision procedure for predicate abstraction of heap-manipulating programs, UBC Dept. Comp. Sci. Tech Report TR-2006-02 (2006), http://www.cs.ubc.ca/cgi-bin/tr/2006/TR-2006-02
39. Rakamarić, Z., Bingham, J., Hu, A.J.: An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 106–121. Springer, Heidelberg (2007)
40. Ranise, S., Zarba, C.G.: A theory of singly-linked lists and its extensible decision procedure. In: SEFM. IEEE Intl. Conf. on Software Engineering and Formal Methods (2006)
41. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, Springer, Heidelberg (2006)

A Generic Constructive Solution for Concurrent Games with Expressive Constraints on Strategies

Sophie Pinchinat*

Computer Sciences Laboratory of RSISE
The Australian National University, Canberra

Abstract. The emerging technology of interacting systems calls for new formalisms to ensure their reliability. Concurrent games are paradigmatic abstract models for which several logics have been studied. However, the existing formalisms show certain limitations in face of the range of strategy properties required to address intuitive situations. We propose a generic solution to specify expressive constraints on strategies in concurrent games. Our formalism naturally extends alternating-time logics while being highly flexible to combine constraints. Our approach is constructive and can synthesize many types of complex strategies, via automata-theoretic techniques.

1 Introduction

Computer-system design currently relies on complex assemblages of *interacting components* which communicate and share resources in order to achieve services. The combinatorics of such systems is so enormous that the development of adequate formal methods to ensure their reliability has become a major challenge. In this context, games are paradigmatic for providing expressive mathematical models of interactive systems, reflecting their operational semantics and offering adequate reasoning tools. In order to reason formally about interactive models, it is necessary to devise appropriate specification languages in which the desirable behaviors of the system can be stated; once the properties are formulated, methods for automated verification and synthesis can be employed to support the design process.

In the past decade, extensions of state-transition based models, such as *Concurrent Game Structures* [AHK02] which extend Kripke structures, have raised considerable interest in virtue of offering mathematical settings to address formal analysis of complex systems. At the same time, alternating-time logics such as ATL, ATL*, AMC and GL [AHK02] have been proposed as a natural extension of standard temporal logics to the multiplayer setting. Noticeable theoretical and practical results exist for these logics, such as effective decision procedures with reasonable cost for ATL [vD03, KP04, GvD06, SF06, LMO07], and implementations [HKQ98, AdAdS+06]. However, it should be made clear that alternating time logics show certain limitations in face of the range of strategy properties required to address intuitive situations. For example, communication protocols often require to consider fairness assumptions, which

* This research was supported by the Marie Curie Scientific Project MASLOG 021669 (FP6-2004-Mobility-6) and Université de Rennes 1.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 253–267, 2007.
© Springer-Verlag Berlin Heidelberg 2007

enable to exclude some undesirable computations of the system. When such assumptions are not expressible in the logic, non trivial efforts are needed to impose the constraints directly in the models [AHK02]. By this type of approach, only very dedicated kind of analysis can be performed, and often, a minor addition of new constraints compels the user to re-design its problem from scratch. Hence, there is a need for a formalism where constraints on strategies can be combined. Other examples of limitations can be borrowed in solution concepts for non-zero sum games. As far as we are concerned, uniqueness of a Nash equilibrium [Cha05], or dominance of strategies [Ber07] cannot be expressed in any respect, because the formalisms do not have strategies as main objects.

In this paper, we propose a generic constructive solution to analyze the strategies of concurrent games. Our formalism is tuned to specify at the same time the strategies, their properties (e.g. fairness), and their objectives in a unified framework. Expressive constraints can henceforth be formulated; for example all the limitations discussed above are overcome. Moreover, all the concepts for nonzero sum games considered by [CHP07] can be captured, since we can express Strategy Logic (SL) in our formalism; notice that SL is limited to turned-based arenas, whereas we also consider general concurrent game structures, and SL is not powerful enough to express, e.g. it cannot express the Alternating Mu-Calulus [AHK02]. We start from the logic D_μ, a traditional propositional mu-calculus [Koz83], augmented with *decision modalities*. By the semantics of the logic, the game is unfold into an infinite tree. The purpose of decision modalities is to specify particular monadic predicates over the nodes of the computation tree of the game, to establish a one-to-one correspondence between these particular predicates and the strategies of sets of players; the outcome of a strategy is the sub-tree whose nodes form the predicate, and still is a concurrent game, but with less players.

Objectives of strategies can be any ω-regular property. In essence, strategies together with their objectives have an *assume-guarantee* flavor: by assuming that a certain strategy is adopted, we guarantee some temporal property of its outcome. From this point of view, we operate on the model (by applying the strategy) and leave the property intact, as done in [CHP07]. In order to decrease the intricacy of the problem, we propose a powerful although simple mechanism to operate on the logic side while leaving the models intact; it is called *relativization*. The benefit is to transform the complex assume-guarantee statement into a mere temporal statement about the model.

Following the original idea of [RP03] for controller synthesis problems, we define the logic $\mathrm{Q}D_\mu$ a monadic second order extension of D_μ, but where fix-points and quantifiers can arbitrarily interleave. Our calculus is then equipped to quantify over strategies, as we show, in a highly expressive manner. In particular, it subsumes alternating time logics, while being amenable to automata constructions, hence to an effective procedure to synthesize the strategies.

The paper is organized as follows: we present the models in Section 2, and the logic in Section 3. *Strategies* and *outcomes* are defined in Section 4, followed by the *relativization* principle. Section 5 is dedicated to significant examples of logical specifications. Section 6 describes automata constructions for $\mathrm{Q}D_\mu$. We complete the work by the embedding of alternating-time logics into our system (Section 7), and a note on a customized automata construction for these logics.

2 The Models

In the following, we assume an infinite countable ordered set $\boldsymbol{P} = \{p, p', \ldots\}$ of players, and an infinite set of *atomic propositions* $\mathrm{Prop} = \{Q, Q_1, Q_2, \ldots\}$. Finite sets C of players are *coalitions*. For any integer $i \geq 1$, let $[i]$ denote the set $\{1, \ldots, i\}$.

A *Concurrent Game Structure* (CGS) over Λ and M is a tuple $\mathcal{S} = \langle \Pi, S, \Lambda, \lambda, M, \delta \rangle$, where:

- $\Pi \subseteq \boldsymbol{P}$ is a non-empty finite set of *players*, whose cardinal is denoted by n; we may represent the ordered set Π by the natural numbers $1, \ldots, n$.
- S is a set of *states*, with typical elements of S written $s, s', \ldots$.
- M is a set of *moves*. Each $j \in M$ is a possible *move* available in each state to each player $p \in \Pi$. A *decision vector* is a tuple $x = \langle j_1, \ldots, j_n \rangle \in M^n$, where j_p is a move of $p \in \Pi$. The value $Card(M)^n$ is the *branching degree* of $\mathcal{S}$.
- $\delta : S \times M^n \to S$ is the *transition function*: given a state $s \in S$ and decision vector $\langle j_1, \ldots, j_n \rangle$, the game moves to the state $\delta(s, x)$. Each $\delta(s, x)$ where $x \in M^n$ is a *successor* of s, and successors of s form the set $Succ(s)$.
- $\Lambda \subseteq \mathrm{Prop}$, and $\lambda : \Lambda \to 2^S$ labels states by propositions. A state s is labelled by $Q \in \Lambda$ whenever $s \in \lambda(Q)$. We let $\lambda(\mathcal{Q}) := \bigcap_{Q \in \mathcal{Q}} \lambda(Q)$, for any set of propositions $\mathcal{Q} \subseteq \Lambda$.

Comparing this with the original definition of [AHK02], we may notice the following:

1. We use the same set of moves for all players, independently of the current state.
2. Each player moves independently of the others.
3. Players make concurrent choices in each state.

However, the proposed models are expressive enough to capture the essential features of concurrent games, as we can actually simulate any concurrent game: In general, each player in a current state s has a set M^p_s of moves. We can simulate this situation with a unique set of moves M by renaming the moves in M^p_s and by qualifying some states *dummy*; the logical statements need being interpreted on the relevant part of the models, namely on computations which do not encounter dummy states. Hence Points 1 and 2 are not restrictive. Since we can restrict players' set of moves from a given state, enforcing all but one player to have a single choice simulates turned-based games; this sorts out Point 3. Notice that asynchronous games are also captured: following [AHK02], we designate a particular player *scheduler* which in every state selects one of the players; the latter then determines the next state. Now, the crucial assumption that the scheduler fairly selects the players can be expressed in the logic, as opposed to [AHK02] where fairness is defined in the models.

Given $s \in S$, $p \in \Pi$, and $j \in M$, we let $Succ_j(s, p) \subseteq Succ(s)$ be the set of successors of s which can be enforced by the move j of player p; formally, it is the set of states of the form $\delta(s, \langle j_1, \ldots, j_n \rangle)$ with $j_p = j$. Consider the classic two-player game *Paper, Rock, and Scissors (PRS)* as depicted in Figure 1: the possible moves of each player range over $M = \{P, R, S\}$ for "paper", "rock" and "scissors" respectively. The propositions 1-Win and 2-Win indicate who is the winner in the current configuration; let us ignore proposition Q for the moment. In this game, $Succ_S(s_0, 2)$

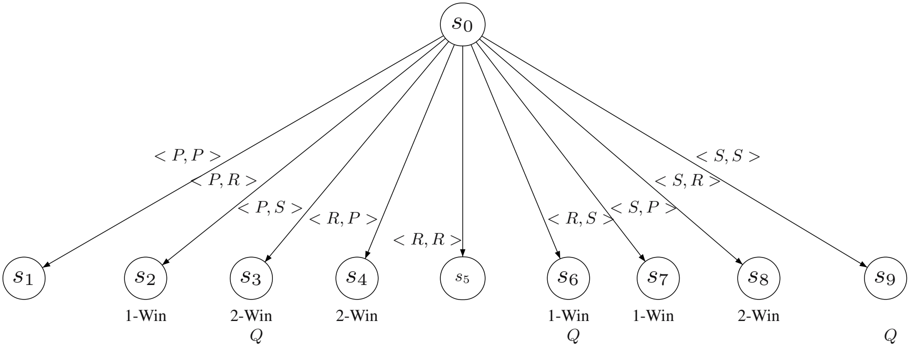


Fig. 1. The Paper, Rock, and Scissors game

$= \{s_3, s_6, s_9\}$ is the set of successors of s_0 player 2 can enforce by playing "S". We let $SuccSets(s,p) \subseteq 2^{Succ(s)}$ be the set of all $Succ_j(s,p)$ where $j \in M$, composed of sets of successors of s which can be enforced by some move of player p. In PRS, $SuccSets(s_0, 2) = \{\{s_1, s_4, s_7\}, \{s_2, s_5, s_8\}, \{s_3, s_6, s_9\}\}$. Since a player $p \notin \Pi$ cannot influence the game, we take the convention that $SuccSets(s,p) = \{Succ(s)\}$. Given a coalition C and $s \in S$, a C-move from s is an element of $\cap_{p \in C} SuccSets(s,p)$; it is a subset of $Succ(s)$ which elements result from fixing a particular move for each player in C. In PRS, a $\{1,2\}$-move from s_0 is $\{s_i\}$, for some $1 \leq i \leq 9$.

3 The Logical Framework

We propose a generalization of [RP03] which is twofold: we enrich the propositional mu-calculus [Koz83] by allowing *decision modalities*, and we consider its monadic second order extension by allowing quantifications over atomic propositions, even under the scope of fix-points operators. We first present the propositional mu-calculus with decision modalities; the second order extension follows in this section.

The logic D_μ is the mu-calculus with *decision modalities* (formulas $\diamond_p \mathcal{Q}$). Given a set Prop of atomic propositions, an infinite set $\boldsymbol{P}$, and a set of variables $\mathrm{Var} = \{Z, Y, \ldots\}$, the syntax of D_μ is:

$$Q \mid \diamond_p \mathcal{Q} \mid \top \mid \neg\beta \mid \beta_1 \vee \beta_2 \mid \mathbf{EX}\,\beta \mid Z \mid \mu Z.\beta(Z)$$

where $Q \in \mathrm{Prop}$, $p \in \boldsymbol{P}$, $\mathcal{Q} \subseteq \mathrm{Prop}$, and β, β_1, β_2 are D_μ formulas. Fix-point formulas ($\mu Z.\beta(Z)$) are such that any occurrence of $Z \in \mathrm{Var}$ in $\beta(Z)$ occurs under an even number of negation symbols $\neg$. A *sentence* is a formula where any occurrence of a variable Z occurs under the scope of a μZ operator. The set of formulas which do not contain any decision modality correspond to the traditional propositional mu-calculus, whence the standard notations $\bot$, $\mathbf{AX}\,\beta$, $\beta_1 \wedge \beta_2$, $\beta_1 \Rightarrow \beta_2$, and $\nu Z.\beta(Z)$ for respectively $\neg\top$, $\neg\mathbf{EX}\,\neg\alpha$, $\neg(\neg\beta_1 \vee \neg\beta_2)$, $\neg\beta_1 \vee \beta_2$, and $\neg\mu Z.\neg\beta(\neg Z)$. Moreover, given $\beta \in D_\mu$, we freely use the concise CTL-like notation $\mathbf{AG}\,(\beta)$ for $\nu Z.(\beta \wedge \mathbf{AX}\,Z)$, which expresses that β is globally true in the future, and $\mathbf{EF}\,(\beta)$ for $\neg\mathbf{AG}\,(\neg\beta)$.

As for the traditional mu-calculus, a formula $\beta \in D_\mu$ is interpreted in a CGS $\mathcal{S} = \langle \Pi, S, \Lambda, \lambda, M, \delta \rangle$ supplied with a valuation $\mathrm{val} : \mathrm{Var} \to 2^S$. Its semantics $[\![\, \beta \,]\!]_{\mathcal{S}}^{\mathrm{val}}$ is a subset of S, defined by induction over the structure of β. The following is very standard as the mu-calculus operators semantics:

$$\begin{aligned}
[\![\, Q \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= \{s \in S \mid s \in \lambda(Q)\} \\
[\![\, \top \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= S \\
[\![\, \neg\beta \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= S \setminus [\![\, \beta \,]\!]_{\mathcal{S}}^{\mathrm{val}} \\
[\![\, \beta_1 \vee \beta_2 \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= [\![\, \beta_1 \,]\!]_{\mathcal{S}}^{\mathrm{val}} \cup [\![\, \beta_2 \,]\!]_{\mathcal{S}}^{\mathrm{val}} \\
[\![\, Z \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= \mathrm{val}(Z) \\
[\![\, \mathbf{EX}\, \beta \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= \{s \in S \mid \exists s' \in Succ(s) \ \wedge\ s' \in [\![\, \beta \,]\!]_{\mathcal{S}}^{\mathrm{val}}\} \\
[\![\, \mu Z.\beta(Z) \,]\!]_{\mathcal{S}}^{\mathrm{val}} &= \cap\{S' \subseteq S \mid [\![\, \beta(Z) \,]\!]_{\mathcal{S}}^{\mathrm{val}(S'/Z)} \subseteq S'\}
\end{aligned}$$

Classically, as a valuation val does not influence the semantics of a sentence $\beta \in D_\mu$, we then simply write $[\![\, \beta \,]\!]_{\mathcal{S}}$.

We now focus on decision modalities which are essential to our logic:

$$[\![\, \diamond_p \mathcal{Q} \,]\!]_{\mathcal{S}}^{\mathrm{val}} = \{s \in S \mid Succ(s) \cap \lambda(\mathcal{Q}) \in SuccSets(s, p)\}$$

By definition, $s \in [\![\, \diamond_p \mathcal{Q} \,]\!]_{\mathcal{S}}$ whenever the set $Succ(s) \cap \{\lambda(Q) \mid Q \in \mathcal{Q}\}$ can alternatively be characterized as a move of player p, namely as $Succ_j(s, p)$ for some move $j \in M$. Notice that the semantics of $\diamond_p \mathcal{Q}$ is well defined even if $p \notin \Pi$, since in this case $SuccSets(s, p)$ equals $\{Succ(s)\}$ by convention. In most of our examples, the set $\mathcal{Q}$ has a single element Q, so we simply write $\diamond_p Q$ for $\diamond_p\{Q\}$.

In the example of the PRS game, Figure 1, $s_0 \notin [\![\diamond_1(\text{1-Win})]\!]_{\mathcal{S}}$ because the predicate 1-Win does not match a set of successors enforced by a move of player 1; as a matter of fact, player 1 does not have a winning strategy, neither does player 2 for symmetrical reasons. On the contrary $s_0 \in [\![\, \diamond_1 Q \,]\!]_{\mathcal{S}}$, since Q matches $Succ_{\mathcal{S}}(s_0, 2)$. When the game is infinite, eg by repeating the game PRS, it can be unfolded as an infinite tree, the nodes of which are histories of an ongoing play. Assume given a predicate Q on the tree nodes such that $\diamond_p Q$ is invariant in the computation tree, that is such that $\mathbf{AG}\,(\diamond_p Q)$ holds in the root. Then any computation inside Q corresponds to a play for a fixed strategy of player p, namely the one designated by Q, and the sub-tree formed by these computations is the outcome of this very strategy. Combining decision modalities for several players characterizes coalition moves: for instance, when a formula like $\diamond_{p_1} Q_1 \wedge \diamond_{p_2} Q_2 \wedge \diamond_{p_3} Q_3$ holds, then the set of successors which satisfy $Q_1 \wedge Q_2 \wedge Q_3$ corresponds to some move of the coalition $\{p_1, p_2, p_3\}$. By extension, if each predicate Q_i designates a strategy of p_i, the sub-tree whose computations remain inside $Q_1 \wedge Q_2 \wedge Q_3$ is the outcome of the coalition strategy.

The logic is extended to the monadic second order to capture strategies as main objects of the logic: stating that there exists a predicate Q such $\mathbf{AG}\,(\diamond_p Q)$ holds expresses the existence of a strategy. This extension of the logic is written $\mathrm{Q}D_\mu$, for "quantified D_μ"; its syntax is as for D_μ but with quantifications over sets of atomic propositions. The syntax of $\mathrm{Q}D_\mu$ is:

$$Q \mid \diamond_p \mathcal{Q} \mid \top \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \mathbf{EX}\, \alpha \mid Z \mid \mu Z.\alpha(Z) \mid \exists \mathcal{Q}.\alpha$$

We write $\forall \mathcal{Q}.\alpha$ for $\neg\exists\mathcal{Q}.\neg\alpha$.

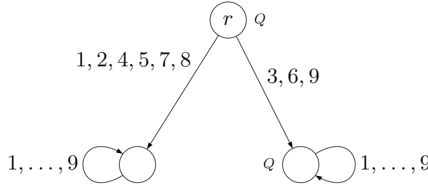


Fig. 2. A Q-labeling $(\mathcal{E}, r)$ of degree 9

The semantics of $\mathrm{Q}D_\mu$ generalizes the one of D_μ: the cases of Q, $\diamond_p \mathcal{Q}$, $\top$, $\neg\alpha$, $\alpha_1 \vee \alpha_2$, **EX** α, Z, and $\mu Z.\alpha(Z)$ are dealt inductively. The semantics of quantification follows the proposal of [RP03]: the mechanism to define new predicates $Q \in \mathcal{Q}$ on a game structure relies on a composition of the structure with a Kripke structure over $\mathcal{Q}$, called a *labeling*.

Definition 1 **($\mathcal{Q}$-labelings).** *Given $\mathcal{Q} \subseteq \mathrm{Prop}$ and $m \geq 1$, a $\mathcal{Q}$-*labeling *(or a* labeling *over $\mathcal{Q}$) is a pair $(\mathcal{E}, r)$ where $\mathcal{E} = \langle E, \mathcal{Q}, \gamma, [m], \delta' \rangle$ is a (one player) CGS structure over $\mathcal{Q}$ and $[m]$, and $r \in E$ is its root. It is a Krikpe structure.*

We compose labelings and CGS's with the same branching degree. We suppose fixed once for all a principle to bijectively relate any set of the form M^n to the set $[Card(M)^n]$ (recall it is $\{1, \ldots, Card(M)^n\}$); for example one can use the coding proposed by [GvD06]. In the following, let us qualify *canonical* a bijection from M^n to $[Card(M)^n]$ whenever it is based on this principle.

Now assume given a rooted CGS $(\mathcal{S}, s)$ with n players over Λ and M, and a labeling $(\mathcal{E}, r)$ over $\mathcal{Q}$ and $[Card(M)^n]$, where $\mathcal{E} = \langle E, \mathcal{Q}, \gamma, [Card(M)^n], \delta' \rangle$; denote by τ the canonical bijection from M^n to $[Card(M)^n]$. The *labeling of* $(\mathcal{S}, s)$ *by* $(\mathcal{E}, r)$ is the synchronous product of the two structures, where x-transitions in $\mathcal{S}$ are synchronized with the $\tau(x)$-transitions in $\mathcal{E}$. Formally,

$$(\mathcal{S}, s) \times (\mathcal{E}, r) = \langle \Pi, S \times E, \Lambda \cup \mathcal{Q}, (\lambda \times \gamma), \delta\text{"} \rangle$$

is the CGS over $\Lambda \cup \mathcal{Q}$ and M rooted at (s, r), where:

- $(\lambda \times \gamma)(Q) = \lambda(Q) \cup \gamma(Q)$, for each $Q \in \Lambda \cup \mathcal{Q}$ with the convention that if $Q \notin \Lambda$ (or $\notin \mathcal{Q}$) then $\lambda(Q)$ (respectively $\gamma(Q)$) is the empty set, and
- $\delta\text{"}((s_1, e_1), x) = (s_2, e_2)$ whenever $\delta(s_1, x) = s_2$ and $\delta'(e_1, \tau(x)) = e_2$.

In the following, composition of a structure with a labeling implicitly assumes that their branching degrees match. Figure 1 shows a (regular) labeling $(\mathcal{E}, r)$ over Q and $[3^2]$, and the labeling of the game structure PRS by $(\mathcal{E}, r)$, with the convention that $\tau(\langle P, P \rangle) = 1$, $\tau(\langle P, R \rangle) = 2$, ..., $\tau(\langle S, R \rangle) = 8$, and $\tau(\langle S, S \rangle) = 9$. The result is depicted in Figure 1.

Notice that since propositions of compound states accumulate, and because $\mathcal{E}$ and $\mathcal{S}$ have the same branching degree, $(\mathcal{S}, s) \times (\mathcal{E}, r)$ is bisimilar to $(\mathcal{S}, s)$ in the usual sense, if we restrict to propositions that are not in $\mathcal{Q}$. In particular if $\mathcal{Q}$ is empty, $(\mathcal{S}, s) \times (\mathcal{E}, r)$

is bisimilar to $(\mathcal{S}, s)$. The composition of $\mathcal{S}$ and labelings is tedious but it only aims at formalizing the means to decoration nodes of the computation tree by propositions; in particular, when $\mathcal{E}$ is a finite state $\mathcal{Q}$-labeling, the predicates Q are regularly placed on the computation tree of the game structure.

We have now the material to define the meaning of quantifiers: $s \in [\![\exists\mathcal{Q}.\alpha]\!]^{\mathrm{val}}_{\mathcal{S}}$ if and only if there exists a $\mathcal{Q}$-labeling $(\mathcal{E}, r)$ such that $(s, r) \in [\![\alpha]\!]^{\mathrm{val}'}_{(\mathcal{S},s)\times(\mathcal{E},r)}$, where $\mathrm{val}'(Z) = \mathrm{val}(Z) \times E$.

Remark that formulas of $\mathrm{Q}D_\mu$ have the same truth value if we unravel the structure $\mathcal{S}$. Besides, the semantics of quantified formulas is a lot more intuitive on the computation tree: $\exists\mathcal{Q}.\alpha$ holds if there is a way to assign the propositions of $\mathcal{Q}$ to the nodes of the computation tree so that α holds.

4 Strategies and Outcomes

In this section, we assume a fixed CGS $\mathcal{S} = \langle \Pi, S, \Lambda, \lambda, M, \delta \rangle$.

We revisit the central concepts of *strategies* and *outcomes* which underlies the semantics of all logics for CGS's: as already explained in Section 3, giving a strategy of player p is equivalent to labeling the structure by some proposition Q where the property $\diamond_p Q$ is invariant. Since invariance is definable in the logic, we obtain the following definition for strategies:

Definition 2 (Strategies). *Given a coalition $C \subseteq \Pi$, and a set $\{Q_p \,|\, p \in C\} \subseteq \mathrm{Prop}$, a C-*strategy from s designated by *$\{Q_p \,|\, p \in C\}$ is a labeling $(\mathcal{E}, r)$ of $(\mathcal{S}, s)$ over $\{Q_p \,|\, p \in C\}$, such that*

$$(s, r) \in [\![\mathbf{AG} \, (\bigwedge_{p \in C} \diamond_p Q_p)]\!]_{(\mathcal{S},s)\times(\mathcal{E},r)} \tag{1}$$

For each C-strategy $(\mathcal{E}, r)$ from s designated by a set $\mathcal{Q}_C = \{Q_p \,|\, p \in C\}$, where $\mathcal{E} = \langle E, \mathcal{Q}, \gamma, [m], \delta' \rangle$, we define its *outcome* as the structure obtained by forgetting all states (s', e) which are not chosen by the coalition C, hence not in the predicate $\bigwedge_{p \in C} Q_p$, and by forgetting the players of C as their moves are fixed by the strategy. Formally, assuming $C \neq \emptyset$, we define $\mathrm{OUT}(\mathcal{Q}_C, \mathcal{S}, s) = \langle \Pi \setminus C, (S \times E) \cap (\lambda \times \gamma)(\mathcal{Q}), \Lambda \cup \mathcal{Q}, M, \delta" \rangle$, with $\delta"((s_1, e_1), y) = \delta((s_1, e_1), y')$ where y' is the decision vector of players in Π obtained by completing the decision vector y of players in $\Pi \setminus C$ by the moves of the players in C recommended by the C-strategy. If $C = \emptyset$ then as expected $\mathrm{OUT}(\mathcal{Q}_\emptyset, \mathcal{S}, s) = (\mathcal{S}, s)$.

Lemma 1. $\mathrm{OUT}(\mathcal{Q}_C, \mathcal{S}, s)$ *is a CGS (rooted at s) over the set of players $\Pi \setminus C$.*

Our definition of outcome is sensible as the set of maximal paths in $\mathrm{OUT}(\mathcal{Q}_C, \mathcal{S}, s)$ coincides with the original definition of 'outcome' in the sense of [AHK02]. However, because our notion retains the structure of a game, contrary to the original definition, we can state any logical statements anew.

In the following, and when it is clear from the context, we simply say "$\mathcal{Q}_C$-strategy" for "C-strategy designated by $\mathcal{Q}_C$", and we write Q_p for $\mathcal{Q}_{\{p\}}$. Also, we concisely write $\mathcal{Q}$ for $\bigwedge_{Q \in \mathcal{Q}} Q$ and define $\bigwedge_{Q \in \emptyset} Q$ as $\top$.

We present now a simple mechanism called the *relativization* which transforms a formula by propagating downward in the formula a set of atomic propositions.

Definition 3 ($\mathcal{Q}$-Relativization of a formula). *For $\mathcal{Q} \subseteq \text{Prop}$, the $\mathcal{Q}$-*relativization *is a mapping $(\cdot|\mathcal{Q}) : \text{Q}D_\mu \to \text{Q}D_\mu$ defined by induction:*

$$\begin{array}{rlr} (Q|\mathcal{Q}) = Q & (\top|\mathcal{Q}) = \top & (Z|\mathcal{Q}) = Z \\ (\neg\alpha|\mathcal{Q}) = \neg(\alpha|\mathcal{Q}) & (\alpha_1 \vee \alpha_2|\mathcal{Q}) = (\alpha_1|\mathcal{Q}) \vee (\alpha_2|\mathcal{Q}) & \\ (\mu Z.\alpha(Z)|\mathcal{Q}) = \mu Z.(\alpha(Z)|\mathcal{Q}) & (\exists\mathcal{Q}'.\alpha|\mathcal{Q}) = \exists\mathcal{Q}'.(\alpha|\mathcal{Q}) & \\ (\diamond_p Q|\mathcal{Q}) = \diamond_p(\mathcal{Q} \wedge Q) & (\mathbf{EX}\,\alpha|\mathcal{Q}) = \mathbf{EX}\,[\bigwedge_{Q\in\mathcal{Q}} Q \wedge (\alpha|\mathcal{Q})] & \end{array}$$

From the above definition, we immediately obtain the equivalences:

$$(\alpha|\emptyset) \equiv \alpha \quad \text{and} \quad (\alpha|\mathcal{Q} \cup \{Q\}) \equiv ((\alpha|\mathcal{Q})|Q). \tag{2}$$

Regarding properties brought about by strategies, Theorem 1 below shows that we can either operate on the model, by considering the outcome and examine its property, or else operate on the formula, by considering the relativization and interpret it on the structure:

Theorem 1. *Given a rooted CGS $(\mathcal{S}, s)$, a coalition C, and a $\mathcal{Q}_C$-strategy $(\mathcal{E}, r)$ from s, we have: for any $\alpha \in \text{Q}D_\mu$, and any valuation* $\text{val} : \text{Var} \to 2^S$*:*

$$[\![(\alpha|\mathcal{Q}_C)]\!]^{\text{val}'}_{(\mathcal{S},s)\times(\mathcal{E},r)} = [\![\alpha]\!]^{\text{val}'}_{\text{OUT}(\mathcal{Q}\ \ ,\mathcal{S},s)}$$

where $\text{val}'(Z) = \text{val}(Z) \times E$.

Proof. The proof of Theorem 1 is conducted by a double induction on the set C and on the structure of α. The case $C = \emptyset$ is trivial and independent of α, since $(\alpha|\mathcal{Q}_C)$ is α by (2), on the one hand, and $(\mathcal{S}, s) \times (\mathcal{E}, r)$ and $\text{OUT}(\mathcal{Q}_C, \mathcal{S}, s)$ are isomorphic to $(\mathcal{S}, s)$, on the other hand. Assume now $C = C' \cup \{p\}$, with $c \notin C$. The $\mathcal{Q}_C$-strategy $(\mathcal{E}, r)$ can be decomposed into $(\mathcal{E}', r') \times (\mathcal{E}_p, r_p)$, where $(\mathcal{E}', r')$ is a $\mathcal{Q}_{C'}$-strategy and $(\mathcal{E}_p, r_p)$ is a Q_p-strategy; let us write $(\mathcal{S}', r')$ for $(\mathcal{S}, s) \times (\mathcal{E}', r')$. By (2):

$$[\![(\alpha|\mathcal{Q}_C)]\!]^{\text{val}'}_{(\mathcal{S},s)\times(\mathcal{E},r)} = [\![((\alpha|\mathcal{Q}_{C'})|Q_p)]\!]^{\text{val}'}_{(\mathcal{S}',s')\times(\mathcal{E}\ \ ,r\ \ )} \tag{3}$$

Lemma 2. *For any rooted CGS $(\mathcal{S}', s')$, any $\{p\}$-strategy $(\mathcal{E}, r)$ designated by Q, any $\alpha \in \text{Q}D_\mu$, and any valuation* $\text{val} : \text{Var} \to 2^S$, $[\![(\alpha|Q)]\!]^{\text{val}'}_{(\mathcal{S}',s')\times(\mathcal{E},r)} = [\![\alpha]\!]^{\text{val}'}_{\text{OUT}(Q,\mathcal{S},s)}$ *where* $\text{val}'(Z) = \text{val}(Z) \times E$.

The proof of this lemma is based on a simple induction over the formulas, in the spirit of [RP03]. Informally, remark first that the relativization is inductively defined for all formulas but those of the form $\mathbf{EX}\,\alpha$. The inductive cases of the lemma follow this line. Regarding statements like $\mathbf{EX}\,\alpha$, the lemma simply expresses that a successor exists in the prune structure $\text{OUT}(Q, \mathcal{S}, s)$ if and only if it already existed in the complete structure and it was labeled by Q.

By Lemma 2, the right hand side of (3) is equal to $[\![(\alpha\,|\,\mathcal{Q}_{C'})]\!]^{\text{val}'}_{\text{OUT}(Q,\mathcal{S}',s')}$. Since $\text{OUT}(Q, \mathcal{S}', s')$ and $\text{OUT}(Q, \mathcal{S}, s) \times (\mathcal{E}', r')$ are isomorphic, it is also equal to $[\![(\alpha \mid \mathcal{Q}_{C'})]\!]^{\text{val}'}_{\text{OUT}(Q,\mathcal{S},s)\times(\mathcal{E}',r')}$ which by induction hypothesis coincides with $[\![\alpha]\!]^{\text{val}'}_{\text{OUT}(\mathcal{Q}\ \ ',\text{OUT}(Q,\mathcal{S},s),(s,r\ \ ))}$. By definition of the outcomes, we have:

Lemma 3. *Given two distinct coalitions C_1, C_2, and any two $\mathcal{Q}_C$ -strategies $(\mathcal{E}_i, r_i)$ $(i \in \{1,2\})$, $\mathrm{OUT}(\mathcal{Q}_{C_1 \cup C_2}, \mathcal{S}, s)$ and $\mathrm{OUT}(\mathcal{Q}_{C_1}, \mathrm{OUT}(\mathcal{Q}_{C_2}, \mathcal{S}, s), (s, r_2))$ are isomorphic.*

Applying Lemma 3 to the term $[\![\, \alpha \,]\!]^{\mathrm{val}'}_{\mathrm{OUT}(\mathcal{Q}\ \ ', \mathrm{OUT}(\mathcal{Q}, \mathcal{S}, s), (s, r\ \ ))}$ yields $[\![\, \alpha \,]\!]^{\mathrm{val}'}_{\mathrm{OUT}(\mathcal{Q}\ \ , \mathcal{S}, s)}$, which concludes the proof of Theorem 1.

Corollary 1. *Given a CGS $\mathcal{S}$, a coalition C, and a sentence $\alpha \in \mathrm{QD}_\mu$. Consider a set of fresh atomic propositions $\mathcal{Q}_C = \{Q_p \,|\, p \in C\}$ indexed over C. The formula*

$$\exists \mathcal{Q}_C.[\mathbf{AG}\,(\bigwedge_{p \in C} \diamond_p Q_p) \wedge (\alpha | \mathcal{Q}_C)]$$

characterizes the states from which there exists a C-outcome of $(\mathcal{S}, s)$ satisfying α.

Proof. By definition, there exists a $\mathcal{Q}_C$-labeling from s, $(\mathcal{E}, r)$ such that (s, r) is a model of $[\mathbf{AG}\,(\bigwedge_{p \in C} \diamond_p Q_p) \wedge (\alpha | \mathcal{Q}_C)]$. Therefore,

$$(s, r) \in [\![\, \mathbf{AG}\,(\bigwedge_{p \in C} \diamond_p Q_p) \,]\!]_{(\mathcal{S}, s) \times (\mathcal{E}, r)}, \text{ and} \tag{4}$$

$$(s, r) \in [\![\, (\alpha | \mathcal{Q}_C) \,]\!]_{(\mathcal{S}, s) \times (\mathcal{E}, r)}. \tag{5}$$

By (4), $(\mathcal{E}, r)$ is a $\mathcal{Q}_C$-strategy. By Theorem 1, (5) is equivalent to $s \in [\![\alpha]\!]_{\mathrm{OUT}(\mathcal{Q}\ \ , \mathcal{S}, s)}$, which concludes. For the reciprocal, simply unroll the reasoning backward.

We make strategies become the main objects of our logic: given a coalition C, and a set $\mathcal{Q}_C = \{Q_p \,|\, p \in C\}$ of propositions, we define the following dual expressions:

$$\hat{\exists} \mathcal{Q}_C.\alpha \stackrel{\text{def}}{=} \exists \mathcal{Q}_C.[\mathbf{AG}\,(\bigwedge_{p \in C} \diamond_p Q_p) \wedge \alpha] \qquad \hat{\forall} \mathcal{Q}_C.\alpha \stackrel{\text{def}}{=} \forall \mathcal{Q}_C.[\mathbf{AG}\,(\bigwedge_{p \in C} \diamond_p Q_p) \Rightarrow \alpha]$$

By Corollary 1, $\hat{\exists} \mathcal{Q}_C.(\alpha | \mathcal{Q}_C)$ expresses the existence of a C-strategy which enforces α. As widely demonstrated in the next section, statements of the form $(\alpha | \mathcal{Q}_C)$ can be combined, and associated with other types of statements (see (6) and (7)). Moreover the property α itself can incorporate specifications about other strategies, hence expressing commitment (see (8)).

5 Expressiveness Issues

This section reveals the high expressiveness of the formalism. We present three significant properties we can express in our formalism, but, we believe, in none of the other logics developed for concurrent games so far. Let us simply write $(\alpha \,|\, Q_1 \wedge Q_2)$ for $(\alpha \,|\, \{Q_1, Q_2\})$, $(Q = Q')$ for $\mathbf{AG}\,(Q \Leftrightarrow Q')$ to denote equality of predicates, and $(Q \neq Q')$ for $\neg(Q = Q')$.

1. *Unique Nash equilibrium in ω-regular games.* Given a two-player game, and an ω-regular objective β [Cha05], the existence of a Nash equilibrium can be stated by $\hat{\exists} Q_1.\hat{\exists} Q_2.Equil(\beta, Q_1, Q_2)$, where

$$Equil(\beta, Q_1, Q_2) \stackrel{\text{def}}{=} \begin{cases} (\beta | Q_1 \wedge Q_2) \\ \wedge \hat{\forall} Q_2'.(Q_2 \neq Q_2') \Rightarrow (\neg\beta | Q_1 \wedge Q_2') \\ \wedge \hat{\forall} Q_1'.(Q_1 \neq Q_1') \Rightarrow (\neg\beta | Q_1' \wedge Q_2) \end{cases}$$

   Uniqueness of the Nash equilibrium is specified by:

$$\hat{\exists} Q_1.\hat{\exists} Q_2.Equil(\beta, Q_1, Q_2) \wedge Unique(Equil(\beta, Q_1, Q_2), Q_1, Q_2) \qquad (6)$$

   where $Unique(\alpha, Q_1, Q_2) = \hat{\forall} Q_1'.\hat{\forall} Q_2'.[\alpha \Rightarrow (Q_1 = Q_1') \wedge (Q_2 = Q_2')]$.
2. *Dominance of strategies.* For instance, a strategy Q weakly dominates another strategy Q' with respect to a goal β [Ber07] whenever (7) holds.

$$\hat{\forall} R.[(\beta | Q' \wedge R) \Rightarrow (\beta | Q' \wedge R)] \wedge \hat{\exists} R.[(\beta | Q \wedge R) \wedge (\neg\beta | Q' \wedge R)] \qquad (7)$$

3. *Communication protocols*. By another reading of Corollary 1, a formula $\hat{\exists}\mathcal{Q}_C.(\alpha | \mathcal{Q}_C)$ states the existence of a C-outcome fixed once for all in which α is interpreted. If α contains a quantified sub-formula $\Delta\mathcal{Q}_{C'}.(\alpha' | \mathcal{Q}_{C'})$ ($\Delta \in \{\hat{\exists}, \hat{\forall}\}$), the statement α' is interpreted in C'-outcomes which lie "inside" the fixed C-outcome. Consider a system with two processors a and b which share a critical resource; we want to specify a protocol $mutex$ in charge of achieving the mutual exclusion. Consider the formula (8):

$$\hat{\exists} Q_{mutex}.(\mathbf{AG}\ (Exclusion \wedge \hat{\exists} Q_a.CritSec_a \wedge \hat{\exists} Q_b.CritSec_b) | Q_{mutex}) \qquad (8)$$

   where $Exclusion = \neg(CritSec_a \wedge CritSec_b)$, $CritSec_z = (\mathbf{AF}\ CritSec_z | Q_z)$. Protocol $mutex$ has a strategy to guarantee the safety property $\mathbf{AG}\ (Exclusion)$, on the one hand, and for each $z \in \{a, b\}$, to enable the recurrent liveness property $\mathbf{AG}\ (\hat{\exists} Q_z.(\mathbf{AF}\ CritSec_z | Q_z)$, on the other hand. Property $(\mathbf{AF}\ CritSec_z | Q_z)$ means that provided processor z adopts policy Q_{mutex}, which consists e.g. in requiring the access to the critical resource and in maintaining this request, it will eventually access to critical section. The *commitment* of $mutex$ to the single strategy Q_{mutex} entails fairness with respect to both processors, although not explicitly specified. Nevertheless, as explained in Section 7, solutions without commitment can also be specified.

   Many other examples of concepts in nonzero-sum games can be expressed in our setting, among which are all the proposals in [CHP07].

6 Automata Constructions

We assume that the reader is familiar with alternating parity tree automata (simply called *automata* when it is clear from the context), and with their relationship with the mu-calculus and parity games (we refer to [AN01], [KVW00], and [Wil01]).

Each formula of our calculus can be represented by alternating parity tree automata, thanks to a powerful construction which generalizes [RP03]. Remark that fix-points and quantifiers do not commute in general: consider the formulas $\alpha_\perp = \exists Q.\nu Z.(\mathbf{AX}\, Z \wedge Q \wedge \mathbf{EX}\, \neg Q)$ and $\alpha_\top = \nu Z.(\mathbf{AX}\, Z \wedge \exists Q.Q \wedge \mathbf{EX}\, \neg Q)$, interpreted on a single infinite path. Whereas the interpretation of $\alpha_\perp$ is the empty set, the one of $\alpha_\top$ is the entire set of states.

We start with an informal presentation of the construction's principles: Existential quantification corresponds to the projection, hence the need to handle non-deterministic automata [Rab69]; by the *Simulation Theorem* [MS95], every alternating automaton is equivalent to a non-deterministic automaton, and the procedure is effective with one exponential blow-up in the size of the automaton. Fix-point operators also have their counterpart on automata: by [AN01, Chapter 7, 7.2], automata can contain variables, we call them *extended automata*; their inputs are like $((\mathcal{S}, s), \mathrm{val})$, where $(\mathcal{S}, s)$ is as usual a model, and $\mathrm{val} : \mathrm{Var} \to 2^S$ is a valuation to interpret the variables, in the same line we interpret non-closed formulas. Extended automata have their own mu-calculus, and fix-point apply on them. Given an extented automaton $\mathcal{A}$, the extended automaton $\mu Z.\mathcal{A}$ can be defined in such a way that e.g. for an automaton $\mathcal{A}$ of a non-closed formula $\exists \mathcal{Q}.\alpha(Z)$, where $Z \in \mathrm{Var}$ is free in $\alpha(Z)$, the automaton $\mu Z.\mathcal{A}$ accepts the models of $\mu Z.(\exists \mathcal{Q}.\alpha(Z))$. Basically, the construction of Theorem 2 relies on three steps. (1) we build the automaton for $\alpha(Z)$; (2) by using the projection operation, we compute the automaton for $\exists \mathcal{Q}.\alpha(Z)$; (3) we build the automaton for $\mu Z.(\exists \mathcal{Q}.\alpha(Z))$. Notice that the automaton obtained for $\alpha(Z)$ may not be non-deterministic in general, either because e.g. $\alpha(Z)$ is of the form $\neg\alpha'(Z)$, or of the form $\alpha_1(Z) \wedge \alpha_2(Z)$. Preliminary to Step (2) we may therefore apply the Simulation Theorem (which by [AN01, Chapter 9] still applies to extended automata)entailing one exponential blow-up .

Theorem 2. *Let $m, n \geq 1$. For any $\alpha \in \mathrm{Q}D_\mu$, write $\kappa \in \mathbb{N}$ for the maximal number of nested quantifiers in α. Then, there exists an alternating parity tree automaton $\mathcal{A}^k_\alpha$ with $\max(\kappa, 0)$-EXPTIME$(|\alpha|)$ states and $\max(\kappa - 1, 0)$-EXPTIME$(|\alpha|)$ priorities, which accepts exactly the models of α of branching degree k, where $k = m^n$, m is the number of moves for each player, and n is the number of players.*

Automata constructions established in Theorem 2 has many interesting corollaries: If we fix the maximal number κ of $\hat{\exists}$ or $\hat{\forall}$ symbols in the formulas, the model-checking problem for $\mathrm{Q}D_\mu$ is κ-EXPTIME-complete (for a fixed branching degree of the structures); more precisely, it is κ-EXPTIME in the size of the formula, but polynomial in the size of the game structure $\mathcal{S}$. Indeed, for the upper bound, the proposed procedure amounts to solving "$\mathcal{S} \in L(\mathcal{A}^k_\alpha)$?", which in the light of [Jur98] for solving two-player parity games can be done with the announced complexity. For the lower bound, simply observe that $\mathrm{Q}D_\mu$ subsumes the proposal in [RP03]. As a consequence, the model-checking problem for $\mathrm{Q}D_\mu$ is non-elementary.

More interestingly, if coalition strategies solutions exist for a given existential $\mathrm{Q}D_\mu$ statements and some game structure, there always exists regular ones, that is describable by finite state machines. Indeed, while model-checking $\mathrm{Q}D_\mu$ formulas (using a classic parity game [Jur98]), any winning strategy for Player 0 in the parity game delivers adequate valuations of the quantified propositions; since parity games always have

memoryless solutions, there always exist regular valuations of the propositions, yielding bounded memory solutions for coalition strategies.

7 Alternating Time Logics

We show that the alternating mu-calculus AMC and the "game logic" from [AHK02] GL are natural fragments of $\mathrm{Q}D_\mu$, as stated by Theorems 3 and 4 – we refer to [Pin07] for details; results for weaker logics such as ATL, Fair ATL, and ATL* follow from their natural embedding either into AMC or GL. As a corollary, automata constructions for alternating time logics can be derived from the procedure presented in Section 6; however, we briefly explain why these automata constructions can be significantly optimized.

For $\mathcal{Q} \subseteq \mathrm{Prop}$, the *bounded $\mathcal{Q}$-relativization* $(\,\cdot\rfloor\mathcal{Q})$ is like the relativization (Definition 3), except that the downward propagation of propositions in the formulas terminates when a quantified sub-formula is encountered:

$$(\exists\mathcal{Q}'.\alpha'\rfloor\mathcal{Q}) = \exists\mathcal{Q}'.\alpha' \tag{9}$$

Relying on the bounded relativization, we define the modality $\hat{\exists}\mathcal{Q}_C(\,\cdot\rfloor\mathcal{Q}_C)$ which has the following semantics: $\hat{\exists}\mathcal{Q}_C(\alpha\rfloor\mathcal{Q}_C)$ states the existence a C-outcome where α holds, but where any further statement $\hat{\exists}\mathcal{Q}_{C'}.\alpha'$ is interpreted in the complete game structure, likewise the modalities of alternating time logics.

7.1 The Alternating-Time μ-Calculus

The syntax of AMC formulas is $Q \mid \top \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid Z \mid \mu Z.\varphi(Z) \mid \langle\!\langle C\rangle\!\rangle \bigcirc \varphi$ with $Q \in \mathrm{Prop}$, $C \subseteq \boldsymbol{P}$, and where each $Z \in \mathrm{Var}$ occurs under an even number of negation symbols $\neg$ in $\varphi(Z)$. These formulas are interpreted over CGS's supplied with a valuation $\mathrm{val} : \mathrm{Var} \to 2^S$. Given $\varphi \in$ AMC, its interpretation $\varphi^{\mathcal{S}}(\mathrm{val}) \subseteq S$ is inductively defined by:

$$Q^{\mathcal{S}}(\mathrm{val}) = \lambda(Q) \quad (\neg\varphi)^{\mathcal{S}}(\mathrm{val}) = S \setminus \varphi^{\mathcal{S}}(\mathrm{val})$$
$$\top^{\mathcal{S}}(\mathrm{val}) = S \qquad Z^{\mathcal{S}}(\mathrm{val}) = \mathrm{val}(Z)$$
$$(\varphi_1 \vee \varphi_2)^{\mathcal{S}}(\mathrm{val}) = \varphi_1^{\mathcal{S}}(\mathrm{val}) \cup \varphi_2^{\mathcal{S}}(\mathrm{val})$$
$$(\mu Z.\varphi(Z))^{\mathcal{S}}(\mathrm{val}) = \bigcap\{S' \subseteq S \mid \varphi(Z)^{\mathcal{S}}(\mathrm{val}[S'/Z]) \subseteq S'\}$$

$(\langle\!\langle C\rangle\!\rangle \bigcirc \varphi)^{\mathcal{S}}(\mathrm{val})$ is the set of states $s \in S$ such that there exists a C-move from s contained in $\varphi^{\mathcal{S}}(\mathrm{val})$.

We define the mapping $\widehat{\cdot} :$ AMC $\to \mathrm{Q}D_\mu$ inductively by: formulas like Q, $\top$ and Z are left unchanged, formulas like $\neg\varphi$, $\varphi_1 \vee \varphi_2$, and $\mu Z.\varphi(Z)$ are dealt inductively, and we set

$$(\widehat{\langle\!\langle C\rangle\!\rangle \bigcirc}\varphi) = \hat{\exists}\mathcal{Q}_C.(\mathbf{AX}\,\widehat{\varphi}\rfloor\mathcal{Q}_C)$$

where $\mathcal{Q}_C = \{Q_p \mid p \in C\}$ is a set of fresh atomic propositions. Notice that the size of $\widehat{\varphi}$ is linear in the size of φ.

Theorem 3. *Given a CGS $\mathcal{S}$, $\varphi \in$ AMC, and a valuation* $\mathrm{val} : \mathrm{Var} \to 2^S$, *we have* $\varphi^{\mathcal{S}}(\mathrm{val}) = [\![\,\widehat{\varphi}\,]\!]^{\mathrm{val}}_{\mathcal{S}}$.

7.2 The Logic GL

Formulas of GL are of three types (the two last types are inherited from CTL*):

State formulas are of the form Q, $\top$, $\neg\varphi$, or $\varphi_1 \vee \varphi_2$ – where φ, φ_1, and φ_2 are state formulas –, and $\exists C.\theta$ – where θ is a tree formula –.
Tree formulas are of the form φ – where φ is a state formula –, $\neg\theta$, or $\theta_1 \vee \theta_2$ – where θ, θ_1, and θ_2 are path formulas –, and $\mathbf{E}\,\psi$ – where ψ is a path formula –.
Path Formulas are of the form θ – where θ is a tree formula –, $\neg\psi$, $\psi_1 \vee \psi_2$, $\bigcirc\psi$, or $\psi_1\,\mathbf{U}\,\psi_2$ – where ψ, ψ_1, and ψ_2 are path formulas –.

We simply sketch the semantics of GL, and we assume that the reader is familiar with CTL* (see [AHK02] for details). Let φ be a state formula, and let $(\mathcal{S}, s)$ be a rooted CGS. $\mathcal{S}, s \models \varphi$, indicating that s satisfies φ in $\mathcal{S}$, is defined by induction over φ. We focus on formulas like $\exists C.\theta$ (the others are dealt inductively or follow the semantics of CTL*): $\mathcal{S}, s \models \exists C.\theta$ whenever there exists a C-outcome $\textsc{out}(\mathcal{Q}_C, \mathcal{S}, s)$ which satisfies θ. Now, θ is a tree formula which in CTL*, up to (non propositional) state sub-formulas $\exists C'.\varphi'$ which must be interpreted back inside $\mathcal{S}$. Let $\varphi^{\mathcal{S}}$ denote the set $\{s \in S \,|\, \mathcal{S}, s \models \varphi\}$.

To lighten the translation of GL into QD_μ, we first establish a translation of GL into a second order extension of CTL* (with decision modalities), written QDCTL*; it generalizes the proposal of [ES84] since quantifications may occur in sub-formulas. In QDCTL*, we denote a tree formula by θ (it may contain quantifications) and a path formula by π, and we write $\mathbf{A}\,\pi$ for $\neg\mathbf{E}\,\neg\pi$, and $\mathbf{G}\theta$ for $\neg(\top\,\mathbf{U}\,\neg\theta)$.

We adapt the definition of the bounded relativization (Section 7) to the syntax of QDCTL*. The relativization of a path formula is conditioned by the path quantifier which binds the formula, as exemplified by the two expressions:

$$(\mathbf{EX}\ \cdot\ |Q) = \mathbf{EX}\,[Q \wedge (\cdot\,|Q)] \qquad (\mathbf{AX}\ \cdot\ |Q) = \mathbf{EX}\,[Q \Rightarrow (\cdot\,|Q)]$$

In order to distinguish the two cases, we define two relativizations of path formulas $(\,\cdot\rfloor_\forall \mathcal{Q})$ and $(\,\cdot\rfloor_\exists \mathcal{Q})$, and set $(\theta\rfloor\mathcal{Q}) = (\theta\rfloor_\forall\mathcal{Q})$ for all tree formula θ. Let $\Delta \in \{\exists, \forall\}$, and θ, θ_1, and θ_2 be tree formulas:

- $(Q\rfloor_\Delta\mathcal{Q}) = Q$, $(\top\rfloor_\Delta\mathcal{Q}) = \top$, and $(\exists\mathcal{Q}'.\theta\rfloor_\Delta\mathcal{Q}) = \exists\mathcal{Q}'.\theta$.
- $(\neg\theta\rfloor_\Delta\mathcal{Q}) = \neg(\theta\rfloor_\Delta\mathcal{Q})$ and $(\theta_1 \vee \theta_2\rfloor_\Delta\mathcal{Q}) = (\theta_1\rfloor_\Delta\mathcal{Q}) \vee (\theta_2\rfloor_\Delta\mathcal{Q})$.
- $(\mathbf{E}\,\pi\rfloor_\forall\mathcal{Q}) = (\mathbf{E}\,\pi\rfloor_\exists\mathcal{Q}) = \mathbf{E}\,(\pi\rfloor_\exists\mathcal{Q})$, and $(\mathbf{A}\,\pi\rfloor_\forall\mathcal{Q}) = (\mathbf{A}\,\pi\rfloor_\exists\mathcal{Q}) = \mathbf{A}\,(\pi\rfloor_\forall\mathcal{Q})$.
- $(\pi_1\,\mathbf{U}\,\pi_2\rfloor_\forall\mathcal{Q}) = [\mathcal{Q}\Rightarrow(\pi_1\rfloor_\forall\mathcal{Q})]\,\mathbf{U}\,[\mathcal{Q}\Rightarrow(\pi_2\rfloor_\forall\mathcal{Q})]$.
- $(\pi_1\,\mathbf{U}\,\pi_2\rfloor_\exists\mathcal{Q}) = [\mathcal{Q} \wedge (\pi_1\rfloor_\exists\mathcal{Q})]\,\mathbf{U}\,[\mathcal{Q} \wedge (\pi_2\rfloor_\exists\mathcal{Q})]$.

(we set similar definitions for path formulas). It can be shown that this definition is consistent with the definition of Section 7. For example, consider the CTL* formula $\mathbf{EF}\,Q_1 \wedge \mathbf{EF}\,Q_2$ which is equivalent to mu-calculus formula $(\mu Z.\mathbf{EX}\,Z \vee Q_1) \wedge (\mu Z.\mathbf{EX}\,Z \vee Q_2)$. Their respective bounded Q-relativization $\mathbf{EF}\,(Q \wedge Q_1) \wedge \mathbf{EF}\,(Q \wedge Q_2)$ (computed according to above) and $(\mu Z.\mathbf{EX}\,(Q{\wedge}Z){\vee}Q_1){\wedge}(\mu Z.\mathbf{EX}\,(Q{\wedge}Z){\vee}Q_2)$ (computed according to Section 7) remain equivalent.

We define $\widehat{\cdot}$: GL $\rightarrow$ QDCTL* by induction: atomic propositions and $\top$ are left unchanged; formulas like $\neg\varphi$, $\varphi_1 \vee \varphi_2$ are dealt inductively, and we define

$$\widehat{\exists C.\theta} = \hat{\exists}\mathcal{Q}_C.(\widehat{\theta}\rfloor\mathcal{Q}_C)$$

Clearly, the size of $\widehat{\varphi}$ is linear in the size of φ, for any $\varphi \in \mathrm{GL}$. Also, since $\hat{\exists}\mathcal{Q}_C.\alpha \in \mathrm{Q}D_\mu$ is definable in QDCTL* provided α is, the co-domain of $\widehat{\cdot}$ is indeed QDCTL*.

Theorem 4. *For any state formula $\varphi \in \mathrm{GL}$, $\varphi^{\mathcal{S}} = [\![\widehat{\varphi}]\!]_{\mathcal{S}}$.*

By an easy adaptation of e.g. the procedure of [Dam94], statements in QDCTL* can be effectively expressed in $\mathrm{Q}D_\mu$.

7.3 A Note on Automata Constructions for Alternating Time Logics

Although our translation $\widehat{\cdot}$ of AMC or GL into $\mathrm{Q}D_\mu$ may generate an arbitrary large number of nested symbols $\hat{\exists}$, the corresponding automata nevertheless remain small, if their construction is carefully conducted; applying Theorem 2 is actually avoidable. Because formulas $\widehat{\varphi}$ are obtained by bounded relativizations of $\mathrm{Q}D_\mu$ formulas, a quantified proposition never occurs in strict quantified sub-formulas. This observation enables us to construct automata in a top-down manner, as opposed to the bottom-up procedure of Theorem 2; due to lack of space, we refer the reader to [Pin07] for the proof details of these constructions, which incidentally match the tight bounds from [AHK02].

Acknowledment

We are extremely grateful to Dietmar Berwanger and Lukasz Kaiser for helpful discussions to improve the first version of the paper. Finally, we thank the reviewers for their highly relevant comments.

References

[AdAdS+06] Adler, B.T., de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Raman, V., Roy, P.: Ticc: A tool for interface compatibility and composition. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 59–62. Springer, Heidelberg (2006)

[AHK02] Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM 49(5), 672–713 (2002)

[AN01] Arnold, A., Niwinski, D.: Rudiments of mu-calculus. North-Holland, Amsterdam (2001)

[Ber07] Berwanger, D.: Admissibility in infinite games. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, Springer, Heidelberg (2007)

[Cha05] Chatterjee, K.: Two-player nonzero-sum omega -regular games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 413–427. Springer, Heidelberg (2005)

[CHP07] Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 59–73. Springer, Heidelberg (2007)

[Dam94] Dam, M.: CTL⋆ and ECTL⋆ as fragments of the modal μ-calculus. Theoretical Computer Science 126(1), 77–96 (1994)

[ES84] Emerson, E.A., Sistla, A.P.: Deciding full branching time logic. Information and Control 61, 175–201 (1984)

[GvD06] Goranko, V., van Drimmelen, G.: Complete axiomatization and decidability of the alternating-time temporal logic. Theoretical Computer Science 353(1), 93–117 (2006)

[HKQ98] Henzinger, T.A., Kupferman, O., Qadeer, S.: From *re*-historic to *ost*-modern symbolic model checking. In: Vardi, M.Y., Hu, A.J. (eds.) CAV 1998. LNCS, vol. 1427, pp. 195–206. Springer, Heidelberg (1998)

[Jur98] Jurdziński, M.: Deciding the winner in parity games is in UP ∩ co-UP. Information Processing Letters 68(3), 119–124 (1998)

[Koz83] Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27(3), 333–354 (1983)

[KP04] Kacprzak, M., Penczek, W.: Unbounded model checking for alternating-time temporal logic. In: Kudenko, D., Kazakov, D., Alonso, E. (eds.) AAMAS 2004, pp. 646–653. IEEE Computer Society, Los Alamitos (2004)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)

[LMO07] Laroussinie, F., Markey, N., Oreiby, G.: On the expressiveness and complexity of ATL. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 243–257. Springer, Heidelberg (2007)

[MS95] Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoretical Computer Science 141(1–2), 69–107 (1995)

[Pin07] Pinchinat, S.: A generic constructive solution for concurrent games with expressive constraints on strategies (full version) August 2007 IRISA Internal Publication 1861, INRIA Research Report (to appear)

[Rab69] Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. 141, 1–35 (1969)

[RP03] Riedweg, S., Pinchinat, S.: Quantified mu-calculus for control synthesis. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 642–651. Springer, Heidelberg (2003)

[SF06] Schewe, S., Finkbeiner, B.: Satisfiability and finite model property for the alternating-time μ-calculus. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, Springer, Heidelberg (2006)

[vD03] van Drimmelen, G.: Satisfiability in alternating-time temporal logic. In: LICS, pp. 208–217. IEEE Computer Society, Los Alamitos (2003)

[Wil01] Wilke. T.: Alternating tree automata, parity games, and modal μ-calculus. Bull. Soc. Math. Belg. 8(2) (May 2001)

Distributed Synthesis for Alternating-Time Logics*

Sven Schewe and Bernd Finkbeiner

Universität des Saarlandes, 66123 Saarbrücken, Germany

Abstract. We generalize the distributed synthesis problem to the setting of alternating-time temporal logics. Alternating-time logics specify the game-like interaction between processes in a distributed system, which may cooperate on some objectives and compete on others. Our synthesis algorithm works for hierarchical architectures (in any two processes there is one that can see all inputs of the other process) and specifications in the temporal logics ATL, ATL*, and the alternating-time μ-calculus. Given an architecture and a specification, the algorithm constructs a distributed system that is guaranteed to satisfy the specification. We show that the synthesis problem for non-hierarchical architectures is undecidable, even for CTL specifications. Our algorithm is therefore a comprehensive solution for the entire range of specification languages from CTL to the alternating-time μ-calculus.

1 Introduction

Program synthesis, which automatically transforms a specification into a correct implementation, has been an active field of research since *Church's solvability problem* [1] in the early sixties. For a given sequential specification over two sets I, O of boolean input and output variables, Church's problem is to find an implementation $f : (2^I)^\omega \to (2^O)^\omega$ such that $(i, f(i))$ satisfies the specification for all possible input sequences $i \in (2^I)^\omega$. Church's problem has been intensively studied in the setting of temporal logics [2,3,4,5,6].

More recently, Church's problem has been extended to distributed systems [7,8,9], where the implementation consists of several independent processes which must choose their actions based on generally incomplete information about the system state. In game-theoretic terms, this type of synthesis solves a multiplayer game, where all players belong to the same team (when synthesizing closed systems), or where the system processes belong to one team and the external environment belongs to the other team (when synthesizing open systems).

However, in many distributed systems the processes do not consistently belong to one team, but rather form different coalitions for different objectives. In security protocols [10,11,12], for example, process Alice may have to deal not only with a hostile environment (which drops her messages from the network),

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 268–283, 2007.
© Springer-Verlag Berlin Heidelberg 2007

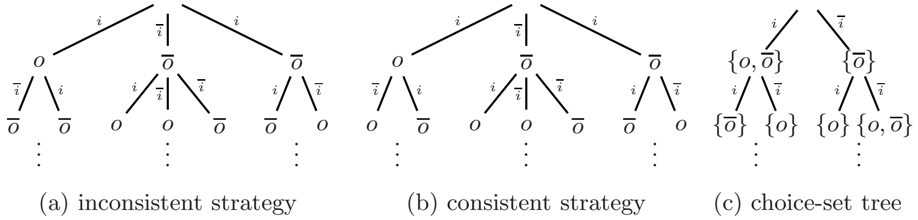


Fig. 1. Figure 1a shows an *inconsistent* nondeterministic strategy: on the leftmost branch (with input $i \cdot \overline{i}$) the process always reacts with output $\overline{o}$, while on the rightmost branch, with identical input $i \cdot \overline{i}$, it reacts with output o. Figure 1b shows a *consistent* nondeterministic strategy. Figure 1c shows the choice-set representation of this strategy.

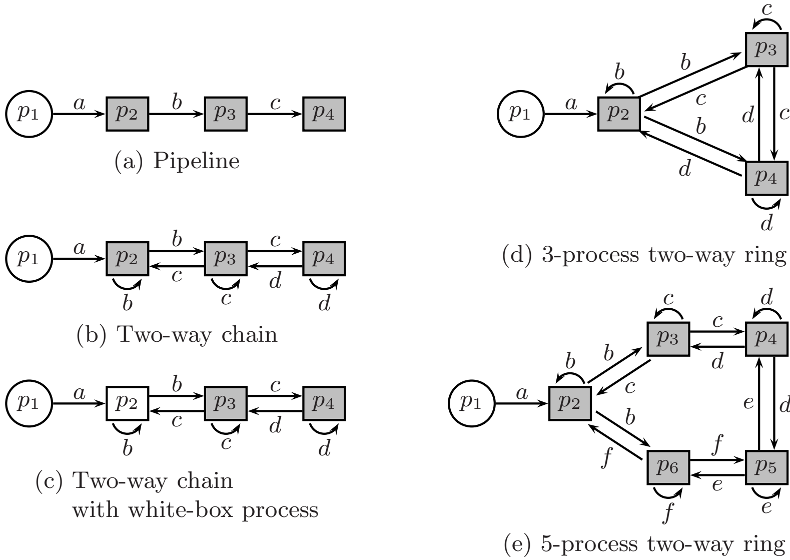
but also with the dishonest process Bob, who cooperates with Alice on some objectives (like transferring money from Alice to Bob), but not on others (like delivering merchandise from Bob to Alice). Such systems can be specified with alternating-time logics, like the alternating-time μ-calculus (AMC) [13], which contain modalities expressing that a process or a coalition of processes has a strategy to accomplish a goal.

In this paper, we solve the synthesis problem for alternating-time logics. For this purpose, we generalize Church's notion of an implementation as a *deterministic strategy* or *function* $f : (2^I)^\omega \to (2^O)^\omega$ to *nondeterministic strategies* or *relations* $r \subseteq (2^I)^\omega \times (2^O)^\omega$, which allow for multiple possible outcomes due to *choices* made by the process.

Church's representation facilitates the development of automata-theoretic synthesis algorithms, because deterministic strategies can be represented as trees that branch according to the possible inputs. Each node carries a label that indicates the output of the process after seeing the input defined by the path to the node. Sets of such trees can be represented as tree automata, and can therefore be manipulated by standard tree automata operations.

Along the same lines, nondeterministic strategies can be unterstood as trees that branch not only according to inputs but also to the choices of the process. However, in this representation, sets of implementations can no longer be represented by tree automata, because tree automata cannot ensure that the choices available to the process are consistent with its observations: a strategy tree is *consistent* if every pair of nodes that are reached on paths labeled by the same input allows the *same set of choices* (for each input). For example, Figure 1a shows an inconsistent strategy tree, while the strategy tree in Figure 1b is consistent. Unfortunately, the consistent trees do not form a regular language, and can therefore not in general be recognized by tree automata.

We solve this problem with a new encoding of nondeterministic strategies as trees where *each node is labeled by the set of possible choices*. Figure 1c shows the representation of the consistent strategy of Figure 1b as such a choice-set tree. Choice-set trees always represent consistent strategies and every consistent strategy can be represented as a choice-set tree (modulo bisimilarity). Using the



(a) Pipeline

(b) Two-way chain

(c) Two-way chain with white-box process

(d) 3-process two-way ring

(e) 5-process two-way ring

Fig. 2. Distributed architectures

choice-set representation, we define an automata-theoretic synthesis algorithm which solves the distributed synthesis problem for all *hierarchical architectures*. Let the system architecture be given as a directed graph, where the nodes represent processes, including the environment as a special process. The edges of the graph are labeled by system variables, which represent the communication of choices: the source process chooses the value and the target process is informed about the choice. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of information. Among the set of system processes, we distinguish two types: a process is black-box if its implementation is unknown and needs to be discovered by the synthesis algorithm. A process is white-box if the implementation is already known and fixed. Figure 2 shows several example architectures, depicting the environment as a circle, black-box processes as filled rectangles, and white-box processes as empty rectangles. We call an architecture *hierarchical* if in each pair of black-box processes, there is one process whose set of input variables is a *subset* of the set of input variables of the other process.

We show that the distributed synthesis problem for alternating-time logics is decidable if and only if the architecture is hierarchical. This is in contrast to our recent result that the distributed synthesis problem for linear and branching-time logics is decidable if and only if the informedness relation of the processes is *fork-free*, i.e., the processes can be completely ordered with respect to their relative informedness [9]. The class of architectures for which the distributed synthesis problem is decidable for alternating-time logics thus forms a

strict subset of the class of architectures for which the problem is decidable for linear and branching-time logics.

For example, the pipeline architecture [7] of Figure 2a is fork-free but not hierarchical: each of the three system processes p_2, p_3, and p_4 has a unique input variable. The two-way chain [8] of Figure 2b is also fork-free and not hierarchical (process p_2 has input $\{a, b, c\}$ and process p_3 has input $\{b, c, d\}$), but becomes hierarchical if process p_2 is made white-box (process p_4 has input $\{c, d\}$, which is contained in the input of p_3), as shown in Figure 2c. Figure 2d and Figure 2e show two ring architectures: The 3-process ring of Figure 2d is both fork-free and hierarchical, while the 5-process ring of Figure 2e satisfies neither criterion.

Related Work. Synthesis algorithms for linear and branching-time logics exploit the finite-model property of these logics: a formula φ is satisfiable iff it has a finite model [3,2]. Our construction builds on the recent result that the finite-model property extends to alternating-time logics [14,15].

The first results for the synthesis of *distributed* systems from temporal formulas are due to Pnueli and Rosner: in their landmark paper [7] they provide a synthesis algorithm for LTL in pipeline architectures and demonstrate the existence of undecidable architectures. An automata-based synthesis algorithm for pipeline and ring architectures and CTL* specifications is due to Kupferman and Vardi [8]. We recently generalized the automata-based construction to all architectures without information forks [9].

2 The Synthesis Problem

In this paper, we solve the distributed synthesis problem for the alternating-time μ-calculus. Given an AMC formula φ and a system architecture, we decide if there exists a distributed implementation that satisfies φ.

2.1 Concurrent Game Structures

In a distributed system where all processes cooperate, we can assume that the behavior of every process is fixed *a priori*: in each state, the next transition follows a deterministic strategy. If we allow for non-cooperating behavior, we can no longer assume a deterministic choice. Instead, we fix the set of *possible decisions* and the effect each decision has on the state of the system. At each point in a computation, the processes choose a decision from the given set and the system continues in the successor state determined by that choice. For two sets of sets X and Y, let $X \oplus Y = \{x \cup y \mid x \in X, y \in Y\}$ denote the set consisting of the unions of their elements. A *concurrent game structure* (CGS) is a tuple $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$, where

- $A = \mathbb{N}_k$ is a finite set of k different processes,
- Π is a finite set of atomic propositions,
- S is a set of states, with a designated initial state $s_0 \in S$,
- $l : S \to 2^{\Pi}$ is a labeling function that decorates each state with a subset of the atomic propositions, and

- α_a defines, for each process $a \in A$, a set of possible decisions.
- $\Delta : S \to \bigoplus_{a\in A}(2^{\alpha_a} \setminus \{\emptyset\})$ maps each state $s \in S$ to a vector of possible decisions for the processes. For $\Delta : s \mapsto \bigoplus_{a\in A} D_a$, $\Delta_{A'}(s)$ denotes the projection of the set $\bigoplus_{a\in A} D_a$ of possible common decisions to the possible decisions $\bigoplus_{a\in A'} D_a$ of a subset $A' \subseteq A$ of the processes.
- Let $\mathfrak{D} = \bigcup_{s\in S} \Delta(s)$ denote the set of all vectors of possible decisions. Then $\tau : S \times \mathfrak{D} \rightharpoonup S$ is a (partial) transition function that maps a state s and a vector d of possible decisions for the processes to a successor state. The partial function τ is defined on $(s,d) \in S \times \mathfrak{D}$ iff $d \in \Delta(s)$.

Architectures. In a distributed system, it is not generally the case that every process is informed about the decisions of all other processes. The system architecture fixes a set of output variables for each process such that every decision corresponds to a certain value of the output variables. An output variable can be an input variable to another process, indicating that the value of the variable is communicated to that process. An *architecture* is a tuple $\mathbf{A} = (A, B, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ with

- a set A of processes, which is partitioned into a set $B \subseteq A$ of *black-box* processes, whose implementations we wish to synthesize, and a set $W = A \setminus B$ of *white-box* processes, which have known and fixed implementations,
- a set Π of system variables or atomic propositions,
- a family $\{I_a\}_{a\in A}$ of sets of input variables, such that $I_a \subseteq \Pi$ denotes the variables visible to agent a, and
- a family $\{O_a\}_{a\in A}$ of non-empty sets of output variables that disintegrates the set Π of system variables.

An architecture is called *hierarchical* if the informedness relation $\preceq = \{(b,b') \in B \times B \mid I_b \subseteq I_{b'}\}$ is linear.

Implementations. An implementation defines for each position of a computation a subset of the output values as the set of possible decisions available to a process. The set of possible decisions must be consistent with the knowledge of the process: an *implementation* of a process $a \in A$ is a function $p_a : (2^{I_a})^* \to 2^{2^{O_a}} \setminus \{\emptyset\} \equiv \mathcal{O}_a$, which assigns a choice-set of possible output values to each history of input values. Occasionally, we consider implementations that have access to a superset I of their input variables. We call a function $p_a : (2^I)^* \to \mathcal{O}_a$ with $I_a \subset I$ a *relaxed* implementation of a with input I.

We identify process implementations with trees. As usual, an Υ-tree is a prefix closed subset $Y \subseteq \Upsilon^*$ of finite words over a predefined set Υ of directions. For given sets Σ and Υ, a *Σ-labeled Υ-tree* is a pair $\langle Y, l\rangle$, consisting of a tree $Y \subseteq \Upsilon^*$ and a labeling function $l : Y \to \Sigma$ that maps every node of Y to a letter of Σ. If Υ and Σ are not important or clear from the context, $\langle Y, l\rangle$ is called a tree. If $Y \neq \emptyset$ is non-empty and, for each $y \in Y$, some successor $y \cdot \upsilon$ ($\upsilon \in \Upsilon$) of y is in Y, then Y and $\langle Y, l\rangle$ are called total. If $Y = \Upsilon^*$, Y and $\langle Y, l\rangle$ are called *full*.

A *distributed implementation* is a set $P = \{p_a\}_{a\in A}$ of process implementations, one for each process a in the architecture. A distributed implementation P defines the concurrent game structure $\mathcal{G}_P = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a\in A}, \Delta, \tau)$ where

– $S = (2^{\Pi})^*$ is the full 2^{Π} tree, with its root $s_0 = \varepsilon$ as initial state,
– each state is labeled with its direction $l(s \cdot \sigma) = \sigma$ (with $l(s_0) = \emptyset$),
– $\alpha_a = 2^{O_a}$,
– $\Delta(s) = \bigoplus_{a \in A} p_a(s_a)$, where $s_a = I_1 I_2 I_3 \dots I_n$ is the local view of process a on $s = V_1 V_2 V_3 \dots V_n$ such that $I_m = V_m \cap I_a$ for all $m \leq n$,
– $\tau(s, d) = s \cdot d$.

2.2 Alternating-Time μ-Calculus

The alternating-time μ-calculus (AMC) extends the classical μ-calculus with modal operators which express that an agent or a coalition of agents has a strategy to accomplish a goal. AMC formulas are interpreted over concurrent game structures.

AMC Syntax. The classical μ-calculus contains two modalities, expressing that a property φ holds in some ($\Diamond\varphi$) or in all ($\Box\varphi$) successor states. In AMC[1], the operators are generalized to $\Box_{A'}\varphi$, expressing that a set $A' \subseteq A$ of agents can enforce that φ holds in the successor state, and $\Diamond_{A'}\varphi$, expressing that it cannot be enforced against the agents A' that φ is violated in the successor state.

Let P and B denote disjoint finite sets of atomic propositions and bound variables, respectively. Then

– *true* and *false* are AMC formulas.
– p and $\neg p$ are AMC formulas for all $p \in P$.
– x is an AMC formula for all $x \in B$.
– If φ and ψ are AMC formulas then $\varphi \wedge \psi$ and $\varphi \vee \psi$ are AMC formulas.
– If φ is an AMC formula and $A' \subseteq A$ then $\Box_{A'}\varphi$ and $\Diamond_{A'}\varphi$ are AMC formulas.
– If $x \in B$ and φ is an AMC formula where x occurs only free then $\mu x.\varphi$ and $\nu x.\varphi$ are AMC formulas.

AMC Semantics. An AMC formula φ with atomic propositions Π is interpreted over a CGS $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$. $\|\varphi\|_{\mathcal{G}} \subseteq S$ denotes the set of nodes where φ holds. A CGS $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$ is a *model* of a specification φ with atomic propositions Π iff $s_0 \in \|\varphi\|_{\mathcal{G}}$, and a distributed implementation P satisfies an AMC specification φ iff $\mathcal{G}_P$ is a model of φ.

– Atomic propositions are interpreted as follows: $\|true\|_{\mathcal{G}} = S$, $\|false\|_{\mathcal{G}} = \emptyset$, $\|p\|_{\mathcal{G}} = \{s \in S \mid p \in l(s)\}$, and $\|\neg p\|_{\mathcal{G}} = \{s \in S \mid p \notin l(s)\}$.
– As usual, conjunction and disjunction are interpreted as intersection and union, respectively: $\|\varphi \wedge \psi\|_{\mathcal{G}} = \|\varphi\|_{\mathcal{G}} \cap \|\psi\|_{\mathcal{G}}$ and $\|\varphi \vee \psi\|_{\mathcal{G}} = \|\varphi\|_{\mathcal{G}} \cup \|\psi\|_{\mathcal{G}}$.

[1] The original definition of alternating-time logics under incomplete information by Alur et al. [13] syntactically restricts the specifications such that the objectives of each process only refer to the atomic propositions that are visible to the process. This restriction ensures that the processes can *state* their respective strategies, while we only require that they *can cooperate* to accomplish their goals. For the specifications allowed in [13], the semantics coincide.

- A node $s \in S$ is in $\|\Box_{A'}\varphi\|_{\mathcal{G}}$ if the agents A' can make a joint decision $\upsilon \in \Delta_{A'}(s)$ such that, for all counter decisions $\upsilon' \in \Delta_{A \smallsetminus A'}(s)$, φ holds in the successor state.
  $\|\Box_{A'}\varphi\|_{\mathcal{G}} = \{s \in S \mid \exists \upsilon \in \Delta_{A'}(s).\, \forall \upsilon' \in \Delta_{A \smallsetminus A'}(s).\, \tau(s,(\upsilon,\upsilon')) \in \|\varphi\|_{\mathcal{G}}\}$.
- A node $s \in S$ is in $\|\Diamond_{A'}\varphi\|_{\mathcal{G}}$ if for all joint decisions $\upsilon \in \Delta_{A \smallsetminus A'}(s)$ of the agents not in A', the agents in A' have a counter decision $\upsilon' \in \Delta_{A'}(s)$ that ensures that φ holds in the successor state.
  $\|\Diamond_{A'}\varphi\|_{\mathcal{G}} = \{s \in S \mid \forall \upsilon' \in \Delta_{A \smallsetminus A'}(s). \exists \upsilon \in \Delta_{A'}(s). \tau(s,(\upsilon,\upsilon')) \in \|\varphi\|_{\mathcal{G}}\}$.
  The modal operators $\Box$ and $\Diamond$ of the classical μ-calculus are equivalent to the modal operators $\Box_{\emptyset}$ and $\Diamond_A$, respectively.
- Let $\mathcal{G}_x^{S_x} = (A, \Pi \cup \{x\}, S, s_0, l_x^{S_x}, \{\alpha_a\}_{a \in A}, \Delta, \tau)$ denote, for $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$ and $x \notin \Pi$, the adapted CGS with the labeling function $l_x^{S_x} : S \to 2^{\Pi \cup \{x\}}$, which is defined by
  - $l_x^{S_x}(s) \cap \Pi = l(s)$ and
  - $x \in l_x^{S_x}(s) \Leftrightarrow s \in S_x \subseteq S$.

  Since, for AMC formulas $\lambda x.\varphi$, x occurs only positive in φ, $\|\varphi\|_{\mathcal{G}_x^{S_x}}$ is monotone in S_x and the following least and greatest fixed points are well-defined:
  $\|\mu x.\varphi\|_{\mathcal{G}} = \bigcap\{S_x \subseteq S \mid \|\varphi\|_{\mathcal{G}_x^{S_x}} \subseteq S_x\}$, and $\|\nu x.\varphi\|_{\mathcal{G}} = \bigcup\{S_x \subseteq S \mid \|\varphi\|_{\mathcal{G}_x^{S_x}} \supseteq S_x\}$.

2.3 Realizability and Synthesis

We call an AMC formula φ *realizable* in a given architecture $\mathbf{A} = (A, B, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ and for a given set $P_W = \{p_w\}_{w \in W}$ of implementations for the white-box processes if there exists a set of implementations $P_B = \{p_b\}_{b \in B}$ for the black-box processes, such that the CGS defined by the distributed implementation $P = P_W \cup P_B$ satisfies φ. $\mathbf{A}$ is called *decidable* if realizability can be decided for all formulas φ and implementations P_W of the white-box processes.

In the following section, we present a *synthesis algorithm*, which determines if a specification is realizable. If the specification is realizable, the synthesis algorithm computes an implementation.

3 The Synthesis Algorithm

In this section, we present a synthesis algorithm for hierarchical architectures. The construction is based on automata over infinite trees and game structures.

3.1 Preliminaries: Automata over Infinite Objects

Automata over Infinite Trees. An *alternating parity tree automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$, where Σ is a finite set of labels, Q is a finite set of states, $q_0 \in Q$ is a designated initial state, δ is a transition function, and $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive boolean combination of states and directions.

The automaton runs on full Σ-labeled Υ-trees. A *run tree* $\langle R, r\rangle$ on a given full Σ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$ is a $Q \times \Upsilon^*$-labeled tree where the root is labeled with (q_0, ε) and where, for each node n with label (q, y) and with the set $L = \{r(n \cdot \rho) | n \cdot \rho \in R\}$ of labels of its successors, the following condition holds: the set $\{(q', \upsilon) \in Q \times \Upsilon \mid (q', y \cdot \upsilon) \in L\}$ satisfies $\delta(q, l(y))$.

An infinite path fulfills the *parity condition*, if the highest color of the states appearing infinitely often on the path is even. A run tree is *accepting* if all infinite paths fulfill the parity condition. A total Σ-labeled Υ-tree is accepted if it has an accepting run tree.

The set of trees accepted by an alternating automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty if its language is empty.

The acceptance of a tree can also be viewed as the outcome of a game, where player *accept* chooses, for a pair $(q, \sigma) \in Q \times \Sigma$, a set of atoms of $\delta(q, \sigma)$, satisfying $\delta(q, \sigma)$, and player *reject* chooses one of these atoms, which is executed. The input tree is accepted iff player *accept* has a strategy enforcing a path that fulfills the parity condition. One of the players has a memoryless winning strategy, i.e., a strategy where the moves only depend on the state of the automaton, the position in the tree and, for player *reject*, on the choice of player *accept* in the same move.

In a *nondeterministic* tree automaton, the image of δ consists only of such formulae that, when rewritten in disjunctive normal form, contain exactly one element of $Q \times \{\upsilon\}$ for all $\upsilon \in \Upsilon$ in every disjunct. For nondeterministic automata, every node of a run tree corresponds to a node in the input tree. Emptiness can therefore be checked with an *emptiness game*, where player *accept* also chooses the letter of the input alphabet. A nondeterministic automaton is empty iff the emptiness game is won by *reject*.

Automata over Concurrent Game Structures. Generalizing symmetric automata [16], automata over concurrent game structures [15] contain *universal atoms* $(\Box, A')$, which refer to *all* successor states for *some* decision of the agents in A', and *existential atoms* $(\Diamond, A')$, which refer to *some* successor state for *each* decision of the agents *not* in A'.

An *automaton over concurrent games structures* (ACG) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$, where Σ, Q, q_0, and α are defined as for alternating parity automata in the previous paragraph. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times (\{\Box, \Diamond\} \times 2^A))$ now maps a state and an input letter to a positive boolean combination of two types of atoms: $(q, \Box, A')$ is a universal atom, and $(q, \Diamond, A')$ is an existential atom.

A run tree $\langle R, r\rangle$ on a given CGS $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$ is a $Q \times S$-labeled tree where the root is labeled with (q_0, s_0) and where, for a node n with label (q, s) and a set $L = \{r(n \cdot \rho) \,|\, n \cdot \rho \in R\}$ of labels of its successors, the following property holds: there is a set $\mathfrak{A} \subseteq Q \times (\{\Box, \Diamond\} \times 2^A)$ of atoms satisfying $\delta(q, l(s))$ such that

– for all universal atoms $(q', \Box, A')$ in $\mathfrak{A}$, there exists a decision $\upsilon \in \Delta_{A'}(s)$ of the agents in A' such that, for all counter decisions $\upsilon' \in \Delta_{A \smallsetminus A'}(s)$, $(q', \tau(s, (\upsilon, \upsilon'))) \in L$, and

- for all existential atoms $(q', \Diamond, A')$ in $\mathfrak{A}$ and all decisions $\upsilon' \in \Delta_{A \smallsetminus A'}(s)$ of the agents not in A', there exists a counter decision $\upsilon \in \Delta_{A'}(s)$ such that $(q', \tau(s, (\upsilon, \upsilon'))) \in L$.

As before, a run tree is accepting iff all paths satisfy the parity condition, and a CGS is accepted iff there exists an accepting run tree.

3.2 Realizability in 1-Black-Box Architectures

We first consider the realizability problem for architectures with a single black-box process. Given such an architecture $\mathbf{A} = (A, \{b\}, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$, an AMC specification φ and a set $P_W = \{p_w\}_{w \in W}$ of implementations for the white-box processes, the following algorithm constructs a nondeterministic automaton $\mathcal{E}$, which accepts an implementation p_b of the black-box process b iff the distributed implementation $P = P_W \cup \{p_b\}$ defines a concurrent game structure that is a model of φ. Realizability can then be checked by solving the emptiness game for $\mathcal{E}$. For convenience, we use $\mathcal{V} = \bigoplus_{a \in A} \mathcal{O}_a$ in the following constructions. The synthesis algorithm uses the following automata operations:

- **From specification to automata.** First, a specification φ is turned into an ACG $\mathcal{A}$ that accepts exactly the models of φ (Theorem 1).
- **From models to implementations.** We then transform $\mathcal{A}$ into an alternating tree automaton $\mathcal{B}$ that accepts a relaxed implementation with input Π iff it defines a model of φ (Lemmata 1 and 2).
- **Adjusting for white-box processes.** In a third step, we construct an alternating automaton $\mathcal{C}$ that accepts an $\mathcal{O}_b$-labeled 2^{Π}-tree iff the $\mathcal{V}$-labeled 2^{Π}-tree obtained by adding the decisions of the white-box processes is accepted by $\mathcal{B}$ (Lemma 3).
- **Incomplete information.** We then transform $\mathcal{C}$ into an alternating automaton $\mathcal{D}$ that accepts an $\mathcal{O}_b$-labeled 2^{I_b}-tree iff its suitable widening is accepted by $\mathcal{C}$ (Lemma 4). In the last step, we construct a nondeterministic tree automaton $\mathcal{E}$ with $\mathcal{L}(\mathcal{E}) = \mathcal{L}(\mathcal{D})$ (Lemma 5).

From Specifications to Automata. AMC formulas can be transformed to equivalent automata over concurrent game structures.

Theorem 1. *[15] Given an alternating-time μ-calculus specification φ with n subformulas, we can construct an ACG $\mathcal{A}$ with $O(n^2)$ states and $O(n)$ colors, which accepts exactly the models of φ.*

From Models to Implementations. The transformation of $\mathcal{A}$ into an alternating tree automaton that accepts a relaxed implementation iff it defines a model of φ consists of two steps: We first restrict for each process a the set of possible decisions to the fixed set $\mathcal{O}_a$ (Lemma 1) and then ensure that the label of each node reflects the preceeding decisions of the processes (Lemma 2).

Lemma 1. *For ACG $\mathcal{A} = (2^{\Pi}, Q, q_0, \delta, \alpha)$ and an architecture $\mathbf{A} = (A, B, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ we can construct an alternating automaton $\mathcal{A}'$*

$= (2^{\Pi} \times \mathcal{V}, Q, q_0, \delta', \alpha)$ *that accepts a tree* $\langle (2^{\Pi})^*, l \times \bigoplus_{a \in A} p_a \rangle$ *iff the concurrent game structure* $\mathcal{G} = (A, \Pi, S, s_0, l, \{\alpha_a\}_{a \in A}, \Delta, \tau)$ *with* $\Delta = \bigoplus_{a \in A} p_a$ *and* $\tau : (s, d) \mapsto s \cdot d$ *is accepted by* $\mathcal{A}$*.*

Proof. Since the potential decisions of the processes are determined by the (relaxed) implementation, the universal and existential atoms can be resolved by boolean combinations of concrete directions.

We obtain $\delta'(q, (V, \bigoplus_{a \in A} \mathbf{O}_a))$ by resolving the $\forall\exists$ and $\exists\forall$ semantics of universal and existential atoms in $\delta(q, V)$ in the following way:

- Each occurrence of $(q', (A', \Box))$ in $\delta(q, V)$ is replaced by $\bigvee_{\bigoplus_{a \in A'} O'_a \in \bigoplus_{a \in A'} \mathbf{O}_a} \bigwedge_{\bigoplus_{a \in A \setminus A'} O'_a \in \bigoplus_{a \in A \setminus A'} \mathbf{O}_a} (q', \bigcup_{a \in A} O'_a)$.
  The outer disjunction refers to the fact that the agents in A' first choose a direction in accordance with the enabled directions in the current state. The inner conjunction refers to the counter choice made by the agents in $A \setminus A'$.
- Likewise, each occurrence of $(q', (A', \Diamond))$ in $\delta(q, V)$ is replaced by $\bigwedge_{\bigoplus_{a \in A \setminus A'} O'_a \in \bigoplus_{a \in A \setminus A'} \mathbf{O}_a} \bigvee_{\bigoplus_{a \in A'} O'_a \in \bigoplus_{a \in A'} \mathbf{O}_a} (q', \bigcup_{a \in A} O'_a)$. □

Let $\langle \Upsilon^*, dir \rangle$ denote the Υ-labled Υ-tree with $dir(y \cdot \upsilon) = \upsilon$ for all $y \in \Upsilon^*$ and $\upsilon \in \Upsilon$, and $dir(\varepsilon) = \upsilon_0$ for some predefined $\upsilon_0 \in \Upsilon$.

Lemma 2. *[17] Given an alternating automaton* $\mathcal{A}' = (\Upsilon \times \Sigma, Q, q_0, \delta, \alpha)$ *over* $\Upsilon \times \Sigma$*-labeled* Υ*-trees, we can construct an alternating automaton* $\mathcal{B} = (\Sigma, Q \times \Upsilon, q'_0, \delta', \alpha')$ *over* Σ*-labeled* Υ*-trees such that* $\mathcal{B}$ *accepts a tree* $\langle \Upsilon^*, l \rangle$ *iff* $\mathcal{A}'$ *accepts* $\langle \Upsilon^*, dir \times l \rangle$*.* □

Adjusting for White-box Processes. In this step, we eliminate the trees that are inconsistent with the decisions of the white-box processes. These decisions are represented by the $\bigoplus_{w \in W} \mathcal{O}_w$ fraction of the label. We assume that the implementations $\{p_w\}_{w \in W}$ of the white-box processes are represented as a deterministic Moore machine with output alphabet $\bigoplus_{w \in W} \mathcal{O}_w$. We construct an automaton that simulates the behavior of this Moore machine, replacing the $\bigoplus_{w \in W} \mathcal{O}_w$ fraction of the label with the output of the Moore machine. The state space of this automaton is linear in the state space of the original automaton and in the state space of the Moore machine, while the set of colors remains unchanged.

Lemma 3. *[18] Given an alternating automaton* $\mathcal{B} = (\Sigma \times \Xi, Q, q_0, \delta, \alpha)$ *over* $\Sigma \times \Xi$*-labeled* Υ*-trees and a deterministic Moore machine* $\mathcal{O}$ *with set* O *of states and initial state* $o_0 \in O$ *that produces a* Ξ*-labeled* Υ*-tree* $\langle \Upsilon^*, l \rangle$*, we can construct an alternating automaton* $\mathcal{C} = (\Sigma, Q \times O, (q_0, o_0), \delta', \alpha')$ *over* Σ*-labeled* Υ*-trees, such that* $\mathcal{C}$ *accepts* $\langle \Upsilon^*, l' \rangle$ *iff* $\mathcal{B}$ *accepts* $\langle \Upsilon^*, l'' \rangle$ *with* $l'' : y \mapsto (l'(y), l(y))$*.*
If $\mathcal{B}$ *is a nondeterministic automaton, so is* $\mathcal{C}$*.* □

Incomplete Information. The output of the black-box process b may only depend on the input I_b visible to b. For a set $\Xi \times \Upsilon$ of directions and a node

$x \in (\Xi \times \Upsilon)^*$, $hide_\Upsilon(x)$ denotes the node in Ξ^* obtained from x by replacing (ξ, υ) by ξ in each letter of x. For a Σ-labeled Ξ-tree $\langle \Xi^*, l \rangle$ we define the Υ-widening of $\langle \Xi^*, l \rangle$, denoted by $widen_\Upsilon(\langle \Xi^*, l \rangle)$, as the Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l' \rangle$ with $l'(x) = l(hide_\Upsilon(x))$.

Lemma 4. *[17] Given an alternating automaton $\mathcal{C} = (\Sigma, Q, q_0, \delta, \alpha)$ over Σ-labeled $\Xi \times \Upsilon$-trees, we can construct an alternating automaton $\mathcal{D} = (\Sigma, Q, q_0, \delta', \alpha)$ over Σ-labeled Ξ-trees, such that $\mathcal{D}$ accepts $\langle \Xi^*, l \rangle$ iff $\mathcal{C}$ accepts $widen_\Upsilon(\langle \Xi^*, l \rangle)$.* □

The resulting alternating automaton can be transformed into an equivalent nondeterministic automaton.

Lemma 5. *[9,19] Given an alternating automaton $\mathcal{D}$ with n states and c colors, we can construct an equivalent nondeterministic automaton $\mathcal{E}$ with $n^{O(c \cdot n)}$ states and $O(c \cdot n)$ colors.* □

3.3 Realizability in Hierarchical Architectures

For a hierarchical architecture $\mathbf{A} = (A, B, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$, the linear informedness relation $\preceq\ = \{(b, b') \in B \times B \mid I_b \subseteq I_{b'}\}$ partitions the black-box processes B into equivalence classes and defines an order on them. If $\preceq$ defines n different equivalence classes, we say that $\mathbf{A}$ has n levels of informedness. We define an ordering function $o : \mathbb{N}_n \to 2^B$, which maps each natural number $i \in \mathbb{N}_n$ to the set of i-th best informed black-box processes. For convenience, we use $\mathcal{O}_i = \bigoplus_{b \in o(\{i,\ldots,n\})} \mathcal{O}_b$ and $I_i = I_b$ for $b \in o(i)$.

The Algorithm. We start by applying the transformations discussed in the previous subsection (Theorem 1 and Lemmata 1 through 3) to construct a tree automaton $\mathcal{C}_0$ that accepts a set of relaxed implementations $P_0 = \{p_b\}_{b \in B}$ (with input Π) iff $P = P_W \cup P_0$ satisfies φ.

Then, we stepwise eliminate the processes in decreasing order of informedness. We successively construct:

- The alternating automaton $\mathcal{D}_i$ that accepts a $\mathcal{O}_i$-labeled 2^{I} -tree iff its widening is accepted by $\mathcal{C}_{i-1}$ (Lemma 4).

  A set $P_i = \{p_b^i \mid b \in B_i\}$ of relaxed implementations with input I_i for the processes in $B_i = o(\{i, \ldots, n\})$ is accepted by $\mathcal{D}_i$ iff there is a set $\overline{P}_i = \{\overline{p}_b^i \mid b \in \overline{B}_i\}$ of implementations for the processes in $\overline{B}_i = o(\mathbb{N}_{i-1})$, such that $P_W \cup P_i \cup \overline{P}_i$ satisfies φ.
- The nondeterministic automaton $\mathcal{E}_i$ with $\mathcal{L}(\mathcal{E}_i) = \mathcal{L}(\mathcal{D}_i)$ (Lemma 5); and
- The nondeterministic automaton $\mathcal{C}_i$ that accepts an $\mathcal{O}_{i+1}$-labeled I_i-tree iff it can be extended to an $\mathcal{O}_i$-labeled I_i-tree accepted by $\mathcal{C}_i$ (Lemma 6).

Narrowing and nondeterminization have been discussed in the previous section, and language projection is a standard operation on nondeterministic automata.

Lemma 6. *Given a nondeterministic automaton* $\mathcal{E} = (\Sigma \times \Xi, Q, q_0, \delta, \alpha)$ *that runs on* $\Sigma \times \Xi$*-labeled* Υ*-trees, we can construct a nondeterministic automaton* $\mathcal{C} = (\Sigma, Q, q_0, \delta', \alpha)$ *that accepts a* Σ*-labeled* Υ*-tree* $\langle \Upsilon^*, l_\Sigma \rangle$ *iff there is a* $\Sigma \times \Xi$*-labeled* Υ*-tree* $\langle \Upsilon^*, l_\Sigma \times l_\Xi \rangle$ *accepted by* $\mathcal{E}$ *with* $\langle \Upsilon^*, l \rangle = proj_\Sigma(\langle \Upsilon^*, l_\Xi \rangle)$.

Proof. $\mathcal{C}$ can be constructed by using δ' to guess the correct tree: we set $\delta' : (q, \sigma) \mapsto \bigvee_{\xi \in \Xi} \delta(q, (\sigma, \xi))$. □

We check realizability by solving the emptiness game for $\mathcal{E}_n$. This step can be extended to the synthesis of implementations $\{p_b\}_{b \in B}$ of the black-box processes.

3.4 Synthesis

The specification is realizable iff player *accept* has a winning strategy in the emptiness game of $\mathcal{E}_n$. From this strategy we obtain by projection a family of implementations $P_n = \{p_a \mid a \in o(n)\}$ for the least-informed processes.

In increasing order of informedness, we obtain implementations for the other processes: After computing implementations for the processes in $o(\{i{+}1, \ldots, n\})$, they are represented as Moore machines. Using Lemma 3, we then construct from $\mathcal{E}_i$ a nondeterministic automaton $\mathcal{F}_i$ that accepts those implementations $\widehat{P}_i$ for the processes in $o(i)$ for which there exists a set of implementations $\overline{P}_{i-1} = \{p_a \mid a \in o(\mathbb{N}_{i-1})\}$ such that $\overline{P}_{i-1} \cup \widehat{P}_i \cup P_{i+1}$ satisfies φ. $\mathcal{F}_i$ is non-empty by construction. From the winning strategy for player *accept* we obtain by projection a family of implementations $P' = \{p_a \mid a \in o(i)\}$ and set P_i to $P' \cup P_{i+1}$.

Theorem 2. *The distributed synthesis problem for an architecture* **A** *with* n *levels of informedness, a specification* φ *given as an AMC formula, and a family* $P_W = \{p_w\}_{w \in W}$ *of implementations of the white-box processes can be solved in time* n*-exponential in the number of subformulas of* φ.

Proof. The specification φ is realizable for an architecture **A** and a given set $\{p_w\}_{w \in W}$ of white-box strategies iff $\mathcal{E}_n$ is not empty. The construction of $\mathcal{E}_n$ involves one transformation of an alternating automaton to a nondeterministic automaton for each $i \in \mathbb{N}_n$, and therefore takes n-exponential time in the number of subformulas of φ. The size of each nondeterministic automaton $\mathcal{F}_i$ is linear in the size of $\mathcal{E}_i$ and the size of the Moore machines for the strategy of the less-informed processes. Each step along the order of informedness therefore again takes n-exponential time. □

The upper bounds for ATL, CTL* and ATL* follow from linear translations to alternation-free AMC [13], exponential translations to the μ-calculus [20], and doubly exponential translations to AMC [21,13], respectively. μ-calculus and CTL form a syntactical subset of AMC and ATL, respectively.

Corollary 1. *The distributed synthesis problem for an architecture* **A** *with* n *levels of informedness and a specification* φ *can be performed in time* n*-exponential in the length of* φ *for specifications in CTL, ATL, or the classical*

μ-calculus, $(n+1)$-exponential in the length of φ for specifications in CTL, and $(n+2)$-exponential in the length of φ for specifications in ATL*.*

A matching nonelementary lower bound (for LTL formulas and pipelines[2]) is provided in [7].

4 Completeness

In the previous section we showed that the linearity requirement on the informedness relation is a sufficient condition for the decidability of an architecture. In this section, we show that the condition is also necessary: we prove that, for non-linear informedness relations, the synthesis problem is already undecidable for the sublogic CTL.

The proof is a variant of the reduction of the synthesis problem for deterministic implementations to the halting problem in [7,9]. In the following we give a brief sketch of this argument before discussing the extension to nondeterministic strategies. In the simplest case, shown in Figure 3a, there are two processes p and q, such that the input i_p and the output o_p of process p is invisible to process q, and, vice versa, i_q and o_q are invisible to p. For a given deterministic Turing machine M, the conjunction ψ_M of the following conditions is realizable iff M halts on the empty input tape:

- The environment can send a *start* signal through i_p and i_q.
- Initially, p and q output the terminal state of M.
- Upon receiving the first *start* signal, p (q) starts to output syntactically legal configurations of M such that
  - the first two configurations are correct, and
  - whenever p and q output two configurations C_p and C_q, such that C_p (C_q) is the successor configuration of C_q (C_p), then the next configurations emitted by p and q have the same property.
- p and q always eventually output the terminal state of M.

For the more complicated case that the processes have access to each other's output (Figure 3b), ψ_M is extended to describe a two-phase protocol: On the input variables, a *start* symbol may be transmitted in the first phase, and an XOR key is sent in the second phase. The output variables are again used to emit sequences of configurations of Turing machines. In the first phase, the output is constantly set to true, and in the second phase it is encrypted by the last received XOR key. In this way, the processes cannot infer the decrypted meaning of the output from the other process, even if they have access to each other's output [9].

[2] For linear-time specifications we can restrict our attention w.l.o.g. to deterministic implementations. In this case, the processes at the beginning of the pipeline have (implicit) knowledge of the output produced by processes later in the pipeline [9]. Turning this knowledge into explicit input does not change the nonelementary complexity.

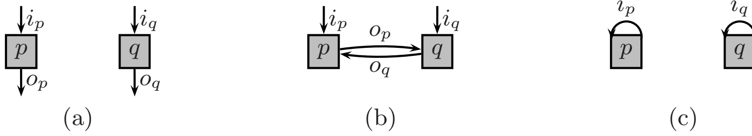


Fig. 3. Three undecidable situations: an architecture is undecidable if it contains two processes with incomparable sets of inputs

We now extend this argument to prove the undecidability of the synthesis problem for nondeterministic strategies and architectures with non-linear informedness relation. In addition to the architectures considered above, we take into account the situation where the two processes do not receive any input from an external environment (Figure 3c). In this case, we specify that the *start*-symbols and XOR keys are chosen completely nondeterministically during the first and second phase. The configurations of the Turing machine are emitted in a separate third phase, where the values of the output variables are specified to be deterministic.

Theorem 3. *The synthesis problem for CTL specifications is undecidable for all architectures with two black-box processes $b, p \in B$ with incomparable sets of input variables ($I_p \not\supseteq I_q \not\supseteq I_p$).*

Proof. The halting problem is reduced to the synthesis problem as follows. W.l.o.g. we fix one input variable for p and q that is invisible to the other process ($i_p \in I_p \setminus I_q$ and $i_q \in I_q \setminus I_p$) and two output variables $o_p \in O_p$ and $o_q \in O_q$. We extend the CTL specification ψ_M (from Theorem 5.3 of [9]) to describe the following three-phase communication pattern:

- A *start* symbol can be transmitted to p and q through i_p and i_q in a first phase.
- A one bit XOR key is transmitted to p and q through i_p and i_q in a second phase.
- p and q output an encoded bit of their output sequence in a third phase.

We extend the specification with the following guarantees:

- Exactly in every third round (and in the third round from the beginning) the values of the variables o_p and o_q are fixed deterministically.
  In the remaining rounds they are set nondeterministically to *true* and *false*.
- The variables in $\{i_p, i_q\} \setminus \{o_p, o_q\}$ are set deterministically to *true* in every third round (and in the third round from the beginning).
  In the remaining rounds they are set nondeterministically to *true* and *false*.
- To rule out the influence of the remaining variables, we require that they are always set to *true*.

If the white-box strategies are chosen consistently with the specification, the synthesis problem has a solution iff M halts on the empty input tape. □

5 Conclusions

This paper provides a comprehensive solution to the distributed synthesis problem for alternating-time temporal logics. The synthesis problem is decidable if and only if the architecture is hierarchical. Our synthesis algorithm is uniformly applicable to all decidable architectures and all specification logics in the range from CTL to the alternating-time μ-calculus.

The central technical innovation is the treatment of nondeterministic implementations. We encode nondeterministic implementations as (deterministic) choice-set trees. This allows us to represent sets of strategies with tree automata and to distribute the global specification over the distributed architecture using standard automata transformations.

Nondeterministic implementations are also of interest if the specification is expressed in a standard branching-time logic like CTL*. In this case, nondeterminism means abstraction: details regarding the interaction with the external environment (including the user) can be omitted, since existential requirements can be demonstrated without immediately establishing the protocol. The resolution of the nondeterminism is moved to later design phases, where, in divide-and-conquer fashion, only a single nondeterministic component needs to be considered at a time.

References

1. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Intl. Congr. Math., Upsala, pp. 23–25 (1963)
2. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
3. Wolper, P.: Synthesis of Communicating Processes from Temporal-Logic Specifications. PhD thesis, Stanford University (1982)
4. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable concurrent program specifications. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) Proc. 16th Int. Colloquium on Automata, Languages and Programming. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
5. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: ACM (ed.) Proc. POPL, pp. 179–190. ACM Press, New York (1989)
6. Kupferman, O., Vardi, M.Y.: μ-calculus synthesis. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 497–507. Springer, Heidelberg (2000)
7. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. FOCS, pp. 746–757. IEEE Computer Society Press, Los Alamitos (1990)
8. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: Proc. LICS, pp. 389–398. IEEE Computer Society Press, Los Alamitos (2001)
9. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proc. LICS, pp. 321–330. IEEE Computer Society Press, Los Alamitos (2005)
10. Kremer, S., Raskin, J.F.: A game-based verification of non-repudiation and fair exchange protocols. Journal of Computer Security 11(3), 399–430 (2003)
11. Mahimkar, A., Shmatikov, V.: Game-based analysis of denial-of-service prevention protocols. In: IEEE Computer Security Foundations Workshop, pp. 287–301 (2005)

12. Kremer, S.: Formal Analysis of Optimistic Fair Exchange Protocols. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium (2003)
13. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM 49(5), 672–713 (2002)
14. van Drimmelen, G.: Satisfiability in alternating-time temporal logic. In: Proc. LICS, pp. 208–217. IEEE Computer Society Press, Los Alamitos (2003)
15. Schewe, S., Finkbeiner, B.: The alternating-time μ-calculus and automata over concurrent game structures. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 591–605. Springer, Heidelberg (2006)
16. Wilke, T.: Alternating tree automata, parity games, and modal μ-calculus. Bull. Soc. Math. Belg. 8(2) (2001)
17. Kupferman, O., Vardi, M.Y.: Church's problem revisited. The bulletin of Symbolic Logic 5(2), 245–263 (1999)
18. Finkbeiner, B., Schewe, S.: Semi-automatic distributed synthesis. In: Peled, D.A., Tsay, Y.K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 263–277. Springer, Heidelberg (2005)
19. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin. McNaughton and Safra. Theor. Comput. Sci. 141(1-2), 69–107 (1995)
20. Bhat, G., Cleaveland, R.: Efficient model checking via the equational μ-calculus. In: Proc. LICS, pp. 304–312. IEEE Computer Society Press, Los Alamitos (1996)
21. de Alfaro, L., Henzinger, T.A., Majumdar, R.: From verification to control: Dynamic programs for omega-regular objectives. In: Proc. LICS, pp. 279–290. IEEE Computer Society Press, Los Alamitos (2001)

Timeout and Calendar Based Finite State Modeling and Verification of Real-Time Systems

Indranil Saha, Janardan Misra, and Suman Roy

HTS (Honeywell Technology Solutions) Research Lab,
151/1 Doraisanipalya, Bannerghatta Road, Bangalore 560 076, India
{indranil.saha,janardan.misra,suman.roy}@honeywell.com

Abstract. To overcome the complexity of verification of real-time systems with dense time dynamics, Dutertre and Sorea proposed timeout and calender based transition systems to model real-time systems and verify safety properties using k-induction. In this work, we propose a canonical finitary reduction technique, which reduces the infinite state space of timeout and calender based transition systems to a finite state space. The technique is formalized in terms of clockless finite state timeout and calendar based models represented as predicate transition diagrams. Using the proposed reduction, we can verify these systems using finite state model checkers and thus can avoid the complexity of induction based proof methodology. We present examples of Train-Gate Controller and the TTA startup algorithm to demonstrate how such an approach can be efficiently used for verifying safety, liveness, and timeliness properties using the finite state model checker Spin.

1 Introduction

Modeling and verification of timeout based real-time systems with continuous dynamics is an important and hard problem that has evoked a lot of prime research interest with industrial focus for many years in the recent past. The problem of faithfully modeling and consequently formally verifying such timeout based real-time systems is rather difficult because the state space of these systems is essentially infinite owing to the diverging valuation required by the timing and timeout variables. Because of this infiniteness of the state space none of the known formal verification techniques can be applied to completely verify some of the interesting properties, e.g., liveness properties, timing deadlocks etc. Although infinite state model checkers like SAL (Symbolic Analysis Laboratory) [10] have been used with limited success for verifying safety properties. The verification process employed by these tools demands significant additional manual efforts in defining supporting lemmas and abstractions for scaling up the model.

Spin [9] is a tool for automatically verifying finite state distributed systems. There are broadly two attempts for extending Spin with time [4,5,16]. Real-time extension of Spin (RT-Spin [16]) is one such work, which provides timed automata (TA) [1] with real-valued clocks as a modeling framework, though is

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 284–299, 2007.
© Springer-Verlag Berlin Heidelberg 2007

incompatible with the partial order reduction implementation of Spin. Another is the work on DT-Spin [4,5], which allows one to quantify (discrete) time elapsed between events, by specifying the time slice in which they occur. DT-Spin is compatible with the partial order reduction of Spin and has been used to verify industrial protocols, e.g., AFDX Frame management protocol [13] and TTCAN [14]. Nonetheless, systems with asynchronous communication with bounded delays between components cannot be modeled directly by using the mechanism of asynchronous channels that Spin provides since there is no explicit provision to capture message transmission delays. One possibility is to model each channel as a separate process with delay as a state variable. In [4], the channels in the example of PAR protocol have been implemented in the same way. But for systems with relatively large number of components and dense connectivity among the components, modeling channels in this way is difficult and state space explosion becomes an unavoidable problem. UPPAAL [2], which can model TA, has the same limitation when modeling asynchronous communications with bounded delays - every channel has to be modeled as a separate TA capturing the message transmission delays.

Dutertre and Sorea [6] proposed timeout based modeling of time triggered systems with dense time dynamics, which have been traditionally used as a model of execution in discrete event system simulations. They presented a modeling approach, where expected delivery delays for all undelivered messages can be stored in a global data structure called *calendar* [6,7]. Formally, a *calendar* is a set of bounded size of the form $C = \{\langle e_1, t_1\rangle, \ldots, \langle e_r, t_r\rangle\}$, where each event e_i is associated with the time point t_i when it is scheduled to occur. The calendar based model along with the timeouts for individual processes has been used to model the TTA startup protocol [7]. Using the infinite bounded model checker of SAL [10], they proved the safety property by k induction. Unfortunately, not all of the safety properties are inductive in nature and therefore may require support of auxiliary lemmas. In [7], proof of the safety property for the TTA startup having just 2 nodes itself required 3 additional lemmas. A verification diagram based abstraction method proposed in [12], was used to prove the invariant property for models having upto 10 nodes. However, liveness properties still remain beyond the scope of this approach. Pike [11] builds on the work of [6] and proposes a new formalism called Synchronizing Timeout Automata (STA) to reduce the induction depth k required for k-induction. STA is defined using shared timeouts such that the resulting transition system does not involve a clock.

Since in timeout and calendar based models, global time and timeouts always increase, such models cannot be directly used for finite state verification. To that end, we propose a finitary reduction technique which effectively reduces the infinite state timeout and calendar based transition systems with discrete dynamics to finite state transition systems. This technique enables us to model a real-time system without considering a clock explicitly. We formalize the timeout and calendar based models as predicate transition diagrams and their behavior in terms of timeout and calendar based transition systems. Such a formal modeling

framework provides background to effectively reason about the correctness of the various possible hypotheses for efficiently verifying these models beyond limited experiments. We demonstrate by examples, how such a modeling approach can be efficiently used for verifying safety, liveness, and timeliness properties using the finite state model checker Spin.

The remainder of the paper is organized as follows: section, In Section 2, we describe the finitary reduction technique and formalize it in terms of clockless modeling in Section 3. In Section 4 we discuss models of time and executablitiy conditions for dense time model. Section 5 presents the experimental results followed by concluding discussion in section 6.

2 Finitary Reduction

With reference to the timeout and calendar based modeling presented in [6,7], notice that although these models can be used to efficiently capture dense time semantics without using a continuously varying clock, it is difficult to use these models for finite state model checking. The difficulty arises because of the fact that the valuations for the global clock t and the timeout variables in $\mathcal{T}$ diverge and thus are not bounded by a finite domain. Unlike TA one cannot reset the global clock or the individual timeouts in these models because straightforward attempts for such resetting results only in incorrect behaviors. One possible solution may be to bound the value of the global clock and the timeouts by appropriate large constants based upon the system specification. But such a upper bound is quite difficult to estimate in case of practical industrial applications and also with such an approach liveness properties cannot be verified.

We propose a finitary reduction technique, which is formalized in terms of clockless modeling and semantics in the next section. This technique effectively reduces the timeout and calendar based transition systems with discrete dynamics into finite state systems, which, in turn, can be expressed and model checked by finite state model checkers.

Informally, the technique can be described as follows: To implement time progress transition, a special process is required to increase the global clock to the minimum of timeouts, when each of the timeout values is strictly greater than the current value of the clock. Other processes wait until their timeouts are equal to the global clock, and when it is so, they take the discrete transitions and updates their timeouts in future. We propose to model the special process which is responsible for time progress transition in such a way that it does not explicitly use the clock variable and prevents the timeout variables to grow infinitely. We call this process *time_progress*. When no discrete transition is possible in the system due to the fact that the discrete transitions for all the systems are scheduled in the future, *time_progress* finds out the minimum of all the timeouts in $\mathcal{T}$ and scales down all these timeouts by the minimum. In this way at least one of the timeouts becomes zero. A process is allowed to take a discrete transition when its timeout becomes zero. When it happens the process updates its timeout and does other necessary jobs.

If the timeouts are always incremented by finite values then it is guaranteed that the value of a timeout will always be in a finite domain. But there are cases when a timeout increment cannot be bounded by finite value. For example, a process may have to wait for an external signal before its next discrete transition. In this case, next discrete transition of the process does not depend on its own timeout, so the timeout of the process is set to the relatively large value, so that it does not affect the next discrete transitions of other processes. In another situation, it may be desired that the next discrete transition of a process may happen at any time in the future, for example, the process may be in a sleeping mode and can wake up at any future point of time. In that case all what we need is to limit the value of the timeout without omitting any of the possible interleaving of the process steps. To do that we limit the timeout value in $[0, M+1]$, where M is the maximum of all the integer constants that are used to define the upper limit of different timeouts for different processes in the system.

The suggested technique gives rise to a canonical representation of the clock and timeout valuations in any state in the sense that for the timeout and calendar based models considered here, there cannot be any further reduction possible without actually loosing the relative timing delay information. This is because this technique effectively reduces timeout valuations into a canonical partial ordering structure and also simultaneously keeps the information on the actual timeout increments intact. This approach can be seamlessly extended for the calendar based models as well.

It should be added that the finitary reduction considered in this work is effective only under discrete dynamics since with dense modeling such a reduction though reduces an infinite region (e.g., $\mathcal{R}^n$) to a finitely bounded region (e.g., $[0,1]^n$), it would still contain infinitely many points resulting into infinite permissible paths.

Above discussion is formalized in terms of "clockless" modeling and associated semantics in the next section.

3 Timeout and Calendar Based Clockless Models

In this section we provide a formalization of timeout and calendar based clockless models as predicate transition diagrams and associated semantics in terms of state transition systems.

3.1 Timeout Based Models: Clockless Modeling

Syntax. The Timeout based Model (ToM) ([6]) can be represented as

$$P: \{\theta\}[P_1||P_2||\dots||P_n],$$

Where each process P_i is a sequential non-deterministic process having τ_i as its local timeout and $\mathcal{X}_i$ as a set of local timing variables used for determining the

relative delay between events. "||" is the parallel composition operator. Formula θ restricts the initial values of variables in

$$\mathcal{U} = \mathcal{T} \cup \mathcal{X} \cup Var,$$

where the set of all timeouts is $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and $\mathcal{X} = \bigcup_i \mathcal{X}_i$. $Var = G \cup L_1 \cup L_2 \cup \ldots \cup L_n$ is the set of other state variables assuming values from finite domains. Variables in G are globally shared among all the processes while L_i contains variables local to process P_i. f^{Var} is the set of computable functions on Var.

Each process P_i is represented using a *predicate transition diagram*, which is a finite directed graph with nodes $Loc_i = \{l_0^i, l_1^i, \ldots, l_{m_i}^i\}$, called *locations.* The entry location is l_0^i. There are two kinds of edges in the graph of a process P_i: *Timeout edges* and *Synchronous Communication edges.* Edge definitions involve an enabling condition or guard ρ, which is a boolean-valued function or a predicate.

Timeout Edges: A timeout edge $(l_j^i, \rho \Rightarrow \langle \tau_i := update_i, \eta, f\rangle, l_k^i)$ in the graph of the process P_i is represented as

$$l_j^i \stackrel{\rho \Rightarrow \langle \tau_i := update_i, \eta, f\rangle}{\longrightarrow} l_k^i,$$

where $update_i$ specifies how timeout τ_i is to be updated on taking a transition on the edge when guard ρ evaluates to True. $\eta \subseteq \mathcal{X}_i$ specifies the local timing variables which capture the relative increment in the value of timeout τ_i while taking transition on the edge. $f \in f^{Var}$ manipulates the state variables in $G \cup L_i$.

$update_i$ is defined using the rule: $update_i = k_1 \mid k_2 \mid \infty \mid \max(\mathcal{M})$, where $l - z \prec k_1 \prec' m - z'$, $\prec, \prec' \in \{<, \leq\}$ and $k_2 \succ l - z$, $\succ \in \{>, \geq\}$; $z, z' := w|0$ and $l, m \in \mathcal{N}_0$ are non-negative integer constants. $\mathcal{M}$ is the set of all the integer constants that are used to define the upper limit of different timeouts for different processes in the system. $\max(\mathcal{M})$ returns the maximum of all the integers in $\mathcal{M}$.

Constraints on k_1, k_2 specify how the new value of timeout τ_i should be determined based upon the value of some local timing variable w, which would have captured the increments in the value of timeout τ_i in some earlier transitions. Setting a timeout to ∞ is used to capture the requirement of indefinite waiting for an external signal/event. Setting the timeout value using $\max(\mathcal{M})$ is used to capture the situation where the next discrete transition of a process may happen at any time in the future, for example, the process may be in a sleeping mode and can wake up at any future point of time.

Synchronous Communication Edges: As rendezvous communication between a pair of processes (P_s, P_r) is represented by having an edge pair (e_s, e_r) s.t. $e_s \in P_s$ and $e_r \in P_r$:

$$e_s : l_j^s \stackrel{\rho \Rightarrow \langle ch!m, \tau_s := update_s, \eta, g\rangle}{\longrightarrow} l_k^s$$

$$e_r : l_j^r \stackrel{True \Rightarrow \langle ch?\bar{m}, \tau_i := update_r, \eta', h\rangle}{\longrightarrow} l_k^r$$

where ch is the channel name, $m \in L_s$ is the message sent, and $\bar{m} \in L_r$ receives the message; $g, h \in f^{Var}$.

Semantics. With a given ToM

$$P : \ \{\theta\}[P_1||P_2||\ldots||P_n]$$

we associate the following transition system $S_P = (\mathcal{V}, \Sigma, \Sigma_0, \Gamma)$, which will be referred to as a *timeout based clockless transition system* :

1. $\mathcal{V} = \mathcal{U} \cup \{\pi_1, \ldots, \pi_n\}$. Each *control variable* π_i ranges over the set $Loc_i \cup \{\bot\}$. The value of π_i indicates the location of the control for the process P_i and $\bot$ denotes before the start of the process.
2. Σ is the set of states. Every state $\sigma \in \Sigma$ is an interpretation of $\mathcal{V}$ such that, for $x \in \mathcal{V}$, $\sigma(x)$ is its value in state σ.
3. $\Sigma_0 \subseteq \Sigma$ is the set of initial states such that for every $\sigma_0 \in \Sigma_0$, θ is true in σ_0 and $\sigma_0(\pi_i) = \bot$ for each process P_i.
4. $\Gamma = \Gamma_e \cup \Gamma_+ \cup \Gamma_0 \cup \Gamma_{syn_comm}$ is the set of transitions. Every transition $\nu \in \Gamma$ is a binary relation on Σ defined further as follows:

Entry Transitions: Γ_e is the set of entry transitions and contains an *entry transition* ν_e^i for every process P_i. In particular $\forall \sigma_0 \in \Sigma_0$,

$$(\sigma_0, \sigma') \in \nu_e^i \Leftrightarrow \begin{cases} 1.\ \forall x \in \mathcal{U} :\ \sigma'(x) = \sigma_0(x) \\ 2.\ \forall \tau \in \mathcal{T} :\ \sigma'(\tau) \geq 0 \\ 3.\ \sigma_0(\pi_i) = \bot \ \text{ and } \sigma'(\pi_i) = l_0^i \end{cases}$$

Time Progress Transition: The first kind of edges $\nu_+ \in \Gamma_+$ are those where all the timeouts are decremented by the minimum of the current timeout values. In particular,

$$(\sigma, \sigma') \in \nu_+ \Leftrightarrow \begin{cases} 1.\ \min\{\sigma(\mathcal{T})\} > 0 \\ 2.\ \forall \tau \in \mathcal{T} :\ \sigma'(\tau) = \sigma(\tau) - \min\{\sigma(\mathcal{T})\} \\ 3.\ \forall x \in \mathcal{X} \cup Var :\ \sigma'(x) = \sigma(x) \\ 4.\ \forall i :\ \sigma'(\pi_i) = \ \sigma(\pi_i) \end{cases}$$

Timeout Increment Transition: If $(l_j^i, \rho \Rightarrow \langle update_i, \eta, f\rangle, l_k^i)$ is an edge in the predicate transition diagram for process P_i, then there is a corresponding edge $\nu_0^i \in \Gamma_0$:

$$(\sigma, \sigma') \in \nu_0^i \Leftrightarrow \begin{cases} 1.\ \rho \text{ holds in } \sigma \\ 2.\ \mathsf{If}\ \sigma(\tau_i) = 0 \ \ \mathsf{then} \\ \quad \sigma'(\tau_i) = update_i > 0 \ \ \mathsf{else} \ \ \sigma'(\tau_i) = \sigma(\tau_i) \\ 3.\ \forall x \in \eta :\ \sigma'(x) = \sigma'(\tau_i) + \sigma(x) \text{ and} \\ \quad \forall x \in \mathcal{X} \setminus \eta :\ \sigma'(x) = \sigma(x) \\ 4.\ \forall v \in G \cup L_i : \sigma'(v) = f(\sigma(v)) \text{ and} \\ \quad \forall v \in \ Var \setminus (G \cup L_i) :\ \sigma'(v) = \sigma(v) \\ 5.\ \sigma(\pi_i) = \ l_j^i \text{ and } \sigma'(\pi_i) = l_k^i \end{cases}$$

If $update_i = k_1$ s.t. $l - z \prec k_1 \prec m - z'$, $update_i$ nondeterministically selects an integer δ such that $l - \sigma(z) \prec \delta \prec m - \sigma(z')$. If $update_i = k_2$ s.t. $k_2 \succ l - z$, $update_i$ nondeterministically selects an integer δ such that $\delta \succ l - \sigma(z)$, else if $update_i = \infty$, it selects a relatively very large integer value and returns it to account for indefinite waiting. If $update_i = \max(\mathcal{M})$, $update_i$ nondeterministically selects any integer δ in $[0, M+1]$, where M is the maximum of all the integers in $\mathcal{M}$ returned by $\max(\mathcal{M})$.

Synchronous Communication: For a pair of processes P_s, P_r having edges (e_s, e_r) as defined before, $\nu^{sr}_{syn_comm} \in \Gamma_{syn_comm}$ exists such that:

$$(\sigma, \sigma') \in \nu^{sr}_{syn_comm} \Leftrightarrow \begin{cases} 1.\ \rho \text{ holds in } \sigma \\ 2.\ \sigma'(\tau_s) = update_s > \sigma(\tau_s) \\ \quad \sigma'(\tau_r) = update_r > \sigma(\tau_r) \\ 3.\ \forall x \in (\eta) :\ \sigma'(x) = \sigma'(\tau_s) + \sigma(x), \quad \text{and} \\ \quad \forall x \in (\eta') :\ \sigma'(x) = \sigma'(\tau_r) + \sigma(x) \ \text{ and} \\ \quad \forall x \in \mathcal{X} \setminus (\eta \cup \eta') :\ \sigma'(x) = \sigma(x) \\ 4.\ \sigma'(\bar{m}) = \sigma(m) \\ 5.\ \forall v \in G \cup L_s : \sigma'(v) = g(\sigma(v)), \quad \text{and} \\ \quad \forall v \in G \cup L_r : \sigma'(v) = h(\sigma(v)) \ \text{ and} \\ \quad \forall v \in Var \setminus (G \cup L_r \cup L_s) :\ \sigma'(v) = \sigma(v) \\ 6.\ \sigma(\pi_s) = l^s_j, \sigma(\pi_r) = l^r_j \text{ and} \\ \quad \sigma'(\pi_s) = l^s_k, \sigma'(\pi_r) = l^r_k \end{cases}$$

This semantic model defines the set of possible computations of the timeout system P as a set of state sequences (possibly infinite) starting with some initial state in Σ_0 and following edges in Γ.

Example: Train-Gate Controller

We will illustrate the timeout based model as formalized above using the example of the Train-Gate Controller (TGC) (adapted from [1].) The example of TGC demonstrates synchronous communication between system components, since the communications between Train and Controller, and between Controller and Gate are assumed to be synchronous.

TGC is an automatic controller that controls the opening and closing of a Gate at railroad crossing. The system is composed of three components: Train, Gate, and Controller. Before entering the railroad crossing the Train sends the signal *approach*. The Controller on receiving this signal is supposed to send the signal *lower* to the Gate within 10 time units and the Gate has to be down within another 10 time units. The Train can enter the crossing at any time after 20 time units since it sent the *approach* signal. While exiting the crossing the Train sends the *exit* signal to the Controller. The requirement is that after sending the *approach* signal the Train must send the *exit* signal within 50 time units. The Controller sends the *raise* signal to the Gate within 10 time units after it receives the *exit* signal. The Gate is required to be up within another

10 time units. All the communications are assumed to be synchronous, that is, there is no message transmission delay.

Figure 1 demonstrates the clockless timeout based model of TGC. The timing requirements are captured by suitably defining the *update* functions on the edges. For example, consider the edge (t_0, t_1) for the train labeled with $(\tau_t = 0) \Rightarrow \langle ch!approach, (\tau_t := k | 20 \leq k \leq 50), x\rangle$. Here $(\tau_t = 0)$ indicates that the system starts when train sends the *approach* signal over the shared channel *ch* and nondeterministically sets its timeout τ_t to some value k between $[20, 50]$ indicating that after sending the *approach* signal it can enter the crossing any time after 20 time units. Upper limit of 50 is used to indicate that the train cannot enter later than 50 time units because it is required that train has to indeed exit the crossing on or before 50 time units. Having spent k units of time in state t_1, train takes transition on the next timeout to state t_2 and resets its timeout to some value k' between $[0, 50 - k]$, which indicates that the train must exit (and send *exit* signal to the controller) from state t_2 no more than before it has spent at most total of 50 units of time in states t_1 and t_2, that is, $0 \leq k + k' \leq 50$. Similarly on taking a transition on edge from g_1 to g_2 for the gate, $\tau_g := \infty$ denotes that the Gate would be waiting for the signal *raise* in state g_2 to be received on channel ch_1 from the Controller.

3.2 Calendar Based Models: Clockless Modeling

Syntax. To capture (lossless) asynchronous communication with bounded message transfer delay, timeout based model is extended with a calendar data structure. A calendar is a linear array of bounded size, where each cell contains the following information: {message, sender_id, receiver_id, expected_delivery_time}. Let $\mathcal{C}$ to denote the calendar array, a globally shared object. We have

$$\mathcal{U} = \mathcal{T} \cup \mathcal{X} \cup Var \cup \mathcal{C}$$

Sending a message is represented in the predicate transition diagram of process P_i using the following edge:

$$l^i_j \overset{\rho \Rightarrow \langle send(m,i,R,\Lambda), \tau_i := update_i, \eta, f\rangle}{\longrightarrow} l^i_k,$$

where $send(..)$ specifies that a message m is to be sent to each of the processes P_r, where $r \in R \subseteq \{1, 2, \ldots n\}$, and with expected delivery time of $\lambda_r \in \Lambda$ for each P_r. On taking a transition on this edge an entry $\{\mathsf{m}, \mathsf{i}, \mathsf{r}, \lambda_r\}$ is added to $\mathcal{C}$ for each $r \in R$.

Corresponding receiving of the message is represented in the predicate transition diagram of each of the processes P_r $(\forall\ r \in R)$ using the following edge:

$$l^r_j \overset{True \Rightarrow \langle receive(m,i,r), \tau_r := update_r, \eta, g\rangle}{\longrightarrow} l^r_k,$$

where $receive(..)$ specifies that a message m sent by process P_i is to be received by the process P_r. When 'time' elapsed in terms of timeout increments approaches some expected delivery time λ_r as specified by the sender process in the calendar $\mathcal{C}$ for entry $e = \{\mathsf{m}, \mathsf{i}, \mathsf{r}, \lambda_r\}$, a transition is taken on this edge and the entry e is deleted from $\mathcal{C}$.

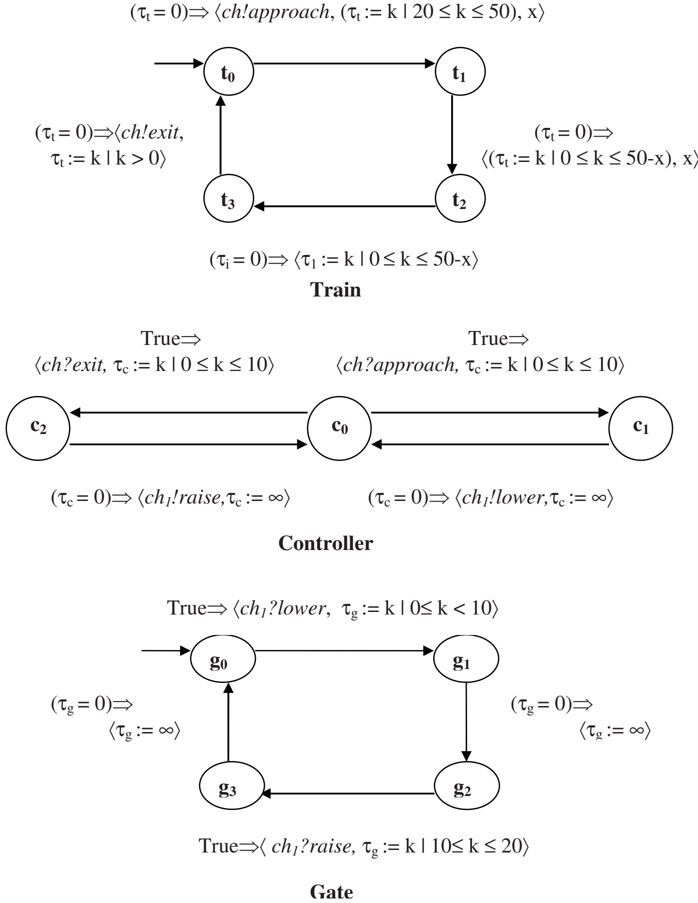


Fig. 1. Clockless model for Train-Gate Controller

Semantics. Given a calendar $\mathcal{C}$, we assume that the set of delays for all undelivered messages at any state σ can be extracted using function $\Delta : \sigma(\mathcal{C}) \rightarrow 2^{\mathcal{N}}$.

Let $\Gamma = \Gamma_e \cup \Gamma_+ \cup \Gamma_0 \cup \Gamma_{syn_comm} \cup \Gamma_{asyn_comm}$ denote the set of transitions in the *calendar based clockless transition system.* Γ_e (the set of Entry Transitions), Γ_{syn_comm} (Synchronous Communications) and Γ_0 (Timeout Increment Transitions) are defined in same way as in the timeout based model. The definition for the Time Progress Transition edges in Γ_+ are modified using calendar object $\mathcal{C}$ as follows:

Time Progress Transition: The edges ν_+ are redefined so that all the timeout and calendar delay entries are decremented by the minimum of all timeouts and the message delays in calendar. Let $\alpha = \min\{\sigma(\mathcal{T}) \cup \Delta(\sigma(\mathcal{C}))\}$,

$$(\sigma, \sigma') \in \nu_+ \Leftrightarrow \begin{cases} 1.\ \alpha > 0 \\ 2.\ \forall \tau \in \mathcal{T}:\ \sigma'(\tau) = \sigma(\tau) - \alpha \\ 3.\ \forall \lambda \in \Delta(\sigma(\mathcal{C})):\ \sigma'(\lambda) = \sigma(\lambda) - \alpha \\ 4.\ \forall z \in Var:\ \sigma'(z) = \sigma(z) \\ 5.\ \text{If } \exists\{\text{m, i, r, } \lambda_r\} \in \sigma(\mathcal{C}) \text{ such that } \alpha = \lambda_r \\ \qquad \text{then } \forall x \in \mathcal{X}_r:\ \sigma'(x) = \sigma(x) + \alpha \text{ and} \\ \qquad\qquad \forall x \in \mathcal{X} \setminus \mathcal{X}_r:\ \sigma'(x) = \sigma(x) \\ \quad \textbf{else } \forall x \in \mathcal{X}:\ \sigma'(x) = \sigma(x) \\ 6.\ \forall i:\ \sigma'(\pi_i) = \sigma(\pi_i) \end{cases}$$

We additionally define new transitions Γ_{asyn_comm} corresponding to $send()$ and $receive()$ to capture asynchronous communication:

Send Transition: If $(l^i_j, \rho \Rightarrow \langle send(m, i, R, \Lambda), update_i, \eta, f\rangle, l^i_k)$ is an edge in process P_i, then we have a corresponding edge ν^i_{send} which adds $|R|$ cells to the calendar array $\mathcal{C}$:

$$(\sigma, \sigma') \in \nu^i_{send} \Leftrightarrow \begin{cases} 1.\ \rho \text{ holds in } \sigma \\ 2.\ \text{If } \min\{\sigma(\mathcal{T})\} = \sigma(\tau_i) = 0 \\ \qquad \text{then } \sigma'(\tau_i) = update_i > 0 \\ \quad \text{else } \sigma'(\tau_i) = \sigma(\tau_i) \\ 4.\ \forall x \in \eta:\ \sigma'(x) = \sigma'(\tau_i) + \sigma(x) \text{ and} \\ \quad \forall x \in \mathcal{X} \setminus \eta:\ \sigma'(x) = \sigma(x) \\ 5.\ \forall v \in G \cup L_i: \sigma'(v) = f(\sigma(v)) \text{ and} \\ \quad \forall v \in Var \setminus (G \cup L_i):\ \sigma'(v) = \sigma(v) \\ 6.\ \forall r \in R: \sigma'(\mathcal{C}) := \sigma(\mathcal{C}) + \{\text{m, i, r, } \lambda_r\} \\ 7.\ \sigma(\pi_i) = l^i_j \text{ and } \sigma'(\pi_i) = l^i_k \end{cases}$$

Receive Transition: If $(l^r_j, True \Rightarrow \langle receive(m, i, r), \tau_r := update_r, \eta, g\rangle, l^r_k)$ is an edge in the graph of process P_r, then we have a corresponding edge $\nu^r_{receive} \in \Gamma_{asyn_comm}$, which deletes the entry $\{m, i, r, \lambda_r\}$ from the calendar array $\mathcal{C}$ when λ_r is 0:

$$(\sigma, \sigma') \in \nu^r_{receive} \Leftrightarrow \begin{cases} 1.\ \exists\{m, i, r, \lambda_r\} \in \sigma(\mathcal{C}) \text{ s.t. } \lambda_r = 0 \\ 2.\ \sigma'(\tau_r) = update_r > 0 \\ 3.\ \forall x \in \eta:\ \sigma'(x) = \sigma'(\tau_r) + \sigma(x) \text{ and} \\ \quad \forall x \in \mathcal{X} \setminus \eta:\ \sigma'(x) = \sigma(x) \\ 4.\ \forall v \in G \cup L_r: \sigma'(v) = f(\sigma(v)) \text{ and} \\ \quad \forall v \in Var \setminus (G \cup L_r):\ \sigma'(v) = \sigma(v) \\ 5.\ \sigma'(\mathcal{C}) := \sigma(\mathcal{C}) - \{m, i, r, \lambda_r\} \\ 6.\ \sigma(\pi_r) = l^r_j \text{ and } \sigma'(\pi_r) = l^r_k \end{cases}$$

Example: TTA Startup Algorithm

Above formalization of the calendar based model can be illustrated using the TTA startup algorithm. TTA startup executes on a logical bus meant for safety-critical applications in both automotive and aerospace industries. In a normal operation, N computers or nodes share a TTA bus using a TDMA schedule. The goal of the startup algorithm is to bring the system from the power-up state, in which all processors are unsynchronized, to the normal operation mode in which all processors are synchronized and follow the same TDMA schedule. For detailed understanding of startup protocol, we refer the reader to [15].

Figure 2 depicts the calendar based clockless predicate transition diagram of the ith node. In the TTA startup algorithm, all the communications are asynchronous and message delivery delays, which are finite and specified by the designer, have to be taken into account for correct operation of the protocol. τ_i^{listen} and τ_i^{cs} represent how much time a node spends in the *Listen* state and the *Coldstart* state respectively, if no external signal is received. τ^{round} denotes the time a node spends in the *Active* state before sending its next massage. $R = \{1, \ldots, N\} \setminus \{i\}$ represents that all the nodes except the sender i are required to receive the message in the network. λ_i's denote the message delivery time for the corresponding send events. In the TTA, message delivery times for all the receivers are considered to be the same, and that is why we have considered a single variable λ_i to represent that delay.

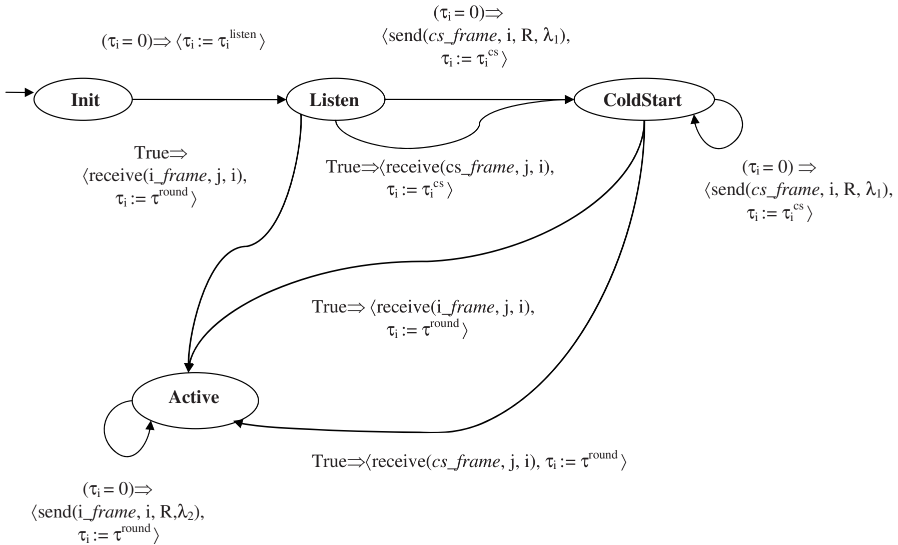


Fig. 2. Clockless model for the ith processor in TTA Startup algorithm

4 Models for Time

It remains unspecified as to what the underlying model of time is (for clock, timeouts, timing variables etc.) while defining the clockless semantics of the timeout and calendar based models.

The choice of dense ($\mathcal{R}^+$) domain versus discrete ($\mathcal{N}_0$) domain critically affects the size of the state space of the model. Indeed, with the dense time model, we need to add the following nonzenoness condition to ensure effective progress in the model: *There must not be infinitely many time progress (or timeout increment) transitions effectively within a finite interval.* Formally,

Nonzenoness: Clockless Semantics:

$$\neg[\exists\ \sigma_0\sigma_1\ldots \text{ s.t. } \exists\ \delta \in \mathcal{R}^+ \text{ and } \Sigma_{i=0}^{\infty} \min\{\sigma_i(\mathcal{T})\} \leq \delta]$$

Another point to note is that clockless semantics reduces infinite state transition system to a finite state transition system only in case of the choice of discrete domain for the clock and timeout variables. This is because for the dense domain, clockless semantics can only limit unbounded set $\mathcal{R}^+$ to a bounded interval. Nonetheless, verification of a real-time system in a dense domain is equivalent to verifying the system in the discrete domain if the behavior of the system captured by the model and the properties considered are digitizable [8]. It can be shown that if we restrict the update function to *weakly constrained* intervals (e.g., $update_i = k_1 \mid k_2 \mid \infty \mid \max(\mathcal{M})$, where $k_1 \in [l, m]$ and $k_2 \geq l$) then similar to the timed transition system of [8] (refer theorem 2), transition systems for timeout and calendar based models also give rise to digitizable behaviors (computations). Also for qualitative properties like the safety and liveness properties, their verification in the discrete domain is equivalent to verifying these properties in dense domain (refer to proposition 1 in [8]).

5 Experimental Results

In this section, we report experimental results of verification of TGC and the TTA startup algorithm using the model checker Spin. We carry out our experiments on an Intel (R) P4 machine with 2.60 GHz speed and 1 GB RAM, and running Windows 2000.

Train-Gate Controller

For the TGC example as discussed before, we consider safety and timeliness properties for verification.

The safety property considered is: "When the Train crosses the line, the Gate should be down". The property can be expressed in LTL as follows:

$$\Box((t_state = t_2) \rightarrow (g_state = g_2))$$

where, *t_state* denotes different states of the Train and it is t_2 when the Train comes into the crossing, *g_state* denotes different states of Gate and is g_2 when the Gate is down.

The timeliness property considered states that the time between two states in execution will by bounded by a particular value. We can find many timeliness properties in this example. We mention one of them here: "The time between the transmission of the *approach* signal by the Train and when the Gate is down should not be more than 20 time units". To verify this property we use two auxiliary flags: $flag_1$ and $flag_2$ in our model. When the first event occurs $flag_1$ is set to *true*. When the second event happens, $flag_2$ is set to *true* and $flag_1$ is reset to $false$. Also, the proctype time_progress is modeled as follows:

```
proctype time_progress () {
  do
  :: timeout ->
  atomic {
      Find out the minimum of all the timeout values
      Subtract the minimum value from all the timeouts
      if
      :: flag1 == true ->
            time_diff = time_diff + min_timeout;
      :: flag2 == true ->
            flag2 = false;
            time_diff = 0;
      :: else ->
      fi;
  }
  od
}
```

A global variable *time_diff* (initially set to 0) captures the time difference between the instants when these two flags are set. During every discrete transition between the two discrete transitions of interest, minimum timeout value is added to *time_diff*. The property is specified as:

$$\Box(time_diff \ \leq \ 20)$$

Table 1 illustrates computational resources and time required to prove the safety and the timeliness property for TGC. Both the properties have been proved by exhaustive verification keeping the option of partial order reduction turned on.

TTA Startup Algorithm

For TTA startup algorithm, we consider the following safety property: "Whenever two nodes are in their *active* states, the nodes agree on the slot time". For two nodes participating in the startup process, the corresponding LTL property is given below:

$$\Box((p_0 \wedge p_1) \wedge (q_0 \wedge q_1) \ \rightarrow \ \Diamond(r \wedge s))$$

Table 1. Computational resources and time required for verification of the Train-Gate Controller

Properties	States stored	States matched	Transitions usage	Total actual memory (in MB)	Time (in seconds)
Safety	246236	422596	668832	19.531	6
Timeliness	253500	415484	668984	21.988	6

Table 2. Computational resources and time required to verify safety and liveness property by bitstate hashing technique in Spin for the TTA Startup

Properties	No of nodes	States stored	States matched	Transitions	Total actual memory usage (in MB)	Time
Safety	2	487	143	630	8.914	6 sec
	3	6142	6490	12632	8.914	7 sec
	4	123452	253057	376509	8.914	7 sec
	5	3.31158e+06	1.03436e+07	1.36552e+07	8.914	47 sec
	6	1.59195e+07	5.93261e+07	7.52457e+07	9.016	3 min
	7	3.44457e+07	1.29191e+08	1.63636e+08	9.016	8 min
	8	4.01727e+07	2.43036e+08	2.83209e+08	9.016	16 min
	9	4.10158e+07	1.29835e+09	1.33936e+09	9.016	97 min
Liveness	2	1445	1036	2481	8.914	7 sec
	3	16582	21980	38562	8.914	7 sec
	4	305893	677235	983128	8.914	8 sec
	5	7.39657e+06	2.38099e+07	3.12064e+07	8.914	1 min
	6	2.73472e+07	1.09554e+08	1.36901e+08	8.914	8 min
	7	3.83552e+07	2.0233e+08	2.40685e+08	9.016	14 min
	8	4.07954e+07	3.47211e+08	3.88006e+08	9.016	24 min
	9	4.10157e+07	2.30705e+09	2.34807e+09	9.016	163 min

where, $p_0 \equiv (pc[0] = state_active)$, $p_1 \equiv (pc[1] = state_active)$, $q_0 \equiv (time_out[0] > 0)$, $q_1 \equiv (time_out[1] > 0)$, $r \equiv (time_out[0] = time_out[1])$, and $s \equiv (slot[0] = slot[1])$. $pc[i]$ denotes the current state of the ith node, $time_out[i]$ denotes the timeout of the ith node, and $slot[i]$ denotes the current time slot viewed by the ith node. *state_active* characterizes the *synchronized* state of a node.

The Safety property ensures that when the nodes are in the *active* state, then they are indeed synchronized. But it does not answer the question whether all the nodes will eventually be synchronized or not. To ensure that all the nodes will eventually be synchronized, it has to be specified in the form of a liveness property: "Eventually all the nodes will be in the *active* state and remain so". This liveness property for two nodes can be specified in LTL as follows:

$$\Diamond\Box((pc[0] = state_active) \wedge (pc[1] = state_active))$$

To verify the safety and the liveness property for the TTA startup we used clockless modeling together with the options of exhaustive verification and bitstate hashing technique offered by Spin, in both the cases keeping the the option of partial order reduction turned on. By exhaustive verification technique, the safety property can be verified for the TTA models with upto 5 nodes and liveness property can be verified upto 4 nodes. Bitstate hashing enables us to verify both the properties for models with upto 9 nodes. For 10 nodes, the verification does not terminate even in 4 hours.

Table 2 describes the computational resources and time required to prove the safety and liveness properties for the TTA Startup protocol using bitstate hashing technique.

Experiments with dense time modeling with clockless reduction using SAL were also carried out on the TGC model presented in [6]. The safety property has been verified at depth 14 as done in [6]. Nonetheless, applying the clockless reduction in SAL models do not scale up the existing results further, primarily because the clockless reduction even though reduces the unbounded set $\mathcal{R}^+$ to a bounded interval, such an interval still will contain uncountably many points giving rise to infinite many possible execution paths of finite lengths.

6 Conclusion

In this work we proposed a canonical finitary reduction technique formalized in terms of clockless modeling and associated semantics, which renders timeout and calendar based models of real-time systems amenable to finite state verification. There exists an equivalence between the corresponding discrete and the dense domain verifications for the qualitative safety and liveness properties considered in this work on the weakly constrained models assuming discrete dynamics. Verification of safety properties for the TTA start up protocol consisting of upto 9 nodes demonstrates the effectiveness of the reduction technique as compared to the dense time modeling and verification results reported in literature [7], which rely on additional efforts to find out appropriate supporting lemmas and abstractions to scale up the model.

Dynamic rescheduling of timeouts to deal with interrupts can further extend the current framework in order to model hardware-software co designs and preemptive scheduling type of scenarios. Work can also be further extended by considering shared timing variables, deadlines to capture urgencies [3], and by considering the timeout update rules, which include the possibility of interactive updation to deal with game theoretic properties.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–236 (1994)
2. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

3. Bornot, S., Sifakis, J., Tripakis, S.: Modeling Urgency in Timed Systems. In: de Roever, W-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 103–129. Springer, Heidelberg (1998)
4. Bošnački, D., Dams, D.: Integrating Real Time into Spin: A Prototype Implementation. In: FORTE/PSTV. Proceedings of the Formal Description Techniques and Protocol Specification, Testing and Verification, pp. 423–439, Kluwer, Dordrecht (1998)
5. Bošnački, D., Dams, D.: Discrete-Time PROMELA and Spin. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 307–310. Springer, Heidelberg (1998)
6. Dutertre, B., Sorea, M.: Timed Systems in SAL. Technical Report, Computer Science Laboratory, SRI Internationalhhe (2004)
7. Dutertre, B., Sorea, M.: Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol using Calendar Automata. In: Lakhnech, Y., Yovine, S. (eds.) FTRTFT 2004. LNCS, vol. 3253, pp. 199–214. Springer, Heidelberg (2004)
8. Henzingerz, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
9. Holzmann, G.J.: The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
10. Moura, L.M., Owre, S., Rue, H., Rushby, J.M., Shankar, N., Sorea, M., Tiwari, A.: Sal 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
11. Pike, L.: Real-Time System Verification by k-Induction. Technical report, NASA Langley Research Center. TM-2005 -213751(2005), Available at http://www.cs.indiana.edu/~lepike/pub_pages/reint.html
12. Rushby, J.: Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 508–520. Springer, Heidelberg (2000)
13. Saha, I., Roy, S.: A Finite State Modeling of AFDX Frame Management using Spin. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006. LNCS, vol. 4346, pp. 227–233. Springer, Heidelberg (2007)
14. Saha, I., Roy, S.: A Finite State Analysis of Time-triggered CAN (TTCAN) Protocol using Spin. In: ICCTA 2007. Proceedings of the International Conference on Computing: Theory and Application, pp. 77–81. IEEE Computer Society Press, Los Alamitos (2007)
15. Steiner, W., Paulitsch, M.: The Transition from Asynchronous to Synchronous System Operation: An Approach for Distributed Fault- Tolerant System. In: ICDCS 2002. Proceedings of the 22nd International Conference on Distributed Computing Systems, pp. 329–336. IEEE Computer Society Press, Los Alamitos (2002)
16. Tripakis, S., Courcoubetis, C.: Extending PROMELA and Spin for Real Time. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 329–348. Springer, Heidelberg (1996)

Efficient Approximate Verification of Promela Models Via Symmetry Markers

Dragan Bošnački[1], Alastair F. Donaldson[2], Michael Leuschel[3], and Thierry Massart[4]

[1] Department of Biomedical Engineering, Eindhoven University of Technology
dragan@win.tue.nl
[2] Codeplay Software Ltd., Edinburgh
ally@codeplay.com
[3] Institut für Informatik, Universität Düsseldorf
leuschel@cs.uni-duesseldorf.de
[4] Département d'informatique, Université Libre de Bruxelles
tmassart@ulb.ac.be

Abstract. We present a new verification technique for Promela which exploits state-space symmetries induced by *scalarset* values used in a model. The technique involves efficiently computing a *marker* for each state encountered during search. We propose a complete verification method which only partially exploits symmetry, and an approximate verification method which fully exploits symmetry. We describe how symmetry markers can be efficiently computed and integrated into the Spin tool, and provide an empirical evaluation of our technique using the TopSpin symmetry reduction package, which shows very good performance results and a high degree of precision for the approximate method (i.e. very few non-symmetric states receive the same marker). We also identify a class of models for which the approximate technique is precise.

1 Introduction

The design of concurrent systems is a non-trivial task where generally a lot of time is spent on simulation to track design errors. *Model checking* methods and tools [Hol03, McM93, CGP99] can be used to help in this effort by automatically analysing finite-state models of a system. In practice, exhaustive exploration of a state-space is impractical due to the infamous *state explosion problem*, which has motivated the development of more efficient exploration techniques. In particular, the model to be checked often consists of a large number of states which are indistinguishable up to rearrangement of process identifiers. As a result, the model is partitioned into classes of states where each member of a given class behaves like every other member of the class (with respect to a logical property that does not distinguish between individual processes). *Symmetry reduction* techniques [CGP99, ID96, CEFJ96, ES96] allow the restriction of model checking algorithms to a *quotient* state-space consisting of one representative state from each symmetric equivalence class. One successful symmetry reduction technique

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 300–315, 2007.
© Springer-Verlag Berlin Heidelberg 2007

[ID96] relies on a special data type, called *scalarset*, used in the specification to identify the presence of symmetry. Symmetry reduction using scalarsets has been initially implemented in the Murϕ verifier [ID96], and in previous work we have adapted the same idea for Promela in the SymmSpin tool [BDH02]. We have also proposed in [DM05] a method to automatically detect, before search, structural symmetries in Promela specifications, and such symmetries can be exploited by the TopSpin tool [DM06]. Both SymmSpin and TopSpin perform symmetry reduction on-the-fly: each time a global state s is reached, its representative state $rep(s)$ is computed and used instead of s. If $rep(t) = rep(s)$ for every state t in the same class as s then symmetry reduction is said to be *full* (i.e. it is memory-optimal). However, the computation of a unique representative for an equivalence class (known as the *constructive orbit problem*) is NP-hard [CEJS98]; and for some practical examples full symmetry reduction strategies can be too time consuming. Therefore, *partial symmetry* reduction strategies have been defined, which take less time to compute a representative element. The price to pay is that multiple representatives may be computed for a given equivalence class and therefore the reduction factor in the number of global states explored can be smaller than with a full reduction method. Fig. 1 illustrates the possible size of the explored state-space for the various verification methods, as a function of the number of replicated processes in the system. The solid line represents exhaustive state-space exploration without symmetry reduction, the dashed line a complete verification of the quotient state-space modulo full symmetry reduction. Any complete verification method using partial symmetry reduction would yield a function in the area between these lines. Recently, we have proposed the *symmetry marker* method [LM07] in the framework of the B specification language [Abr96]. This reduction technique is inspired by the success of Spin's bitstate hashing technique [Hol88], which regards two states as equivalent if they hash to the same value. Bitstate hashing reduces the per-state storage requirement to a single bit, at the expense of excluding a small percentage of states due to hash collisions. In a similar vein, we define a *marker* function on global states, invariant under symmetry, which can be computed efficiently. We avoid the underlying complexity induced by the constructive orbit problem by assuming that two states with the same marker value are symmetric. As this assumption

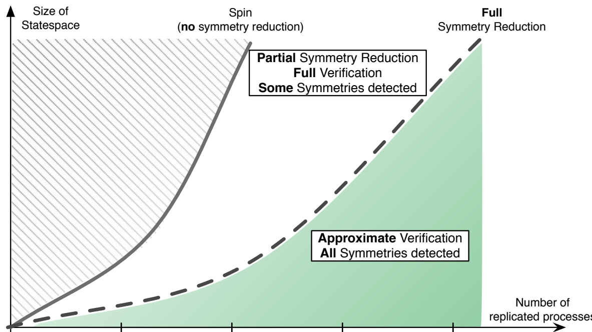


Fig. 1. Illustrating partial symmetry reduction and approximate verification

can be wrong in general, we only have an *approximate* verification technique (in the sense that collisions may occur where non-symmetric states obtain the same marker value); but a very fast one. If we refer to Fig. 1, our approximate verification method would give a function on the dashed line when it provides complete verification, or below this line in case of collisions when it only provides approximate verification.

In this paper, inspired by our previous works on symmetry, we propose two new symmetry reduction methods for Promela: a *complete* verification method which may compute more than one representative for each symmetric equivalence class, and an *approximate* verification method which guarantees unique representatives, but results in a small number of collisions between equivalence classes. We detail a TopSpin-based implementation of these methods, and provide encouraging experimental results. Note that although we present our methods in the context of Promela/Spin, they are clearly transferrable to other explicit-state model checking frameworks.

In the remainder of the paper, we briefly recall in Sect. 2 some relevant features of Spin and Promela, the notion of a *scalarset* and the main methods employed by the SymmSpin and TopSpin tools. We outline in Sect. 3 the initial *symmetry marker* method presented in [LM07], then describe in Sect. 4 a first naïve method for Promela and Spin, which is directly inspired by our symmetry marker method. In Sect. 5 we describe our "complete" new methods based on markers for Promela and Spin. In Sect. 5 we also provide theoretical results for particular classes of systems where our method is precise, and in Sect. 6 some results which empirically validate both methods compared to methods without symmetry reduction and existing methods implemented in SymmSpin and TopSpin.

2 Scalarsets in Promela

The Spin model checker [Hol03] allows verification of concurrent systems specified in Promela — a C like language extended with Dijkstra's guarded commands and communication primitives from Hoare's CSP. In Promela, system components are specified as *processes* that can interact either by message passing, via buffered or rendez-vous *channels*, or memory sharing, via *global variables*. Concurrency is asynchronous and modelled by interleaving. Spin can verify various safety and liveness properties of a Promela model including any LTL formula. To cope with the problem of state-space explosion, standard Spin employs several reduction techniques, such as partial-order reduction, state-vector compression, and bitstate hashing. In Spin each state has an explicit representation called the *state vector*. The state vector has the form $(G, R_1, \ldots, R_n)$, where G comprises the values of global variables, and $R_1, \ldots, R_n$ are records corresponding to the processes in the system. Each process record contains the parameters, local variables and the program counter for the particular process. The marker algorithms that we present in the sequel are independent of this representation, but in the presentation we will refer to the process vector structure and in particular its form for symmetric models.

SymmSpin [BDH02] is an extension of Spin with symmetry reduction based on the *scalarset* data type [ID96], by which the user can point out (full) symmetries to the verification tool. The values of scalarsets are finite in number, unordered, and allow only a restricted number of operations which do not break symmetry. Intuitively, in the context of Spin, the scalarsets are process identifiers (pids) of a family of symmetric processes. Such a family is obtained by instantiating a parameterized process type. One restriction is that applying arithmetic operations to pids is forbidden. Also, since formally there is no ordering between the scalarset values, the pids can be tested only for equality. We consider models in Promela that are collections of parallel processes of the form $B\|P_1\|\ldots\|P_n$. Processes P_i are instances of a parameterized process template and differ only in their pid. Process B is a *base* process and it represents the "non-symmetric" part of the model (though B must behave symmetrically with respect to the P_i). Further, we assume that each state is represented explicitly by a state vector as described above. To illustrate the main ideas behind the strategies for finding representatives in SymmSpin, consider the following example adapted from [BDH02]. Let us assume that we want to choose as a (unique) representative of each symmetry class (orbit) the state from that class represented by the lexicographically minimal state vector. Further, suppose that there is an array M (we call it the *main* array) at the very beginning (of the global part G) of the state vector of our model. Let M be indexed by pids (scalarsets, here 1 to 5), but the elements of M are not of scalarset type. The *sorted* strategy computes a representative of a state s by sorting M and applying the pid permutation p corresponding to this sorting operation to the rest of the state vector. This involves sorting the process records, and rearranging the values of scalarset variables. Fig. 2 shows the state vector before and after sorting an example main array M with permutation $p = \{1 \mapsto 1, 2 \mapsto 5, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4\}$. Note that when M contains several instances of the same value (here 2 and 7), several orbit representatives can be computed for the same class of states. Hence, the sorted strategy only performs partial symmetry reduction. Suppose we applied *all* pid permutations to the upper state vector in Fig. 2. Then the lexicographical minimum among the resulting states, $s_{\min}$ say, would start with the same values as the lower state vector, namely 2, 2, 7, 7, 8. However, the rest of $s_{\min}$ need not coincide with the rest of the lower state vector. The reason is that there are other pid permutations that yield the same ordering of the array M, for example $p' = \{1 \mapsto 2, 2 \mapsto 5, 3 \mapsto 4, 4 \mapsto 1, 5 \mapsto 3\}$, but may give smaller results than p when applied to the rest of the state vector. The *segmented* strategy applies all pid permutations which sort M (in this example there are four

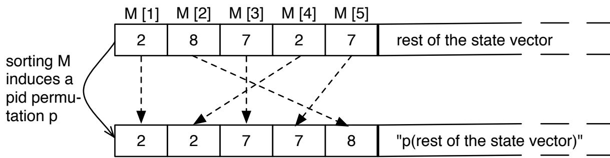


Fig. 2. A state vector before and after sorting the main array M

of them) to the whole state vector, and selects the smallest among the resulting states, which is then guaranteed to be $s_{\min}$. The price to pay for this *full* reduction strategy is factorial complexity in the worst case: if all values of M are identical in a state then all $n!$ scalarset permutations must be considered in order to compute a representative for the state. However, for many states very few permutations need to be considered, so this approach is more efficient than the basic approach of considering every pid permutation at each state. Note that a main array is always available and can be selected automatically by the model checker – if no suitable array is explicitly declared in the model then the array of process program counters can be used, by default.

The TopSpin symmetry reduction package [DM06] builds on the ideas introduced with SymmSpin, supporting more general types of symmetry than just full symmetry, and providing support for automatic symmetry detection based on techniques presented in [DM05]. We use TopSpin for implementation of our techniques, so it is important to briefly explain the relationship between TopSpin and SymmSpin. TopSpin includes symmetry reduction strategies based on the SymmSpin *sorted* and *segmented* strategies. The TopSpin *sorted* strategy is a generalisation of the SymmSpin *sorted* strategy: instead of sorting with respect to one particular *main* array, sorting is performed in a more general fashion by repeatedly applying swap permutations to the entire target state. This generalised *sorted* strategy sometimes performs better than the original SymmSpin *sorted* strategy, but both approaches share the problem that symmetry reduction may result in storage of multiple orbit representatives. The TopSpin *segmented* strategy generalises the SymmSpin *segmented* approach, and is described in detail in [DM07]. For the special case of full symmetry, the TopSpin and SymmSpin *segmented* strategies are analogous.

3 Symmetry Markers

Our symmetry marker technique, initially proposed in [LM07] for the B language, is partially inspired by Holzmann's successful bitstate hashing technique [Hol88]. In our case, the hash value is replaced by a *marker*. This marker has a more complicated structure, but integrates the notion of symmetry: two symmetric states will have the same marker and there is a "small chance" that two non-symmetric states have the same marker. Our adapted model checking algorithm stores those markers rather than the states and checks a new state only if its marker has not yet been seen before. The advantages over classical symmetry reduction are two-fold. First, a precise symmetry marker can be computed very efficiently (depending on the system, basically linear or quadratic in the size of the state for which the marker is computed), while classical symmetry reduction has an inherent factorial complexity (in terms of the number of the symmetric data values). The second advantage is the size of the state-space explored with our marker method, which is equal or less than the size of the state space explored with a full symmetry reduction method (smaller if collisions occur). The price we pay is that – just as with the bitstate hashing technique – we no longer have a complete verification method: two non-symmetric states s_1, s_2 can have

the same marker meaning that the second state s_2 would not be checked, even though it could lead to an error while no error is reachable from s_1.

In the B language [Abr96], the *deferred sets* construct gives rise to *symmetric* data values similar to scalarsets [LM07]. The value of a global state s can be given as a vector of values of its global variables and constants, which are classified by the following types: simple non-symmetric (e.g. booleans, and integer subranges), simple scalarset (i.e. a deferred set), pair, or finite (multi)sets. In this setting, a set of pairs defines a relation and an *array* is defined as a total function (i.e. a particular type of relation) between its indexes and the value of its components. We adopt a standard structured view of a state as a *rooted acyclic graph* whose nodes are labelled by their type and whose leaves are values. The root has n ordered children corresponding to n variables or constants. Simple values are *leaves* of the graph, pair values have two ordered children and (multi)sets have unordered children, one for each element in the (multi)set. The idea of our marking function is to transform a state s into a marker by replacing the scalarset values by so-called *vertex invariants*. In graph theory, a vertex invariant *inv* is a function which labels the vertices of an arbitrary graph with values so that symmetrical vertices are assigned the same label. Examples of simple vertex invariants include the in-degree and the out-degree for the specified vertex. Our technique uses a more involved vertex invariant for scalarset elements. Informally, a symmetry marker $m(s)$ for a given state s is computed as follows: (1) For every scalarset element d used in s, compute structural information about its occurrence in s, invariant under symmetry. This is computed as the multiset of *paths* that lead to an occurrence of d in s. (2) Replace all scalarset elements by the structural information computed above and compute a marker with an algorithm similar to the computation of the canonical form in the tree isomorphism problem [Val02]. The resulting complexity is quadratic in the size of the state in the worst case. We have proved in [LM07] that our definition is indeed invariant under symmetry, i.e. that if s_1 and s_2 are symmetric states in a system M then $m(s_1) = m(s_2)$; we have also identified classes of systems where the marker method is precise, i.e. it provides a full symmetry reduction method. Note that the method is quite general and abstract and could be instantiated in other contexts than those of SPIN/Promela and B.

4 A First Naïve Approach

As a stepping stone towards our approximate techniques, we first describe a naïve strategy that we call *flattened*. In this approach we "flatten" the state vector by assigning to each scalarset variable the same value. (The choice of the concrete value is irrelevant – it can be from the range of the scalarset or some other "neutral" value.) Basically, this amounts to distinguishing processes by the values of their non-scalarset local variables only. Then we apply to the flattened state vector TopSPIN's *sorted* strategy, described in Sect. 2. Obviously, because of the flattening, states that are not symmetric may have the same representative. As a result a full state-space coverage is not guaranteed. We illustrate this basic technique using Peterson's n-process mutual exclusion protocol [Pet81],

```
byte flag[6] = 0;    // an array from PID to byte (flag[0] is not used)
pid turn[5] = 0;     // an array from [0..4] to PID
byte inCR = 0        // number of processes in critical region
proctype user () {
   byte k; bool ok;
   do :: k = 1;
         do :: k < 5 ->  flag[_pid] = k; turn[k] = _pid;
again:         atomic {
                  ok = ((_pid==1)||(_pid!=1 && flag[1]<k))&&
                       ((_pid==2)||(_pid!=2 && flag[2]<k))&&
                       ((_pid==3)||(_pid!=3 && flag[3]<k))&&
                       ((_pid==4)||(_pid!=4 && flag[4]<k))&&
                       ((_pid==5)||(_pid!=5 && flag[5]<k));
                  if :: ok || turn[k] != _pid
                     :: else -> goto again
                  fi
               };
               k++
            :: else -> break
         od;
         atomic { inCR++; assert(inCR == 1) };  inCR--;  flag[_pid] = 0;
   od;
}
init {   // start the processes
   atomic{ run user(); run user(); run user(); run user(); run user(); }
}
```

Fig. 3. Promela code for Peterson's mutual exclusion protocol, with five processes

which has been used for experiments with SymmSpin [BDH02]. We consider various configurations of a Promela specification of this protocol, adapted from the specification used in [BDH02]. Promela code for the specification with five processes is given in Fig. 3. We check the mutual exclusion property, which is embedded into the specification using an assertion, and also verify that the model associated with each specification is deadlock-free. For various configurations of the protocol, Fig. 4 shows the state-space size and time (in seconds) for unreduced verification, and verification using the SymmSpin *segmented* and *sorted* strategies, the TopSpin *sorted* strategy, and our *flattened* strategy. An entry marked '–' indicates that memory requirements for verification exceeded available resources, or that verification did not complete within five hours. All experiments were performed on a PC with a 1.7 GHz Pentium processor, 760 Mb of main memory, using Spin version 4.2.6. Recall that the SymmSpin *segmented* strategy is guaranteed to give memory-optimal symmetry reduction. For this

Peterson	**Spin (unreduced)**		**SymmSpin segmented**		**SymmSpin sorted**		**TopSpin sorted**		**TopSpin flattened**	
n	states	time	states	time	states	time	states	time	states	time
3	2636	0.4	494	0.3	907	0.4	494	0.4	251	0.1
4	60577	0.6	3106	0.4	9373	0.4	3106	0.4	1177	0.1
5	1.56×10^6	11	17321	1	95303	2	17321	1	5148	0.3
6	4.48×10^7	2666	89850	7	885399	18	89850	7	21752	2
7	-	-	442481	85	7.94×10^6	383	442481	56	89969	10
8	-	-	2.09×10^6	1166	-	-	2.09×10^6	412	366424	63
9	-	-	9.62×10^6	16673	-	-	9.62×10^6	3034	1.47×10^6	393

Fig. 4. Experimental results for Peterson's mutual exclusion protocol, using SymmSpin, TopSpin, and a naïve "markers"-based approach

example, therefore, we see that TopSpin *sorted* also provides memory-optimal symmetry reduction. In comparison, SymmSpin *sorted* performs visibly poorly on the *Peterson 7* configuration, and could not be practically applied to larger configurations. The *flattened* strategy yields very fast verification in comparison to all other strategies. Correspondingly, the state-space explored using this strategy is much smaller than the symmetry-reduced state-space explored using the SymmSpin *segmented* strategy. The speed-up gained using this approach is encouraging, but state-space coverage is clearly too low for this technique to be acceptable in practice. Motivated by the efficiency of the *flattened* strategy, we now develop more sophisticated symmetry marker techniques for Promela.

5 The New Marker Methods for Promela

The marker method developed for B [LM07] inspires the definition of efficient symmetry reduction techniques for other specification languages, such as Promela. However, adapting the techniques for Promela is not trivial. The requirement to extend Spin (more precisely SymmSpin or TopSpin) to include the new concepts means that we cannot simply use data structures like multisets of paths as defined in [LM07], but need to define an efficient encoding in the context of the data structures used for state-space representation by Spin. The methods we propose respect this constraint through transformations of state vectors which preserve its structure. We also derive a new technique, which can be used as a complete verification method.

Datatypes and state representation. We first define the following simple Promela datatypes:

– a single scalarset I (whose elements are called pids) of cardinality N with values 1..N. Sometimes we also need to use the special value 0, representing an undefined value (as in [ID96]). We define $I_0 = I \cup \{0\}$.
– simple non-scalar datatypes such as byte, bool and mtype (an enumerated message type included in the Promela language), denoted by NS.

We assume that we do not have nested arrays or queues (i.e., the elements of arrays or queues cannot be in turn arrays or queues), and that our Promela model is composed with a base process G in parallel with instances P_i of a parameterized family of processes.

Definition 1. *The state of a Promela specification is described by the following quadruple* $\langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ *where*

– $\overrightarrow{n}$ *is a vector of values from* NS *(i.e., of non-scalar type)*
– $\overrightarrow{s}$ *is a vector of values in* I_0
– $\overrightarrow{sn}$ *is a vector of arrays indexed by the scalarset* I *and with range values from* NS
– $\overrightarrow{ss}$ *is a vector of arrays indexed by the scalarset* I *and with range from* I_0

One can make the following observations. Conceptually there are no local process variables: they are treated as entries of a global array indexed by the pids. In

other words, the local variables become part of $\overrightarrow{sn}$ and $\overrightarrow{ss}$. The program counter pc_i of each process i is conceptually handled as part of $\overrightarrow{sn}$.

Some datastructures are missing from Def. 1. However, without loss of generality, they can be incorporated into the state as follows:

1. Arrays $NS \to I_0$ from non-scalar to scalarset values can be seen as part of $\overrightarrow{s}$ by expanding out the array and treating each array element as a distinct variable.
2. Similarly, arrays $NS \to NS$ from non-scalar to non-scalar values can be viewed as part of $\overrightarrow{n}$ by expanding them out.
3. Queues of (scalar or non-scalar) values are translated into arrays (indexed by non-scalar values) of the same size, padded with zeroes if the queue is not full, together with an integer to record the current length of the queue. For example, $queue = [2, 3]$ of length 4 becomes $array = [2, 3, 0, 0]$, $length = 2$. The resulting arrays can then be expanded according to points 1 and 2, depending on the type of values which they contain. A queue for which messages consist of multiple fields can be handled using a series of arrays, one per field.
4. Records can be handled by treating each field as an individual variable.

Example 1. Consider the Promela code for Peterson's mutual exclusion protocol [Pet81] with 5 processes, shown in Fig. 3 and introduced in Sect. 4. For this Promela specification the components of a state s from Def. 1 will look as follows (where x^s denotes the value of the global variable x in the state s and y_i^s denotes the value of the local variable y for process i in s):

- $\overrightarrow{n} = \langle inCR^s \rangle$
- $\overrightarrow{s} = \langle turn^s[0], \ldots, turn^s[4] \rangle$
- $\overrightarrow{sn} = \langle [pc_1^s, \ldots, pc_5^s], [flag_1^s, \ldots, flag_5^s], [k_1^s, \ldots, k_5^s], [ok_1^s, \ldots, ok_5^s] \rangle$
- $\overrightarrow{ss} = \langle \rangle$

The structure of s is also depicted in Fig. 5 below.

To compute our markers (see algorithm 5.1), we use the notions of *permutation* and *mapping* of a state s as defined below where $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ with $\overrightarrow{sn} = \langle \overrightarrow{sn_1}, \ldots \overrightarrow{sn_k} \rangle$ and $\overrightarrow{ss} = \langle \overrightarrow{ss_1}, \ldots \overrightarrow{ss_\ell} \rangle$.

Definition 2. *A* permutation π *is a bijection from I to I.*
We extend the application of a permutation π to a data value v, denoted by v^π, as follows: v^π =

- *v if v is a non-scalar value or $v = 0$*
- *$\pi(v)$ if $v \in I$*
- *$\langle v_1^\pi, \ldots, v_k^\pi \rangle$ if $v = \langle v_1, \ldots, v_k \rangle$ is an array or vector indexed by non-scalars*
- *$\langle (v_{\pi^{-1}(1)})^\pi, \ldots, (v_{\pi^{-1}(k)})^\pi \rangle$ if $v = \langle v_1, \ldots, v_k \rangle$ is an array indexed by scalarset values*

The application of a permutation π to the state s is defined by $s^\pi = \langle \overrightarrow{n}, \overrightarrow{s}^\pi, \langle \overrightarrow{sn_1}^\pi, \ldots \overrightarrow{sn_k}^\pi \rangle, \langle \overrightarrow{ss_1}^\pi, \ldots \overrightarrow{ss_\ell}^\pi \rangle \rangle$.

Finally, we say that state s' is symmetric to s *iff there exists a permutation* π *such that* $s^\pi = s'$.

We sometimes write permutations (and mappings) in explicit form as follows: $\{1 \mapsto j_1, \ldots, N \mapsto j_N\}$.

For example, let $\pi = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\}$ and let $a = [1, 2, 2]$ be an array $NS \to I$. Then $a^\pi = [2, 1, 1]$. However, if a is of type $I \to I$ then $a^\pi = [1, 2, 1]$.

Definition 3. *A* mapping ρ *is a function (which may not be a bijection) from* I *to* I*. We extend the application of a mapping* ρ *to a data value* v *and state* s*, denoted resp. by* $\rho(v)$ *and* $\rho(s)$ *in a way similar to what we did for permutations, except for arrays* $v = \langle v_1, \ldots, v_k\rangle$ *indexed by scalarset values which is defined by* $\langle \rho(v_1), \ldots, \rho(v_k)\rangle$.

Note that, contrary to permutations, a mapping does not necessarily permute the indexes of vectors indexed by scalarset values.

Marker algorithms for approximate and exact verification. We will now present a way to efficiently compute for any given state s a marker $m(s)$. The central idea of our approach is to analyse the current state s of a Promela specification in order to compute information about every scalarset value p. This information $m_s(p)$ is called the marker of p in s and captures structurally how p is used within s.

Definition 4. *The marker* $m_s(p)$ *of a scalarset value* $p \in I$ *in the state* $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss}\rangle$ *is the triple* $\langle \overrightarrow{s'}, \overrightarrow{sn'}, \overrightarrow{ss}'\rangle$ *where*

- $\overrightarrow{s'}$ *is a vector of bits of the same length as* $\overrightarrow{s}$*, where* $\overrightarrow{s'}_i = 1$ *iff* $\overrightarrow{s}_i = p$
- $\overrightarrow{sn'}$ *is a vector of non-scalar values and of the same length as* $\overrightarrow{sn}$ *where* $\overrightarrow{sn'}_i = \overrightarrow{sn}_i[p]$
- $\overrightarrow{ss'}$ *is a vector of non-scalar values and of the same length as* $\overrightarrow{ss}$ *where* $\overrightarrow{ss'}_i =$ *number of occurrences of* p *in the range of* $\overrightarrow{ss_i}$

For a particular Promela specification with possible states S *we define the set of scalarset markers* $\mathcal{M} = \{m_s(p) \mid s \in S \wedge p \in I\}$*. By* $<_\mathcal{M}$ *we denote a total order relation* $<$ *on* $\mathcal{M}$.[1]

Consider the Peterson-5 example (Fig. 3 and Ex. 1). For $p \in I$ we have that (see also Fig. 5):

- $\overrightarrow{s'}$ is a vector of 5 bits, one for each entry of *turn*, with $\overrightarrow{s'_i} = 1 \Leftrightarrow turn^s[i] = p$
- $\overrightarrow{sn'} = \langle pc^s_p, flag^s_p, k^s_p, ok^s_p\rangle$
- $\overrightarrow{ss'} = \langle\rangle$

Our algorithm takes a state s of a Promela specification and computes the marker $m(s)$ for the state. Ideally we want the property that if two states are

[1] Such an order is easy to define, e.g. using lexicographical ordering, as no scalarset values occur inside the markers.

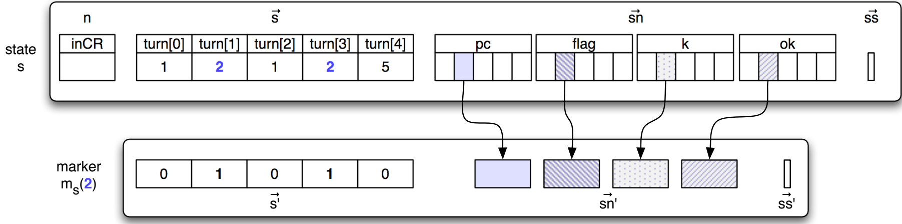


Fig. 5. The structure of Promela states and markers for Peterson-5

symmetric then they have the same marker, and vice versa. However, in order to make the computation of the marker more efficient, we are willing to accept a tradeoff. We will present two possible tradeoffs below. The method which uses the *exact markers* $m_{exact}(s)$ does not always detect that two symmetrical states are symmetric; the method which uses the *approximate marker* $m(s)$ may merge two states which are not symmetrical.

Below, by $|\ \overrightarrow{v}\ |$ we denote the length of a vector $\overrightarrow{v}$. We compute the markers for all $i \in I$ and based on the markers (which contain no scalarset values) find a way to permute the values in I. To handle the case where two values in I have the same marker, we also compute the information $local(i)$ for every $i \in I$ which captures which other markers i refers to in its entries of $\overrightarrow{ss}$ (which is usually part of its local state).

Algorithm 5.1[Computation of the markers $m(s)$ and $m_{exact}(s)$ for s]

Input: A state $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ of a Promela specification
Output: The markers $m(s)$ and $m_{exact}(s)$ for s
let $a = \langle m_s(1), \ldots m_s(N) \rangle$; **sort** a according to $<_{\mathcal{M}}$
let $mval_s(i)$ = **if** i=0 **then** 0 **else** $max(\{j \mid a[j] = m_s(i)\})$ **fi** ;
for $i \in I$ **do** *% compute which other markers does i refer to in its part of $\overrightarrow{ss}$*
  let $local_s[i] = \langle mval_s(\overrightarrow{ss}_1[i]), \ldots, mval_s(\overrightarrow{ss}_{|\overrightarrow{ss}|}[i]) \rangle$
od ;
let $b = \langle (m_s(1), local_s[1], 1), \ldots, (m_s(n), local_s[n], n) \rangle$;
sort b where $(m_1, l_1, n_1) < (m_1, l_2, n_2)$ iff $m_1 <_{\mathcal{M}} m_2$ or $m_1 = m_2$ and
  $l_1 <_{local_s} l_2$ (using some total order $<_{local_s}$ on arrays of numbers);
let $newval_s(i) = max(\{j \mid \exists k.b[j] = (m_s(i), local_s[i], k)\})$;
let $pos(i)$ = value j such that $b[j] = (m_s(i), local_s[i], i)$;
let $\pi = \{1 \mapsto pos(1), \ldots, N \mapsto pos(N)\}$;
let $m_{exact}(s) := s^{\pi}$; *% Apply permutation π*
let $\rho = \{pos(1) \mapsto newval_s(1), \ldots, pos(N) \mapsto newval_s(N)\}$; *% may not be a perm.*
let $m(s) := \rho(m_{exact}(s))$ *% Apply mapping ρ*

Example 2. Take the state s partially illustrated in Ex. 1. If the markers computation gives that $m_s(3) < m_s(4)$, we have $\pi = \{1 \mapsto 5, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 3\}$ and $m(s) = m_{exact}(s)$ as outlined by Fig. 6 where for $\overrightarrow{sn}$ we just showed that values initially in position 2 are now in position 4. Note that since $\overrightarrow{ss}$ is empty, a big part of the algorithm can be simplified.

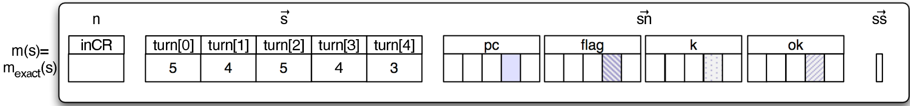


Fig. 6. $m(s)$ and $m_{exact}(s)$ for s in Fig 5

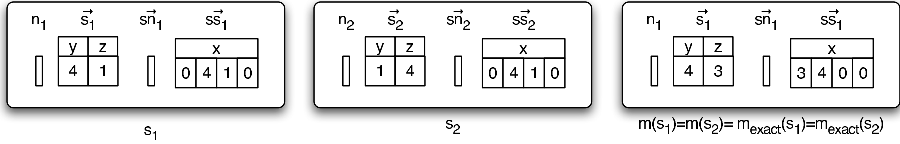


Fig. 7. $m(s)$ and $m_{exact}(s)$ for $s1$ and s_2

Example 3. In Fig. 7, the states s_1 and s_2 are symmetrical through the permutation $\pi = \{1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto 1\}$. However, both for states s_1 and s_2, $m_s(2) = m_s(3)$ and hence without $local_s$ it would be unclear in which order to put the scalarset values 2 and 3. Our algorithm will guarantee that s_1 and s_2 have the same marker, as shown in the Fig. 7 and detailed in the following table:

element	value for s_1	value for s_2
a sorted	$m_{s_1}(2) = m_{s_1}(3) < m_{s_1}(1) < m_{s_1}(4)$	$m_{s_2}(2) = m_{s_2}(3) < m_{s_2}(4) < m_{s_2}(1)$
$local_s$	$\langle 0, 4, 3, 0\rangle$	$\langle 0, 3, 4, 0\rangle$
b sorted	$\langle\langle m_{s_1}(3), 3, 3\rangle, \langle m_{s_1}(3), 4, 2\rangle,$ $\langle m_{s_1}(1), 0, 1\rangle, \langle m_{s_1}(4), 0, 4\rangle\rangle$	$\langle\langle m_{s_2}(3), 3, 2\rangle, \langle m_{s_2}(3), 4, 3\rangle,$ $\langle m_{s_2}(4), 0, 4\rangle, \langle m_{s_2}(1), 0, 1\rangle\rangle$
π	$\langle 1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1, 4 \mapsto 4\rangle$	$\langle 1 \mapsto 4, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 3\rangle$

Proposition 1. *Let s, s' be states. Then the following hold:*

1. *$m(s) = m(s^\pi)$ for any permutation π*
2. *$m_{exact}(s) = m_{exact}(s') \Rightarrow \exists\pi.s' = s^\pi$*
3. *$m_{exact}(s) = m_{exact}(s') \Rightarrow m(s) = m(s')$*

Proof. Point 1 can be proven as follows. It is easy to see that $m_s(\pi(i)) = m_{s^\pi}(i)$ and hence the sorted arrays a in Alg. 5.1 are identical for s and s^π. Hence, $mval_s(\pi(i)) = mval_{s^\pi}(i)$. This in turn implies $local_s[\pi(i)] = local_{s^\pi}[i]$ and that $newval_s(\pi(i)) = newval_{s^\pi}(i)$. The only potential difference between i and $\pi(i)$ could be the value of pos. However, in that case there must exist another $j \in I$ with $m_s(j) = m_s(i) \wedge local_s[i] = local_s[j] \wedge newval(j) = newval(i)$ with the same value of pos as $\pi(i)$; and hence the resulting markers must be identical. Point 2 can be proven by composing π from Alg. 5.1 for s with the inverse of π from Alg. 5.1 for s'. Point 3 follows directly from the two other points ($m(s') = m(s^\pi) = m(s)$).

Point 1 means that all symmetries are detected by our approximate markers. Point 2 means that using exact markers yields a complete verification method.

In general the ordinary markers do not provide a complete verification method, but in the next proposition we establish a class of models for which ordinary markers do:

Peterson	SPIN (unreduced)		SymmSpin segmented		TopSPIN sorted		TopSPIN markers	
n	states	time	states	time	states	time	states	time
3	2636	0.4	494	0.3	494	0.4	494	0.3
4	60577	0.6	3106	0.4	3106	0.4	3106	0.4
5	1.56×10^6	11	17321	1	17321	1	17321	1
6	4.48×10^7	2666	89850	7	89850	7	89850	3
7	-	-	442481	85	442481	56	442481	24
8	-	-	2.09×10^6	1166	2.09×10^6	412	2.09×10^6	175
9	-	-	9.62×10^6	16673	9.62×10^6	3034	9.62×10^6	1333

Fig. 8. Symmetry markers applied to Peterson's mutual exclusion example

Proposition 2. *Let s, s' be two states. If $s = \langle \overrightarrow{n}, \overrightarrow{s}, \overrightarrow{sn}, \overrightarrow{ss} \rangle$ with $\overrightarrow{ss} = \langle\rangle$ and $m(s) = m(s')$ then $\exists \pi . s' = s^\pi$.*

Proof. We will prove that $\overrightarrow{ss} = \langle\rangle$ implies that for any s: $m_{exact}(s) = m(s)$. Hence, by Point 2 of Proposition 1 we have proven our result. First it is clear to see that if for all $i \in I$ all markers are distinct, then $m_{exact}(s) = m(s)$ as $newval(i) = pos(i)$. Let $j \in I$ be such that $newval(j) \neq pos(j)$ (i.e., there must be at least one other $k \in I$ with $k \neq j$ with $m_s(k) = m_s(j)$). In this case we know that j does not occur as a value in $\overrightarrow{s}$ (otherwise we cannot have another $k \neq j$ with the same marker). But then, as $\overrightarrow{ss} = \langle\rangle$, applying ρ has no effect for j. This reasoning can be applied to all $j \in I$ and hence our result holds.

6 Empirical Results

We have implemented symmetry reduction using symmetry markers in the TopSPIN symmetry reduction package. The result is two new TopSPIN strategies: *exact markers* and *approx markers*. Use of the *exact markers* strategy results in complete verification since the strategy guarantees that at least one state from each symmetric equivalence class is explored. On the other hand, the *approx markers* strategy does *not* guarantee sound model checking since several equivalence classes may be represented by the same state. However, our results show that this strategy can provide a reduction in verification time whilst maintaining high state-space coverage. We illustrate our implementation using two families of Promela specifications: the Peterson mutual exclusion protocol (see Sect. 4), and an email system adapted from [CM03] (and similar to an example used for experiments in [DM06]). A configuration of the email example consists of n *client* processes which exchange messages via a *mailer* process. The *mailer* is modelled using the Promela *init* process, and can be viewed as part of the *base* process discussed earlier. Experiments were carried out on the platform described in Sect. 4, and once again a '−' result indicates that verification was intractable, or took longer than 5 hours, for a given configuration. For each configuration we check basic safety properties expressed using assertions, and for deadlock-freedom. Note that symmetry reduction can be used, in principle, for model checking symmetric CTL* formulas [ES96]; our implementation is limited to basic safety properties due to restrictions of SPIN and TopSPIN [DM06].

Email	**Spin (unreduced)**		**TopSpin segmented**		**TopSpin sorted**		**TopSpin exact markers**		**TopSpin approx markers**	
n	states	time	states	time	states	time	states	time	states	time
2	938	0.5	471	0.3	471	0.3	471	0.3	470	0.4
3	37793	0.5	6335	0.4	6361	0.5	6337	0.4	6316	0.4
4	1.33×10^6	11	56631	4	60566	2	57398	1	55711	1
5	-	-	399534	64	481964	30	430212	12	380040	9
6	-	-	2.42×10^6	1415	3.40×10^6	366	3.05×10^6	131	2.21×10^6	82
7	-	-	-	-	-	-	-	-	1.17×10^7	766

Fig. 9. Symmetry markers applied to a Promela email specification

Fig. 8 shows state-space sizes and verification times for various configurations of the Peterson protocol. To ease readability, some of the results from Fig. 4 are duplicated in Fig. 8. For the Peterson examples, the set $\overrightarrow{ss}$ is empty. Therefore, by Propositions 1 and 2, we anticipate that the *exact markers* and *approx markers* strategies should both provide full symmetry reduction *and* complete verification. This is indeed the case – the *states* column for the TopSpin *exact markers* and SymmSpin *segmented* strategies are identical. Results for the *approx markers* strategy are not shown in Fig. 8, as they are the same as for the *exact markers* strategy. Verification using symmetry markers is significantly faster than using the TopSpin *sorted* or SymmSpin *segmented* strategies.

Results for configurations of the email example are given in Fig. 9. It was not possible to apply SymmSpin to these examples due to limitations of the prototype SymmSpin implementation; therefore we used the TopSpin *segmented* strategy to compute (where practical) the optimal symmetry-reduced state-space for each configuration (see Sect. 2). Fig. 9 and Fig. 10 back up the predictions of our theory: that, with systems with arrays both indexed by scalarset and range from scalarset, the *approx markers* strategy may sometimes regard inequivalent states as equivalent, whereas the *exact markers* strategy may not always recognise equivalent states as such. The left of Fig. 10 in particular (see also Fig. 1) highlights the precision of our methods – note how close the respective curves lie to the curve for full symmetry reduction (TopSpin *segmented*). For this example, TopSpin *sorted* also computes multiple representatives from each orbit. The results of Fig. 9 clearly illustrate the benefits of using symmetry markers:

– The *exact markers* strategy outperforms TopSpin *sorted* both in terms of memory requirements and verification time
– The difference between the state-space size using full symmetry reduction compared with *exact markers* is relatively small
– The *approx markers* strategy provides very good coverage of the symmetry-reduced state-space, and runs significantly faster than the other strategies.

The value of the *approx markers* strategy is further illustrated by the fact that exact verification of the *email 7* configuration was not possible: full symmetry reduction using the *segmented* strategy is not feasible (for some states, as many as 7! symmetries would need to be considered), and the state-spaces generated using *exact markers* and standard TopSpin strategies exceed memory requirements.

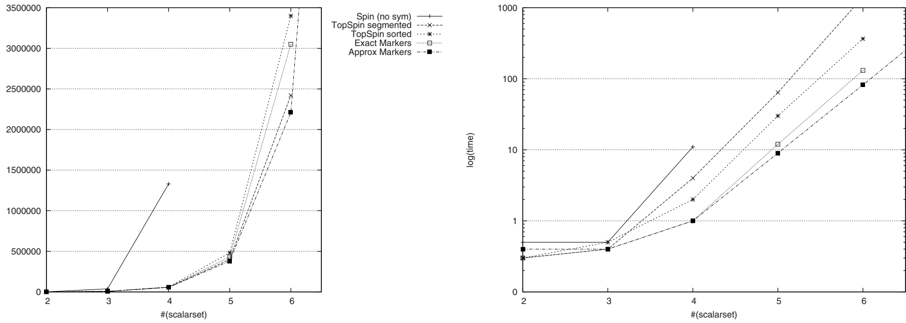


Fig. 10. Number of states and log of model checking times for the Email benchmark

Verification of deadlock-freedom using *approx markers* does not *guarantee* deadlock-freedom for the full model, but the high coverage rate of this strategy provides us with a reasonable degree of confidence that the model does not deadlock.

7 Related and Future Work

The SMC symmetry reduction tool [SGE00] employs a somewhat similar approach to our symmetry markers in order to determine state equivalence. Given two states to be tested for equivalence, SMC computes a *checksum* for each state. If the checksums are not equal then the states are not symmetrically equivalent. This simple pre-test quickly identifies many inequivalent states, but (as with markers), equality of checksums does not guarantee equivalence. The tool applies an approximate strategy to conservatively determine whether states with equal checksums are genuinely equivalent. Symmetry markers are more precise than the checksums computed by SMC, and can effectively handle the complications introduced by pid variables and pid-indexed arrays, which are not supported in the SMC input language.

Symmetry markers are currently limited to apply to fully symmetric systems. Although full symmetry occurs most frequently in practice, concurrent systems with ring or tree-like structures can exhibit other forms of structural symmetry [CEJS98, DM05]. It should be straightforward to extend the markers approach to handle multiple families of symmetric processes. A more challenging research topic involves generalising the markers approach to apply in the presence of an arbitrary symmetry group.

References

[Abr96] Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)

[BDH02] Bosnacki, D., Dams, D., Holenderski, L.: Symmetric Spin. STTT 4(1), 92–106 (2002)

[CEFJ96] Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. 9(1-2), 77–104 (1996)

[CEJS98] Clarke, E., Emerson, E., Jha, S., Sistla, A.: Symmetry reductions in model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)

[CGP99] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[CM03] Calder, M., Miller, A.: Generalising feature interactions in email. In: FIW'03, pp. 187–204. IOS Press, Amsterdam (2003)

[DM05] Donaldson, A., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 481–496. Springer, Heidelberg (2005)

[DM06] Donaldson, A., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 541–556. Springer, Heidelberg (2006)

[DM07] Donaldson, A., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. ENTCS 185, 63–76 (2007)

[ES96] Emerson, E., Sistla, A.: Symmetry and model checking. Formal Methods in System Design 9(1/2), 105–131 (1996)

[Hol88] Holzmann, G.: An improved protocol reachability analysis technique. Softw. Pract. Exper. 18(2), 137–161 (1988)

[Hol03] Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison-Wesley, Reading (2003)

[ID96] Ip, C., Dill, D.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)

[LM07] Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proc. of the International Symmetry Conference, Edinburgh, UK, pp. 71–85 (January 2007)

[McM93] McMillan, K.: Symbolic Model Checking. PhD thesis, Boston (1993)

[Pet81] Peterson, G.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)

[SGE00] Sistla, A., Gyuris, V., Emerson, E.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Trans. Softw. Eng. Methodol. 9(2), 133–166 (2000)

[Val02] Valiente, G.: Algorithms on Trees and Graphs. Springer, Heidelberg (2002)

Latticed Simulation Relations and Games

Orna Kupferman and Yoad Lustig

Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
{orna,yoadl}@cs.huji.ac.il

Abstract. Multi-valued Kripke structures are Kripke structures in which the atomic propositions and the transitions are not Boolean and can take values from some set. In particular, latticed Kripke structures, in which the elements in the set are partially ordered, are useful in abstraction, query checking, and reasoning about multiple view-points. The challenges that formal methods involve in the Boolean setting are carried over, and in fact increase, in the presence of multi-valued systems and logics. We lift to the latticed setting two basic notions that have been proven useful in the Boolean setting. We first define *latticed simulation* between latticed Kripke structures. The relation maps two structures M_1 and M_2 to a lattice element that essentially denotes the truth value of the statement "every behavior of M_1 is also a behavior of M_2". We show that latticed-simulation is logically characterized by the universal fragment of latticed μ-calculus, and can be calculated in polynomial time. We then proceed to defining latticed two-player games. Such games are played along graphs in which each transition have a value in the lattice. The value of the game essentially denotes the truth value of the statement "the $\vee$-player can force the game to computations that satisfy the winning condition". An earlier definition of such games involved a zig-zagged traversal of paths generated during the game. Our definition involves a forward traversal of the paths, and it leads to better understanding of multi-valued games. In particular, we prove a min-max property for such games, and relate latticed simulation with latticed games.

1 Introduction

Several recent verification methods involve reasoning about *multi-valued Kripke structures* in which an atomic proposition is interpreted at a state as a *lattice* element, rather than a Boolean value[1]. The multi-valued setting arises directly in systems in which the designer can give to the atomic propositions rich values like "unknown" or "don't care" (c.f., the IEEE Standard Multivalue Logic System for VHDL Model Interoperability [20]), and arise indirectly in applications like abstraction methods, in which it is useful to allow the abstract system to have unknown assignments to atomic propositions and transitions [15,2], query checking [8], which can be reduced to model checking over multi-valued Kripke structures, and verification of systems from inconsistent view-points [19], in which the value of the atomic propositions is the composition of their values in the different viewpoints.

[1] A lattice $\langle \mathcal{L}, \leq \rangle$ is a partially ordered set in which every two elements $a, b \in \mathcal{L}$ have a least upper bound (a join b) and a greatest lower bound (a meet b). We will define lattices in detail in Section 2.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 316–330, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The various applications use various types of lattices (see Figure 1). For example, in the abstraction application, researchers have used three values ordered as in $\mathcal{L}_3$ [2], as well as its generalization to linear orders [6]. In query checking, the lattice elements are sets of formulas, ordered by the inclusion order [3]. When reasoning about inconsistent viewpoints, each viewpoint is Boolean, and their composition gives rise to products of the Boolean lattice, as in $\mathcal{L}_{2,2}$ [11]. Finally, in systems with rich values of the atomic propositions, several orders may be used, allowing the modeling of uncertainty, disagreement, and relative importance [7]. In the most general setting, both the atomic propositions and the transitions in the Kripke structure are elements in a lattice [12]. We refer to such structures as *latticed Kripke structures*.

Properties of latticed Kripke structures can be specified using multi-valued logics. In particular, [4] introduced a latticed version of the μ-calculus. The value of a *latticed μ-calculus* formula ψ in a latticed Kripke structure M, denoted $[\![M, \psi]\!]$, is an element in $\mathcal{L}$ — the lattice with respect to which M and ψ are defined. Several model-checking algorithms for latticed μ-calculus are studied in the literature [4,27]. Automated tools for reasoning about multi-valued logics include theorem provers for first-order multi-valued logics (cf. [16,29]) and symbolic multi-valued model checkers (cf. [5]). Naturally, the challenges that formal methods involve in the Boolean setting are carried over, and in fact increase, in the presence of multi-valued systems and logics.

In 1971, Milner defined the *simulation* pre-order between systems [25]. Simulation enjoys many appealing properties, making it a key notion in reasoning about systems and their specifications. First, simulation has a fully abstract semantics: a Kripke structure M_2 simulates a Kripke structure M_1 iff every computation tree embedded in the unrolling of M_1 can be embedded also in the unrolling of M_2. Second, simulation has a logical characterization: M_2 simulates M_1 iff every universal branching-time formula satisfied by M_2 is satisfied also by M_1 [26,1,14]. It follows that simulation is a suitable notion of implementation, and it is the coarsest abstraction of a system that preserves universal branching-time properties. Third, simulation can be defined locally by means of a game that relates states with their immediate successor states. Based on its local definition, simulation between finite-state systems can be checked in polynomial time [9] and symbolically [17]. Finally, simulation implies trace-containment, which cannot be defined locally and requires polynomial space for verification [28]. Hence simulation is widely used both in manual and in automatic verification.

An adoption of the advantages of simulation to the multi-valued setting was partially suggested in the context of abstraction. There, *modal transition systems* (MTS) are used in order to model systems at different levels of abstraction [23]. Accordingly, atomic propositions have three possible values (false, unknown, and true), and transitions have three possible values (false, may, and must). Researchers have defined *mixed simulation* between MTSs [10,13] and use it as a precision order: M_1 is simulated by M_2 iff M_2 is more precise (less abstract) than M_1. The adoption is partial in the sense that it fits only for the special case of MTS, and it returns a Boolean value: either M_2 simulates M_1 or it does not.

In this work we define and study *latticed simulation* in general. Given two Kripke structures M_1 and M_2 over a lattice $\mathcal{L}$, the simulation value of M_1 by M_2, denoted $sim_val(M_1, M_2)$, is an element in $\mathcal{L}$ that essentially denotes the truth value of the

statement “every behavior of M_1 is also a behavior of M_2”. Technically, for two states q_1 of M_1 and q_2 of M_2, the simulation value of q_1 by q_2 refers to both the agreement between the states on the values of the atomic propositions, and to the value in which successors of q_1 can be matched with successors of q_2. The logical characterization of simulation is extended to the latticed setting: for every sentence ψ in the universal fragment of latticed μ-calculus, we have that $sim_val(M_1, M_2) \leq [\![M_2, \psi]\!] \rightarrow [\![M_1, \psi]\!]$. Thus, the greater the simulation value is, the more likely it is that the value of ψ in M_1 is not smaller than its value in M_2. The characterization is tight, in the sense that if $sim_val(M_1, M_2) \not\geq l$, for a lattice value $l \in \mathcal{L}$, then there is a sentence ψ in the universal latticed μ-calculus such that $[\![M_2, \psi]\!] \rightarrow [\![M_1, \psi]\!] \not\geq l$. In [21], we defined the *implication value* between two latticed Kripke structures M_1 and M_2, denoted $imp_val(M_1, M_2)$. Essentially, $imp_val(M_1, M_2)$ denotes the truth value of the statement “every computation of M_1 is also a computation of M_2”; thus it is the latticed counterpart of trace containment. The computational advantage of simulation with respect to trace containment is carried over to the latticed setting: $sim_val(M_1, M_2) \leq imp_val(M_1, M_2)$, and while the calculation of $imp_val(M_1, M_2)$ is PSPACE-complete [21], $sim_val(M_1, M_2)$ can be calculated in PTIME. We also define *latticed bisimulation* and study its properties.

It is easy to see that Boolean simulation and its properties are a special case of latticed simulation and its properties. This may create an impression as if the extension to the latticed setting is straightforward. To see that this is not the case, note that there are quite many extensions of the Boolean setting to a latticed setting that coincide with the Boolean setting for the Boolean lattice. For example, we could have defined latticed simulation so that if the simulation value of M_1 by M_2 is a lattice element l, then for all universal formulas ψ, if $[\![M_2, \psi]\!] \geq l$, then $[\![M_1, \psi]\!] \geq l$. To see that this definition is different, consider the three-valued linearly-ordered lattice $\{0, \frac{1}{2}, 1\}$ and assume there is a formula ψ such that $[\![M_2, \psi]\!] = \frac{1}{2}$ and $[\![M_1, \psi]\!] = 0$. A simulation value of $\frac{1}{2}$ is possible according to our definition (indeed, $\frac{1}{2} \geq (\frac{1}{2} \rightarrow 0)$) but is not possible according to the alternative definition (indeed, $[\![M_2, \psi]\!] \geq \frac{1}{2}$ yet $[\![M_1, \psi]\!] \not\geq \frac{1}{2}$). Our search for a good definition have eventually converged to the suggested one — the only definition that enjoys all the helpful properties of Boolean simulation.

Recall that in the Boolean case, simulation can be defined by means of a game between two players. We define and study *latticed games* and show that latticed simulation can be defined by means of such games. An earlier definition of latticed games is presented in [27]. As in our setting, the game graph is a latticed Kripke structure whose states are partitioned into $\vee$-states and $\wedge$-states. Also, the game is defined so that its value is a lattice element that essentially denotes the truth value of the statement “the $\vee$-player can force the games into computations that satisfy the winning condition”. The definition of the value of a game in [27], however, is conceptually different from the definition of the winner in a Boolean two-players game. Indeed, the value of a play in the game in [27] is defined as the limit of a sequence $\{val_i\}_{i=0}^{\infty}$, where each val_i is computed by backward traversal of the prefix $v_0, v_1, \ldots, v_i$ of the path generated during the play. Thus, while in Boolean games the generated path is traversed in a forward manner, here the need to calculate a lattice value has forced a zig-zagged traversal.

Our definition of a value of a game avoids the zig-zagged traversal and is based on a mutual definition of two values: one for the $\vee$-player and one for the $\wedge$ player. The values are updated during the play, and the values after the i-th transition depends on the values before the i-th transition and the value of the edge traversed during the i-th transition. The value of a game according to our definition coincides with the value in [27]. The fact our definition resembles the forward traversal in Boolean games leads to a better understanding of latticed games. In particular, we prove a min-max theorem for latticed games: the value of a game for the $\vee$-player complements the value of the game for the $\wedge$-player.[2] We note that this result is technically very challenging. In particular, unlike Boolean games, the value of a game need not be achieved with a single strategy. Beyond the relation between latticed games and latticed simulation, they are of independent interest. In particular, as discussed in [27], model checking of latticed μ-calculus can be reduced to latticed-game solving.

Due to space limitations, proofs and examples are omitted and can be found in the full version at the authors' web pages.

2 Preliminaries

Let $\langle A, \leq \rangle$ be a partially ordered set, and let P be a subset of A. An element $a \in A$ is an *upper bound* on P if $a \geq b$ for all $b \in P$. Dually, a is a *lower bound* on P if $a \leq b$ for all $b \in P$. An element $a \in A$ is the *least element of* P if $a \in P$ and a is a lower bound on P. Dually, $a \in A$ is the *greatest element of* P if $a \in P$ and a is an upper bound on P. A partially ordered set $\langle A, \leq \rangle$ is a *lattice* if for every two elements $a, b \in A$ both the least upper bound and the greatest lower bound of $\{a, b\}$ exist, in which case they are denoted $a \vee b$ (a *join* b) and $a \wedge b$ (a *meet* b), respectively. A lattice is *complete* if for every subset $P \subseteq A$ both the least upper bound and the greatest lower bound of P exist, in which case they are denoted $\bigvee P$ and $\bigwedge P$, respectively. In particular, $\bigvee A$ and $\bigwedge A$ are denoted $\top$ (*top*) and $\bot$ (*bottom*), respectively. A lattice $\langle A, \leq \rangle$ is finite if A is finite. Note that every finite lattice is complete. A lattice is *distributive* if for every $a, b, c \in A$, we have $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.

The traditional disjunction and conjunction logic operators correspond to the join and meet lattice operators. In a general lattice, however, there is no natural counterpart to negation. A *De Morgan* lattice is a lattice in which every element a has a unique complement $\neg a$ such that $\neg\neg a = a$, De Morgan rules hold, and $a \leq b$ implies $\neg b \leq \neg a$. In the rest of the paper we consider only finite[3] distributive De Morgan lattices.

In Figure 1 we describe some (finite distributive De Morgan) lattices. The elements of the lattice $\mathcal{L}_2$ are the usual truth values 1 (**true**) and 0 (**false**) with the order $0 \leq 1$. The lattice $\mathcal{L}_3$ contains in addition the value $\frac{1}{2}$, with the order $0 \leq \frac{1}{2} \leq 1$, and with negation defined by $\neg 0 = 1$ and $\neg\frac{1}{2} = \frac{1}{2}$. The lattice $\mathcal{L}_{2,2}$ is the Cartesian product of two $\mathcal{L}_2$ lattices, thus $(a, b) \leq (a', b')$ if both $a \leq a'$ and $b \leq b'$. Also,

[2] In multi-valued games, *determinancy* is generalized to having a min-max property.

[3] Note that focusing on finite lattices is not as restrictive as may first seem. Indeed, even when the lattice is infinite, the problems we consider involve only finite Kripke structures. Therefore, only a finite number of lattice elements appear in a problem, and since the lattice is distributive, the closure of logical operations on these values is still finite.

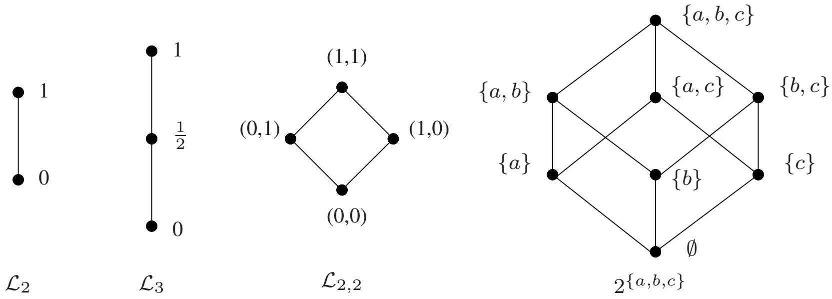


Fig. 1. Some lattices

$\neg(a,b) = (\neg a, \neg b)$. Finally, the lattice $2^{\{a,b,c\}}$ is the power set of $\{a,b,c\}$ with the set-inclusion order (that is, the transitive closure of the edges in the figure). Complementation is interpreted as set complementation relative to $\{a,b,c\}$. In this lattice, for example, $\{a\} \vee \{b\} = \{a,b\}$, $\{a\} \wedge \{b\} = \bot$, $\{a,c\} \vee \{b\} = \top$, and $\{a,c\} \wedge \{b\} = \bot$.

A *join irreducible* element is a value $l \in \mathcal{L}$ such that $l \neq \bot$ and for all $l_1, l_2 \in \mathcal{L}$, if $l_1 \vee l_2 \geq l$, then $l_1 \geq l$ or $l_2 \geq l$. For example, in $\mathcal{L}_3$ (and in every linear order), all elements are join irreducible. On the other hand, in the lattice $2^{\{a,b,c\}}$, the elements $\{a\}, \{b\}, \{c\}$, and are join irreducible, but $\{a,b\}, \{b,c\}$, and $\{a,c\}$ are not join irreducible. To see the latter, note that $\{a\} \vee \{b,c\} \geq \{a,c\}$ but $\{a\} \not\geq \{a,c\}$ and $\{b,c\} \not\geq \{a,c\}$. Birkhoff's representation theorem for finite distributive lattices implies that in order to prove that $l_1 = l_2$ it is sufficient if to prove that for every join irreducible element l it holds that $l_1 \geq l$ iff $l_2 \geq l$. We denote the set of join irreducible elements of $\mathcal{L}$ by $JI(\mathcal{L})$. A *meet irreducible* element $l \in \mathcal{L}$ is a value for which if $l_1 \wedge l_2 \leq l$ then either $l_1 \leq l$ or $l_2 \leq l$. Note that in a De Morgan lattice, an element is meet irreducible iff its complement is join irreducible.

Consider a lattice $\mathcal{L}$ (we abuse notation and refer to $\mathcal{L}$ also as a set of elements, rather than a pair of a set with an order on it). For a set X of elements, an *$\mathcal{L}$-set over X* is a function $S : X \to \mathcal{L}$ assigning to each element of X a value in $\mathcal{L}$. It is convenient to think about $S(x)$ as the truth value of the statement "x is in S". We say that an $\mathcal{L}$-set S is *Boolean* if $S(x) \in \{\top, \bot\}$ for all $x \in X$. The usual set operators can be lifted to $\mathcal{L}$-sets as expected. Given two $\mathcal{L}$-sets S_1 and S_2 over X, we define join, meet, and complementation so that for every element $x \in X$, we have $S_1 \vee S_2(x) = S_1(x) \vee S_2(x)$, $S_1 \wedge S_2(x) = S_1(x) \wedge S_2(x)$, and $comp(S_1)(x) = \neg S_1(x)$.[4]

A *latticed Kripke structure* is a 6-tuple $M = \langle \mathcal{L}, AP, Q, Q_0, R, \Theta \rangle$, where $\mathcal{L}$ is a lattice, AP is a set of atomic propositions, Q is set of states, Q_0 is an $\mathcal{L}$-set of initial states, $R : Q \times Q \to \mathcal{L}$ is an $\mathcal{L}$-set of transitions, and $\Theta : AP \to \mathcal{L}^Q$ maps each atomic proposition p to an $\mathcal{L}$-set of states, describing the truth value of p in each state.

The μ-calculus [22] is an expressive temporal logic that subsumes logics like LTL and CTL*. A multi-valued variant of the μ-calculus, was suggested and studied in [4].

[4] If S_1 and S_2 are over different domains X_1 and X_2, we can view them as having the same domain $X_1 \cup X_2$ and let $S_1(x) = \bot$ for $x \in X_2 \setminus X_1$ and $S_2(x) = \bot$ for $x \in X_1 \setminus X_2$.

We call this logic *latticed μ-calculus* (LMC, for short)[5]. For technical convenience, we consider LMC formulas in a positive form in which negation applies only to atomic propositions. Given a set AP of atomic propositions and a set Var of variables, LMC formulas have the following syntax, with $p \in AP$ and $y \in Var$.

$$\varphi = p|\neg p|\varphi_1 \vee \varphi_2|\varphi_1 \wedge \varphi_2|AX\varphi|EX\varphi|\mu y.\varphi(y)|\nu y.\varphi(y)$$

Note that the operators $\vee$ and $\wedge$ stand for "join" and "meet", rather than "or" and "and". In fixed-point formulas $\mu y.\varphi(y)$ and $\nu y.\varphi(y)$, the operators μ and ν bind free occurrences of y in φ. We use $\varphi_1 \to \varphi_2$ to abbreviate (the positive normal form of) $\neg\varphi_1 \vee \varphi_2$.

A *valuation* $\mathcal{V} : Var \to \mathcal{L}^Q$ over a lattice $\mathcal{L}$ maps the variables in Var into $\mathcal{L}$-sets of states. We write $\mathcal{V}_\perp$ for the valuation that maps every variable to the $\mathcal{L}$-set that maps every state to $\perp$ (that it, for every $y \in Var$ and $q \in Q$, it holds that $\mathcal{V}_\perp(y)(q) = \perp$). For a variable $y \in Var$ and an $\mathcal{L}$-set l, we write $\mathcal{V}[y = l]$ for the valuation that agrees with $\mathcal{V}$ except that it maps y to l.

The semantics of an LMC formula φ is defined with respect to a latticed Kripke structure M and a valuation $\mathcal{V}$ to the free variables in φ. Given such a valuation, the formula induces an $\mathcal{L}$-set of states, denoted $[\![M, \varphi]\!]_\mathcal{V}$, in which each state s of M is mapped to a value in $\mathcal{L}$ describing the truth value of the formula in s. Accordingly, given $\mathcal{V}$, a formula φ with a free variable y can be viewed as a transformer $f_\varphi^{\mathcal{V},y} : \mathcal{L}^Q \to \mathcal{L}^Q$, defined by $f_\varphi^{\mathcal{V},y}(g) = [\![M, \varphi]\!]_{\mathcal{V}[y=g]}$. We use $\mu f_\varphi^{\mathcal{V},y}$ and $\nu f_\varphi^{\mathcal{V},y}$ to denote the the least and greatest fixed-points of $f_\varphi^{\mathcal{V},y}$ with respect to $\mathcal{V}$. By the Tarksi-Knaster Theorem, these fixed-points exist.

Formally, the *interpretation* $[\![M, \varphi]\!]_\mathcal{V}$ of an LMC formula φ in a latticed Kripke structure $M = \langle \mathcal{L}, AP, Q, Q_0, R, \Theta \rangle$ and valuation $\mathcal{V}$ over a complete lattice $\mathcal{L}$ is defined as follows:[6]

$$\begin{aligned}
[\![M, p]\!]_\mathcal{V} &= \Theta(p) \\
[\![M, \neg p]\!]_\mathcal{V} &= comp(\Theta(p)) \\
[\![M, \mu y.\varphi]\!]_\mathcal{V} &= \mu f_\varphi^{\mathcal{V},y} \\
[\![M, \nu y.\varphi]\!]_\mathcal{V} &= \nu f_\varphi^{\mathcal{V},y} \\
[\![M, y]\!]_\mathcal{V} &= \mathcal{V}(y)
\end{aligned}
\qquad
\begin{aligned}
[\![M, \varphi_1 \vee \varphi_2]\!]_\mathcal{V} &= [\![M, \varphi_1]\!]_\mathcal{V} \vee [\![M, \varphi_2]\!]_\mathcal{V} \\
[\![M, \varphi_1 \wedge \varphi_2]\!]_\mathcal{V} &= [\![M, \varphi_1]\!]_\mathcal{V} \wedge [\![M, \varphi_2]\!]_\mathcal{V} \\
[\![M, EX\varphi]\!]_\mathcal{V} &= \lambda s. \bigvee_{s' \in Q} (R(s, s') \wedge [\![M, \varphi]\!]_\mathcal{V}(s')) \\
[\![M, AX\varphi]\!]_\mathcal{V} &= \lambda s. \bigwedge_{s' \in Q} (R(s, s') \to [\![M, \varphi]\!]_\mathcal{V}(s'))
\end{aligned}$$

A formula in which every variable is in the scope of a fixed-point operator is a *sentence*. If φ is a sentence, we write $[\![(M, s), \varphi]\!]$ for the value $[\![M, \varphi]\!]_{\mathcal{V}_\perp}(s)$. Since we almost exclusively deal with sentences, we abuse notation and omit the valuation $\mathcal{V}_\perp$.

The *value* of an LMC sentence φ in a latticed Kripke structure M, denoted $[\![M, \varphi]\!]$, is $\bigwedge_{q \in Q}(Q_0(q) \to [\![M, \varphi]\!](q))$. Note that we get the standard Boolean semantics of μ-calculus as a special case.

The universal fragment of LMC, termed ALMC, consists of the formulas that do not contain the EX operator. Note that since we assume that formulas are in a positive normal form, the above syntactic restriction implies that indeed formulas of ALMC can only impose universal requirements. Finally, the fixed-point free fragment consists of

[5] The logic is termed μL in [4], and is termed $\mathcal{L}_\mu$ in [27]. We prefer a terminology that does not involve mathematical symbols.

[6] We use $\lambda s.\theta(s)$ to denote the $\mathcal{L}$-set in which each state s is mapped to $\theta(s)$.

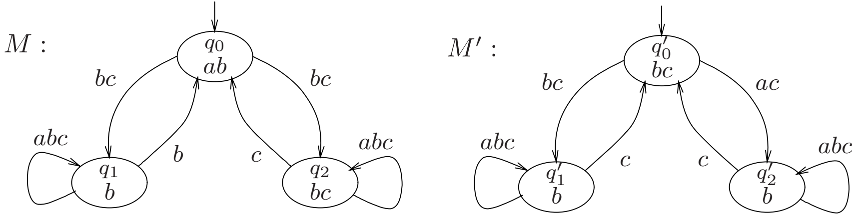


Fig. 2. Latticed Kripke structures over $\mathcal{L} = 2^{\{a,b,c\}}$

formulas that do not contain a fixed-point operator. Thus, it corresponds to a latticed version of Modal Logic, we term it LML, and term its universal fragment ALML.

Example 1. Consider the latticed Kripke structure $M = \langle 2^{\{a,b,c\}}, \{p\}, \{q_0, q_1, q_2\}, Q_0, R, \Theta\rangle$, where $Q_0(q_0) = \top$ and $Q_0(q_1) = Q_0(q_2) = \bot$, appearing in Figure 2. The values of R and Θ are described in the figure (we describe only transitions with value greater than $\emptyset$; in the description of the value of the transitions and the value of p inside the states, we omit the $\{\}$ notation). For example, $\Theta(p)(q_0) = \{a, b\}$ and $R(q_0, q_1) = \{b, c\}$. Also, $Q_0(q_0) = \{a, b, c\}$ and the initial value of q_1 and q_2 is $\emptyset$.

Below we describe the truth value of some LMC formulas in M. Since q_0 is the only state with $Q_0(q_0) \neq \bot$ and $Q_0(q_0) = \{a, b, c\}$, we have $[\![M, \varphi]\!] = [\![(M, q_0)]\!](\varphi)$.

- $[\![M, p]\!] = \{a, b\}$
- $[\![M, EXp]\!] = (\{b, c\} \wedge \{b\}) \vee (\{b, c\} \wedge \{b, c\}) = \{b, c\}$
- $[\![M, AXp]\!] = (\{b, c\} \to \{b\}) \wedge (\{b, c\} \to \{b, c\}) = (\{a, b\} \wedge \{a, b, c\}) = \{a, b\}$

Note that in the Boolean case, a state may satisfy $AX\theta$ without satisfying $EX\theta$ only if it does not have successors. In the latticed setting, the transitions to the successors have values. This is why the value of $EX\theta$ may not be greater than the value of $AX\theta$, as we see here.

Let us now calculate $[\![M, \nu z.p \wedge AXz]\!]$; that is, the truth value of "p holds at all reachable states". Let $\theta(z) = p \wedge AXz$. We start with z_0 that maps all states to $\{a, b, c\}$. We then iterate θ as described in the table in the right. A fixed-point is reached when $z_2 = z_3$, and $[\![M, \nu z.p \wedge AXz]\!] = \{a, b\}$. Intuitively, p holds at all reachable states from both viewpoints a and b: From viewpoint b, the proposition p indeed holds at all states. From viewpoint a, the proposition p does not hold in states q_1 and q_2, but these states are not reachable.

	q_0	q_1	q_2
z_0	{a,b,c}	{a,b,c}	{a,b,c}
z_1	{a,b}	{b}	{b,c}
z_2	{a,b}	{b}	{b}
z_3	{a,b}	{b}	{b}

□

3 Latticed Simulation

In this section we define latticed simulation — an extention of the definition of the simulation pre-order of [25] to the latticed context. Let $M_1 = \langle \mathcal{L}, AP, Q_1, Q_0^1, R_1, \Theta_1\rangle$ and $M_2 = \langle \mathcal{L}, AP, Q_2, Q_0^2, R_2, \Theta_2\rangle$ be two latticed Kripke structures. In the Boolean case, a relation $S \subseteq Q_1 \times Q_2$ is a simulation relation if two conditions are satisfied:

First, simulating states satisfy the same propositions. Second, if $q_2 \in Q_2$ simulates $q_1 \in Q_1$ then for every successor q_1' of q_1 there exists a successor q_2' of q_2 such that q_2' simulates q_1'. In the latticed setting, the simulation relation is an $\mathcal{L}$-set $S \in \mathcal{L}^{Q \times Q}$. Intuitively, $S(q_1, q_2)$ describes the truth value of the statement "every behavior of M_1 is also a behavior of M_2". As in the Boolean case, the simulation value of a pair of states q_1 and q_2 depends both in agreement on the values of the atomic propositions in q_1 and q_2 and in the ability to match successors of q_1 with successors of q_2.

We capture the first condition by the value

$$S_{AP}(q_1, q_2) = \bigwedge_{p \in AP} [\![M_1, p]\!](q_1) \leftrightarrow [\![M_2, p]\!](q_2).$$

We capture the second condition by the value

$$S_R(q_1, q_2) = \bigwedge_{q_1' \in Q_1} \left(R_1(q_1, q_1') \to \bigvee_{q_2' \in Q_2} (R_2(q_2, q_2') \wedge S(q_1', q_2')) \right).$$

An $\mathcal{L}$-relation $S : Q_1 \times Q_2 \to \mathcal{L}$ is an *$\mathcal{L}$-simulation* from M_1 to M_2 if for all $q_1 \in Q_1$ and $q_2 \in Q_2$, we have $S(q_1, q_2) = S_{AP}(q_1, q_2) \wedge S_R(q_1, q_2)$.

In the Boolean setting, $S(q_1, q_2)$ guarantees that all universal μ-calculus formulas that are satisfied in q_2 are also satisfied in q_1. In the latticed setting, we have the following

Theorem 2. *Consider an $\mathcal{L}$-simulation $S : Q_1 \times Q_2 \to \mathcal{L}$. For all states $q_1 \in Q_1$ and $q_2 \in Q_2$ and for all ALMC sentences ψ, we have $S(q_1, q_2) \leq [\![M_2, \psi]\!](q_2) \to [\![M_1, \psi]\!](q_1)$.*

Note that the relation S_R depends on S, thus there may be several latticed-simulation relations. We define the *maximal simulation relation* to be the relation S^* that maps every pair of states to the join of their image under every simulation relation. Formally, we define $S^*(q_1, q_2) = \bigvee_{S \in SL(M_1, M_2)} S(q_1, q_2)$, where $SL(M_1, M_2)$ is the set of simulation relations from M_1 to M_2.

We now justify the definition by showing that the maximal simulation relation is indeed a simulation relation, and furthermore, it can be easily computed.

Theorem 3. *The relation S^* is a simulation relation, and it can be computed in polynomial time.*

We now show that logical characterization by ALMC is tight for the maximal simulation. While the idea is similar to the logical characterization in the Boolean case, the technical details are complicated. In particular, we rely on Birkhoff's representation theorem and, as in the proof of Theorem 2, rely heavily on the lattice being a De Morgan distributive lattice.

Theorem 4. *For every pair of states $q_1 \in Q_1$ and $q_2 \in Q_2$, and value $l \in \mathcal{L}$, if $S^*(q_1, q_2) \not\geq l$, then there exists an ALML formula φ such that $[\![M_2, \varphi]\!](q_2) \to [\![M_1, \varphi]\!](q_1) \not\geq l$.*

For two Lattice Kripke Structures $M_1 = \langle \mathcal{L}, AP, Q_1, Q_0^1, R_1, \Theta_1 \rangle$ and $M_2 = \langle \mathcal{L}, AP, Q_2, Q_0^2, R_2, \Theta_2 \rangle$, we define the *simulation value* of M_1 by M_2 to be

$$S^*(M_1, M_2) = \bigwedge_{q_1 \in Q_1} \left(Q_0^1(q_1) \to (\bigvee_{q_2 \in Q_2} (Q_0^2(q_2) \wedge S^*(q_1, q_2))) \right),$$

where S^* is the maximal simulation relation of the two structures.

Theorem 5 below gives the full logical characterization of latticed simulation. It follows from Theorems 2 and 4, and the distributivity of the lattice.

Theorem 5. *Let M_1 and M_2 be two Kripke structures.*

1. *For all ALMC sentences ψ, we have $S^*(M_1, M_2) \leq [\![M_2, \psi]\!] \to [\![M_1, \psi]\!]$.*
2. *For all $l \in \mathcal{L}$, if $S^*(M_1, M_2) \not\geq l$, then there exists an ALML formula ψ such that $[\![M_2, \psi]\!] \to [\![M_1, \psi]\!] \not\geq l$.*

In [21], we defined the implication value between latticed automata. The definition extends to latticed Kripke structures: for two latticed Kripke structures M_1 and M_2 over a lattice $\mathcal{L}$, let $imp_val(M_1, M_2)$ be the implication value of M_2 by M_1. Essentially (for details, see [21]), each word $w \in (2^{AP})^\omega$ has a "membership value" in M_1 and in M_2, and $imp_val(M_1, M_2)$ denotes the truth value of the statement "for all words, the membership value in M_1 implies the membership value in M_2". As in the Boolean case, the branching setting is more general than the linear setting. Ee have the following.

Theorem 6. *For all latticed Kripke structures M_1 and M_2, we have $S^*(M_1, M_2) \leq imp_val(M_1, M_2)$.*

Thus, latticed simulation, which can be calculated in polynomial time, is a lower bound to the implication value, whose calculation is PSPACE-complete.

3.1 Latticed Bisimulation

The Boolean simulation pre-order has a symmetric version, namely the bisimulation relation. Two Kripke structures that are bisimilar have exactly the same behaviors. Adding symmetry to our definition of latticed simulation results in a *latticed bisimulation* relation. Formally, for two lattice kripke structures $M_1 = \langle \mathcal{L}, AP, Q_1, Q_0^1, R_1, \Theta_1 \rangle$ and $M_2 = \langle \mathcal{L}, AP, Q_2, Q_0^2, R_2, \Theta_2 \rangle$, an $\mathcal{L}$-relation $S : Q_1 \times Q_2 \to \mathcal{L}$ is an *$\mathcal{L}$-bisimulation* between M_1 and M_2 if

$$S(q_1, q_2) = S_{AP}(q_1, q_2) \wedge S_R(q_1, q_2) \wedge S_R(q_2, q_1),$$

where S_{AP} and S_R are as in $\mathcal{L}$-simulation.

The logical characterization of $\mathcal{L}$-simulation extends $\mathcal{L}$-bisimulation, now with LMC.

Theorem 7. *Let S be the maximal $\mathcal{L}$-bisimulation relation between M_1 and M_2.*

1. *For all LMC sentences φ, it holds that $S(q_1, q_2) \leq [\![M_1, \varphi]\!](q_1) \leftrightarrow [\![M_2, \varphi]\!](q_2)$.*
2. *For all $l \in \mathcal{L}$, if $S(M_1, M_2) \not\geq l$, then there exists an LML formula ψ such that $[\![M_2, \psi]\!] \leftrightarrow [\![M_1, \psi]\!] \not\geq l$.*

Other properties of latticed simulation extend easily to latticed bisimulation: it can be computed in polynomial time, and the bisimulation value between latticed Kripke structures is a lower bound to the equivalence value (two-sided implication) between them.

4 Latticed Games

A *latticed game graph* is a pair $G = \langle V, E \rangle$, where V is a set of vertices and $E : V \times V \to \mathcal{L}$ is an $\mathcal{L}$-set of transitions. The vertices are partitioned into two sets, $V_\vee$ and $V_\wedge$ (referred to as the $\vee$-vertices and the $\wedge$-vertices). A *latticed game* is a latticed game graph together with an initial state $v_0 \in V$ and a acceptance condition α. We postpone the description of the acceptance condition since for that we need the notion of a play that we define next.

Intuitively, a play of the game proceeds as follows: a token is put on some initial vertex. If the token is placed on a $\vee$-vertex then the $\vee$-player chooses an edge originating at the vertex on which the token is on, and the token is advanced along that edge. Similarly, if the token is placed on a $\wedge$-vertex, then the $\wedge$-player is doing the choosing. After the token is advanced to the successor vertex, the process repeats. This proceeds forever and the *play* of the game is a sequence of vertices $p = \{v_i\}_{i=0}^{\infty}$ (the sequence of vertices the token has traversed during the play). Each play is assigned a value, in a fashion we will describe next. Intuitively, the $\vee$-player's objective is to maximize the value of the play, while the $\wedge$-player's objective is to minimize it. We proceed to define the value of a play formally. Note that the value of a play is defined from the $\vee$-player perspective, and is in fact the value of the game for the $\vee$-player. We postpone the definition of the value of the game for the $\wedge$-player.

The acceptance value of the play, denoted $acc(p)$, stands for the value with which the play satisfies the acceptance condition. For example, an $\mathcal{L}$-Büchi acceptance condition is an $\mathcal{L}$-set of states $F \in \mathcal{L}^V$, and the acceptance value of the play p is $\bigwedge_{i=0}^{\infty} \bigvee_{j>i} F(v_i)$.

The value of a play is set not only by its acceptance value, but also by the values of the edges traversed during the play. Intuitively, when a player traverses an edge with low value he gives up more then if would have traversed an edge with a higher value. Assume, for example, that the underlying lattice is $2^{\{a,b,c\}}$ and in the first move the $\vee$-player traversed an edge with value $\{a\}$. This means that the $\vee$-player gives up all values that are not smaller or equal to $\{a\}$, and "is willing" that at the end of the play the acceptance value would be met with $\{a\}$. By dual reasoning, if the first move is done by the $\wedge$-player over an edge with value $\{a\}$, then the $\wedge$-player (whose objective is to lower the total play value) "is willing" that at the end of the game the value would be joined with $\neg\{a\} = \{b, c\}$. Furthermore, the order in which edges are traversed is important: if one player gave up some value l, the other player is assured the value l, and may move freely along edges that would have implied giving up l on other circumstances.

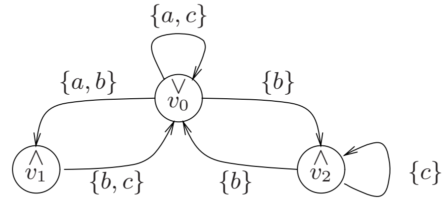
We therefore define two "given up" values, $r^\vee$ and $r^\wedge$. These values are defined inductively along the play, where at the beginning neither player has given up anything, thus $r_0^\vee = \top$ and $r_0^\wedge = \bot$. If $p_i \in V_\vee$, then the next transition is taken by the $\vee$-player. If the $\vee$-player chooses to traverse an edge with value different than $\top$, then he gives up the value of the edge traversed, except the parts of the value already given up by the $\wedge$-player. Thus, $r_{i+1}^\vee = r_i^\vee \wedge (E(p_i, p_{i+1}) \vee r_i^\wedge)$. The $\wedge$-player, on the other hand,

gives up nothing, and therefore $r^{\wedge}_{i+1} = r^{\wedge}_i$. Similarly, if $p_i \in V_{\wedge}$, then $r^{\vee}_{i+1} = r^{\vee}_i$, and $r^{\wedge}_{i+1} = r^{\wedge}_i \vee (\neg E(p_i, p_{i+1}) \wedge r^{\vee}_i)$. Since both $\{r^{\vee}_i\}$ and $\{r^{\wedge}_i\}$ are monotonic, their limit is defined. Let $r^{\vee} = \bigwedge_{i=0}^{\infty} r^{\vee}_i$ and $r^{\wedge} = \bigvee_{i=0}^{\infty} r^{\wedge}_i$

To define the value of a play we need one more technical observation. Let $val^{\vee\wedge}(p) = (acc(p) \wedge r^{\vee}) \vee r^{\wedge}$, and $val^{\wedge\vee}(p) = (acc(p) \vee r^{\wedge}) \wedge r^{\vee}$. Similarly, let $val^{\vee\wedge}_i(p) = (acc(p) \wedge r^{\vee}_i) \vee r^{\wedge}_i$, and $val^{\wedge\vee}_i(p) = (acc(p) \vee r^{\wedge}_i) \wedge r^{\vee}_i$. We will shortly prove that $val^{\wedge\vee}(p) = val^{\vee\wedge}(p)$ and define the *value* of the play, denoted $val(p)$, to equal both.

Example 8

Consider the game G over the lattice $2^{\{a,b,c\}}$ appearing on the right. Assume that all the computations of G are accepting with value $\{a, b, c\}$. We calculate the value of some plays in G.



- Consider the play $p = (v_0, v_0, v_1)^{\omega}$. By definition, $r^{\vee}_0 = \{a, b, c\}$ and $r^{\wedge}_0 = \emptyset$. Since the first transition is by the $\vee$-player, we have, $r^{\vee}_1 = r^{\vee}_0 \wedge (E(v_0, v_0) \vee r^{\wedge}_0) = \{a, b, c\} \wedge (\{a, c\} \vee \emptyset) = \{a, c\}$. Also, $r^{\wedge}_1 = r^{\wedge}_0 = \emptyset$. The second transition is also by the $\vee$-player, thus $r^{\vee}_2 = r^{\vee}_1 \wedge (E(v_0, v_1) \vee r^{\wedge}_1) = \{a, c\} \wedge (\{a, b\} \vee \emptyset) = \{a\}$. Also, $r^{\wedge}_2 = r^{\wedge}_1 = \emptyset$. The third transition is by the $\wedge$-player, thus $r^{\wedge}_3 = r^{\wedge}_2 \vee (\neg E(v_1, v_0) \wedge r^{\vee}_2) = \emptyset \vee (\{a\} \wedge \{a\}) = \{a\}$. Also, $r^{\vee}_3 = r^{\vee}_2 = \{a\}$. At this point, the sequences $r^{\vee}_i$ and $r^{\wedge}_i$ reach a fixed-point, thus $r^{\vee} = r^{\wedge} = \{a\}$. Hence, $val^{\vee\wedge}(p) = val^{\wedge\vee}(p) = \{a\}$.
- Consider now the play $p = v_0^{\omega}$. Here, $r^{\vee} = \{a, c\}$ and $r^{\wedge} = \emptyset$. Accordingly, $val^{\vee\wedge}(p) = (\{a, b, c\} \wedge \{a, c\}) \vee \emptyset = \{a, c\}$, which equals $val^{\wedge\vee}(p) = (\{a, b, c\} \vee \emptyset) \wedge \{a, c\} = \{a, c\}$.
- Consider now the play $p = (v_0, v_2)^{\omega}$. Here, $r^{\vee} = \{b\}$ and $r^{\wedge} = \emptyset$. Accordingly, $val^{\vee\wedge}(p) = val^{\wedge\vee}(p) = \{b\}$. □

Another definition of the value of a play was introduced by [27]. In [27], the authors define the value of a finite prefix of the play $p_0, \ldots, p_i$ in a backward manner. First, $val^i_i(p) = acc(p)$. Then, for $j \leq i$, we have $val^i_{j-1}(p) = val^i_j(p) \wedge E(p_{j-1}, p_j)$ if p_{j-1} is a $\vee$-vertex, or $val^i_{j-1} = val^i_j \vee \neg E(p_{j-1}, p_j)$ if p_{j-1} is a $\wedge$-vertex. The value of the prefix $p_0, \ldots, p_i$ is then $val^i = val^i_0$. It can be shown that the sequence $\{val^i\}_{i=0}^{\infty}$ stabilizes, and the value of the play is taken to be the limit. Thus, for the entire play, the value is calculated in a zig-zagged manner: in each iteration, a vertex is added, and then the calculation proceeds backwardly. Our definition, on the other hand, involves a forward traversal, and is similar to the definition in the Boolean case. As the next lemma shows, the intermediate values we get in our forward traversal coincide with these that [27] gets in the zig-zagged traversal.

Lemma 1. *For every play $p = p_0 p_1 \ldots$, and every $i \geq 0$, we have $val^{\vee\wedge}_i(p) = val^{\wedge\vee}_i(p) = val^i$.*

Since both $\{r^{\vee}_j\}_{j=1}^{\infty}$ and $\{r^{\wedge}_j\}_{j=1}^{\infty}$ are monotone, they must stabilize eventually, implying an equivalence among the three values:

Corollary 1. $val^{\wedge\vee}(p) = val^{\vee\wedge}(p) = \lim val^i$.

We define the value of a play p to be $val^{\wedge\vee}(p)$ and denote it by $val(p)$. We also define the *value of a play* p *for the* $\wedge$*-player*, denoted $val_\wedge(p)$, as the negation of the value of the game for the $\vee$-player, i.e., $\neg val(p)$.

A *strategy* for a player is a function from prefixes of plays ending in one of his vertices, to the set of vertices. Thus, a $\vee$-player strategy is $f : V^* \cdot V_\vee \to V$. A prefix $p_0, \ldots, p_n$ is consistent with a strategy f of the $\vee$-player, if for all $j \geq 0$ it holds that if p_j is a $\vee$-vertex then $p_{j+1} = f(p_0, \ldots, p_j)$. Similarly, a strategy for the $\wedge$-player is a function $f : V^* \cdot V_\wedge \to V$, and a prefix $p_0, \ldots, p_n$ is consistent f, if for all $j \geq 0$ it holds that if p_j is a $\wedge$-vertex then $p_{j+1} = f(p_0, \ldots, p_j)$. A play is consistent with a strategy if all its prefixes are consistent the strategy. It is easy to see that for every two strategies, one for the $\vee$-player and one for the $\wedge$-player, there is exactly one play consistent with both strategies. Thus, two strategies induce a play.

The *value of a strategy* f, denoted $val(f)$, is the meet of all the plays compatible with f (this holds for strategies of either player). The *value of a game* for a player is the join of the values of that player's strategies. Thus, the value of a game G for the $\vee$-player, denoted $val_\vee(G)$, is the join of all values of strategies of the $\vee$-player. Similarly, the value of a game for the $\wedge$-player, denoted $val_\wedge(G)$, is the join of all values of strategies of the $\wedge$-player. In Theorem 12 we prove that $val_\wedge(G) = \neg val_\vee(G)$.

Example 9. Consider again the game G discussed in Example 8. Consider a strategy f for the $\vee$-player that whenever the play is in v_0, goes to v_2. There are infinitely many plays that are compatible of this strategy (depending on the move of the $\wedge$-player whenever the play is in v_2). All these plays, however, have value $\{b\}$. Thus, $val(f) = \{b\}$. Consider now a strategy f' the $\vee$-player that whenever the play is in v_0, goes to v_0. Only the play v_0^ω is compatible with f. The value of this play is $\{a, c\}$, Thus, $val(f') = \{a, c\}$. It follows that $val_\vee(G) = \{a, b, c\}$. □

Remark 10. Unlike the Boolean case, the $\vee$-player might not have a single strategy ensuring him the value of the game. In fact, it might be the case that the value of the game cannot be obtained as the value of a single play. □

4.1 Properties of Lattice Games

In the latticed case, unlike the Boolean case, it is not necessarily true that a game value can be obtained in a single play. Therefore, it does not make sense to search for a single strategy as a solution to the game. What is true, is that for each join irreducible element $l \in JI(\mathcal{L})$ it holds that if the game value is greater than or equal to l, then there exists a single strategy that ensures that a value of at least l, is obtained. Thus, Birkhoff's representation theorem enables us to decompose a latticed game to several Boolean games. Formally, we have the following.

Theorem 11. *For a latticed game* $G = \langle V, E\rangle$*, over a lattice* $\mathcal{L}$*, there exists a family of Boolean games* $\{G_l = \langle V, E_l\rangle\}_{l \in JI(\mathcal{L})}$ *all sharing the same state space, such that the* $\vee$*-player has a winning strategy in* G_l *iff there is a strategy in* G *ensuring the* $\vee$*-player a value greater than or equal to* l*.*

Furthermore, for every $l \in JI(\mathcal{L})$, the game G_l can be computed in logarithmic space from G. In addition, the strategy ensuring a value greater than or equal to l can be computed, in logarithmic space, from a winning strategy in G_l and vice versa.

As expected, the state space of G_l agrees with the state space of G, and the partition to $\vee$-vertices and $\wedge$-vertices is also as in G. The challenging part is to define the set of edges according to l. The set of edges is a subset of the edges in G, and an edge $\langle v, u\rangle$ of G exists in G_l if either $v \in V_\vee$ and the edge $\langle v, u\rangle$ has value greater than or equal to l in G, or $v \in V_\wedge$ and the edge $\langle v, u\rangle$ does not have value less than or equal to $\neg l$ in G.

Theorem 11 suggests a way to solve a latticed game by decomposing it into Boolean games and solving each of those. A different algorithm is suggested in [27].

Recall that Boolean Büchi games are determined: in every game, one of the players has a winning strategy. Extending this result to the latticed setting amounts to proving that for every value l, if the value of the game for the $\vee$-player is greater than l, then $\neg l$ is greater than the value of the game for the $\wedge$-player.

Theorem 12. *For a lattice game G, we have $val_\wedge(G) = \neg val_\vee(G)$.*

4.2 The Simulation Game

For two latticed Kripke structures $M_1 = \langle \mathcal{L}, AP, Q_1, Q_0^1, R_1, \Theta_1\rangle$ and $M_2 = \langle \mathcal{L}, AP, Q_2, Q_0^2, R_2, \Theta_2\rangle$, the *simulation game for M_1 and M_2* is a latticed game defined as follows. Intuitively, the $\wedge$-player "claims" that M_1 is not simulated by M_2, while the $\vee$-player "claims" that M_1 is simulated by M_2. The game value is then $S^*(M_1, M_2)$. Thus, in the beginning of the game, the $\wedge$-player chooses an initial state q_1 in M_1, and the $\vee$-player chooses an initial state q_2 in M_2 that is supposed to resemble q_1. We measure the resemblance value between q_1 and q_2 by $S_{AP}(q_1, q_2) = \bigwedge_{p\in AP}([\![M_1, p]\!](q_1) \leftrightarrow [\![M_2, p]\!](q_2))$. The game proceeds by the $\wedge$-player choosing a successor to q_1, denoted q_1', followed by the $\vee$-player choosing a successor to q_2, denoted q_2'. Again, q_2' is supposed to resemble q_1'. This process is iterated ad infinitum.

Naturally, the edges values correspond to the transitions taken, thus the edge chosen by the $\wedge$-player to move from q_1 to q_1' has the value of the transition $R_1(q_1, q_1')$. The values of the $\vee$-player transitions reflect not only the value of the corresponding transition in M_2, but also the resemblance between the state in M_1 to the state in M_2. Therefore, the value of the $\vee$-player transition is the value of the transition in M_2 meet the value $S_{AP}(q_1', q_2')$. We now to define the game formally.

The game graph is $G_{\langle M_1, M_2\rangle} = \langle V, E\rangle$, where $V = (Q_1 \times Q_2 \times \{\wedge, \vee\}) \cup \{in_\wedge\} \cup (Q_1 \times \{in_\vee\})$. The $\wedge$-vertices are $(Q_1 \times Q_2 \times \{\wedge\}) \cup \{in_\wedge\}$, and the $\vee$-vertices are $(Q_1 \times Q_2 \times \{\vee\}) \cup (Q_1 \times \{in_\vee\})$. The initial position is $in_\wedge$, and the edges are defined as follows. For every $q_1 \in Q_1$, there exists an edge from $in_\wedge$ to $\langle q_1, in_\vee\rangle$ with value $Q_0^1(q_1)$. For every $q_1 \in Q_1$ and $q_2 \in Q_2$ there is an edge from $\langle q_1, in_\vee\rangle$ to $\langle q_1, q_2, \wedge\rangle$ with value $Q_0^2(q_2) \wedge S_{AP}(q_1, q_2)$. For $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$, the edge from $\langle q_1, q_2, \wedge\rangle$ to $\langle q_1', q_2, \vee\rangle)$ has value $R_1(q_1, q_1')$, and the edge from $\langle q_1', q_2, \vee\rangle$ to $\langle q_1', q_2', \wedge\rangle)$ has value $R_2(q_2, q_2') \wedge S_{AP}(q_1', q_2')$. All other edges have value $\bot$. The acceptance criteria is Büchi in which all vertices are accepting with value $\top$ (making the acceptance value of every play $\top$).

We now claim that the value of the simulation game is the simulation value of M_1 by M_2. The proof, detailed in the full paper, shows that for every join irreducible element $l \in \mathcal{L}$, the value of the simulation game is greater than or equal to l iff $S^*(M_1, M_2)$ is greater than or equal to l.

Theorem 13. $val_\vee(G_{\langle M_1, M_2 \rangle}) = S^*(M_1, M_2)$.

Thus, as in the Boolean setting, latticed simulation can be defined in terms of a game between two players.

5 Discussion

We lifted the notions of simulation and games to a multi-valued setting. We considered values taken from a lattice, and we were able to lift the known properties of simulation and games to the latticed setting. In the Boolean setting, bisimulation is an equivalence relation, and an abstraction of a system (one that agrees with the original system on all μ-calculus specifications) can be obtained by merging bisimilar states to one state. In the latticed setting, bisimulation associates with each two states a lattice element denoting their bisimulation value. Therefore, even if we settle on a lattice element l and seek an abstraction whose bisimulation value with the original system is l, it is not clear how to define the state space and the transitions of the abstraction. Finding a satisfying definition would enable us to find coarsest abstractions that may not agree with the original system on all specifications, but for which we can provide a lower bound on the value of agreement (i.e., most view-points agree).

Another open question is the extension of latticed simulation to Kripke structures with fairness. In the Boolean setting, the relation between simulation and games has led to a definition of fair simulation that retains the logical characterization and the computational advantages of simulation [18]. While the relation between simulation and games is maintained in the latticed setting, it is an open question whether latticed games can be used in a definition of latticed fair simulation. Indeed, the definition in [18] relates a strategy that generates computations in the simulated structure with a strategy that generates computations in the simulating structure. In the latticed setting, the value of the game may depend on different strategies, thus a game-based definition of fair simulation has to take all strategies into an account.

References

1. Bensalem, S., Bouajjani, A., Loiseaux, C., Sifakis, J.: Property preserving simulations. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 260–273. Springer, Heidelberg (1993)
2. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
3. Bruns, G., Godefroid, P.: Temporal logic query checking. In: Proc. 16th LICS, pp. 409–420. IEEE Computer Society Press, Los Alamitos (2001)
4. Bruns, Godefroid: Model checking with multi-valued logics. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 281–293. Springer, Heidelberg (2004)

5. Chechik, M., Devereux, B., Easterbrook, S.: Implementing a multi-valued symbolic model checker. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 404–419. Springer, Heidelberg (2001)
6. Chechik, M., Devereux, B., Gurfinkel, A.: Model-checking infinite state-space systems with fine-grained abstractions using SPIN. In: Dwyer, M.B. (ed.) SPIN. LNCS, vol. 2057, pp. 16–36. Springer, Heidelberg (2001)
7. Chechik, M., Easterbrook, S., Petrovykh, V.: Model checking over multi-valued logics. In: Formal Methods Europe (2001)
8. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
9. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: A semantics-based tool for the verification of concurrent systems. ACM TOPLAS 15, 36–72 (1993)
10. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM TOPLAS 19(2), 253–291 (1997)
11. Easterbrook, S., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: Proc. 23rd Int. Conf. on Software Engineering, pp. 411–420. IEEE Computer Society Press, Los Alamitos (2001)
12. Fitting, M.C.: Many-valued modal logics. Fundamenta Informaticae XV, 235–254 (1991)
13. Godefroid, P., Jagadeesan, R.: Automatic abstraction using generalized model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 137–150. Springer, Heidelberg (2002)
14. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
15. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
16. Hähnle, R.: Automated deduction in multiple-valued logics. International Series of Monographs on Computer Science 10 (1994)
17. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. 36th FOCS, pp. 453–462 (1995)
18. Henzinger, T.A., Kupferman, O., Rajamani, S.: Fair simulation. I& C 173(1), 64–81 (2002)
19. Hussain, A., Pradhan, S.: Consistent partial model checking. ENTCS 73, 45–85 (2004)
20. IEEE. IEEE standard multivalue logic system for VHDL model interoperability (Std_logic_1164) (1993)
21. Kupferman, O., Lustig, Y.: Lattice automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
22. Kozen, D.: Results on the propositional μ-calculus. TCS 27, 333–354 (1983)
23. Larsen, K.G., Thomsen, G.B.: A modal process logic. In: Proc. 3rd LICS (1988)
24. Martin, D.A.: Borel determinacy. Annals of Mathematics 65, 363–371 (1975)
25. Milner, R.: An algebraic definition of simulation between programs. In: Proc. 2nd Int. Joint Conf. on Artificial Intelligence, pp. 481–489. British Computer Society (1971)
26. Pnueli, A.: Linear and branching structures in the semantics and logics of reactive systems. In: Brauer, W. (ed.) ICALP. LNCS, vol. 194, pp. 15–32. Springer, Heidelberg (1985)
27. Shoham, S., Grumberg, O.: Multi-valued model checking games. In: Peled, D.A., Tsay, Y.K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 354–369. Springer, Heidelberg (2005)
28. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proc. 5th STOC, pp. 1–9 (1973)
29. Sofronie-Stokkermans, V.: Automated theorem proving by resolution for finitely-valued logics based on distributive lattices with operations. Multiple-Valued Logics: An International Journal 5(2) (2000)

Providing Evidence of Likely Being on Time: Counterexample Generation for CTMC Model Checking

Tingting Han[1,2] and Joost-Pieter Katoen[1,2]

[1] Software Modelling and Verification, RWTH Aachen University, Germany
[2] Formal Methods and Tools, University of Twente, The Netherlands
{tingting.han,katoen}@cs.rwth-aachen.de

Abstract. Probabilistic model checkers typically provide a list of individual state probabilities on the refutation of a temporal logic formula. For large state spaces, this information is far too detailed to act as useful diagnostic feedback. For quantitative (constrained) reachability problems, sets of paths that carry enough probability mass are more adequate. We recently have shown that in the context of discrete-time probabilistic processes, such sets of smallest size can be efficiently computed by (hop-constrained) k-shortest path algorithms. This paper considers the problem of generating counterexamples for continuous-time Markov chains. The key contribution is a set of approximate algorithms for computing small sets of paths that indicate the violation of time-bounded (constrained) reachability probabilities.

1 Introduction

A major strength of model checking is the possibility to generate counterexamples in case of a property violation. In fact, it is this facility that makes model checking an effective bug hunting technique. Even if only a fragment of the entire model can be searched, such counterexamples provide useful diagnostic feedback. Efficient algorithms for generating (succinct) counterexamples therefore have received considerable attention by the model checking community, cf. [5,9,18]. For probabilistic models, though, counterexample generation is far less developed.

Model checking of probabilistic models is focused on verifying system models in which transitions are equipped with random information. Popular models are discrete- and continuous-time Markov chains (DTMCs and CTMCs, respectively), and variants thereof which exhibit nondeterminism. Most probabilistic model checkers support variants of CTL [3,4,11]. For quantitative properties such as "the (maximal) probability to reach a set of goal states by avoiding certain states is at most p", alternative algorithms have to be employed. In case such property is refuted, the idea is to provide a set of paths—such path is called an *evidence*—that all together carry a probability mass that exceeds p. As such sets could be huge, the interest is in generating small sets, possibly the smallest possible. Preferably, the probability mass of such sets deviates significantly from the bound p.

Recently, we have shown [10] that for DTMCs wrt. the quantitative (hop-constrained) until formulas, most probable evidences—thus contributing the most to

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 331–346, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the violation—can be determined efficiently using either well-known (hop-constrained) shortest path (SP or HSP) algorithms, or Viterbi's algorithm. In addition, smallest counterexamples—containing the least number of evidences while maximally deviating from p among all counterexamples containing the same number of evidences—can be determined using k-SP (k-HSP) algorithms. Here, k is the size of the counterexample and is determined on-the-fly. Similar results hold for properties where p is a lower bound, where sets of paths are considered that indicate the violation of the "dual" of the formula to be checked; see [10] for details.

This paper considers the generation of evidences and counterexamples for model checking CSL [3,4] on CTMCs. For (hop-constrained) reachability properties expressed in CSL, the algorithms of [10] can be exploited. Properties that involve time, however, require other strategies. The continuous-time setting is unfortunately different and more complicated than the discrete one. First, an evidence cannot be a single timed path (an alternating sequence of states and time instants) as such paths have zero probability. Instead, we consider *symbolic evidences* for $\Phi \, \mathsf{U}^{\leqslant t} \Psi$, i.e., time-abstract paths—finite state sequences—that satisfy $\Phi \, \mathsf{U} \, \Psi$. A symbolic evidence induces a set of concrete evidences, viz. the set of timed paths on the same state sequence whose duration does not exceed t. Counterexamples are sets of symbolic evidences that exceed probability p. The main contribution of the paper is a set of algorithms for computing informative (symbolic) evidences and counterexamples, i.e., evidences with large probability and small counterexamples. We first indicate how the likelihood of symbolic evidences can be computed, both numerically and analytically. The latter approach exploits the fact that symbolic evidences are in fact acyclic CTMCs for which closed-form solutions exist [15]. We then consider the problem of how to find symbolic evidences such that small counterexamples result. First, we (naively) apply the strategy from [10], i.e., use k-SP algorithms on a discretized CTMC (obtained by uniformization [12]). This yields a simple algorithm, though may result in large counterexamples. A first variant exploits timing information and generates paths in the discretized CTMC that correspond to symbolic evidences. The advantage of this approach is that one can guarantee that counterexamples are obtained that contain the smallest number of evidences wrt. to their probability contribution in the CTMC. As probable paths of this kind usually correspond to probable symbolic evidences, this yields small counterexamples. Finally, we present a heuristic to improve the time and memory efficiency of this algorithm.

Organization of the paper. Section 2 summarizes the main steps of counterexample generation for DTMCs, and defines the main concepts of CTMCs needed for the rest of the paper. Section 3 defines symbolic evidences and counterexamples. Computing probabilities of symbolic evidences is treated in Section 4. Section 5 and 6 present the algorithms for determining symbolic evidences. Section 7 concludes the paper.

2 Preliminaries

Counterexample generation in DTMCs. Let AP denote a fixed, finite set of atomic propositions ranged over by $a, b, c, \ldots$.

Definition 1 (DTMC). *A (labelled)* discrete-time Markov chain *(DTMC)* $\mathcal{D}$ *is a triple* $(S, \mathbf{P}, L)$ *with* S *a finite set of states,* $\mathbf{P} : S \times S \rightarrow [0, 1]$ *a stochastic matrix, and* $L : S \rightarrow 2^{AP}$ *a labelling function.*

For a DTMC, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$, i.e. it is *stochastic*. If $\sum_{s' \in S} \mathbf{P}(s, s') \in [0, 1)$, then we call the model a *fully probabilistic system (FPS)* and it is *sub-stochastic*. A state s is absorbing if $\mathbf{P}(s, s) = 1$, i.e., if s only has a self-loop. A path σ in $\mathcal{D}$ is a state sequence $s_0\, s_1\, s_2$... such that $\mathbf{P}(s_i, s_{i+1}) > 0$, for all i. The probability $\Pr\{\sigma\}$ for finite $\sigma = s_0\, s_1 \ldots s_n$ is defined as $\mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot \ldots \cdot \mathbf{P}(s_{n-1}, s_n)$. For finite set of paths C, $\Pr(C) = \sum_{\sigma \in C} \Pr\{\sigma\}$. $\sigma[i]$ denotes the $(i + 1)$-st state on σ.

For PCTL [11] formula $\mathcal{P}_{\leqslant p}(\phi)$ where ϕ is a path formula, we have:

$$s \nvDash \mathcal{P}_{\leqslant p}(\phi) \quad \text{iff} \quad \Pr\{\sigma \mid \sigma[0] = s \text{ and } \sigma \models \phi\} > p.$$

So, $\mathcal{P}_{\leqslant p}(\phi)$ is refuted by state s whenever the total probability mass of all ϕ-paths that start in s exceeds p. This indicates that a counterexample for $\mathcal{P}_{\leqslant p}(\phi)$ is a *set* of paths starting in s and satisfying ϕ. As ϕ is a path formula whose validity can be witnessed by finite state sequences, *finite* paths suffice.

Definition 2 (Evidence). *An* evidence *for* $\mathcal{P}_{\leqslant p}(\phi)$ *in state* s *is a finite path* σ *that starts in* s *and minimally satisfies* ϕ. *A* strongest evidence *is an evidence* σ^* *such that* $\Pr\{\sigma^*\} \geqslant \Pr\{\sigma\}$ *for any evidence* σ.

A finite path σ minimally satisfies ϕ if it satisfies ϕ, but no proper prefix of σ does so.

Definition 3 (Counterexample). *A* counterexample *for* $\mathcal{P}_{\leqslant p}(\phi)$ *in state* s *is a set* C *of evidences such that* $\Pr(C) > p$. C^* *is a* smallest (most indicative) counterexample *if* $|C^*| \leqslant |C|$ *for all counterexamples* C *and* $\Pr(C^*) \geqslant \Pr(C')$ *for any counterexample* C' *with* $|C'| = |C^*|$.

The intuition is that a smallest counterexample is mostly exceeding the required probability bound given that it has the smallest number of paths. To compute the strongest evidence and smallest counterexample, the DTMC $\mathcal{D}$ is transformed to a weighted digraph $\mathcal{G}_D = (V, E, w)$, where V and E are finite sets of vertices and edges, respectively. $V = S$ and $(v, v') \in E$ iff $\mathbf{P}(v, v') > 0$, and $w(v, v') = \log(\mathbf{P}(v, v')^{-1})$. Multiplication of transition probabilities is thus turned into the addition of edge weights along paths. Now:

Lemma 1. *For any path* σ *from* s *to* t *in DTMC* $\mathcal{D}$, $k \in \mathbb{N}_{>0}$, *and* $h \in \mathbb{N} \cup \{\infty\}$*:* σ *is a* k*-th most probable path of at most* h *hops in* $\mathcal{D}$ *iff* σ *is a* k*-th shortest path of at most* h *hops in* $\mathcal{G}_D$.

Consider $\phi = \Phi \, \mathsf{U}^{\leqslant h} \Psi$ for PCTL state-formulas Φ, Ψ and hop bound $h \in \mathbb{N} \cup \{\infty\}$. If $s \nvDash \mathcal{P}_{\leqslant p}(\phi)$, then a strongest evidence can be found by a shortest path (SP) algorithm once all Ψ-states and all $(\neg\Phi \wedge \neg\Psi)$-states in DTMC $\mathcal{D}$ are made absorbing. Similarly, a smallest counterexample can be determined by k-SP algorithms that allow k to be determined on-the-fly. If $h \neq \infty$, hop-constrained SP and k-SP algorithms need to be employed; they have pseudo-polynomial time complexity in $\mathcal{O}(hm)$ and $\mathcal{O}(hm + hk \log(\frac{m}{n}))$, respectively, where $n = |S|$ and m is the number of non-zero entries in $\mathbf{P}$.

CTMCs

Definition 4 (CTMC). *A (labelled) continuous-time Markov chain (CTMC) $\mathcal{C}$ is a quadruple $(S, \mathbf{P}, E, L)$ with $(S, \mathbf{P}, L)$ a DTMC and $E : S \rightarrow \mathbb{R}_{\geqslant 0}$ a rate vector, assigning exit rates to states.*

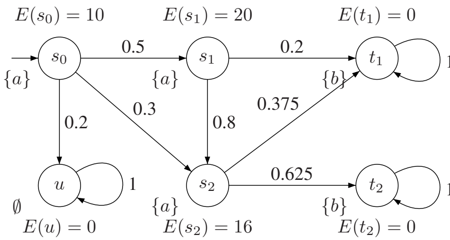


Fig. 1. CTMC $\mathcal{C}$

$(S, \mathbf{P}, L)$ is the *embedded* DTMC of $\mathcal{C}$. $E(s)$ denotes the rate of firing a transition from s, which, in other words, specifies the average delay of transitions. More precisely, with probability $(1 - e^{-E(s)\cdot t})$, a transition is enabled within the next t time units provided that the current state is s. If $\mathbf{P}(s, s') > 0$ for more than one state s', a *race* between the outgoing transitions from s exists. The probability of transition $s \rightarrow s'$ winning this race in time interval $[0, t]$ is given by:

$$\mathbf{P}(s, s', t) = \mathbf{P}(s, s')\cdot\big(1 - e^{-E(s)\cdot t}\big).$$

The probability density function is $p(s, s', t) = \mathbf{P}(s, s')\cdot E(s)\cdot e^{-E(s)\cdot t}$. Note that $\mathbf{P}(s, s', t) = \int_0^t p(s, s', t_1)\cdot dt_1$. We sometimes use $\mathbf{R}(s, s') = \mathbf{P}(s, s')\cdot E(s)$ to denote the rate of the transition $s \rightarrow s'$.

Remark 1. Except for absorbing states, all states in a CTMC are assumed to have no self-loops. The reason for this assumption will become clear later. Note that this is not a severe restriction as self-loops can be removed without affecting the transient and the steady-state probabilities of the CTMC.

Example 1. An example CTMC $\mathcal{C}$ is shown in Fig. 1. $S = \{s_i, t_1, t_2, u\}$; $L(s_i) = \{a\}$, $L(t_1) = L(t_2) = \{b\}$ and $L(u) = \emptyset$ with $0 \leqslant i \leqslant 2$; $E(s_0) = 10$, $E(s_1) = 20$, and so on. States u, t_1 and t_2 are absorbing.

Paths and probability measure

Definition 5 (Timed paths in CTMCs). *Let $\mathcal{C} = (S, \mathbf{P}, E, L)$ be a CTMC. An* infinite *timed path σ is a sequence $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots$ with $s_i \in S$ and $t_i \in \mathbb{R}_{\geqslant 0}$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for $i \geqslant 0$. A* finite *timed path σ is a finite prefix of an infinite path ending in an absorbing state.*

Let $|\sigma|$ denote the length of the path σ, i.e., $|s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots s_{l-1} \xrightarrow{t_{l-1}} s_l| = l$, $|s_0| = 0$ and $|\sigma| = \infty$ for infinite σ. For (finite or infinite) path σ and $i < |\sigma|$, let $\sigma[i] = s_i$ be the $(i{+}1)$-st state of σ, and $\delta(\sigma, i) = t_i$ be the time spent in s_i. For $t \in \mathbb{R}_{\geqslant 0}$ and k the smallest index with $t < \sum_{j=0}^{k} t_j$, let $\sigma@t = \sigma[k]$ denote the state in σ occupied at time t. For finite path σ and $l = |\sigma|$, $\delta(\sigma, l) = \infty$; and for $t \geqslant \sum_{j=0}^{l-1} t_j$, $\sigma@t = s_l$.

A *time-abstract path* is obtained by omitting all timing information from a timed path. The function α performs this, i.e., $\alpha(s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots) = s_0 s_1 \ldots$. Let $Paths^{\mathcal{C}}$ denote

the set of all timed paths in CTMC $\mathcal{C}$ and $Paths^{\mathcal{C}}_{abs}$ all time-abstract paths in $\mathcal{C}$. The superscript is omitted if $\mathcal{C}$ is clear from the context. $Paths(s)$ and $Paths_{abs}(s)$ denote the set of timed and time-abstract paths starting from s, respectively. We use ρ to range over time-abstract paths.

A σ-algebra and probability measure of the timed paths of a CTMC can be defined using the standard cylinder set construction, cf. [4]. It follows that time-convergent paths, i.e., paths on which time does not diverge, have probability 0.

CSL. Continuous Stochastic Logic (CSL) [4] is a variant of the logic originally proposed by Aziz et al. [3] and extends PCTL by path operators that reflect the real-time nature of CTMCs: in particular, a time-bounded until operator.

Syntax. The syntax of CSL state-formulae is defined as follows:

$$\Phi ::= \mathrm{tt} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{\unlhd p}(\phi),$$

where $p \in [0,1]$ is a probability, $\unlhd \in \{<, \leqslant, >, \geqslant\}$. For t a non-negative real number or $t = \infty$, ϕ is a path-formula defined according to the following grammar:

$$\phi ::= \Phi \,\mathsf{U}^{\leqslant t}\, \Psi \mid \Phi \,\mathsf{W}^{\leqslant t}\, \Psi.$$

The path formula $\Phi \,\mathsf{U}^{\leqslant t} \Psi$ asserts that Ψ is satisfied within t time units and that at all preceding time instants Φ holds. $\mathsf{W}^{\leqslant t}$ is the weak counterpart which does not require Ψ to eventually become true. For the sake of simplicity, the next-operator and the steady-state operator [4] are not considered here.

Semantics. CSL state-formulae are interpreted over the states of a CTMC. Let $\mathcal{C} = (S, \mathbf{P}, E, L)$ with labels in AP, and $Sat(\Phi) = \{s \in S \mid s \models \Phi\}$. The semantics of CSL state-formulae is defined for path-formula ϕ as:

$s \models \mathrm{tt}$	iff	*true*	$s \models \Phi \wedge \Psi$	iff	$s \models \Phi$ and $s \models \Psi$
$s \models a$	iff	$a \in L(s)$	$s \models \mathcal{P}_{\unlhd p}(\phi)$	iff	$Prob(s, \phi) \unlhd p$
$s \models \neg\Phi$	iff	not $(s \models \Phi)$			

$Prob(s, \phi)$ denotes the probability measure of all paths $\sigma \in Paths$ starting in state s and satisfying ϕ, i.e., $Prob(s, \phi) = \Pr\{\sigma \in Paths(s) \mid \sigma \models \phi\}$. For a timed path σ in $\mathcal{C}$, the satisfaction relation for CSL path-formulae is defined as:

$$\sigma \models \Phi \,\mathsf{U}^{\leqslant t}\Psi \quad \text{iff} \quad \sigma@x \models \Psi \text{ for some } x \leqslant t \text{ and } \sigma@y \models \Phi \text{ for all } y < x,$$

$$\sigma \models \Phi \mathsf{W}^{\leqslant t}\Psi \quad \text{iff} \quad \text{either } \sigma \models \Phi \,\mathsf{U}^{\leqslant t}\Psi \text{ or } \sigma@x \models \Phi \text{ for all } x \leqslant t.$$

The until and weak until operators are closely related. This follows from the following equations. For any CSL-formulae Φ and Ψ we have:

$$\mathcal{P}_{\geqslant p}(\Phi \,\mathsf{W}^{\leqslant t}\Psi) \equiv \mathcal{P}_{\leqslant 1-p}\big((\Phi \wedge \neg\Psi)\,\mathsf{U}^{\leqslant t}(\neg\Phi \wedge \neg\Psi)\big)$$

$$\mathcal{P}_{\geqslant p}(\Phi \,\mathsf{U}^{\leqslant t}\Psi) \equiv \mathcal{P}_{\leqslant 1-p}\big((\Phi \wedge \neg\Psi)\mathsf{W}^{\leqslant t}(\neg\Phi \wedge \neg\Psi)\big)$$

Counterexamples for $\mathcal{P}_{\geqslant p}(\Phi \,\mathsf{U}^{\leqslant t}\Psi)$ can be obtained by considering a formula of the form $\mathcal{P}_{\leqslant p'}(\Phi' \,\mathsf{U}^{\leqslant t}\, \Psi')$. This can be seen as follows. Extend the labels of $\mathcal{C}$ with a new atomic proposition, at_B, say, where at_B is a new atomic proposition such that $s \models at_B$ iff (i) either $s \models \neg\Phi \wedge \neg\Psi$ (ii) or $s \in B$ where B is a bottom strongly connected

component (BSCC) such that $B \subseteq Sat(\Phi \wedge \neg\Psi)$, or shortly $B_{\Phi\wedge\neg\Psi}$. A BSCC B is a maximal strong component that has no transitions leaving B. Then:

$$\mathcal{P}_{\geqslant p}(\Phi \,\mathsf{U}^{\leqslant t}\Psi) \equiv \mathcal{P}_{\leqslant 1-p}((\Phi \wedge \neg\Psi)\mathsf{W}^{\leqslant t}(\neg\Phi \wedge \neg\Psi)) \equiv \mathcal{P}_{\leqslant 1-p}((\Phi \wedge \neg\Psi)\,\mathsf{U}^{\leqslant t} at_B)$$

Intuitively, to show that the set of $(\Phi\,\mathsf{U}^{\leqslant t}\Psi)$-paths has probability $\geqslant p$, it is sufficient to show that the paths violating $\Phi\,\mathsf{U}^{\leqslant t}\Psi$ have probability $\leqslant 1-p$.

Note that for $t = \infty$, $\Phi\,\mathsf{U}^{\leqslant t}\Psi$ denotes the standard-until operator. As this operator can be verified on the embedded DTMC, counterexamples can be obtained as for DTMCs. In the sequel, we therefore consider $t \neq \infty$.

3 Evidences and Counterexamples

Assume $s \not\models \mathcal{P}_{\leqslant p}(\phi)$ for CSL path-formula ϕ. Unlike in DTMCs, a timed path could not be an evidence since it always has probability 0. Instead, we consider *symbolic* evidences that represent a set of (concrete) finite timed paths satisfying ϕ. For time-abstract path ρ, let $\rho{\downarrow_k}$ denote the prefix of ρ of length k, i.e., $(s_0 s_1 \ldots){\downarrow_k} = s_0 s_1 \ldots s_k$.

Definition 6 (Symbolic evidence). *A* symbolic evidence *for $\mathcal{P}_{\leqslant p}(\phi)$ in state s is a finite time-abstract path that starts in s and minimally satisfies ϕ. Let $Paths_{abs}(s,\phi)$ denote the set of symbolic evidences starting from s for ϕ.*

Actually, a symbolic evidence for $\phi = \Phi\,\mathsf{U}^{\leqslant t}\,\Psi$ is a finite time-abstract path that goes along Φ-states and halts at the first encountered Ψ-state. A symbolic evidence for $\phi = \Phi\,\mathsf{U}^{\leqslant t}\,\Psi$ represents a set of (infinite) timed paths in the CTMC:

$$Paths_{\leqslant t}(\rho) = \{\sigma \in Paths \mid \rho = \alpha(\sigma){\downarrow_l} \wedge \sum_{i=0}^{l-1} \delta(\sigma, i) \leqslant t\} \quad \text{where } l = |\rho|.$$

The timed paths induced by ρ have a common initial state sequence, viz. ρ, and the total duration of this prefix is at most t, i.e., the last state of ρ is reached within t. We define the probability of a symbolic evidence ρ to be $\Pr_{\leqslant t}(\rho)$, and for the set C of symbolic evidences, the probability is $\Pr(C) = \sum_{\rho\in C}\Pr_{\leqslant t}(\rho)$. A *strongest* symbolic evidence is a symbolic evidence of maximal probability.

Lemma 2. *For CTMC $\mathcal{C}$ and $\phi = \Phi\,\mathsf{U}^{\leqslant t}\,\Psi$: $Prob(s,\phi) = \sum_{\rho\in Paths_{abs}(s,\phi)} \Pr_{\leqslant t}(\rho)$.*

For state s in CTMC $\mathcal{C}$ and formula $\mathcal{P}_{\leqslant p}(\phi)$ we now have:

$$s \not\models \mathcal{P}_{\leqslant p}(\phi) \quad \text{iff} \quad Prob(s,\phi) > p \quad \text{iff} \quad \sum_{\rho\in Paths_{abs}(s,\phi)} \Pr_{\leqslant t}(\rho) > p.$$

As $Paths_{abs}(s,\phi)$ only contains finite time-abstract paths, counterexamples are sets of symbolic evidences of sufficient probability mass.

Definition 7 (Symbolic counterexample). *A* symbolic counterexample *for $\mathcal{P}_{\leqslant p}(\phi)$ where $\phi = \Phi\,\mathsf{U}^{\leqslant t}\,\Psi$ is a set C of symbolic evidences for ϕ such that* $\Pr(C) > p$.

Example 2. For the CTMC $\mathcal{C}$ in Fig. 1 and CSL formula $\mathcal{P}_{\leqslant 0.45}(a \,\mathsf{U}^{\leqslant 1} b)$ the symbolic evidences are $\rho_1 = s_0 s_2 t_2$, $\rho_2 = s_0 s_1 s_2 t_2$, $\rho_3 = s_0 s_1 t_1$, and so on. These paths all satisfy $a \,\mathsf{U}\, b$. For instance, $s_0 \xrightarrow{0.5} s_1 \xrightarrow{0.25} s_2 \xrightarrow{0.05} t_2 \in Paths_{\leqslant 1}(\rho_2)$. Without specifying the details (see next section), the probabilities of the symbolic evidences are: $\Pr_{\leqslant 1}(\rho_1) = 0.24998$, $\Pr_{\leqslant 1}(\rho_2) = 0.24994$ and $\Pr_{\leqslant 1}(\rho_3) = 0.16667$. $C = \{\rho_1, \rho_2\}$ is a counterexample since $\Pr(C) > 0.45$, but $C' = \{\rho_1, \rho_3\}$ is not.

The remainder of the paper is concerned with determining (symbolic) counterexamples and symbolic evidences. As in conventional model checking, the intention is to obtain *comprehensible* counterexamples. We interpret this as counterexamples of minimal size, i.e., minimal cardinality. An algorithmic skeleton to generate such counterexamples iteratively is given below:

(1)	$k := 1; pr := 0;$
(2)	**while** $pr \leqslant p$ **do**
(3)	determine symbolic evidence ρ^k;
(4)	compute $\Pr_{\leqslant t}(\rho^k)$;
(5)	$pr := pr + \Pr_{\leqslant t}(\rho^k)$;
(6)	$k := k + 1$;
(7)	**od**;
(8)	**return** $(\rho^1, \ldots, \rho^{k-1})$

The termination of this algorithm is guaranteed as the violation of the property has been already established prior to invoking it. Evidently, the smaller the index k, the more succinct the counterexample. The next section presents a way to determine $\Pr_{\leqslant t}(\rho)$, i.e., the probability of a symbolic evidence (cf. line (4)). In subsequent sections, we present algorithms that aim to finding probable symbolic evidences, cf. line (3) of the algorithm. Stated differently, we aim to terminating with a small value of k.

4 The Likelihood of a Symbolic Evidence

Assume we have symbolic evidence $\rho = s_0\, s_1\, s_2 \ldots s_l$ at our disposal. The probability $\Pr_{\leqslant t}(\rho)$ of this evidence—in fact, the probability of all concrete evidences of ρ up to time t—is given by:

$$\int_0^t \left(p(s_0, s_1, t_0) \cdot \left(\ldots (\int_0^{t - \sum_{i=0}^{l-2} t_i} p(s_{l-1}, s_l, t_{l-1}) \cdot dt_{l-1}) \ldots \right) \right) dt_0 \qquad (1)$$

where $p(s_0, s_1, t_0)$ denotes the probability density function of $s_0 \rightarrow s_1$ winning the race at time instant t_0 in the interval $[0, t]$. The corresponding probability is thus derived by the outermost integral. Suppose the transition $s_0 \rightarrow s_1$ takes place at time instant t_0. Then the possible time instant for the second transition $s_1 \rightarrow s_2$ to take place is in $[0, t - t_0]$. This determines the range of the second outermost integral. The rest is likewise. The innermost integral determines the residence time in state s_{l-1}, the one-but-last state in ρ.

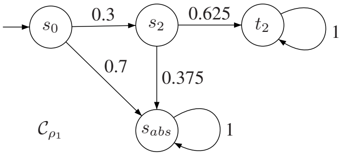
To avoid computing this (somewhat involved) integral directly by numerical techniques we resort to a simpler technique. The main idea is to isolate the time-abstract path ρ from the entire CTMC. This yields a simple acyclic CTMC, i.e., an acyclic phase-type distribution [16] which can be solved either analytically or numerically.

Transformation into an acyclic CTMC. As a first step, we transform the CTMC as suggested in [4]. (The same strategy was applied in Section 2 for DTMCs prior to applying SP algorithms.) Consider CTMC $\mathcal{C}$ and CSL path-formula $\phi = \Phi \, \mathsf{U}^{\leqslant t} \, \Psi$. All Ψ-states as well as all $(\neg\Phi \wedge \neg\Psi)$-states are made absorbing in $\mathcal{C}$, i.e., their outgoing transitions are replaced by a self-loop. It is not difficult to establish that the validity of $\mathcal{P}_{\leqslant p}(\phi)$ remains invariant under this modification. In the rest of the paper, CTMCs are assumed to have been subject to this transformation.

Definition 8 (CTMC induced by symbolic evidence). *Let CTMC* $\mathcal{C} = (S, \boldsymbol{P}, E, L)$ *and* $\rho = s_0 \, s_1 \ldots s_l$ *a symbolic evidence in CTMC* $\mathcal{C}$ *in which all states are pairwise distinct*[1]. *The CTMC* $\mathcal{C}_\rho$ *induced by* ρ *on* $\mathcal{C}$ *is defined by:* $\mathcal{C}_\rho = (S_\rho, \mathbf{P}_\rho, E_\rho, L_\rho)$ *with:*

- $S_\rho = \{s_0, \ldots, s_l, s_{abs}\}$ *with* $s_{abs} \notin S \cup \{s_0, \ldots, s_l\}$,
- $\mathbf{P}_\rho(s_i, s_{i+1}) = \mathbf{P}(s_i, s_{i+1})$, $\mathbf{P}_\rho(s_i, s_{abs}) = 1 - \mathbf{P}_\rho(s_i, s_{i+1})$ *for* $0 \leqslant i < l$ *and* $\mathbf{P}_\rho(s, s) = 1$ *for* $s = s_l$ *or* $s = s_{abs}$
- $E_\rho(s_i) = E(s_i)$ *and* $E_\rho(s_{abs}) = 0$ *and* $L_\rho(s_i) = L(s_i)$ *and* $L_\rho(s_{abs}) = \{abs\}$.

Stated in words, $\mathcal{C}_\rho$ is the CTMC obtained from $\mathcal{C}$ by incorporating all states in ρ, and deleting all outgoing transitions from these states except $s_i \rightarrow s_{i+1}$. The total probability mass of these omitted transitions becomes the probability to move to the trap state s_{abs}. It follows directly that $\mathcal{C}_\rho$ is acyclic when ignoring the self-loops of the absorbing states.



Example 3. Consider CTMC $\mathcal{C}$ in Fig. 1 and symbolic evidence $\rho_1 = s_0 \, s_2 \, t_2$. The induced CTMCs $\mathcal{C}_{\rho_1}$ is shown on the left.

The following result states that computing the probability of symbolic evidence ρ boils down to a (standard) transient analysis of the induced CTMC by ρ.

Lemma 3. *For CTMC* $\mathcal{C}$ *and symbolic evidence* ρ *for* $\phi = \Phi \, \mathsf{U}^{\leqslant t} \, \Psi$*:*

$$\Pr^{\mathcal{C}}_{\leqslant t}(\rho) \;=\; \pi^{\mathcal{C}_\rho}(s, s_l, t)$$

where $\pi^{\mathcal{C}_\rho}(s, s_l, t)$ *is the transient probability of state* s_l*, the last state of* ρ*, at time* t *under the condition that* $\mathcal{C}_\rho$ *started in* s*.*

This result enables us to exploit well-known algorithms for the transient analysis of CTMCs to determine the likelihood of a symbolic evidence. In fact, as CSL model checking of time-bounded until-formulas is reduced to transient analysis (see [4]), the desired likelihood can be determined by verifying the property $\Diamond^{\leqslant t} \, at_{s_l}$ on the CTMC $\mathcal{C}_\rho$. (Here, at_{s_l} is an atomic proposition that only holds in state s_l.) This yields an approximate solution up to an a priori user-defined accuracy and is part of the standard machinery in model checkers such as PRISM [14] and MRMC [13]. Alternatively, we can exploit the fact that $\mathcal{C}_\rho$ is acyclic (ignoring the self-loops at the absorbing states) and use the closed-form expression for transient distributions in acyclic CTMCs as proposed by Marie *et al.* [15]. This yields an exact solution.

[1] This is not a restriction since it is always possible to rename a state along ρ while keeping e.g. its exit rate and its labeling the same.

5 A First Attempt to Find Probable Symbolic Evidences

It remains to clarify how symbolic evidences can be obtained and how to obtain them in such a way that small counterexamples result. As symbolic evidences are just state sequences, the first idea is to adapt the strategy for DTMCs [10], cf. Section 2. That is, the CTMC under consideration is discretized. This is done using uniformization [12], a technique to transform a CTMC into a DTMC whose transient behaviour is equal (up to some accuracy ε)[2]. k-SP algorithms are then exploited to obtain symbolic evidences in ascending order of likelihood (in the obtained DTMC). k is determined on-the-fly as the minimal natural number such that $\sum_{i=1}^{k} \Pr_{\leqslant t}(\rho^i) > p$ where p is the lower bound of the property that is refuted. Let us first briefly present uniformization.

Uniformization (also known as Jensen's method or randomization) [12] is a well-known method for computing the transient probabilities of a CTMC at specific time t. Its formulation involves construction of a DTMC and Poisson process from an original CTMC. Uniformization is attractive because of its excellent numerical stability and the fact that the computational error is well-controlled and can be specified in advance.

For CTMC $\mathcal{C} = (S, \mathbf{P}, E, L)$, the uniformized DTMC is $\mathcal{U} = \mathit{unif}(\mathcal{C}) = (S, \mathbf{U}, L)$, where $\mathbf{U}$ is defined by $\mathbf{U} = \mathbf{I} + \frac{\mathbf{Q}}{q}$ with $q \geqslant \max_i\{E(s_i)\}$ and $\mathbf{Q} = \mathbf{R} - \mathit{diag}(\underline{E})$. For the special case $q = 0$, $\mathbf{U}(s, s) = 1$ for any $s \in S$. In the rest of the paper, we always use $\mathcal{U}$ to denote $\mathit{unif}(\mathcal{C})$. The uniformization rate q can be chosen to be any value exceeding the shortest mean residence time. All rates in the CTMC are normalized with respect to q. For each state s with $E(s) = q$, one epoch in the uniformized DTMC corresponds to a single exponentially distributed delay with rate q, after which one of its successor states is selected probabilistically. As a result, such states have no additional self-loop in the DTMC. If $E(s) < q$, i.e., state s has, on average, a longer state residence time than $\frac{1}{q}$, one epoch in the DTMC might not be "long enough"; hence, in the next epoch, these states might be revisited with some positive probability. This is represented by equipping these states with a self-loop with probability $1 - \frac{E(s)}{q} + \frac{\mathbf{R}(s,s)}{q}$.

Remark 2 (Self-loops). As a CTMC is assumed to have no self-loops on non-absorbing states, all self-loops in the uniformized DTMC are caused by uniformization.

After uniformization, the vector of state probabilities $\underline{\pi}^{\mathcal{C}}(t)$ at time t, namely the *transient probability* vector, is computed as:

$$\underline{\pi}^{\mathcal{C}}(t) = \alpha_0 \cdot \sum_{i=0}^{\infty} PP(i, qt)\mathbf{U}^i = \sum_{i=0}^{\infty} PP(i, qt)\underline{\pi}^{\mathcal{U}}(i), \tag{2}$$

where $PP(i, qt) = e^{-qt}\frac{(qt)^i}{i!}$ is the ith Poisson probability that i epochs occur in $[0, t]$ when the average rate is $\frac{1}{qt}$ and $\underline{\pi}^{\mathcal{U}}(i)$ is the state probability distribution vector after i epochs in $\mathcal{U}$ with transition matrix $\mathbf{U}$ determined recursively by $\underline{\pi}^{\mathcal{U}}(i) = \underline{\pi}^{\mathcal{U}}(i-1) \cdot \mathbf{U}$ with the initial distribution $\underline{\pi}^{\mathcal{U}}(0) = \alpha_0$.

[2] An alternative discretization is to use the embedded DTMC, but as this does not involve any timing aspects, this is senseless.

The Poisson probabilities can be computed in a stable way with the Fox-Glynn algorithm [8], thus avoiding numerical instability. The infinite summation problem is solved by introducing a required accuracy ε, so that $\|\underline{\pi}^{\mathcal{C}}(t) - \tilde{\underline{\pi}}^{\mathcal{C}}(t)\| \leqslant \varepsilon$, where $\tilde{\underline{\pi}}^{\mathcal{C}}(t) = \sum_{i=0}^{N_\varepsilon(t)} PP(i, qt) \cdot \underline{\pi}^{\mathcal{U}}(i)$ is the approximation of $\underline{\pi}^{\mathcal{C}}(t)$ and $N_\varepsilon(t)$ is the number of terms to be taken in Equation (2) for time t, which is the smallest value satisfying:

$$\sum_{i=0}^{N_\varepsilon(t)} \frac{(qt)^i}{i!} \geqslant \frac{1-\varepsilon}{e^{-qt}} = (1-\varepsilon) \cdot e^{qt}. \tag{3}$$

If qt is larger, $N_\varepsilon(t)$ tends to be of the order $\mathcal{O}(qt)$.

Let θ denote a path in $\mathcal{U}$, $Paths^{\mathcal{U}}$ denote the set of all paths in $\mathcal{U}$ and $Paths^{\mathcal{U}}(s)$ the paths in $\mathcal{U}$ starting in s.

Model transformation. Given a CTMC $\mathcal{C}$ and a CSL formula $\phi = \Phi \, \mathsf{U}^{\leqslant t} \, \Psi$, we take the uniformized DTMC $\mathcal{U}$ of $\mathcal{C}$ and remove all its self-loops. The resulting DTMC is referred to as $\mathcal{U}^{\otimes}$, which is an FPS instead of a DTMC. If $\mathcal{U}^{\otimes}$ would be normalized, we obtain the embedded DTMC of $\mathcal{C}$. The probability in the embedded DTMC only considers the race of transitions after the delay, while the probability in $\mathcal{U}^{\otimes}$ takes delays into consideration. We remove self-loops in $\mathcal{U}$ as many paths in $\mathcal{U}$ correspond to the same time-abstract path in $\mathcal{C}$. Every path in $\mathcal{U}^{\otimes}$ is a time-abstract path in $\mathcal{C}$ and satisfies ϕ. Besides, the information of the self-loops (viz., delays) can be recovered easily by taking the difference between the total probability of a state and one.

Algorithm by pure graph analysis. For $s \not\models \mathcal{P}_{\leqslant p}(\phi)$, a counterexample can be computed as follows: The k most probable paths in $\mathcal{U}^{\otimes}$ are computed, each corresponding to a symbolic evidence in $\mathcal{C}$, i.e., symbolic evidences are computed in such an order $\rho^1, \rho^2, ..., \rho^k$ that $\Pr\{\rho^1\} \geqslant \Pr\{\rho^2\} \geqslant ... \geqslant \Pr\{\rho^k\}$. k is determined on the fly, as the smallest number such that $\sum_{i=1}^{k} \Pr_{\leqslant t}(\rho^i) > p$. The k most probable paths problem can be reduced to k-SP problem by the standard transformation in Section 2 which also applies to FPS $\mathcal{U}^{\otimes}$. The resulting algorithm becomes:

(1) $k := 1$; $pr := 0$;
(2) **while** $pr \leqslant p$ **do**
(3) determine symbolic evidence ρ^k as the k-th most probable path in $\mathcal{U}^{\otimes}$;
(4) compute $\Pr_{\leqslant t}(\rho^k)$;
(5) $pr := pr + \Pr_{\leqslant t}(\rho^k)$;
(6) $k := k + 1$;
(7) **od**;
(8) **return** $(\rho^1, \ldots, \rho^{k-1})$

The time complexity for computing the k most probable paths is as the k-SP problem, cf. [7], $\mathcal{O}(m + n \log n + k)$. The transformation from ρ to $\mathcal{C}_\rho$ takes $\mathcal{O}(|\rho|)$ time. It takes $\mathcal{O}(|\rho| qt)$ to compute the probability of a symbolic evidence ρ, where $\mathcal{O}(qt)$ is the number of terms before truncation (i.e., $N_\varepsilon(t)$, cf. [4]) and $\mathcal{O}(|\rho|)$ time is needed for vector-vector multiplication. There are k symbolic evidences, which gives rise to the total time complexity $\mathcal{O}(m + n \log n + k|\rho| qt)$.

In most of the cases, probable paths in $\mathcal{U}^{\otimes}$ correspond to probable symbolic evidences in $\mathcal{C}$. However, this is not always the case, since the time bound in the property

is not considered. In particular, this approach does not guarantee $\Pr_{\leqslant t}(\rho^i) \geqslant \Pr_{\leqslant t}(\rho^j)$ for $i < j$. An example is given as follows:

Example 4. Consider our running example. The uniformized DTMC $\mathcal{U}$ is illustrated on the left. The uniformization rate is chosen as $q = E(s_1) = 20$, since s_1 has the largest exit rate. For symbolic evidences $\rho_2 = s_0 s_1 s_2 t_2$ and $\rho_3 = s_0 s_2 t_1$ of Example 3, the probabilities in $\mathcal{U}^{\otimes}$ are $\Pr\{\rho_2\} = 0.100$ and $\Pr\{\rho_3\} = 0.045$, respectively. For CSL path formula $\phi = a \, \mathsf{U}^{\leqslant 1} b$, $\Pr_{\leqslant 1}(\rho_2) = 0.24994$ and $\Pr_{\leqslant 1}(\rho_3) = 0.16362$. For $\phi' = a \, \mathsf{U}^{\leqslant 0.1} b$, $\Pr_{\leqslant 0.1}(\rho_2) = 0.04478$ and $\Pr_{\leqslant 0.1}(\rho_3) = 0.06838$. Thus, for $t = 1$, $Paths_{\leqslant 1}(\rho_2)$ is more probable than $Paths_{\leqslant 1}(\rho_3)$, whereas for $t = 0.1$, the reverse holds.

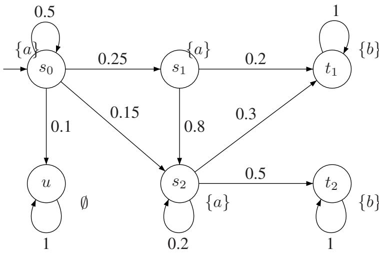


Fig. 2. $unif(\mathcal{C}) = \mathcal{U}$

This implies that for symbolic evidences ρ and ρ' and arbitrary time bound t, $\Pr\{\rho\} > \Pr\{\rho'\}$ cannot guarantee that $\Pr_{\leqslant t}(\rho) > \Pr_{\leqslant t}(\rho')$. A direct consequence is that the algorithm might terminate with a large value of k. The counterexamples may thus be less comprehensive, because evidences with large probability might not be included. The algorithm in the next section attempts to overcome this problem by taking the time bound into account.

6 Involving Time Bounds

The previous algorithm ignores the time bound t in determining the order of generating the symbolic evidences $\rho^1, \rho^2, \ldots$. By definition, however, every transition in the uniformized DTMC $\mathcal{U}$ takes $\frac{1}{q}$ time units. In fact, using the Poisson probabilities we can determine the probability of path θ in $\mathcal{U}$ to have a duration of at most t:

Definition 9 ([17]). *Given a CTMC $\mathcal{C}$, for $\theta \in Paths^{\mathcal{U}}$ with $|\theta| = l$ and $t \in \mathbb{R}_{>0}$, the probability of θ occurring in $[0, t]$ with rate q is defined as:*

$$\Pr_{\leqslant t}(\theta, qt) = PP(l, qt) \cdot \Pr^{\mathcal{U}}\{\theta\}.$$

Intuitively, given that $|\theta|$ transitions occur in the interval $[0, t]$, the likelihood of θ occurring in $\mathcal{U}$ is $\Pr^{\mathcal{U}}\{\theta\}$. As $\mathcal{U}$ is a DTMC, $\Pr^{\mathcal{U}}\{\theta\}$ is simply $\prod_{i=0}^{|\theta|-1} \mathbf{U}(s_i, s_{i+1})$ for $\theta = s_0 s_1 s_2 \ldots$.

It remains to establish a connection between $\Pr_{\leqslant t}(\rho)$ and the probabilities obtained in the uniformized DTMC $\mathcal{U}$, i.e., $\Pr_{\leqslant t}(\theta, qt)$, where θ relates to ρ. This can be done as follows. Consider symbolic evidence ρ, say of length l. Paths in $\mathcal{U}$ that correspond to $\rho = s_0 s_1 \ldots s_l$ visit the same state sequence $s_0 s_1 \ldots s_l$ but may take the self-loop in s_i zero or more times. Recall that the purpose of this self-loop is to mimic the probability for the CTMC to reside longer in s_i. The set of paths in $\mathcal{U}$ that correspond to (or can mimic) ρ is defined by:

$$mimic(\rho) = \{s_0^{n_0} s_1^{n_1} \ldots s_l^{n_l} \in Paths^{\mathcal{U}} \mid n_i > 0 \text{ for } 0 \leqslant i \leqslant l\},$$

where $l = |\rho|$ and $s_0^{n_0}$ is short for the n_0-time replication of s_0. Then:

$$\Pr^{\mathcal{C}}_{\leqslant t}(\rho) = \sum_{\theta \in mimic(\rho)} \Pr^{\mathcal{U}}_{\leqslant t}(\theta, qt) = \sum_{i=|\rho|}^{\infty} PP(i, qt) \cdot \sum_{\theta \in mimic(\rho) \wedge i=|\theta|} \Pr^{\mathcal{U}}\{\theta\}$$

Note the similarity to Equation (2). The intuition is also similar: given a symbolic evidence ρ of $\mathcal{C}$, there are paths in $\mathcal{U}$ that can mimic ρ. These paths can have $i(= |\rho|)$ hops, $i+1$ hops, and so forth. The extra hops are self-loops in $\mathcal{U}$ which simulate the longer residence time in a state in $\mathcal{C}$.

To truncate the infinite summation, which lengths i do we need to consider? A natural criterion for this is fortunately provided by the uniformization process. As the probability of any path longer than $N_\varepsilon(t)$ is negligible – given an accuracy ε – this suggests to only consider paths up to length $N_\varepsilon(t)$. By taking this approach, it is guaranteed that the total probability mass of the not considered paths is less than ε.

An algorithm involving time. In the following, we give an algorithm that determines paths in a decreasing order with respect to $\Pr_{\leqslant t}(\theta, qt)$. Since we are interested in paths without self-loops, we consider paths in $\mathcal{U}^{\otimes}$.

Let ϖ_h^j denote the j-th most probable path in $\mathcal{U}^{\otimes}$ of h hops, i.e. $\Pr\{\varpi_h^j\} \geqslant \Pr\{\varpi_h^{j+1}\}$. Since the Poisson probability is fixed for a given h, $\Pr_{\leqslant t}(\varpi_h^j, qt) \geqslant \Pr_{\leqslant t}(\varpi_h^{j+1}, qt)$. Let τ^k denote the path in $\mathcal{U}^{\otimes}$ with k-th largest probability $\Pr_{\leqslant t}(\tau^k, qt)$. Then:

$$\tau^k = \arg\max_{\theta} \left\{ \Pr_{\leqslant t}(\theta, qt) \mid \theta \in Q^k \right\}, \tag{4}$$

where Q^k is the candidate path set defined as:

$$Q^k = \begin{cases} \left\{ \varpi_h^1 \mid 0 \leqslant h \leqslant N_\varepsilon(t) \right\} & \text{if } k = 1 \\ \left(Q^{k-1} - \{\tau^{k-1}\} \right) \cup \left\{ \varpi_h^{j+1} \right\} & \text{if } k > 1 \text{ and } \tau^{k-1} = \varpi_h^j \end{cases}$$

where j and h are the index and path length of τ^{k-1}, the previous path computed.

The algorithm starts with a "candidate" path set Q^1 which contains all ϖ_h^1 paths, the most probable path of length h, for $0 \leqslant h \leqslant N_\varepsilon(t)$. τ^1 is picked out as the one with the maximal probability in Q^1, according to Equation (4). To compute the next evidence τ^2, Q^2 is computed on the basis of Q^1. As $\varpi_{l^*}^1$ has been removed from Q^1, where $l^* = |\tau^1|$, another path of exactly l^* hops replaces it. This new path is $\varpi_{l^*}^2$, i.e., the second most probable path with the same length l^* as τ^1. Then τ^2 can be picked from Q^2. Since each path in $\mathcal{U}^{\otimes}$ is an evidence in $\mathcal{C}$, the algorithm will terminate when the probability of the first k evidences exceeds p.

Candidate paths are stored in a priority queue pq sorted by the keys $\Pr_{\leqslant t}(\varpi_h^j, qt)$. The *enqueue* function inserts a new path to its proper position and the *dequeue* function returns the pair (h, j) of the corresponding path with the highest probability in pq. Function $\varpi(h, j, qt)$ computes the j-th h-hop most probable path ϖ_h^j, which can be reduced to computing j-th shortest h-hop path in $\mathcal{G}_{\mathcal{U}^{\otimes}}$ and can be solved by adapted REA, see [10] for more details.

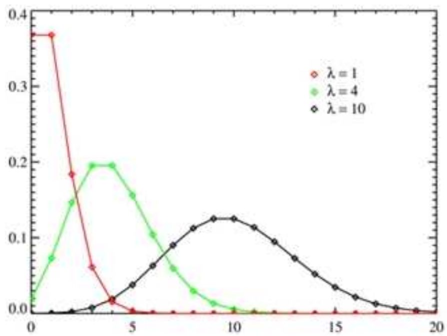
(1)	$k := 0;$ $pr := 0;$ $h := 0;$ PriorityQueue pq;
(2)	**for** $h := 0$ **to** $N_\varepsilon(t)$ **do** $pq.\, enqueue(\varpi(h, 1, qt));$ **od**; * Q^1 *\
(3)	**while** $pr \leqslant p$ **do** * Q^2 to Q^k *\
(4)	$(h', j') := pq.\, dequeue();$ $k := k + 1;$ $\rho^k := \varpi_{h'}^{j'};$
(5)	$\varpi := \varpi(h', j' + 1, qt);$ $pq.\, enqueue(\varpi);$ $pr := pr + \Pr_{\leqslant t}(\varpi);$ **od**;
(6)	**return** $(\rho^1, \ldots, \rho^{k-1});$

Note that the resulting evidence sequence $\rho^1, \rho^2, \ldots$ coincides with $\tau^1, \tau^2, \ldots$.

Time complexity. The time complexity for computing Q^1 is $\mathcal{O}(q^2t^2m)$, since there are $N_\varepsilon(t)+1$ most probable paths to compute and the time to compute one probable path is $\mathcal{O}(qtm)$. Note that $N_\varepsilon(t)$ is linear in $\mathcal{O}(qt)$ [4]. To compute ρ^k, there are at most $N_\varepsilon(t)$ paths in Q^k, so it takes $\mathcal{O}(qt\log(qt))$ time [10]. There are $k-1$ such paths (ρ^2 through ρ^k) to be computed. This yields a total time complexity $\mathcal{O}(q^2t^2m + kqt\log(qt))$.

A refined algorithm. The above algorithm will generate a sequence of evidences $\rho^1, \ldots \rho^k$ by the decreasing order of their probability product $\Pr_{\leqslant t}(\rho^i, qt)$. However, $N_\varepsilon(t)$ is usually large (typically, a few hundred or thousand) yielding a large set Q^1. As a result, the above algorithm is costly. We now suggest a heuristic to improve this strategy. The basic idea is to use the Poisson probability function to obtain smaller counterexamples.

Observation 1: We first notice that for fixed qt, the Poisson probability $PP(h, qt)$ is maximal when $h = \lceil qt \rceil$ or $h = \lfloor qt \rfloor$, which is the expectation of h. If $h < \lceil qt \rceil$, $PP(h, qt)$ is monotonically increasing and if $h > \lfloor qt \rfloor$, $PP(h, qt)$ is monotonically decreasing, cf. the figure below where the horizontal axis is the hop count h. The function is only non-zero at integer values of h. The connecting lines do not indicate continuity. This observation justifies the heuristics that we start from the crest of the function $(h = \lceil qt \rceil)$ and proceed in two directions, in which way the hop counts for larger Poisson probabilities are explored first, as a consequence, the probability product will usually be large. This bidirectional increments will stop when the bounds 0 and $N_\varepsilon(t)$ have been reached.



Observation 2: When the value qt is small, the Poisson probability is almost monotonically decreasing, cf. case $\lambda = 1$ in the figure. Then $h = \lceil qt \rceil$ is not suitable as the starting point any more.

Let $\varpi_{l^*}^1$ be the most probable path in $\mathcal{U}^\otimes$. It means that paths with $h \neq l^*$ have less or equal probability than $\varpi_{l^*}^1$. Therefore, $h = l^*$ is also considered as a starting point. Due to Observation 1 and 2, our algorithm starts from exploring paths with $H_0 = \max\{\lceil qt \rceil, l^*\}$.

We use the priority queue pq to keep track of the paths which have been explored but not yet expanded. A path is "explored" when its probability is computed and added

to the total counterexample probability. Note that every path that is explored is already taken as an evidence. This is different from the previous algorithm where we might explore many more paths (the huge basic set Q^1) than actually needed. That also partly explains why this algorithm is more efficient. A path is called "expanded" when it is dequeued from pq and its successor is computed. When a path is dequeued, it means that it has the largest probability product among all the paths in the queue; and this fact makes the expansion reasonable.

New path(s) or evidence(s) will be added to the counterexample in each iteration. The increments are in two dimensions. In one dimension, we have to increase the index of some most probable path. More specifically, we dequeue the path ϖ_h^j with the highest probability $\Pr_{\leqslant t}(\varpi_h^j, qt)$ from pq, and add its successor ϖ_h^{j+1}, namely the $(j+1)$-st most probable path with the same hop count. This happens in each iteration. In the other dimension, the minimal and maximal number of hops of paths are incremented. We use $H_{\min}$ and $H_{\max}$ to denote the minimal and maximal hop counts explored so far. Two more new paths with $H_{\min} - 1$ and $H_{\max} + 1$ hops are added, namely, $\varpi^1_{H_{\min}-1}$ and $\varpi^1_{H_{\max}+1}$ when the bounds 0 and $N_\varepsilon(t)$ have not yet been reached. The more iterations the algorithm proceeds, the farther away $H_{\min}$ and $H_{\max}$ are from H_0 and the less Poisson probability the path will have.

The sketch of the improved algorithm is shown as follows:

(1) Compute most probable path $\varpi^1_{l^*}$ in $\mathcal{U}^\otimes$; * Initialization: *\
(2) $pr := \Pr_{\leqslant t}(\varpi^1_{l^*})$; PriorityQueue $pq.enqueue(\varpi^1_{l^*})$;
(3) $H_{\min} := H_{\max} := \max\{\lceil qt \rceil, l^*\}$; $k := 1$; $\rho^k = \varpi^1_{l^*}$;
(4) **while** $pr \leqslant p$ **do** * Main body: *\
(5) $(h', j') := pq.\ dequeue()$; $\varpi_1 := \varpi(h', j'+1, qt)$; * Increments on j *\
(6) $pq.enqueue(\varpi_1)$; $pr := pr + \Pr_{\leqslant t}(\varpi_3)$; $\rho^k := \varpi_1$; $k := k+1$;
(7) **if** $H_{\min} > 0$ **then** * Decrease of hop count *\
(8) $H_{\min} := H_{\min} - 1$; $\varpi_2 := \varpi(H_{\min}, 1, qt)$; $pq.enqueue(\varpi_2)$;
(9) $pr := pr + \Pr_{\leqslant t}(\varpi_2)$; $\rho^k := \varpi_2$; $k := k+1$;
(10) **if** $H_{\max} < N_\varepsilon(t)$ **then** * Increase of hop count: *\
(11) $H_{\max} := H_{\max} + 1$; $\varpi_3 := \varpi(H_{\max}, 1, qt)$; $pq.enqueue(\varpi_3)$;
(12) $pr := pr + \Pr_{\leqslant t}(\varpi_3)$; $\rho^k := \varpi_3$; $k := k+1$; **od**;
(13) **return** $(\rho^1, ..., \rho^{k-1})$;

Note that $\varpi^1_{l^*}$ in Line (1) can be computed by SP algorithms, say Dijkstra's [6], in $\mathcal{G}_{\mathcal{U}^\otimes}$.

7 Conclusion

Comparison of algorithms. This paper presented a set of approximate algorithms for computing small sets of paths that indicate the violation of time-bounded constrained reachability probabilities. The algorithm involving time bounds for computing informative evidences considers Poisson probability besides the probability of paths themselves, which characterizes the significance of the paths in $\mathcal{U}$, thus provides a clue of the significance of the corresponding evidences in $\mathcal{C}$. As we mentioned, the first algorithm of pure graph analysis *cannot* guarantee that for the sequence of paths that computed by our algorithm in order: $\rho^1, ..., \rho^k$, it holds that $\Pr_{\leqslant t}(\rho^1) \geqslant ... \geqslant \Pr_{\leqslant t}(\rho^k)$. Unfortunately, the one involving time also cannot guarantee this, however, it *can* guarantee that

$\Pr_{\leqslant t}(\rho^1, qt) \geqslant ... \geqslant \Pr_{\leqslant t}(\rho^k, qt)$ which is usually very close to the target sequence. The refined algorithm exploits the monotonicity of the Poisson probability function to obtain small counterexamples. Experimental research of the proposed algorithms is to be carried out as the future work.

Related work. Aljazzar et al. [1][2] applied directed explicit-state search algorithms to determine a set of diagnostic traces which carry large amount of probability. Their algorithms are guided by heuristics which exploit stochastic information on the traces. In contrast, we have proposed several algorithms according to different levels of knowledge about the CTMCs, which to some extent shows the significant role of probability and time. The uniformization technique discretizes the continuous-time setting and makes the efficient algorithms for DTMC counterexample-generation [10] adaptable here. Moreover, the analysis and utilization of Poisson probability distribution gives rise to an almost decreasing order of the evidence probabilities, which enables the incremental exploration of the candidate evidence set.

Acknowledgment. Holger Hermanns and Boundewijn R. Haverkort are thanked for useful discussions. This research has been performed as part of the QUPES project that is financed by the Netherlands Organization for Scientific Research (NWO).

References

1. Aljazzar, H., Hermanns, H., Leue, S.: Counterexamples for timed probabilistic reachability. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 177–195. Springer, Heidelberg (2005)
2. Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 33–51. Springer, Heidelberg (2006)
3. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Model-checking continous-time Markov chains. ACM Trans. Comput. Log. 1(1), 162–170 (2000)
4. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. 29(6), 524–541 (2003)
5. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29 (2002)
6. Dijkstra, E.W.: A note on two problems in connection with graphs. Num. Math. 1, 395–412 (1959)
7. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. 28(2), 652–673 (1998)
8. Fox, B.L., Glynn, P.W.: Computing Poisson probabilities. Comm. ACM 31(4), 440–445 (1988)
9. Gurfinkel, A., Chechik, M.: Proof-like counter-examples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 160–175. Springer, Heidelberg (2003)
10. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
11. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. 6(5), 512–535 (1994)
12. Jensen, A.: Markoff chains as an aid in the study of Markoff processes. Skand. Aktuarietidskrift 36, 87–91 (1953)
13. Katoen, J.-P., Khattri, M., Zapreev, I.S.: A Markov reward model checker. In: QEST 2005, pp. 243–244. IEEE Computer Society Press, Los Alamitos (2005)

14. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 2.0: A tool for probabilistic model checking. In: QEST 2004, pp. 322–323. IEEE Computer Society Press, Los Alamitos (2004)
15. Marie, R.A., Reibman, A.L., Trivedi, K.S.: Transient analysis of acyclic Markov chains. Perform. Eval. 7(3), 175–194 (1987)
16. Neuts, M.F.: Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach. The Johns Hopkins Univ. Press (1981)
17. Qureshi, M., Sanders, W.: A new methodology for calculating distributions of reward accumulated during a finite interval. In: FTCS, pp. 116–125. IEEE Computer Society, Los Alamitos (1996)
18. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 275–287. Springer, Heidelberg (2003)

Assertion-Based Proof Checking of Chang-Roberts Leader Election in PVS*

Judi Romijn[1], Wieger Wesselink[1], and Arjan Mooij[2]

[1] Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{j.m.t.romijn,j.w.wesselink}@tue.nl
[2] School of Computer Science, The University of Nottingham
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom
arjan.mooij@cs.nott.ac.uk

Abstract. We report a case study in automated incremental assertion-based proof checking with PVS. Given an annotated distributed algorithm, our tool ProPar generates the proof obligations for partial correctness, plus a proof script per obligation. ProPar then lets PVS attempt to discharge all obligations by running the proof scripts.

The Chang-Roberts algorithm elects a leader on a unidirectional ring with unique identities. With ProPar, we check its correctness with a very high degree of automation: over 90% of the proof obligations is discharged automatically. This case study underlines the feasibility of the approach and is, to the best of our knowledge, the first verification of the Chang-Roberts algorithm for arbitrary ring size in a proof checker.

1 Introduction

Checking proofs with proof assistant tools is a recognized and valuable activity. Manual proofs tend to have small mistakes, and sometimes more serious ones. Tools such as PVS provide powerful generic strategies, but the interaction with these is usually rather involved. For correctness of distributed systems, the tedious and cumbersome task of proof checking may benefit greatly from automation if there are general proof structures that apply and if a formalization of the specification language is available. E.g., the TAME strategies [1] for PVS relieve the book keeping in interactive proof checking for I/O automata.

In [15], we introduced a front-end tool supporting assertion-based verification in the style of Owicki and Gries [18]. Given an annotated program, our tool ProPar (Proof checking of Parallel programs) generates the proof obligations for partial correctness, plus a proof script per obligation. It feeds the resulting specification to the proof checker PVS [19], which attempts to discharge each proof obligation by running the supplied proof script.

There is a growing interest to use general purpose provers like PVS as a back-end for dedicated proof checking tasks. For example, [10] discusses

* This research was supported by the NWO under project 016.023.015: "Improving the Quality of Protocol Standards".

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 347–361, 2007.
© Springer-Verlag Berlin Heidelberg 2007

customizations of and extensions to PVS that would support this, while [16] describes a tool that supports batch proving using PVS.

Closer to our work is [17,20], in which the Owicki-Gries theory is formalized in the Isabelle prover. Their work is mainly theoretical, while we aim at supporting the proof checking of concrete examples in an effective way, and at dealing with the incremental nature of constructing a correct annotation.

This paper reports a case study: we apply our tool ProPar to the ring leader election algorithm by Chang and Roberts [3]. Our interest in the correctness of this algorithm is twofold. For this simple algorithm, some of the correctness proofs require rather involved reasoning on the ring structure, making it a non-trivial case study. In addition, to the best of our knowledge, no existing proof for this protocol for arbitrary ring size has ever been checked mechanically.

We construct a correct annotation of the algorithm in several steps, while illustrating the use of ProPar and PVS. Of the final annotation, over 90% of the proof obligations has been discharged automatically. In four cases we have to supply a proof in PVS ourselves, only two of these proofs are non-trivial.

Compared to [15,13], the effectiveness and user-friendliness of ProPar has been improved greatly, according to the findings of this case study.

Overview. This paper is organised as follows. In Section 2, we recall the Owicki-Gries theory and explain ProPar. In Section 3, the Chang-Roberts algorithm is introduced, and related work on correctness of this algorithm is discussed. Sections 4, 5 and 6 present the actual annotation of Chang-Roberts and our ProPar/PVS efforts. Section 7 has the conclusions and future work.

2 The Owicki-Gries Theory and the Tool ProPar

We check the correctness of annotated programs with the tool ProPar[1] (Proof checking of Parallel programs), which we introduced in [15]. ProPar takes an annotated program as input, and generates proof obligations for the PVS proof checker[2] [19] for local and global correctness of the annotation. Per proof obligation, ProPar generates a proof script to enable running PVS in batch mode.

2.1 Annotated Programs

ProPar accepts the following language constructs in annotated programs:

- empty statement (**skip**)
- sequential composition of two or more statements (. . . **;** . . .)
- alternative selection of one or more guarded statements (**if** . . . **fi**)
- repetition of one or more guarded statements (**do** . . . **od**)
- parallel composition over the elements of a type (**par** x: . . . **rap**)

[1] ProPar is available from the authors upon request.

[2] ProPar can also generate Isabelle output, but that feature was not exploited here.

- parallel composition of two or more statements[3] (**co** ... **oc**)
- atomic statements

Multiple guarded statements are separated by []; a guard is separated from the corresponding statement by $\rightarrow$. Atomic statements are statements whose execution cannot be interfered by executions of statements in other processes. An example is the multiple assignment $x, y := a, b$. It is up to the user to determine the atomic statements, and to model them in the language of the prover.

Programs can be annotated using assertions, that may be placed at the control points of a program, i.e. at locations right before or after a statement. Assertions are predicates on the state of the program. For repetitions and parallel compositions there is also a notion of invariants. An invariant must be placed at the control point i before keyword **do** (resp **par**), and is equivalent to an assertion placed at i and at all control points within the repetition (resp. parallel composition). This gives the user a convenient shorthand and allows ProPar to combine many duplicate proof obligations.

2.2 Proof Obligations

An annotated program is to be considered correct if all assertions are correct. An assertion at a control point is correct if the state of the program satisfies the assertion, whenever execution is at the control point. Note that termination of programs is not considered. The tool ProPar automatically generates proof obligations from which the correctness of a program can be derived.

All proof obligations are expressed in terms of Hoare triples. A Hoare triple $\{P\}\ S\ \{Q\}$ is a boolean that is *true* if and only if each terminating execution of statement S that starts from a state satisfying predicate P is guaranteed to end up in a state satisfying predicate Q. The weakest liberal precondition $wlp.S.Q$, is the weakest precondition P such that $\{P\}\ S\ \{Q\}$ is a correct Hoare triple. More formally $\{P\}\ S\ \{Q\} \equiv [P \Rightarrow wlp.S.Q]$, where $[\ldots]$ is a shorthand for "for all states", i.e. a universal quantifier binding all free variables.

According to Owicki-Gries [18] an assertion Q in a process is correct iff:

- *local correctness*: If Q is an initial assertion, Q is implied by the precondition of the program. If Q is preceded by atomic statement $\{P\}\ S$ (with P an assertion preceding statement S), then $\{P\}\ S\ \{Q\}$ is a correct Hoare triple.
- *global correctness*: For each atomic statement $\{P\}\ S$ in a different process, $\{P \wedge Q\}\ S\ \{Q\}$ is a correct Hoare triple.

ProPar obtains the proof obligations for local correctness by rewriting (parts of) the annotated program according to Table 1, while applying each line as rewrite rule from left to right. The rewriting ends when no statements remain. Nested parallel compositions are treated seamlessly in this manner. The Hoare

[3] A **co** statement can be easily expressed in terms of **par**. However, in many cases the number of proof obligations is smaller for **co**.

Table 1. The local correctness proof obligations per statement type

$$\{P\}\text{skip}\{R\} \quad \Leftarrow \quad [P \Rightarrow R]$$

$$\{P\}\text{atomic-statement-S}\{R\} \quad \Leftarrow \quad [P \Rightarrow \text{wlp.atomic-statement-S}.R]$$

$$\{P\}S_0; \{Q\}S_1\{R\} \quad \Leftarrow \quad \begin{cases} \{P\}S_0\{Q\} \\ \{Q\}S_1\{R\} \end{cases}$$

$$\begin{array}{l} \{P\} \\ \textbf{do } B_0 \rightarrow \{P_0\}S_0 \\ [\,]\ B_1 \rightarrow \{P_1\}S_1 \\ \textbf{od} \\ \{R\} \end{array} \quad \Leftarrow \quad \begin{cases} [P \wedge B_0 \Rightarrow P_0] \\ [P \wedge B_1 \Rightarrow P_1] \\ [P \wedge \neg(B_0 \vee B_1) \Rightarrow R] \\ \{P_0\}S_0\{P\} \\ \{P_1\}S_1\{P\} \end{cases}$$

$$\begin{array}{l} \{P\} \\ \textbf{if } B_0 \rightarrow \{Q_0\}S_0 \\ [\,]\ B_1 \rightarrow \{Q_1\}S_1 \\ \textbf{fi} \\ \{R\} \end{array} \quad \Leftarrow \quad \begin{cases} [P \wedge B_0 \Rightarrow Q_0] \\ \{Q_0\}S_0\{R\} \\ [P \wedge B_1 \Rightarrow Q_1] \\ \{Q_1\}S_1\{R\} \end{cases}$$

$$\begin{array}{l} \{P\} \\ \textbf{par } x : \\ \qquad \{Q_0.x\}S.x\{Q_1.x\} \\ \textbf{rap} \\ \{R\} \end{array} \quad \Leftarrow \quad \begin{cases} [\forall x : P \Rightarrow Q_0.x] \\ \forall x : \{Q_0.x\}S.x\{Q_1.x\} \\ [(\forall x : Q_1.x) \Rightarrow R] \end{cases}$$

$$\begin{array}{l} \{P\} \\ \textbf{co} \\ \qquad \textbf{proc } \{P_0\}S_0\{Q_0\} \textbf{ corp} \\ \qquad \textbf{proc } \{P_1\}S_1\{Q_1\} \textbf{ corp} \\ \textbf{oc} \\ \{R\} \end{array} \quad \Leftarrow \quad \begin{cases} [P \Rightarrow P_0] \\ [P \Rightarrow P_1] \\ \{P_0\}S_0\{Q_0\} \\ \{P_1\}S_1\{Q_1\} \\ [(Q_0 \wedge Q_1) \Rightarrow R] \end{cases}$$

triples corresponding to global correctness are related to atomic statements, and can therefore be expressed directly in wlps of atomic statements.

To illustrate the proof obligations generated by ProPar, we consider the program fragment of the parallel composition in Table 1, where we assume that assertions P, Q_0, Q_1 and R are placed at labels 0, 1, 2 and 3. The three local correctness proof obligations on the right hand side are encoded in PVS as follows, where we assume that S is an atomic statement:

loc_Q0_stat_0: **lemma**
  forall $(s : state)$: **forall** $(x : X)$: $lab_0(s) \Rightarrow Q0(x)(s)$
loc_Q1_stat_1: **lemma**
  forall $(s : state)$: **forall** $(x : X)$: $lab_1(x)(s) \Rightarrow \text{wlp_}S(x)(Q1(x))(s)$
loc_R_stat_2: **lemma**
  forall $(s : state)$: (**forall** $(x : X)$: $lab_2(x)(s)) \Rightarrow R(s)$

Here, the variable s ranges over all possible program states, and is introduced to model the brackets $[\ldots]$ in Table 1. The logical variable lab_i represents the conjunction of the assertions located at the control point with label i.

2.3 Proof Scripts

ProPar generates PVS proof scripts for each of the generated proof obligations, and writes them to a .prf file in the PVS proof format. PVS is then run in batch execution mode to check whether each proof obligation is discharged.

As the first step in a proof script, ProPar automatically selects assertions and invariants from relevant control points and inserts them with the `lemma` command. Here, ProPar also instantiates quantifier variables for the state with appropriate skolem variables. Contrary to the previous version, ProPar now always explicitly chooses the skolem variables itself. Experience has taught us that the automatic choices of PVS are not always suitable.

The second step is a simplification step, in which commands like `replace`, `assert` and `simplify` are used. We now discuss an example of the difficulties encountered when trying to automate this step. Consider the proof state

```
{-1}  FORALL (c: component): lab_6(c)(s!1) => inv_0a(s!1)
[-2]  FORALL (c: component): lab_6(c)(s!1)
  |-------
[1]   ass_7a(s!1)
```

Given the assumption that type *component* is non-empty, one would expect there are high-level commands available that simplify this into

```
{-1} inv_0a(s!1)
[-2] FORALL (c: component): lab_6(c)(s!1)
  |-------
[1]  ass_7a(s!1)
```

However, we did not succeed in generating proof scripts that achieve this simplification in all the different contexts that may occur. We solved this problem by introducing two custom lemmas as follows, with t a generic type:

quantifier_lemma_1: **lemma**
$$\textbf{forall}\ (P : \mathit{bool}) : (\textbf{forall}\ (x : t) : P) = ((\exists (x : t) : \mathit{true}) \Rightarrow P)$$
quantifier_lemma_2: **lemma**
$$\textbf{forall}\ (P : [t \rightarrow \mathit{bool}], Q : [t \rightarrow \mathit{bool}]) : (\textbf{forall}\ (x : t) : P(x)) \Rightarrow (\textbf{forall}\ (x : t) : Q(x)) = \textbf{forall}\ (x : t) : P(x) \Rightarrow Q(x)))$$

This is a generalization of the approach we used in previous versions [15,13].

The third and last part of a proof script consists of a generalization of the `grind` command to perform the hard work. This is the same as in [15].

2.4 User Input

The input of ProPar consists of

- An annotated program, written in a prover independent language.
- An import file containing prover-dependent definitions.
- A file containing proof hints (optional, see below).
- A file containing manual proofs (optional, see below).

The annotated program contains only references to assertions, guards and wlp's of atomic statements. In the import file, the user has to express these in the language of the prover. Usually this is a straightforward task.

Proof Hints. When the automatic proof of a lemma fails, the user may provide proof hints, i.e. high-level directions to help the prover for a certain lemma. Compared to [15], the use of proof hints is now much more generic. Proof hints apply to the sequence of lemmas introduced in the beginning of a proof script. If ProPar introduces too many lemmas, the prover becomes very inefficient. The order of the lemmas can also influence the performance of the prover. Moreover, additional lemmas from other theories may be needed. Through proof hints the prover can focus on the appropriate lemmas, in the optimal order.

Finally, if all else fails, the user can supply a manual proof.

3 The Chang-Roberts Leader Election Algorithm

The leader election algorithm introduced by Chang and Roberts in [3] is designed for a uni-directional ring consisting of components with unique identities. A strict total order $>$ on the identities is assumed. The algorithm elects the greatest identity present according to $>$.

Each component may send its own unique identity to its neighbour in the ring. Components only forward messages with identities which are greater than any identity received thus far. The component that receives its own identity concludes that its identity is the greatest in the ring and wins the election.

The algorithm was intended as an improvement on the leader election part of Le Lann's token passing algorithm [8]. Here, due to faulty connections, multiple elections can overlap and correctness is not guaranteed. Like [3], we restrict ourselves to reliable connections and one election round only.

3.1 The Algorithm in Assertion-Based Style

In Figure 1, the Chang-Roberts leader election algorithm is shown, including assertions for the correctness specification. The program is a parallel composition of the repetition to be executed by each component. The program terminates only when all components have finished the repetition. The labels 0a, 0b, etc. that precede assertions are generated by ProPar, we stick to that labelling in the remainder of this paper. Likewise, we refer to the guards in the selection statement at location 2 as guard 2a, 2b and 2c.

```
    var leader : [comp → nat],   ready : [comp → bool]
0:  {ass 0a: (∀_{c:comp} : leader_c = id(c))}
    {ass 0b: (∀_{c:comp} : ¬ready_c)}
    par (c : comp):
1:      do ¬ready_c  →
2:          if  ready_{prev(c)}  →
3:                ready_c := true
            []  leader_{prev(c)} = id(c)  →
4:                ready_c := true
            []  leader_{prev(c)} > leader_c  →
5:                leader_c := leader_{prev(c)}
            fi
        od
6:  rap
7:  {ass 7a: (∀_{c1,c2:comp} : leader_{c1} = leader_{c2}}
    {ass 7b: (∀_{c1,c2:comp} : leader_{c1} ≥ id(c2))}
    {ass 7c: (∃_{c:comp} : leader_c = id(c)}
```

Fig. 1. The Chang-Roberts leader election algorithm

Assertions 0a, 0b, 7a, 7b and 7c express the correctness of the algorithm. These enforce that upon termination of the parallel statement, all components have elected the same leader, i.e. the greatest identity present in the ring.

To start, each component has its own identity for leader. Inside the repetition at control point 1, the election takes place. Each component c monitors the leader identity of its immediate neighbour $prev(c)$ in the unidirectional ring. When the neighbour's leader identity is greater than the component's own leader identity, guard 2c evaluates to true, and the component may copy it in statement 5. In this manner, candidate leader identities spread over the ring, until they are overtaken by a better identity. The greatest identity spreads over the ring until it reaches the originating component. Then this component signals it has won because guard 2b evaluates to true. At this point, the election is finished.

A component can only terminate the repetition after its *ready* flag has been set to true. In this way the algorithm ensures that all components can find out that the election has ended, an aspect that is often ignored in the literature.

We model communication by having the receiver poll the sender's current leader identity. This is clearly equivalent to synchronous communication with explicit messages. Moreover, for this particular algorithm, a version with asynchronous communication simulates our polling version: the sending of a message in the asynchronous case can be related to the polling moment in our version.

Ring structure. We assume the type *comp* for the components on the ring, and a constant function $id : [comp \to nat]$ mapping each component to its unique identity which is a natural number. In this way we immediately have the total order $>$. For the ring structure, we assume the constant function $prev : [comp \to comp]$ which points at the predecessor of a component in the

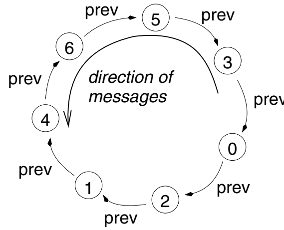


Fig. 2. A unidirectional ring

ring. An example ring is shown in Figure 2. Here, the unique identity of a component is in the node, and the arrows between components indicate the predecessor relation.

3.2 Related Work

Lynch et al. have studied Chang-Roberts in the I/O automata language. [12] has a correctness proof, but proofs are only sketched. In [11,9] a performance analysis of a timed version is given but the proofs are not checked (in contrast to other results in both publications). An IOA model is online at the IOA homepage (`http://theory.csail.mit.edu/tds/ioa/`). In [7], a finite instance of this IOA model is checked in Isabelle/IOA.

Garavel et al. [6] have model checked Le Lann's full token passing algorithm for concrete ring sizes in the formal language LOTOS and proposed an improvement. They find mistakes in the presence of faulty connections that lead to overlapping election rounds, which is outside the scope of our case study. However, the mistakes found and the improvement proposed suggest that having a proper notification of the election termination should be part of the protocol. We have this in our version of the leader election, as explained above.

Sen [21] reports on another model checking experiment in his thesis, comparing the tool POTA to SPIN.

A correctness proof for arbitrary ring size is given by Chen et al in [4] in the formal language μCRL, but it has not been proof checked in any tool. In fact, this paper turns out to contain some essential mistakes in both specification and proof, thus underlining the need for machine-checked verifications.

Many papers study the performance of Chang-Roberts and related algorithms (like [2]). That is outside the scope of this paper.

Apparently, despite the attention in the literature, the Chang-Roberts algorithm has not been verified mechanically for arbitrary ring size. It seems that this algorithm, while small and simple, has the appropriate degree of complexity to make it a nice proof checking challenge for our tool ProPar.

4 A Correct Chang-Roberts Annotation (Phase 1)

We must extend the annotation given in Figure 1 with assertions and invariants until it is provably correct, i.e. each proof obligation generated by ProPar is discharged either automatically by running PVS on the generated proof script, or manually by proving it in PVS ourselves. In the coming sections, we extend the annotation and then run ProPar to see how far PVS gets in batch mode.

In the first annotation step, we work from the annotation in Figure 1 towards the one in Figure 3 (new parts marked $*$).

We start by adding invariants. We weaken assertion 7b to a local version in invariant 0a. The two are equivalent when assertion 7a holds. Assertion 7a becomes true when at least one component signals that the election is finished. We state this in invariant 0b with no more than two quantified variables, using transitivity of $=$. When exiting the parallel statement, by successful termination of each repetition, the premise in invariant 0b holds for all components hence establishing assertion 7a. For assertion 7c, we observe that each leader identity stored by a component is the identity of some component in the ring. We state this as invariant 0c. Combining assertion 7a and invariant 0c yields assertion 7c.

To establish the correctness of these invariants, we add assertions to the control points inside the parallel statement. We start by stating each selection guard as an assertion following the guard's execution. We can do so if the guard continues to hold regardless of what the other components do. For each of the guards 2a, 2b and 2c this is indeed the case. Similarly, we state the negation of the repetition guard at location 6.

In addition, we state in assertion 4b that the current component's leader identity is equal to its neighbour's leader identity. We also state in assertion 5b that none of the components have finished the repetition.

4.1 ProPar Results

We run the ProPar tool on the files containing the annotated algorithm from Figure 3. ProPar generates 36 proof obligations and proof scripts. Of these, 29 are discharged automatically by PVS. The seven remaining obligations are:

- local correctness of assertion 4b (when executing guard 2b),
- local correctness of assertion 7c (when exiting the parallel statement), and
- global correctness of assertions 4a and 4b when executing assignment 5,
- global correctness of assertion 5b when executing assignment 4,
- global correctness of invariant 0b when executing assignment 3 or 4.

For assertion 4b, when guard 2b holds, clearly c still has its own identity for leader. If this was not the case, then it copied a better identity from its predecessor, and then guard 2b cannot hold. Reasoning about this requires information on the leader identities that a component has had between the first (its own) and the last (the winner). We can do so by adding a history variable, which is discussed in the next section. We choose this as our next move, in the hope that it will help in discharging the other proof obligations.

var $leader : [comp \rightarrow nat]$, $ready : [comp \rightarrow bool]$

0: {**ass** 0a: $(\forall_{c:comp} : leader_c = id(c))$}
{**ass** 0b: $(\forall_{c:comp} : \neg ready_c)$}
{**inv** 0a: $(\forall_{c:comp} : leader_c \geq id(c))$} *
{**inv** 0b: $(\forall_{c_1,c_2:comp} : ready_{c_1} \Rightarrow leader_{c_1} = leader_{c_2})$} *
{**inv** 0c: $(\forall_{c_1:comp} : (\exists_{c_2:comp} : leader_{c_1} = id(c_2)))$} *
par $(c : comp)$:
1: **do** $\neg ready_c \rightarrow$
2: **if** $ready_{prev(c)} \rightarrow$
3: {**ass** 3a: $ready_{prev(c)}$} *
$ready_c := true$
[] $leader_{prev(c)} = id(c) \rightarrow$
4: {**ass** 4a: $leader_{prev(c)} = id(c)$} *
{**ass** 4b: $leader_c = leader_{prev(c)}$} *
$ready_c := true$
[] $leader_{prev(c)} > leader_c \rightarrow$
5: {**ass** 5a: $leader_{prev(c)} > leader_c$} *
{**ass** 5b: $(\forall_{c_1:comp} : \neg ready_{c_1})$} *
$leader_c := leader_{prev(c)}$
fi
od
6: {**ass** 6a: $ready_c$} *
rap
7: {**ass** 7a: $(\forall_{c_1,c_2:comp} : leader_{c_1} = leader_{c_2}$}
{**ass** 7b: $(\forall_{c_1,c_2:comp} : leader_{c_1} \geq id(c_2))$}
{**ass** 7c: $(\exists_{c:comp} : leader_c = id(c)$}

Fig. 3. Chang-Roberts algorithm (phase 1, marked $*$)

In order to maintain global correctness of invariant 0b under assignment 4, we calculate the wlp. We find a stronger version of assertion 4b: when guard 2b holds, it is in fact the case that the winning identity has traversed the entire ring is the leader for each component. This is added as assertion 4c in the following section. This assertion will help in discharging the remaining global correctness obligations for assertions 4a, 4b and 5b. Note that reasoning to show local correctness of the new assertion 4c requires induction on the ring structure. Here, the history variable can be useful too.

5 A Correct Chang-Roberts Annotation (Phase 2)

The first step for dealing with the remaining proof obligations from Section 4.1, for which PVS cannot successfully execute the ProPar proof script, is to introduce a history variable $leaders_c$ for each component c in the ring. The history variable records all the values that the program variable $leader_c$ takes on during execution. This enables us to compare current and past leader identities of a component and its neighbour. We can add any history variable if it does not change the program's behaviour.

We now state invariants 0d to 0g, expressing that each new leader identity accepted by component c is better than c's own identity, has been copied from c's predecessor, and hence must be in the $leaders_{prev(c)}$ collection.

The new annotation is Figure 4 with everything from the previous annotation (unmarked) and the new parts added in this phase (marked $*$). Note that the part marked $**$ belongs with the final annotation (see Section 6). The two annotations are merged in Figure 4 to save space.

As announced, we also add assertion 4c which helps to maintain invariant 0b under assignment 4, and which implies assertion 4b. Assertion 4a is weakened to make use of variable *leaders*, by combining guard 2b and invariant 0d.

For history variable *leaders*, we ensure the proper initial value with the new assertion 0c. Its value is updated in statement 5: whenever a new leader identity is copied it is also added to the *leaders* collection. Invariants 0d to 0g express the additional information that we require for proving correctness.

5.1 ProPar Results

We run ProPar on the incremented annotated algorithm from Figure 4 (except the part marked $**$). Of the 52 proof obligations generated, PVS discharges 46 automatically through the ProPar proof scripts.

We notice that PVS fails for local correctness of assertions 5b and 7a whereas it ran successfully for the previous annotation. PVS gets confused by the growing number of applicable lemmas: this annotation has seven invariants instead of three. If we give ProPar proof hints for these failing proof, we can easily generate the successful script again. We supply only the invariants from the previous annotation plus the assertions of the location prior to the current statement:

```
loc_ass_5b_stat_2: lab_2_inv_0a lab_2_inv_0b lab_2_inv_0c
loc_ass_7a_stat_6: lab_6_ass_6a lab_6_inv_0a lab_6_inv_0b lab_6_inv_0c
```

With these proof hints, the proofs generated by ProPar are accepted by PVS.

Of the 52 proof obligations, PVS has now 48 discharged automatically. The four obligations for which PVS fails are ($*$ marks the new obligation):

– local correctness of assertion 4b (when executing guard 2b),
$*$ local correctness of assertion 4c (when executing guard 2b),
– local correctness of assertion 7c (when exiting the parallel statement), and
– global correctness of invariant 0b when executing assignment 3.

Apparently, all previous global correctness obligations are now discharged except the one for invariant 0b under assignment 4, assertions, and only one of the new proof obligations remains unproved.

We establish local correctness of assertion 7c with local correctness of assertion 7a and invariant 0c as proof hints. Since 7a and 7c are at the same control point, with 7a preceding 7c, we can use local correctness of 7a as a lemma here. We adjusted ProPar to allow such proof hints.

For local correctness of assertion 4c, we need one final invariant. This is described in the following section.

var $\mathit{leader} : [\mathit{comp} \to \mathit{nat}]$, $\mathit{leaders} : [\mathit{comp} \to \mathit{setof}(\mathit{nat})]$, $\mathit{ready} : [\mathit{comp} \to \mathit{bool}]$	
0: $\{$**ass** 0a: $(\forall_{c:comp} : \mathit{leader}_c = \mathit{id}(c))\}$	
$\{$**ass** 0b: $(\forall_{c:comp} : \neg \mathit{ready}_c)\}$	
$\{$**ass** 0c: $(\forall_{c:comp} : \mathit{leaders}_c = \{\mathit{leader}_c\})\}$	
$\{$**inv** 0a: $(\forall_{c:comp} : \mathit{leader}_c \geq \mathit{id}(c))\}$	
$\{$**inv** 0b: $(\forall_{c_1,c_2:comp} : \mathit{ready}_{c_1} \Rightarrow \mathit{leader}_{c_1} = \mathit{leader}_{c_2})\}$	
$\{$**inv** 0c: $(\forall_{c_1:comp} : (\exists_{c_2:comp} : \mathit{leader}_{c_1} = \mathit{id}(c_2)))\}$	
$\{$**inv** 0d: $(\forall_{c:comp} : \mathit{leader}_c \in \mathit{leaders}_c)\}$	*
$\{$**inv** 0e: $(\forall_{c:comp,n:nat} : n \in \mathit{leaders}_c \Rightarrow \mathit{leader}_c \geq n)\}$	*
$\{$**inv** 0f: $(\forall_{c:comp} : (\mathit{leaders}_c - \{\mathit{id}(c)\}) \subseteq \mathit{leaders}_{prev(c)})\}$	*
$\{$**inv** 0g: $(\forall_{c:comp,n:nat} : n \in \mathit{leaders}_c \Rightarrow n \geq \mathit{id}(c))\}$	*
$\{$**inv** 0h: $(\forall_{c_1,c_2:comp} : \mathit{leader}_{c_2} = \mathit{id}(c_1)$	
$\Rightarrow (\forall_{c_3:comp} : \mathit{mp}(c_1,c_3) \leq \mathit{mp}(c_1,c_2) \Rightarrow c_1 \in \mathit{leaders}_{c_3}))\}$	**
par $(c : \mathit{comp})$:	
1: **do** $\neg \mathit{ready}_c \to$	
2: **if** $\mathit{ready}_{prev(c)} \to$	
3: $\{$**ass** 3a: $\mathit{ready}_{prev(c)}\}$	
$\mathit{ready}_c := \mathit{true}$	
[] $\mathit{leader}_{prev(c)} = \mathit{id}(c) \to$	
4: $\{$**ass** 4a: $\mathit{id}(c) \in \mathit{leaders}_{prev(c)}\}$	*
$\{$**ass** 4b: $\mathit{leader}_c = \mathit{leader}_{prev(c)}\}$	
$\{$**ass** 4c: $(\forall_{c_1:comp} : \mathit{leader}_{c_1} = \mathit{leader}_c)\}$	*
$\mathit{ready}_c := \mathit{true}$	
[] $\mathit{leader}_{prev(c)} > \mathit{leader}_c \to$	
5: $\{$**ass** 5a: $\mathit{leader}_{prev(c)} > \mathit{leader}_c\}$	
$\{$**ass** 5b: $(\forall_{c_1:comp} : \neg \mathit{ready}_{c_1})\}$	
$\mathit{leader}_c, \mathit{leaders}_c := \mathit{leader}_{prev(c)}, \mathit{leaders}_c \cup \{\mathit{leader}_{prev(c)}\}$	*
fi	
od	
6: $\{$**ass** 6a: $\mathit{ready}_c\}$	
rap	
7: $\{$**ass** 7a: $(\forall_{c_1,c_2:comp} : \mathit{leader}_{c_1} = \mathit{leader}_{c_2}\}$	
$\{$**ass** 7b: $(\forall_{c_1,c_2:comp} : \mathit{leader}_{c_1} \geq \mathit{id}(c_2))\}$	
$\{$**ass** 7c: $(\exists_{c:comp} : \mathit{leader}_c = \mathit{id}(c))\}$	

Fig. 4. Chang-Roberts algorithm (phase 2 marked $*$, and phase 3 marked $**$)

6 A Correct Chang-Roberts Annotation (Phase 3)

We add a final invariant to enable the manual proof for local correctness of assertion 4c. The annotation obtained in this way is Figure 4, now including the line marked $**$. Invariant 0h expresses for component c_2 that has the identity of c_1 for leader, that all components on the predecessor path from c_2 to c_1 have also seen the identity of c_1. Note the use of function $\mathit{mp}(c_1, c_2)$ which computes the distance in the ring between c_1 and c_2 by counting the number of *prev* steps back from c_2 to c_1.

6.1 ProPar Results

Local correctness of assertion 7b is lost, and mended with proof hints, as in Section 5.1. Of the 55 current proof obligations, PVS discharges 51 automatically, 4 of these through our use of proof hints. The obligations for which PVS fails are (∗ marks the new obligation):

– local correctness of assertion 4b (when executing guard 2b),
– local correctness of assertion 4c (when executing guard 2b),
– global correctness of invariant 0b when executing assignment 3.
∗ global correctness of invariant 0h when executing assignment 5.

Only the second obligation seems to require an involved proof, but the others turn out to be too tricky for PVS with ProPar proof scripts, even with our proof hints. All of these require a manual proof.

We run ProPar on the annotation of Figure 4 with the proof hints and manual proofs (discussed in the remainder of this section), to find that all proof obligations are discharged, hence correctness of Chang-Roberts is now established.

6.2 Manual Proofs

When starting a manual proof, the proof script generated by ProPar is very helpful. We use the PVS option to step through the generated proof, until we reach the point where ProPar calls on the semidecision procedures (like `grind`). Here we take over and manually instruct PVS step by step until we have `Q.E.D.`

We have created a theory for the ring structure with domain-specific knowledge. In this theory, we have proved some useful lemmas which are used in the manual proofs for invariant 0b and assertion 4c.

The proofs for local correctness of assertion 4b and for global correctness of invariant 0b under statement 3 are easy. Global correctness of invariant 0h under statement 5 requires a complicated case distinction on the three quantified variables but this is very manageable.

As announced, for local correctness of assertion 4c we must use some kind of induction on the ring structure. We apply `measure-induct+` and use as measure function the distance $mp(c_1, c_2)$. The base case is easy. For the induction step, based on the assumption that we have the same leader for c_1 and $prev(c_2)$ we can then prove the induction step using all invariants except 0b. This requires a complicated proof of about 90 PVS proof steps.

7 Conclusions and Future Work

We successfully applied our method to check correctness of the Chang-Roberts algorithm for arbitrary ring size, by creating a proof in an incremental and automated fashion. In the end, 51 out of 55 proofs were handled automatically. Five of the 51 proofs required straightforward proof hints from the user. In addition, four manual proofs were needed, two of which were rather involved.

The most significant parts of the proof effort were developing a correct annotation, and manually proving the remaining proof obligations. Other activities (creating an import file, creating proof hints and interacting with the tool) were negligible compared to these.

During the case study we made pragmatic improvements to ProPar, that resulted in a higher rate of automatically handled proofs, without changing anything in the theory. E.g., the proof hints mechanism now allows for user-defined lemmas to be introduced, and for local correctness results of pre-assertions at the current control point to be exploited. Custom lemmas are applied to smoothen the quantifications encountered which differ for control points in the body versus the control point at the end of a parallel composition. In addition, the proof scripts have been changed to make them behave in a more predictable way, by explicitly instantiating with skolem variables.

7.1 PVS Discussion

In contrast to manual proofs, when dealing with large numbers of generated proofs, it is essential that proof strategies are robust and behave in a predictable way. In our previous case study [15] we discovered a bug in PVS that prevented certain proofs to be completed automatically. This has been repaired since version 4.0 (PVS bug 920). In this case study another bug has been encountered in PVS 4.0 and submitted (PVS bug 979, reportedly fixed but yet not released).

In some cases unpredictable behavior of PVS caused problems. For example, certain proofs suddenly failed after hiding an unnecessary antecedent. Still many seemingly simple proof obligations exist that PVS cannot handle automatically. To deal with this, we introduced new proof hints, and applied custom lemmas to support the prover.

A more powerful proof script language for PVS is desirable. For example, it would be useful to be able to enumerate possible instantiations of quantifier variables, and to specify in which order PVS should try to use them. PVS commands like `use` and the higher level `grind` often choose the right instantiations, but for proofs in batch mode "often" is not good enough.

7.2 Future Work

There are several directions for future work. Termination of programs (or more generally progress conditions [5]) could be supported by generating proof obligations for the decrease of a user supplied norm function. Another topic is to study to what extent inductive proofs (e.g., for security protocols like in [14]) can be supported, if the user supplies a measure function. Soundness could be warranted by automatically verifying that the generated proof obligations are sufficient, similar to the approach in [17].

Finally, PVS has weak support for controlling instantiations of quantifier variables, which sometimes causes the tail part of the generated proofs to fail. Until PVS is improved in this respect, we can possibly circumvent this by arranging for the user to supply suitable instantiation hints.

References

1. Archer, M., Heitmeyer, C., Riccobene, E.: Proving invariants of I/O automata with TAME. Automated Software Engineering 9(3), 201–232 (2002)
2. Chan, M.Y., Chin, F.Y.L.: Optimal resilient distributed algorithms for ring election. IEEE Trans. on Parallel and Distributed Systems 4(4), 475–480 (1993)
3. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema finding in circular configurations of processes. Comm. of the ACM 22(5), 281–283 (1979)
4. Chen, T., Han, T., Lu, J.: Analysis of a leader election algorithm in μCRL. In: Proceedings of CIT 2005, pp. 841–847. IEEE Computer Society, Los Alamitos (2005)
5. Dongol, B., Mooij, A.J.: Progress in deriving concurrent programs: emphasizing the role of stable guards. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 140–161. Springer, Heidelberg (2006)
6. Garavel, H., Mounier, L.: Specification and verification of various distributed leader election algorithms for unidirectional ring networks. Science of Computer Programming 29(1–2), 171–197 (1997)
7. Hamberger, T.: Integrating theorem proving and model checking in Isabelle/IOA. Technical Report TUM-I, T.U. Munich (1999)
8. Le Lann, G.: Distributed systems - towards a formal approach. In: 1977 IFIP Congress Proceedings, Information Processing, vol. 77, pp. 155–160. North-Holland, Amsterdam (1977)
9. Luchangco, V.: Using Simulation to Prove Timing Properties. PhD thesis, Massachusetts Institute of Technology (1995)
10. Lüttgen, G., Muñoz, C., Butler, R., Vito, B.D., Miner, P.: Towards a customizable PVS. Technical Report ICASE 2000-4, CR-2000-209851. NASA Langley (2000)
11. Lynch, N.A.: Proving performance properties (even probabilistic ones). In: Proceedings of FORTE 1994, pp. 3–20. Chapman and Hall (1995)
12. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Mooij, A.J.: Constructive formal methods and protocol standardization. PhD thesis, Technische Universiteit Eindhoven (2006)
14. Mooij, A.J.: Constructing and reasoning about security protocols using invariants. In: Proceedings of REFINE 2007, ENTCS. Elsevier, Amsterdam (to appear, 2007)
15. Mooij, A.J., Wesselink, J.W.: Incremental verification of Owicki/Gries proof outlines using PVS. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 390–404. Springer, Heidelberg (2005)
16. C. Muñoz. Batch proving and proof scripting in PVS. Technical Report NIA 2007-03, CR-2007-214546, NASA Langley (2007)
17. Nipkow, T., Prensa Nieto, L.: Owicki/Gries in Isabelle/HOL. In: Finance, J.-P. (ed.) FASE 1999. LNCS, vol. 1577, pp. 188–203. Springer, Heidelberg (1999)
18. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6, 319–340 (1976)
19. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
20. Prensa Nieto, L.: Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL. PhD thesis, T.U. Munich (2002)
21. Sen, A.: Techniques for Formal Verification of Concurrent and Distributed Program Traces. PhD thesis, The University of Texas at Austin (2004)

Continuous Petri Nets: Expressive Power and Decidability Issues

Laura Recalde[2], Serge Haddad[1], and Manuel Silva[2,⋆]

[1] LAMSADE-CNRS UMR 7024, University Paris-Dauphine, France
haddad@lamsade.dauphine.fr
[2] GISED, University of Zaragoza, Spain
{lrecalde,silva}@unizar.es

Abstract. State explosion is a fundamental problem in the analysis and synthesis of discrete event systems. Continuous Petri nets can be seen as a relaxation of discrete models. The expected gains are twofold: improvements in comlexity and in decidability. This paper concentrates on the study of decidability issues. In the case of autonomous nets it is proved that properties like reachability, liveness or deadlock-freeness remain decidable. When time is introduced in the model (using an infinite server semantics) decidability of these properties is lost, since continuous timed Petri nets are able to simulate Turing machines.

1 Introduction

State explosion problems represent a main drawback in the study of heavily loaded discrete event dynamic systems, modeled for example as Petri nets. Trying to alleviate these problems, different relaxations have been used, in particular, continuous Petri nets. The expected gains are twofold: improvements in computability and in decidability. An example of computability gain is the study of reachability, that under very general conditions (consistent continuous nets with all the transitions fireable) can be computed in polynomial time [8]. Another property that is improved, now for timed nets, is the computation of an initial marking that maximizes a linear function of the throughput (production), the steady-state marking (work in process) and the initial marking (investment). In continuous equal conflict nets (the weighted version of free choice nets) this initial marking can be computed using a linear programming problem [11]. This paper concentrates on the study of decidability issues. The autonomous model is studied first, and it is proved that properties like reachability, liveness or deadlock-freeness, that are decidable in discrete Petri net systems [4], are also decidable in continuous Petri net systems, as it was expected.

In discrete Petri nets, time has been introduced in many different ways. Here we will concentrate on one of them: a delay is associated to transitions according to a distribution function. Discrete PNs with general probability distribution functions associated to transitions (which includes deterministic timing)

⋆ This work was partially supported by project CICYT and FEDER DPI2003-06376.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 362–377, 2007.
© Springer-Verlag Berlin Heidelberg 2007

are equivalent to Turing machines [3]. However, this is based on the existence of transitions that fire in zero time (immediate transitions), that introduce a notion of priority. In continuous timed Petri nets, these transitions will not be considered, so the model is more restrictive in this sense.

In [2] decidability problems of hybrid Petri nets were studied, proving that reachability was decidable. The model was hybrid i.e., it combined a discrete with a continuous part. Regarding the interpretation, the discrete part was untimed, while the dynamics of the continuous part was based on finite server semantics. In some sense, the model used here is simpler (all the transitions are continuous), but more complex in the dynamics used for the marking evolution (the so called infinite server semantics). Under infinite servers semantics the evolution of the marking can be represented as a set of constant parameters linear differential equations. The switching among them is associated to minimum operators. In [6], an alternative model was defined (timed differentiable Petri nets), and it was shown that it could model a Turing machine. Here it is proved that these two models are equivalent. Moreover, self-loop arcs can be avoided in the model. That is, the modelling power of pure continuous Petri nets under infinite server semantics is equivalent to that of Turing machines.

The structure of the paper is as follows: in Section 2 autonomous and timed continuous Petri nets are introduced, illustrating the kind of behavior they can model with several examples. Decidability of properties like reachability, liveness or deadlock-freeness of the autonomous model is studied in Section 3. Section 4 deals with the timed model, proving that timed continuous Petri nets and timed differentiable Petri nets are in fact equivalent. This is used in Section 5 to deduce that timed continuous Petri nets have Turing machine modelling power.

2 Continuous Petri Nets

2.1 Autonomous Continuous Petri Nets

We assume that the reader is familiar with Petri nets (PNs) (for notation we use the standard one, see for instance [10]).

The structure $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}\rangle$ of *(autonomous) continuous Petri nets* (ACPN) is the same as the structure of discrete PNs. That is, P is a finite set of places, T is a finite set of transitions with $P \cap T = \emptyset$, $\mathbf{Pre}$ and $\mathbf{Post}$ are $|P| \times |T|$ sized, natural valued, *pre- and post- incidence matrices*. The usual PN system, $\langle \mathcal{N}, \mathbf{m_0}\rangle$, will be said to be *discrete* so as to distinguish it from a continuous PN system, in which $\mathbf{m_0} \in (\mathbb{R}_{\geq 0})^{|P|}$. The main difference between both formalisms is in the evolution rule, since in continuous PNs firing is not restricted to be done in integer amounts. As a consequence the marking is not forced to be integer. More precisely, a transition t is *enabled* at $\mathbf{m}$ iff for every $p \in {}^\bullet t$, $\mathbf{m}[p] > 0$, and its *enabling degree* is $\text{enab}(t, \mathbf{m}) = \min_{p \in {}^\bullet t}\{\mathbf{m}[p]/\mathbf{Pre}[p, t]\}$. The firing of t in a certain amount $\alpha \leq \text{enab}(t, \mathbf{m})$ leads to a new marking $\mathbf{m'} = \mathbf{m} + \alpha \cdot \mathbf{C}[P, t]$, where $\mathbf{C} = \mathbf{Post} - \mathbf{Pre}$ is the token-flow matrix.

As in discrete systems, right and left natural annullers of the token flow matrix are called T- and P-semiflows, respectively. When $\mathbf{y} \cdot \mathbf{C} = \mathbf{0}$, $\mathbf{y} > \mathbf{0}$ the net is said

to be *conservative*, and when $\mathbf{C} \cdot \mathbf{x} = \mathbf{0}$, $\mathbf{x} > \mathbf{0}$ the net is said to be *consistent.* A set of places Θ is a *trap* iff $\Theta^\bullet \subseteq {}^\bullet\Theta$. Similarly, a set of places Σ is a *siphon* iff ${}^\bullet\Sigma \subseteq \Sigma^\bullet$. The support of a vector $\mathbf{v} \geq \mathbf{0}$ will be denoted as $\|\mathbf{v}\|$ and represents the set of positive elements of $\mathbf{v}$.

In continuous PNs the reachability concept is not so immediate as in discrete nets. For example, in a continuous net it may happen that the marking of a place can be done smaller and smaller, but never reaches 0. This idea of getting as close as desired to a marking, even if it is never reached with a finite firing sequence leads in [9] to the definition of limit reachability, further refined in [8].

Definition 1. *Let $\langle \mathcal{N}, \mathbf{m_0} \rangle$ be a continuous system. Two reachability concepts are defined:*

- $\mathrm{RS}(\mathcal{N}, \mathbf{m_0}) = \{\, \mathbf{m} \in (\mathbb{R}_{\geq 0})^{|P|} \mid$ *a finite fireable sequence* $\sigma = \alpha_1 t_{a_1} \ldots \alpha_k t_{a_k}$ *exists such that* $\mathbf{m_0} \xrightarrow{\alpha_1 t_{a_1}} \mathbf{m}_1 \xrightarrow{\alpha_2 t_{a_2}} \ldots \xrightarrow{\alpha_k t_{a_k}} \mathbf{m}_k = \mathbf{m}$ *with* $t_i \in T$ *and* $\alpha_i \in \mathbb{R}_{\geq 0}\}$.
- $\text{lim-RS}(\mathcal{N}, \mathbf{m_0}) = \{\mathbf{m} \in (\mathbb{R}_{\geq 0})^{|P|} |$ *a sequence of reachable markings* $\{\mathbf{m}_i\}_{i \geq 1}$ *exists verifying* $\mathbf{m_0} \xrightarrow{\sigma_1} \mathbf{m}_1 \xrightarrow{\sigma_2} \mathbf{m}_2 \cdots \mathbf{m}_{i-1} \xrightarrow{\sigma_i} \mathbf{m}_i \cdots$ *and* $\lim_{i \to \infty} \mathbf{m}_i = \mathbf{m}\}$.

The set of reachable (and lim-reachable) markings in continuous PNs satisfies some properties that do not hold for discrete nets. For example, the reachability set (and the lim-reachability set) of a continuous system is a convex set [9].

Many basic properties of discrete PNs can be extended to continuous PNs.

Definition 2. *Let $\langle \mathcal{N}, \mathbf{m_0} \rangle$ be a continuous system.*

- $\langle \mathcal{N}, \mathbf{m_0} \rangle$ *is* (lim-) *deadlock-free iff for every* $\mathbf{m} \in$ (lim-) $\mathrm{RS}(\mathcal{N}, \mathbf{m_0})$ *there exists* $t \in T$ *such that* $\mathrm{enab}(t, \mathbf{m}) > 0$.
- $\langle \mathcal{N}, \mathbf{m_0} \rangle$ *is* (lim-) *live iff for every* $\mathbf{m} \in$ (lim-) $\mathrm{RS}(\mathcal{N}, \mathbf{m_0})$ *and for every* $t \in T$ *there exist* $\mathbf{m}' \in$ (lim-) $\mathrm{RS}(\mathcal{N}, \mathbf{m})$ *such that* $\mathrm{enab}(t, \mathbf{m}') > 0$.
- $\langle \mathcal{N}, \mathbf{m_0} \rangle$ *is* (lim-) *reversible iff for every* $\mathbf{m} \in$ (lim-) $\mathrm{RS}(\mathcal{N}, \mathbf{m_0})$ *then* $\mathbf{m_0} \in$ (lim-) $\mathrm{RS}(\mathcal{N}, \mathbf{m})$.

Since continuous PN are a relaxation of discrete PN, for those properties based on universal (existential) quantifiers the continuous PN will provide sufficient (necessary) conditions. For example, if the continuous PN is bounded, so will be the discrete PN. For a marking to be reachable in the discrete model, reachability in the continuous one must be guaranteed. However, for those properties formulated interleaving universal and existential quantifiers the analysis of the continuous PN may not provide information about the behavior of its discrete counterpart. For example, liveness of the continuous PN is neither necessary nor sufficient for liveness of the discrete model [9].

Besides the situation in which the properties of the discrete and the continuous net are not related, it also happens that some properties of discrete PN cannot be observed in continuous systems, as mutex relationships. Moreover, the distinction between two properties may be lost in the continuous model. For example, under broad conditions, lim-liveness and lim-reversibility are equivalent.

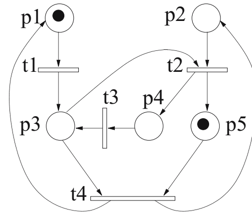


Fig. 1. This system is not reversible as discrete, or as continuous with finite number of firings, but it is lim-reversible

Indeed, reformulating Theorem 21 in [8], the following result can be proved:

Theorem 1. $\langle \mathcal{N}, \mathbf{m_0} \rangle$ *is consistent and* lim-*live iff it is* lim-*reversible and every transition is fireable at least once.*

However, liveness and consistency are not sufficient conditions for reversibility in discrete systems, and neither are for continuous net systems if finite firing sequences are considered. For example, the system in Fig. 1 is consistent and live as discrete, however once t_1 has fired it is impossible to get back to the initial marking. In the continuous net system, to go back to the initial marking from $\mathbf{m} = [0, 0, 1, 0, 1]$, an infinite sequence $\frac{1}{2}t_4 \frac{1}{2}t_2, \frac{1}{2}t_3, \frac{1}{4}t_4, \frac{1}{4}t_2, \frac{1}{4}t_3, \ldots \frac{1}{2^k}t_4, \frac{1}{2^k}t_2, \frac{1}{2^k}t_3, \ldots$ has to be fired.

2.2 Timed Continuous Petri Nets

A simple and interesting way to introduce time in discrete PNs is to assume that all the transitions are timed with exponential probability distribution functions. For the timing interpretation of continuous PNs we will use a first order (or deterministic) approximation of the discrete case, assuming that the delays associated to the firing of transitions can be approximated by their mean values. These mean delays will be assumed to be positive, i.e., immediate transitions are not allowed.

Definition 3. *A* Timed Continuous Petri Net (TCPN) *is a continuous PN together with a vector* $\boldsymbol{\lambda} \in \mathbb{R}_{>0}^{|T|}$.

A TCPN with an initial marking $\langle \mathcal{N}, \boldsymbol{\lambda}, \mathbf{m_0} \rangle$ *will be denoted a TCPN system.*

Since it is an interpretation of the autonomous net, the evolution of a TCPN has to fulfill the state equation: $\mathbf{m}(\tau) = \mathbf{m}(0) + \mathbf{C} \cdot \boldsymbol{\sigma}(\tau)$, where $\mathbf{m}$ and σ now depend on τ, the actual time. Deriving, $\dot{\mathbf{m}}(\tau) = \mathbf{C} \cdot \mathbf{f}(\tau)$, where $\mathbf{f}(\mathbf{m}) = \dot{\boldsymbol{\sigma}}(\tau)$ is the flow obtained by firing the transitions. Different semantics have been used to define this flow, the two most important being *infinite server* (or *variable speed*) and *finite server* (or *constant speed*) [1,11]. Here infinite server semantics will be considered.

Like in purely markovian discrete net models, under *infinite server semantics*, the flow through a timed transition t is the product of the speed, $\boldsymbol{\lambda}[t]$, and $\text{enab}(t, \mathbf{m})$, the instantaneous enabling of the transition, i.e.,

$$\mathbf{f}(\mathbf{m})[t] = \boldsymbol{\lambda}[t] \cdot \text{enab}(t, \mathbf{m}) = \boldsymbol{\lambda}[t] \cdot \min_{p \in {}^\bullet t} \{\mathbf{m}[p] / \mathbf{Pre}[p, t]\}.$$

For the flow to be well defined, every transition must have at least one input place, hence in the following we will assume $\forall t \in T, |{}^\bullet t| \geq 1$.

Notice that the flow is a piecewise linear function, i.e., the continuous timed model is *technically hybrid*. The change of behavior happens when in a synchronization the place representing the minimum changes. Hence, the switching among the linear systems is given by an internal event. A system without synchronizations (i.e., for every t $|{}^\bullet t| = 1$) would be linear.

Let us introduce the concept of *configurations*: a configuration assigns to a transition one place that for some markings will control its firing rate. Thus the number of configurations is $\Pi_{t \in T} |{}^\bullet t|$. The reachability space can be divided into *regions* according to the configurations. These regions are polyhedrons, and are disjoint, except on the borders. Inside each polyhedron, the evolution of the system is defined by a linear differential equation.

Different kind of behaviors can be modeled with TCPN. In the following we will show some examples to illustrate their modelling power.

Steady-state with finite number of switches. Let us consider the continuous relaxed view of the PN system in Fig. 2 with $\boldsymbol{\lambda} = [1, 2, 1, 1, 0.5]$. The flows through transitions are given by:

$$\begin{aligned}
\dot{\mathbf{m}}[p_1] &= \mathbf{f}[t_3] - \mathbf{f}[t_1] = \mathbf{m}[p_3] - \mathbf{m}[p_1] \\
\dot{\mathbf{m}}[p_2] &= \mathbf{f}[t_1] - \mathbf{f}[t_2] = \mathbf{m}[p_1] - 2 \cdot \min(\mathbf{m}[p_2], \mathbf{m}[p_6]) \\
\dot{\mathbf{m}}[p_3] &= \mathbf{f}[t_2] - \mathbf{f}[t_3] = 2 \cdot \min(\mathbf{m}[p_2], \mathbf{m}[p_6]) - \mathbf{m}[p_3] \\
\dot{\mathbf{m}}[p_4] &= \mathbf{f}[t_5] - \mathbf{f}[t_4] = 0.5 \cdot \mathbf{m}[p_5] - \min(\mathbf{m}[p_4], \mathbf{m}[p_6]) \\
\dot{\mathbf{m}}[p_5] &= \mathbf{f}[t_4] - \mathbf{f}[t_5] = \min(\mathbf{m}[p_4], \mathbf{m}[p_6]) - 0.5 \cdot \mathbf{m}[p_5] \\
\dot{\mathbf{m}}[p_6] &= \mathbf{f}[t_3] + \mathbf{f}[t_5] - \mathbf{f}[t_2] - \mathbf{f}[t_4] = \\
&\qquad = \mathbf{m}[p_3] + 0.5 \cdot \mathbf{m}[p_5] - 2 \cdot \min(\mathbf{m}[p_2], \mathbf{m}[p_6]) - \min(\mathbf{m}[p_4], \mathbf{m}[p_6])
\end{aligned}$$

With the initial marking shown in Fig. 2, there is one switch when $\mathbf{m}[p_4] = \mathbf{m}[p_6]$, and then the system approaches its steady-state and never switches again.

Steady state but infinite switches. Looking at the previous example, one may wonder whether the asymptotical behavior of a net approaching a steady-state can be reduced to the behavior of the set of configurations. This would be the case if switches between configurations occurred only a finite number of times, as happens in the previous example. However, it can be proved that the net system of Fig. 3 approaches to a steady state while infinitely switching between configurations. This net has two regions, namely $\{\mathbf{m} \mid \mathbf{m}[x_1] \geq \mathbf{m}[x_2]\}$ and $\{\mathbf{m} \mid \mathbf{m}[x_1] \leq \mathbf{m}[x_2]\}$. The steady state belongs to both regions, i.e., it is a border point, and the system keeps switching between them.

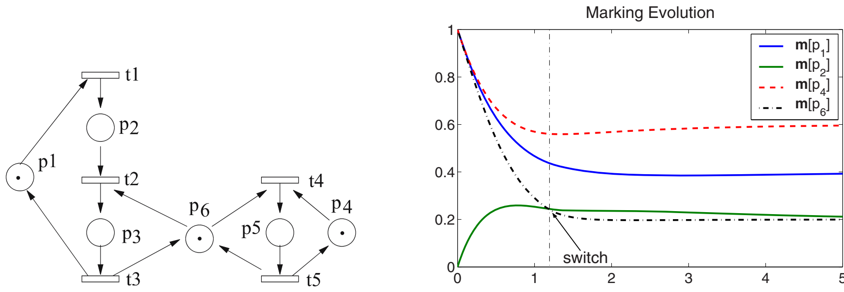


Fig. 2. A TCPN and its marking evolution with $\boldsymbol{\lambda} = [1, 2, 1, 1, 0.5]$. The system switches only once, and approaches to a steady-state.

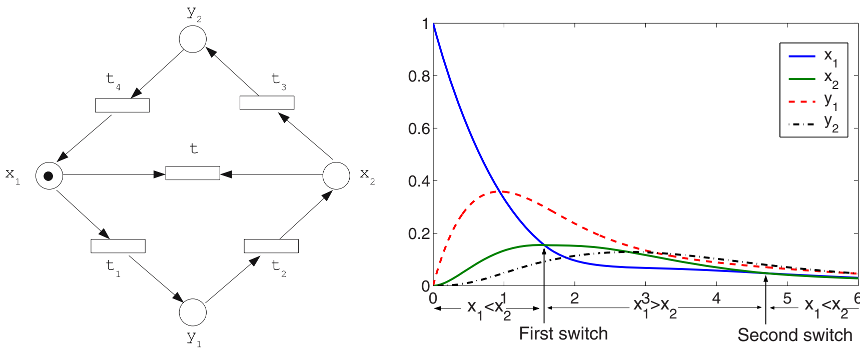


Fig. 3. A TCPN and its marking evolution with $\boldsymbol{\lambda} = [1, 1, 1, 1, 1]$. The system switches an infinite number of times while approaching to a steady-state.

Periodic behavior without switches. A periodic behavior in which the configuration does not change can also be modeled with TCPN. The ordinary differential equation corresponding to the net system of Fig. 4 is:

$$\begin{aligned}
\dot{\mathbf{m}}[x_1] &= 2a\omega \cdot \min\left\{\frac{\mathbf{m}[x_2]}{2a}, \mathbf{m}[y_1]\right\} - a\omega \cdot \min\left\{\mathbf{m}[x_1], \frac{\mathbf{m}[pk]}{a}\right\} \\
\dot{\mathbf{m}}[x_2] &= a\omega \cdot \min\left\{\mathbf{m}[y_2], \frac{\mathbf{m}[pk]}{a}\right\} - 2a\omega \cdot \min\left\{\mathbf{m}[x_2], \frac{\mathbf{m}[x_1]}{2a}\right\} \\
\dot{\mathbf{m}}[y_1] &= a\omega \cdot \min\left\{\mathbf{m}[x_1], \frac{\mathbf{m}[pk]}{a}\right\} - 2a\omega \cdot \min\left\{\frac{\mathbf{m}[x_2]}{2a}, \mathbf{m}[y_1]\right\} \\
\dot{\mathbf{m}}[y_2] &= 2a\omega \cdot \min\left\{\mathbf{m}[x_2], \frac{\mathbf{m}[x_1]}{2a}\right\} - a\omega \cdot \min\left\{\mathbf{m}[y_2], \frac{\mathbf{m}[pk]}{a}\right\}
\end{aligned} \tag{1}$$

This net has sixteen configurations. However, for $1 \leq a \leq b \leq 2a - 1$, the system never switches and always remains in the configuration defined by $\mathbf{m_0}$.

Infinite repetitive switches (not Approaching a Steady State). The behavior of the system represented by the model in Fig. 5 shows that infinite switches without approaching to a final marking can also be modeled with

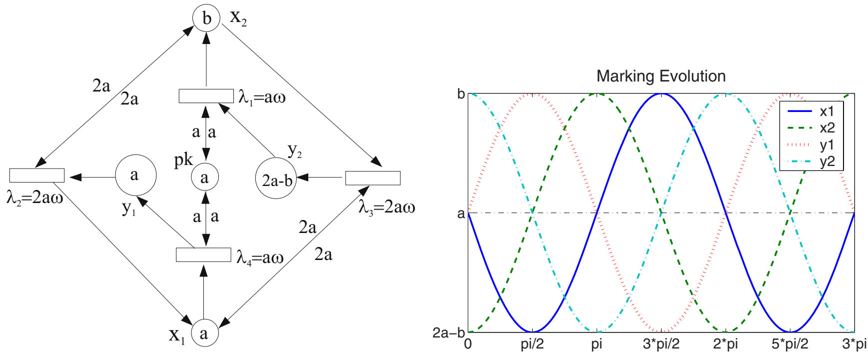


Fig. 4. A TCPN and its marking evolution with $\boldsymbol{\lambda} = [a\omega, 2a\omega, 2a\omega, a\omega]$. The system has a periodic behavior but always remains inside the same configuration.

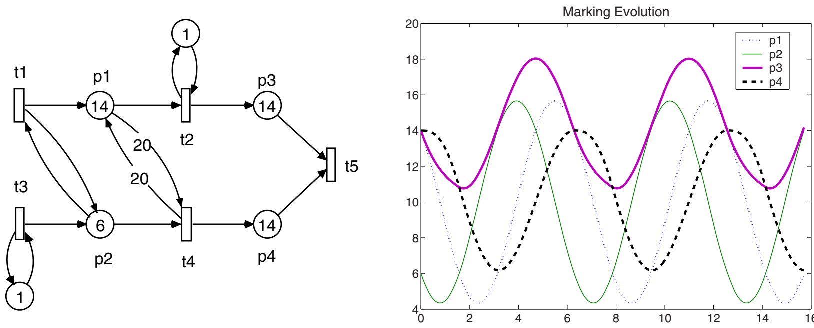


Fig. 5. A TCPN and its marking evolution with $\boldsymbol{\lambda} = [1, 10, 10, 20, 1]$. The system switches indefinitely, and does not approach to a limit marking.

TCPN. In this net there are two configurations that are commuting indefinitely (observe that sometimes $\mathbf{m}[p_3] < \mathbf{m}[p_4]$, and sometimes $\mathbf{m}[p_4] < \mathbf{m}[p_3]$).

3 Decidability of Basic Properties of Autonomous Continuous Petri Nets

The idea under continuization is that it leads to "easier to analyze" models. For that, we need to ensure that properties can be analyzed at least. This section will be devoted to proving that properties like (lim-)reachability, (lim-)liveness and (lim-)deadlock-freeness are decidable.

In [8], a characterization of reachability and lim-reachability is presented.

Theorem 2. *Let $\langle \mathcal{N}, \mathbf{m_0} \rangle$ be an ACPN system. Marking $\mathbf{m} \in \mathrm{RS}(\mathcal{N}, \mathbf{m_0})$ iff*

1. $\mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \boldsymbol{\sigma} \geq \mathbf{0},\ \boldsymbol{\sigma} \geq \mathbf{0}$
2. $\|\boldsymbol{\sigma}\| \in FS(\mathcal{N}, \mathbf{m_0})$
3. *no trap in* $P \setminus \|\mathbf{m}\|$ *intersects with* $\|\mathbf{m_0}\| \cup \|\boldsymbol{\sigma}\|^{\bullet}$

where $FS(\mathcal{N}, \mathbf{m_0})$ *is the set of the supports of sequences fireable from* $\mathbf{m_0}$*, which is a finite set that can be effectively constructed.*

A marking $\mathbf{m} \in \text{lim-RS}(\mathcal{N}, \mathbf{m_0})$ *iff it verifies conditions (1) and (2).*

For the computation of $FS(\mathcal{N}, \mathbf{m_0})$, first add all the combinations of transitions that are enabled at $\mathbf{m_0}$. Then, take one of these sets and fire all the transitions, but in an amount smaller than the enabling degree. This will possibly enable other transitions, so new sets are added to FS. Repeat the procedure till all the sets in FS have been checked.

The only difference between reachability and lim-reachability is on traps, which can be emptied in the limit, but not with a finite sequence. In [8], decidability of reachability is proved. This result can be generalized to lim-reachability.

Corollary 1. *Reachability and* lim-*reachability are decidable for ACPN systems.*

The characterization of (lim-)reachability allows to address also the problem of (lim-)deadlock freeness.

Theorem 3. *For ACPN systems deadlock-freeness and* lim-*deadlock-freeness are decidable.*

Proof. Let us see that checking deadlock-freeness can be reduced to solving a set of linear programming problems.

Let $DP = \{DP_i\}_{i \in I}$ be the set of all the sets of places that have at least one input place per transition. Hence applying Theorem 2, $\mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \boldsymbol{\sigma}$ is a deadlock iff there exist $DP_i \in DP$ and $FS_j \in FS$ such that

$$\mathbf{m_0}[DP_i] + \mathbf{C}[DP_i, FS_j] \cdot \boldsymbol{\sigma}[FS_j] = \mathbf{0} \tag{2}$$

$$\mathbf{m_0}[P \setminus DP_i] + \mathbf{C}[P \setminus DP_i, FS_j] \cdot \boldsymbol{\sigma}[FS_j] > \mathbf{0} \tag{3}$$

$$\boldsymbol{\sigma}[FS_j] > \mathbf{0} \tag{4}$$

$$\text{For every trap } \Theta \subseteq DP_i, (\|\mathbf{m_0}\| \cup FS_j{}^{\bullet}) \cap \Theta = \emptyset \tag{5}$$

Applying the characterization of traps in [12], (5) is equivalent to checking whether the solution of the following linear programming problem is zero,

$$\begin{aligned} \text{maximize } & \varepsilon_{i,j} \\ \text{subject to: } & \mathbf{y} \cdot \mathbf{C}_\Theta \geq \mathbf{0} \\ & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{y}[P \setminus DP_i] = \mathbf{0} \\ & \sum_{p \in \|\mathbf{m_0}\| \cup FS_j{}^{\bullet}} \mathbf{y}[p] \geq \varepsilon_{i,j} \end{aligned} \tag{6}$$

where C_Θ is the token flow matrix of the net $\mathcal{N}_\Theta = \langle P, T, \mathbf{Pre}, \mathbf{Post}_\Theta \rangle$ with $\mathbf{Post}_\Theta[p,t] = 0$ iff $\mathbf{Post}[p,t] = 0$, and $\mathbf{Post}_\Theta[p,t] \geq \sum_{p' \in {}^{\bullet}t} \mathbf{Pre}[p',t]$.

Equations (2), (3) and (4) are equivalent to:

$$\begin{aligned} &\text{maximize } \delta_{i,j} \\ &\text{subject to: } \mathbf{m_0}[DP_i] + \mathbf{C}[DP_i, FS_j] \cdot \boldsymbol{\sigma}[FS_j] = \mathbf{0} \\ &\qquad \mathbf{m_0}[P \setminus DP_i] + \mathbf{C}[P \setminus DP_i, FS_j] \cdot \boldsymbol{\sigma}[FS_j] \geq \delta_{i,j} \cdot \mathbf{1} \\ &\qquad \boldsymbol{\sigma}[FS_j] \geq \delta_{i,j} \cdot \mathbf{1} \end{aligned} \tag{7}$$

Therefore, to prove that the system is deadlock-free, check that for each $DP_i \in DP$ and each $FS_j \in FS$ either (6) has a positive solution or (7) does not have a positive solution.

Regarding lim-deadlock-freeness, the only difference is that there is no restriction with respect to the traps, so clearly it is also decidable. □

Theorem 4. *Liveness and* lim-*liveness are decidable for ACPN systems.*

Proof. Let us study liveness first. For continuous PNs, a transition t is not fireable for any successor of $\mathbf{m}$ iff there exists an empty siphon in $\mathbf{m}$ that contains a place $p \in {}^{\bullet}t$ (Lemma 11 in [8]). Hence, the system is non live iff there exist $FS_j \in FS$ and a siphon Σ such that

$$\mathbf{m} = \mathbf{m_0} + \mathbf{C}[P, FS_j] \cdot \boldsymbol{\sigma}[FS_j] \tag{8}$$

$$\mathbf{m} \geq \mathbf{0} \tag{9}$$

$$\boldsymbol{\sigma}[FS_j] > \mathbf{0} \tag{10}$$

$$\mathbf{m}[\Sigma] = 0 \tag{11}$$

$$\text{For any trap } \Theta \text{ such that } (\|\mathbf{m_0}\| \cup FS_j{}^{\bullet}) \cap \Theta = \emptyset, \exists p \in \Theta \text{ with } \mathbf{m}[p] = 0 \tag{12}$$

Let $\{\Sigma_i\}_{i \in I}$ be the set of minimal siphons (i.e., siphons that are not contained in other siphons), and let $\{\Theta_k\}_{k \in K}$ be the set of traps. Then for each $FS_j \in FS$, each trap Θ_k verifying $(\|\mathbf{m_0}\| \cup FS_j{}^{\bullet}) \cap \Theta_k \neq \emptyset$, and each siphon Σ_i, check whether the following linear programming problem has a positive solution:

$$\begin{aligned} &\text{maximize } \varepsilon_{i,j} \\ &\text{subject to: } \mathbf{m} = \mathbf{m_0} + \mathbf{C}[P, FS_j] \cdot \boldsymbol{\sigma}[FS_j] \\ &\qquad \mathbf{m} \geq \mathbf{0} \\ &\qquad \mathbf{m}[\Sigma_i] = 0 \\ &\qquad \boldsymbol{\sigma}[FS_j] \geq \varepsilon_{i,j} \cdot \mathbf{1} \\ &\qquad \mathbf{m}[p] = 0 \end{aligned}$$

Regarding lim-liveness, notice that in fact it is equivalent that a transition is not fireable from any reachable marking or from any lim-reachable marking. Then, the lim-liveness problem is analogous, changing the reachability condition, which amounts to forgetting the traps, and removing the last equation ($\mathbf{m}[p] = 0$) in each one of the linear programming problems. □

Combining this result with Theorem 1, the following corollary is obtained.

Corollary 2. lim-*reversibility is decidable in ACPN systems in which all the transitions are fireable.*

4 Timed Continuous Petri Nets and Timed Differentiable Petri Nets

The structure and equations of TCPN were somehow inherited from those of discrete Petri nets. Petri nets are based on a production/consumption logic, and it is the flow of material that is mainly represented. However, these material flow channels can also be used to simulate control flows, and self-loop structures appear. This kind of structures can be used in continuous PNs to "force" behaviors. For example, in the net in Fig. 4, places y_1 and y_2 never define the flow of their output transitions (i.e., never belong to the active configuration), and this has been achieved by a tuning in the self-loop arcs that connect the other places to the transitions. That means that for example t_2 has two input places, but one of them is used to control the speed of the transition, while the other is the one that "suffers" the changes.

Moreover, the information that appears in the structure of the net is redundant. Assume a place-transition-place-transition subnet, and let us denote them as $p_1 - t_1 - p_2 - t_2$. The marking evolution of p_2 due to the output flow does not depend on the weight of the arc (p_2, t_2), and its input flow really depends on the quotient of the weights (p_1, t_1) and (t_1, p_2).

Hence, the notation of TCPN "looks cumbersome", and the evolution rules seem convoluted and counterintuitive in some cases. Trying to clarify the behavior, in [6], a different way of introducing time in a PN structure was presented, in which two different kinds of arcs are used: one to model the *control*, and the other to model the *marking evolution*. This is similar to what is done in Forrester diagrams, in which *control* and *material* flows are kept separate [5].

Definition 4. *A Timed Differentiable Petri Net (TDPN)* $\mathcal{D} = \langle P, T, \mathbf{C}, \mathbf{W} \rangle$ *is defined by:*

– $\langle P, T, \mathbf{C} \rangle$*, a pure PN (thus* $\mathbf{C}$ *is the incidence matrix),*
– $\mathbf{W}$*, the speed control matrix: a mapping from* $P \times T$ *to* $\mathbb{R}_{\geq 0}$ *such that*
  1. $\forall t \in T, \exists p \in P, \mathbf{W}(p,t) > 0$
  2. $\forall t \in T, \forall p \in P, \mathbf{C}(p,t) < 0 \Rightarrow \mathbf{W}(p,t) > 0$

A TDPN with an initial marking $\langle \mathcal{D}, \mathbf{m_0} \rangle$ *will be denoted a TDPN system.*

The first requirement about $\mathbf{W}$ ensures that the firing rate of any transition is defined, whereas the second one ensures that the marking remains non negative. Notice that these weights are defined as real numbers, while in TCPN all the arc weights are natural numbers.

We are now in position to give semantics to TDPNs.

Definition 5. *Let* $\mathcal{D}$ *be a TDPN. A trajectory is a continuously differentiable mapping* $\mathbf{m}$ *from time (i.e.,* $\mathbb{R}_{\geq 0}$*) to the set of markings (i.e.,* $(\mathbb{R}_{\geq 0})^P$*) which satisfies the following differential equation system:*

$$\begin{aligned} \dot{\mathbf{m}} &= \mathbf{C} \cdot \mathbf{f}(\mathbf{m}) \\ \mathbf{f}(\mathbf{m})[t] &= \min(\mathbf{W}(p,t) \cdot \mathbf{m}[p] \mid \mathbf{W}(p,t) > 0) \end{aligned} \tag{13}$$

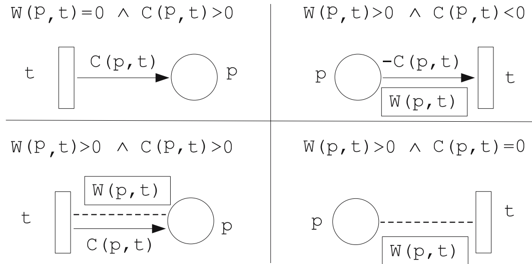


Fig. 6. Graphical notations: it will be seen that only the two on the top are needed

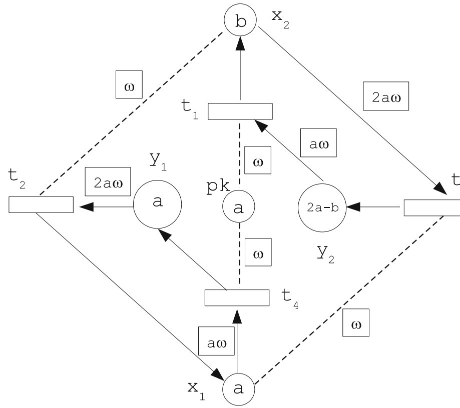


Fig. 7. A periodic TDPN

Graphical notations. We extend the graphical notations of PN to take into account $\mathbf{W}$. These arcs are not oriented, since they are always defined as preconditions of transitions the orientation is implicit. Like in Forrester diagrams, to help distinguishing the $\mathbf{W}$ arcs from the **Pre** and **Post** arcs, $\mathbf{W}$ arcs will be drawn with dotted lines. To distinguish between the labels, $\mathbf{W}(p,t)$ will be drawn inside a box. There are four possible patterns illustrated in Fig. 6. When $\mathbf{W}(p,t) = 0 \wedge \mathbf{C}(p,t) > 0$, place p receives tokens from t and does not control its firing rate. When $\mathbf{W}(p,t) > 0 \wedge \mathbf{C}(p,t) < 0$, place p provides tokens to t. So it *must control* its firing rate. Hence, the non oriented arc between p and t is redundant, and we will not draw it and represent only an oriented arc from p to t both labelled by $-\mathbf{C}(p,t)$ and $\mathbf{W}(p,t)$. When $\mathbf{W}(p,t) > 0 \wedge \mathbf{C}(p,t) > 0$, place p receives tokens from t and controls its firing rate. There is both an oriented arc from t to p and a non oriented arc between p and t with their corresponding labels. When $\mathbf{W}(p,t) > 0 \wedge \mathbf{C}(p,t) = 0$, place p controls the firing rate of t and t does not modify the marking of p, so there is a non oriented arc between p and t. As usual, we omit labels when equal to 1.

As an example, the equations in (1) correspond to the TDPN in Fig. 7.

The Petri net structure of TCPN is similar to that of TDPN. Moreover, the minimum operator appears related to the marking evolution. Hence, it would not be surprising that both models are equivalent, as long as the elements of the speed control matrix are in the rationals. To prove that, observe first that fractions in the **Pre** and **Post** matrices do not pose a problem, since they can be easily avoided multiplying the columns of the **Pre** and **Post** matrices by the right number (this does not change the dynamics of the system).

Proposition 1. *For any TCPN, a TDPN with the same number of places and transitions exists which has the timed evolution, and vice versa (as long as the weights of the TDPN are rational numbers).*

Proof. Let $\mathcal{D} = \langle P, T, \mathbf{C}, \mathbf{W} \rangle$ be a TDPN, and let us construct a TCPN $\mathcal{S} = \langle P, T, \mathbf{Pre}', \mathbf{Post}', \boldsymbol{\lambda} \rangle$ with the same differential equations. For each $t \in T$, define $\mathbf{W}^*(t) \geq \max(-\mathbf{C}(p,t) \cdot \mathbf{W}(p,t) \mid \mathbf{C}(p,t) < 0)$ (i.e., $\mathbf{W}^*(t)$ can be defined as any value that fulfills this inequality). By definition it is greater than zero. For each $p \in P$ and $t \in T$, define

- $\mathbf{Pre}'(p,t) = \mathbf{W}^*(t)/\mathbf{W}(p,t)$ if $\mathbf{W}(p,t) \neq 0$, and $\mathbf{Pre}'(p,t) = 0$ otherwise.
- $\mathbf{Post}'(p,t) = \mathbf{C}(p,t) + \mathbf{W}^*(t)/\mathbf{W}(p,t)$ if $\mathbf{W}(p,t) \neq 0$, and $\mathbf{Post}'(p,t) = \mathbf{C}(p,t)$ otherwise.
- $\boldsymbol{\lambda}(t) = \mathbf{W}^*(t)$

Let now $\mathcal{S} = \langle P, T, \mathbf{Pre}, \mathbf{Post}, \boldsymbol{\lambda} \rangle$ be a TCPN, and let us construct a TDPN $\mathcal{D} = \langle P, T, \mathbf{C}', \mathbf{W} \rangle$ with the same differential equations. For each $p \in P$ and $t \in T$, define

- $\mathbf{C}'(p,t) = \mathbf{Post}(p,t) - \mathbf{Pre}(p,t)$
- $\mathbf{W}(p,t) = \boldsymbol{\lambda}(t)/\mathbf{Pre}(p,t)$ if $p \in {}^\bullet t$, and 0 otherwise. □

As an example, the TCPN in Fig. 4 and the TDPN in Fig. 7 are obtained one from the other with the previous procedure.

The patterns appearing at the bottom of Fig. 6 represent situations in which a place acts as a control place of a transition for which it is not an input place (in the sense that the transition is not removing tokens from the place). That is, they represent non-consuming control arcs. However, these patterns can be simulated with the two ones on top.

Proposition 2. *Let $\mathcal{D} = \langle P, T, \mathbf{C}, \mathbf{W} \rangle$ be a TDPN. Then another TDPN $\mathcal{D}' = \langle P', T', \mathbf{C}', \mathbf{W}' \rangle$ can be defined, using only the two patterns in the top of Fig. 6, with the same timed evolution as $\mathcal{D}$ but for some duplicated places. Moreover, the transformation is linear in size, since it at most doubles the number of places and transitions.*

Proof. Define the new TDPN as follows:

- $P' = \{p^-, p^+ \mid p \in P\}$
- $T' = \{t^-, t^+ \mid t \in T\}$

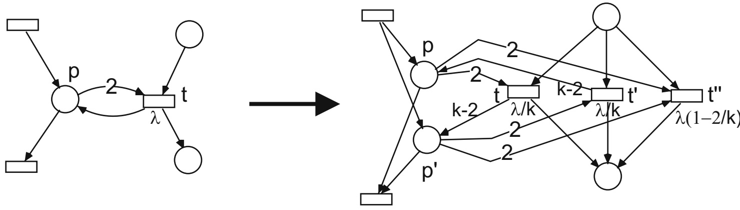


Fig. 8. TCPN can always be transformed into pure nets

- To define $\mathbf{C}'$, distinguish two cases:
  - If $\mathbf{C}(p,t) < 0 \vee \mathbf{W}(p,t) = 0$ then
    $\mathbf{C}'(p^-,t^-) = \mathbf{C}'(p^+,t^+) = \mathbf{C}(p,t)$ and $\mathbf{C}'(p^-,t^+) = \mathbf{C}'(p^+,t^-) = 0$
  - If $\mathbf{C}(p,t) \geq 0 \wedge \mathbf{W}(p,t) > 0$ then
    $\mathbf{C}'(p^-,t^-) = \mathbf{C}'(p^+,t^+) = -1$ and $\mathbf{C}'(p^-,t^+) = \mathbf{C}'(p^+,t^-) = \mathbf{C}(p,t)+1$
- $\mathbf{W}'(p^-,t^-) = \mathbf{W}'(p^+,t^+) = \mathbf{W}(p,t)$ and $\mathbf{W}'(p^-,t^+) = \mathbf{W}'(p^+,t^-) = 0$ □

The transformation that has been proposed doubles the number of places and transitions. In practice, sometimes it is not necessary to duplicate all of them. A similar transformation can also be applied to *remove self-loop arcs* in TCPN. Let us see the idea in the case of one self-loop (see Fig. 8). The transformation replaces the self-loop with two places and three transitions. These two places will have the same marking the original place had, and the flow of the original transition is now split into the flow of these three transitions. Any other input/output place of the transition is input/output of all the transitions. Any input/output transition of the place is now input/output of all the places. The rates of these new transitions are defined so that the sum of their flows is equal to the flow of the original transition.

Again, this is a linear transformation. The procedure can be easily generalized to the case in which several places are connected with self-loops to the same transition (just duplicate each self-loop place and split the transition in three), or when one place is engaged in several self-loops (duplicate the self-loop place and split each transition in three).

Proposition 3. *For any TCPN a pure TCPN can be defined which has the same timed evolution, but for some duplicated places. Moreover, the transformation is linear in size, since it at most doubles the number of places and triples the number of transitions.*

The results have been summarized in the schema in Fig. 9.

Theorem 5. *The expressive power of TDPN, TDPN constrained to the use of the two basic constructions (the two on the top in Fig. 6), TCPN and pure TCPN are identical. Moreover, the transformations range from keeping the places and transitions to a linear increase in size (see the diagram in Fig. 9).*

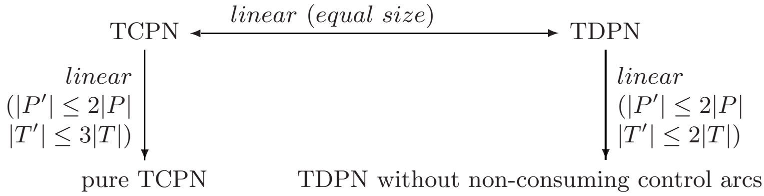


Fig. 9. Relationships between TDPN and TCPN and their versions without self-loops or non-consuming control arcs

5 Decidability Issues on Timed Continuous Petri Nets

In [6], it has been proved that TDPN can simulate two (non negative integer) counter machines (equivalent to Turing machines [7]). More precisely, different simulations of two counter machines have been defined, in order to fulfill opposite requirements like *robustness* (allowing some perturbation of the simulation) and *boundedness* of the simulating net system (the simulation of an infinite-state system cannot be *simultaneously robust and bounded*). Furthermore these simulations can be performed by a net with a constant number of places, i.e., the dimension of the associated ordinary differential equation (ODE) is constant.

Theorem 6. *[6] Given a two counter machine $\mathcal{M}$, one can build a TDPN $\mathcal{D}$, with a constant number of places, whose size is linear w.r.t. the machine, with one of these two properties:*

- *its associated ODE has dimension* 6 *and $\mathcal{D}$ robustly simulates $\mathcal{M}$.*
- *it is a bounded net system, its associated ODE has dimension* 14*, and $\mathcal{D}$ simulates $\mathcal{M}$.*

Applying the equivalences developed in the previous section, the same result can be stated for TCPN with respect to the robust simulation. Regarding the bounded one, the proof in [6] requires the use of non rational numbers, and we have not been able to extend this result to TCPN up to now. As in [6], these results can be used to obtain undecidability results.

Corollary 3. *Let $\langle \mathcal{N}, \boldsymbol{\lambda}, \mathbf{m_0} \rangle$ be a TCPN whose associated ODE has dimension at least* 6*, $\mathbf{m}_1$ a marking, p a place and $k \in \mathbb{N}$. The problem whether there is a τ such that the trajectory starting at $\mathbf{m_0}$ fulfills*

- $\mathbf{m}(\tau)(p) = k$ *is undecidable.*
- $\mathbf{m}(\tau)(p) \geq k$ *is undecidable.*
- $\mathbf{m}(\tau) \geq \mathbf{m}_1$ *is undecidable.*

Corollary 4. *Let $\langle \mathcal{N}, \boldsymbol{\lambda}, \mathbf{m_0} \rangle$ be a TCPN whose associated ODE has dimension at least* 8*. Then the problem whether the trajectory $\mathbf{m}$ starting at $\mathbf{m_0}$ is such that $\lim_{\tau \to \infty} \mathbf{m}(\tau)$ exists, is undecidable.*

6 Conclusions

Regarding autonomous (i.e., untimed) nets, it has been proved that reachability, deadlock-freeness and liveness are decidable in continuous nets, both if firing sequences are limited to be finite or if infinitely long sequences are allowed (Corollary 1, Theorem 3 and Theorem 4). Therefore, continuous and discrete autonomous nets are similar in decidability terms.

For timed nets, it has been proved that continuous pure nets are able to model Turing machines (Theorem 6). This means that many properties are undecidable (Corollary 3, Corollary 4). Timed continuous Petri nets can seen as a fluidified version of discrete Petri nets in which time is associated to transitions using a probability distribution function with positive mean value. Therefore, the fluidification has not increased decidability in the timed model.

However, fluidification *does* simplify some problems, and in particular some inverse problems. For example, the computation of an initial marking that maximizes a linear function of the throughput, the steady-state marking and the initial marking is easier for some nets in the continuous (timed) model than in the discrete one. More work is needed to characterize classes of problems that fluidification makes simpler, and why this happens.

References

1. Alla, H., David, R.: Continuous and hybrid Petri nets. Journal of Circuits, Systems, and Computers 8(1), 159–188 (1998)
2. Balduzzi, F., Di Febraro, A., Giua, A., Seatzu, C.: Decidability results in first-order hybrid Petri nets. Discrete Event Dynamic Systems 11(1–2), 41–57 (2001)
3. Ciardo, G.: Toward a definition of modeling power for stochastic Petri net models. In: Procs. of the Int. Workshop on PNPM, IEEE-Computer Society Press, Los Alamitos (1987)
4. Esparza, J., Nielsen, M.: Decidability issues for Petri nets - a survey. Bulletin of the EATCS 52, 245–262 (1994)
5. Forrester, J.W.: Principles of Systems. Productivity Press (1968)
6. Haddad, S., Recalde, L., Silva, M.: On the computational power of Timed Differentiable Petri nets. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 230–244. Springer, Heidelberg (2006)
7. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Addison-Wesley, Reading (1969)
8. Júlvez, J., Recalde, L., Silva, M.: On reachability in autonomous continuous Petri net systems. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 221–240. Springer, Heidelberg (2003)
9. Recalde, L., Teruel, E., Silva, M.: Autonomous continuous P/T systems. In: Donatelli, S., Kleijn, J.H.C.M. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 107–126. Springer, Heidelberg (1999)
10. Silva, M.: Introducing Petri nets. In: Practice of Petri Nets in Manufacturing, pp. 1–62. Chapman & Hall (1993)

11. Silva, M., Recalde, L.: Continuization of timed Petri nets: From performance evaluation to observation and control. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 26–46. Springer, Heidelberg (2005)
12. Silva, M., Teruel, E., Colom, J.M.: Linear algebraic and linear programming techniques for the analysis of net systems. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 309–373. Springer, Heidelberg (1998)

Quantifying the Discord: Order Discrepancies in Message Sequence Charts*

Edith Elkind[1], Blaise Genest[2], Doron Peled[3], and Paola Spoletini[4]

[1] School of Electronics and Computer Science University of Southampton, UK
[2] CNRS/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[3] Department of Computer Science, University of Warwick
Coventry CV4 7AL, United Kingdom
and Department of Computer Science, Bar Ilan University,
Ramat Gan 52900, Israel
[4] Dipartimento di Elettronica e Informazione, Politecnico di Milano
via Ponzio 34/5 - 20133, Milano, Italy

Abstract. Message Sequence Charts (MSCs) and High-level Message Sequence Charts (HMSC) are formalisms used to describe scenarios of message passing protocols. We propose using Allen's logic to study the temporal order of the messages. We introduce the concept of *discord* to quantify the order discrepancies between messages in different nodes of an HMSC and study its algorithmic properties. We show that while discord of a pair of messages is hard to compute in general, the problem becomes polynomial-time computable if the number of nodes of the HMSC or the number of processes is constant. Moreover, for a given HMSC, it is always computationally easy to identify a pair of messages that exhibits the worst-case discord, and compute the discord of this pair.

1 Introduction

Message Sequence Charts (MSCs) and High-level Message Sequence Charts (HMSC) are very useful tools for describing executions of communication protocols. They provide an intuitive visual notation, which is widely used in practice and has been formally described in the MSC standard [11]. A related notation was also adopted as part of the UML standard. Intuitively, an MSC is described by a set of *processes* and a set of *messages* between these processes. The notation allows us to specify the order in which each process sends and receives messages. An HMSC is a graph whose nodes are labeled with MSCs. An execution of an HMSC is a concatenation of MSCs that appear on a path in this graph. Using HMSC notation, one can describe alternative behaviors of systems, or even use it as a scenario-based programming formalism [10]. The reader is referred to Section 2 for formal definitions.

Besides being used in practice, MSCs and HMSCs have been extensively studied from theoretical perspective over the past few years. This research has pointed out several difficulties with these formalisms. One such example is the problem of detecting race conditions in MSCs [2], i.e., the possibility that messages arrive out of order due to lack of synchronization. This problem has also been generalized to HMSCs [14] and

* Work partly supported by the ESF project Automatha and the ANR-06-SETI project DOTS.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 378–393, 2007.
© Springer-Verlag Berlin Heidelberg 2007

sets of MSCs [7]. Another problem is related to global choice [4,3], where some processes behave according to one MSC scenario and other processes behave according to another MSC scenario, resulting in new behaviors.

Continuing this line of research, in this paper we identify another ambiguity of the MSC notation. Namely, in the definition of an HMSC, a concatenation of MSCs along a path intuitively suggests that messages that appear in an earlier MSC precede in time any message that appears in a later MSC. In fact, in some frameworks such as *live sequence charts* [6] there is a hidden assumption of such synchronous nature. However, according to the MSC semantics, this is is not the case: independence between events happening in different sets of processes may allow messages in later MSCs to overlap or even sometimes appear earlier than messages in previous MSCs. Moreover, it is not clear how to achieve this kind of synchronization without an additional mechanism or extra messages. Clearly, this discrepancy may result in users misinterpreting the notation and, as a result, designing protocols that do not work as intended. This is reminiscent of the concept of race conditions: the straightforward visual interpretation of concatenation is different from the intended semantics. However, unlike for race conditions, this discrepancy has not been studied before.

In this paper, we provide a formal treatment of this issue. We introduce the notion of *discord* of a pair of messages in different nodes of an HMSC. Intuitively, the discord of two messages is the worst possible discrepancy between their order in an execution and their "ideal" order, in which the message in the MSC that appears earlier on the path precedes the message in the MSC later on the path. To formalize this intuition, we need several tools that we introduce below.

We start our study of the message order in MSCs and HMSCs by defining the concept of a *chain*. Informally, a chain is a sequence of events where any adjacent pair of events is ordered either by being a send-receive pair, or by belonging to the same process line. Hence, a chain represents a possible flow of information. Clearly, the order between messages is determined not only by the relevant messages themselves, but also by chains between their endpoints. We characterize the possible message orders by describing the possible communication patterns between their endpoints. We then project each such pattern on a global timeline and classify the resulting scenarios. To do so, we use a subset of *Allen's interval logic* [1]. Allen's logic is a formalism for describing the relative order of time intervals. For example, Allen's logic formula $A\mathbf{d}B$ expresses the fact that A happens during B, i.e., A starts after B starts and ends before B ends. It has been widely studied in the context of artificial intelligence and knowledge representation, and its expressive power and computational properties are well understood [12]. As messages can easily be seen as time intervals, it provides a convenient language for describing the message order. We introduce a natural ordering on Allen's logic primitive predicates and define the discord of a pair of messages in an MSC as the worst possible Allen's logic primitive predicate (according to this ordering) that corresponds to the communication pattern of this pair.

We study the concept of discord from the algorithmic perspective. First, we show that computing the discord of a pair of messages is coNP-complete. Our reduction assumes that both the number of nodes in the HMSC and the number of processes are part of the input. We show that this is inevitable: if either of these numbers is fixed, the discord

can be computed in polynomial time. We then focus on characterizing the discord of an HMSC by a single parameter. To this end, we define the discord of an HMSC as the worst possible discord of a pair of messages in this HMSC. Surprisingly, it turns out that this quantity can be computed in time polynomial both in the size of the MSC graph and the number of processes. Intuitively, the reason for that is that it is easy to identify a pair of messages that exhibits the worst-case behavior for a given HMSC and compute the discord of such a pair. The study of discords provides also a generic study of the existence of communication chains, which we believe will be interesting in its own right in studies of layered combination of communication algorithms.

2 Preliminaries

2.1 Message Sequence Charts

Following [11], we formally define message sequence charts (MSCs), MSC concatenation, and high-level message sequence charts (HMSCs).

Definition 1. *A* Message Sequence Chart *(MSC) is a tuple $C = (\mathcal{P}, E, P, \mathcal{M}, <_{p:p\in\mathcal{P}})$, where*

- *$\mathcal{P}$ is a finite set of* processes*;*
- *E is a finite set of* events*;*
- *$P : E \mapsto \mathcal{P}$ is a function that maps every event to the process on which it occurs;*
- *$\mathcal{M}$ is a finite set of messages. Each message $m \in \mathcal{M}$ consists of a pair of events (s, r) for* send *and* receive*;*
- *For each process $p \in \mathcal{P}$, $<_p$ is a total order on the events of that process.*

We define a relation $<$ as $<= \bigcup_{p\in\mathcal{P}} <_p \cup\{(s, r) \mid (s, r) \in M\}$ and let $<^*$ be the transitive closure of $<$. We require $<^*$ to be acyclic. We assume that MSCs are FIFO, that is, if two messages (s_1, r_1) and (s_2, r_2) are between the same processes, i.e., $P(s_1) = P(s_2)$ and $P(r_1) = P(r_2)$, then $s_1 < s_2$ implies $r_1 < r_2$.

We will occasionally abuse notation and write $m \in C$ instead of $m \in \mathcal{M}$.

Definition 2. *Let C_1, C_2 be two MSCs where $C_1 = (\mathcal{P}^1, E^1, P^1, \mathcal{M}^1, <^1_{p:p\in\mathcal{P}^1})$, $C_2 = (\mathcal{P}^2, E^2, P^2, \mathcal{M}^2, <^2_{p:p\in\mathcal{P}^2})$ with $\mathcal{P}^1 = \mathcal{P}^2 = \mathcal{P}$ and $E^1 \cap E^2 = \emptyset$. Define their* concatenation *as an MSC $(C_1; C_2) = (\mathcal{P}, E, P, \mathcal{M}, <_{p:p\in\mathcal{P}})$, where $E = E^1 \cup E^2$, $\mathcal{M} = \mathcal{M}^1 \cup \mathcal{M}^2$, the function P is given by $P(e) = P^1(e)$ if $e \in E^1$ and $P(e) = P^2(e)$ if $e \in E^2$, and for each $p \in \mathcal{P}$ we define $<_p = <^1_p \cup <^2_p \cup\{(e_1, e_2)|P^1(e_1) = P^2(e_2)\}$.*

Notice that there are no sends in one MSC that are received in the other. This definition can be naturally extended to sequences $C_1, C_2, \ldots, C_n$ of three or more MSCs by setting $(C_1; C_2; \ldots; C_n) = ((\ldots(C_1; C_2); C_3) \ldots)$.

Definition 3. *A High-level Message Sequence Chart (HMSC) is a tuple $H = (\mathcal{G}, \mathcal{C}, \mathcal{V}_0, \lambda)$, where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph with the vertex set $\mathcal{V} = \{v_1, \ldots, v_n\}$ and the edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, $\mathcal{C} = \{C_1, \ldots, C_n\}$ is a collection of MSCs with a common set of processes and mutually disjoint sets of events, $\mathcal{V}_0 \subseteq \mathcal{V}$ is a set of* initial

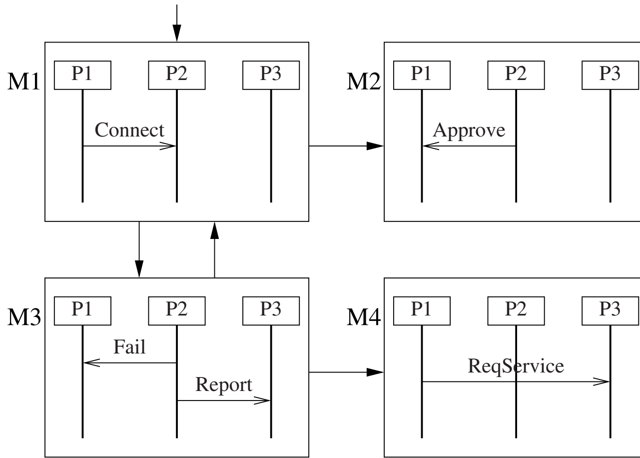


Fig. 1. An HMSC

nodes, and $\lambda : \mathcal{V} \mapsto \mathcal{C}$ is a bijective mapping between the nodes of the graph and the MSCs in $\mathcal{C}$. To simplify notation, we assume $\lambda(v_i) = C_i$. Each vertex of $\mathcal{G}$ is reachable from one of the initial nodes. An execution *of the HMSC is a finite MSC $(C_i; \ldots ; C_j)$ obtained by concatenating the MSCs in the nodes of a path $v_i, \ldots, v_j$ of the HMSC that starts with some initial node $v_i \in \mathcal{V}_0$. The* size *$|H|$ of an HMSC H is defined as $|H| = |E_1| + \cdots + |E_n| + |\mathcal{V}| + |\mathcal{E}|$, where E_i is the set of events of the MSC C_i.*

Given a path $L = (v_i, \ldots, v_j)$ in $\mathcal{G}$ of length at least 2, we denote by $\lambda(L)$ the MSC that is obtained by concatenating the MSCs along L, i.e., $(C_i; \ldots ; C_j)$. The set of executions of an HMSC is also referred to as the set of MSCs *generated* by that HMSC.

Figure 1 shows an example of an HMSC. The node in the upper left corner, denoted M1, is the starting node, hence it has an incoming edge that is connected to no other node. Initially, process P1 sends a message to P2, requesting a connection (e.g., to an internet service), according to the node M1. This can result in either an approval message from P2, according to the node M2, or a failure message, according to the node M3. In the latter case, a report message is also sent from P2 to some supervisory process P3. There are two progress choices, corresponding to the two arrows out of the node M3. We can decide to try and connect again, by choosing the arrow from M3 to M1, or to give up and send a service request (from process P1 to process P3), by choosing to progress according to the arrow from M3 to M4. Note how the HMSC description abstracts away from internal process computation, and presents only the communications. Consider the path (M1, M3, M4). According to the HMSC semantics, process P2 does not necessarily have to send its Report message in M3 before process P1 has progressed according to M4 to send its Req_service message. However, process P3 must receive the Report message before the Req_service message.

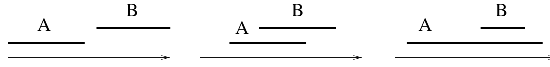


Fig. 2. Allen's logic relationships: $A\mathbf{p}B$, $A\mathbf{o}B$, and $A\mathbf{d}^{-1}B$

2.2 Allen's Logic

Allen's logic [1] is a formalism that allows one to express temporal relationships between time intervals. It has 13 primitive relations that correspond to possible relationships between two intervals, such as "A precedes B" or "A happens during B". Each primitive relation describes a total order between the endpoints of these intervals. When working with MSCs, we normally assume that no two events can happen at the same time, i.e., no two intervals have a common endpoint. Therefore, to represent relationships between two messages $m_1 = (s_1, r_1)$ and $m_2 = (s_2, r_2)$, we will only use 6 of these primitives, namely:

$\mathbf{p}$ — m_1 precedes m_2 (i.e., $s_1 < r_1 < s_2 < r_2$);
$\mathbf{p}^{-1}$ — m_1 is preceded by m_2 (i.e., $s_2 < r_2 < s_1 < r_1$);
$\mathbf{o}$ — m_1 overlaps m_2 (i.e., $s_1 < s_2 < r_1 < r_2$);
$\mathbf{o}^{-1}$ — m_1 is overlapped by m_2 (i.e., $s_2 < s_1 < r_2 < r_1$);
$\mathbf{d}$ — m_1 is during m_2 (i.e., $s_2 < s_1 < r_1 < r_2$);
$\mathbf{d}^{-1}$ — m_1 contains m_2 (i.e., $s_1 < s_2 < r_2 < r_1$).

Observe that for $\mathbf{t} \in \{\mathbf{p}, \mathbf{o}, \mathbf{d}\}$ the predicate $A\mathbf{t}B$ is equivalent to $B\mathbf{t}^{-1}A$.

An Allen's logic formula consists of concatenation of one or more of these 6 letters, and is interpreted as a disjunction of the corresponding predicates. For example, the formula $A\mathbf{pod}^{-1}B$ says that either A precedes B, or A overlaps B, or B happens during (is included in) A. Given the semantics of the primitive predicates, it is easy to see that this formula says that A starts before B, but may end before ($\mathbf{p}$), during ($\mathbf{o}$), or after ($\mathbf{d}^{-1}$) B. There are several operations that can be performed on Allen's logic formulas, such as composition and intersection. However, in this paper we only use the Allen's logic as a means to describe the relationships between messages. Therefore, we will not formally define these operations.

3 Relationships Between Messages

In this section, we will show how to use Allen's logic to reason about the relationship between a given pair of messages.

Given an MSC C, a *chain* from $x \in E$ to $y \in E$ is a sequence of events $(x = e_{i_1}, e_{i_2}, \ldots, e_{i_{k-1}}, e_{i_k} = y)$ such that $e_{i_j} \in E$ for $j = 1, \ldots, k$, and every adjacent pair $(e_{i_j}, e_{i_{j+1}})$ in the chain is either a send and the corresponding receive, or e_{i_j} appears before (above) $e_{i_{j+1}}$ in the same process line. Clearly, $x <^* y$ if and only if there is a chain of messages from x to y. Now, consider a pair of messages (s_1, r_1) and (s_2, r_2). By definition, there is always a chain from s_1 to r_1 and from s_2 to r_2. Moreover, for any $(a, b) \in \{s_1, r_1\} \times \{s_2, r_2\}$, we have one of the following three cases: (1) there is a chain of messages from a to b; (2) there is a chain of messages from b to a; (3) there is no

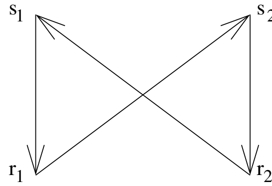


Fig. 3. Impossible relation between messages

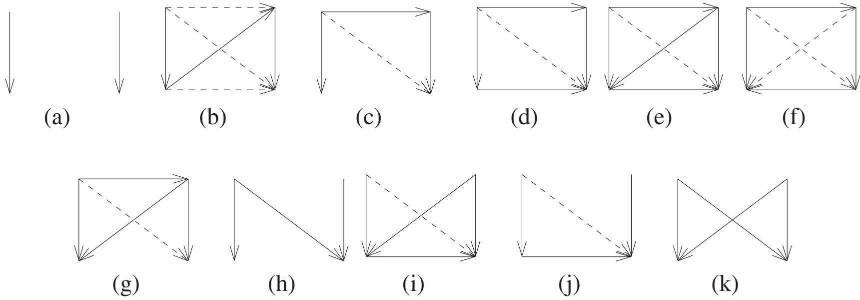


Fig. 4. The possible orders between messages (up to symmetry)

chain in either direction. As there are four pairs of points, this corresponds to $3^4 = 81$ combinations. However, not all of them are possible, as MSCs do not admit cycles (see Figure 3). In fact, for two messages there are exactly twenty possible combinations of orders between their endpoints. We list them in Figure 4. In these figures, the messages correspond to the vertical arrows, and dashed arrows correspond to relationships derived by transitive closure.

The patterns in Figure 4 correspond to the following Allen's logic relationships: (a) $\mathbf{pp^{-1}oo^{-1}dd^{-1}}$; (b) $\mathbf{p}$; (c) $\mathbf{pod^{-1}}$; (d) $\mathbf{po}$; (e) $\mathbf{o}$; (f) $\mathbf{d^{-1}}$; (g) $\mathbf{od^{-1}}$; (h) $\mathbf{poo^{-1}dd^{-1}}$; (i) $\mathbf{od}$; (j) $\mathbf{opd}$; (k) $\mathbf{oo^{-1}dd^{-1}}$. Except for cases (a) and (k), which are symmetric, each other case has a symmetric twin that can be obtained by swapping the left and the right message.

To decide between these cases, it suffices to calculate the transitive closure relation $<^*$. While in general transitive closure algorithms run in cubic time [9,16], it has been observed [2] that in the MSC case one can be more efficient since each event has at most two successors. Formally, we have the following proposition.

Proposition 1. Given an MSC M with messages $m_1, \cdots, m_t$, one can decide in time $O(t^2)$ the relation between every m_i, m_j, $1 \leq i, j \leq t$.

4 Definition of Discord

Concatenating two MSCs C_1 and C_2 does not necessarily mean that *all* the messages of C_1 precede in time all the messages of C_2: for example, if C_1 consists of a single

message from p_1 to p_2, and C_2 consists of a single message from p_3 to p_4, the relation $<$ does not provide any information about the relative order of these two messages. In what follows, we propose an Allen logic-based formalism that allows us to quantify the ordering discrepancies that occur when concatenating MSCs. We start by considering sequences of MSCs, and then extend our analysis to HMSCs.

Consider a concatenated MSC $(C_1; C_2)$. For any two messages $m_1 = (s_1, r_1) \in C_1$ and $m_2 = (s_2, r_2) \in C_2$, we know that $s_1 < r_1$ and $s_2 < r_2$. Now, intuitively, the best possible scenario for $(C_1; C_2)$ is when all messages in C_1 precede all messages in C_2. In this case, we also have $r_1 < s_2$, and thus we obtain $s_1 < r_1 < s_2 < r_2$. This corresponds to case (b) in Figure 4. Note that this scenario is only possible when C_1 has a unique maximal event e, C_2 has a unique minimal event e', and e and e' occur on the same process, i.e., $P(e) = P(e')$.

Conversely, the worst possible case is when some message m_2 in C_2 may be completely unordered with respect to a message m_1 in C_1. That is, for some m_1 and m_2 as above, the situation is described by case (a) in Figure 4, or by the Allen's logic formula $m_1 \mathbf{pp}^{-1}\mathbf{oo}^{-1}\mathbf{dd}^{-1} m_2$. In this case, at worst, the Allen logic formula allows m_2 to actually precede m_1, since the disjunction permits in particular that $m_1 \mathbf{p}^{-1} m_2$. All remaining scenarios lie, as will be formulated below, between these two cases. We will now introduce a measure of discrepancy, which we call the *discord*, which will allow us to order them more precisely,

Given a concatenation of two MSCs $(C_1; C_2)$, two messages $m_1 = (s_1, r_1) \in C_1$ and $m_2 = (s_2, r_2) \in C_2$ are said to be *out of order* if r_1 does not precede s_2, i.e., $\neg m_1 \mathbf{p} m_2$. In Figure 4, this happens in cases (a), (c), (d), (h), and (j). Note that in our setting, the cases (e), (f), (g), (i), and (k) are impossible: in each of these cases, there are chains of messages starting from events of m_2 and ending in events of m_1, which cannot happen under concatenation.

We now classify all primitive Allen logic predicates according to how well they order the endpoints of the projected intervals, i.e., represent the order between the events of the two messages m_1 and m_2. Recall that in the ideal case, i.e., when the order between the intervals is described by the Allen logic predicate $\mathbf{p}$, we have $s_1 < r_1 < s_2 < r_2$. In this case, there are zero events in $\{s_2, r_2\}$ that precede those in $\{s_1, r_1\}$. Under the worst case, i.e., if m_2 fully precedes m_1, we count four inversions: namely, $s_2 < s_1$, $r_2 < r_1$, $r_2 < s_1$ and $s_2 < r_1$. We thus order the predicates according to how many of these four relationships are inverted. In case of a tie, we give preference to the relationships that involve s_1 to those that involve r_1.

Definition 4. *The total order $\prec$ is the transitive closure of the partial order $\prec_0$ given by* $\prec_0 = \{(\mathbf{p}, \mathbf{o}), (\mathbf{o}, \mathbf{d}^{-1}), (\mathbf{d}^{-1}, \mathbf{d}), (\mathbf{d}, \mathbf{o}^{-1}), (\mathbf{o}^{-1}, \mathbf{p}^{-1})\}$.

Remark 1. Observe that the number of inversions in $\mathbf{p}^{-1}$ is 4, as explained above, in $\mathbf{o}^{-1}$ it is 3, in $\mathbf{d}$ and $\mathbf{d}^{-1}$ it is 2, in $\mathbf{o}$ it is 1, and in $\mathbf{p}$ it is 0. Therefore, our decision that $\mathbf{d}^{-1} \prec \mathbf{d}$ may appear quite arbitrary. We made this choice for two reasons. First, we do think that the time when the messages are sent is more important than the time when they are received, as the designer has more control over the former, and second, it is convenient to have a total order to work with. However, we believe that many of our ideas and results will apply for different orders, including some that are not total.

Definition 5. *Consider a sequence of MSCs* $(C_1, \ldots, C_k)$ *and a pair of messages* $m_1 \in C_1$*,* $m_2 \in C_k$ *such that in the MSC* $C = (C_1; \ldots; C_k)$ *we have* $m_1 \mathbf{R} m_2$*, where* $\mathbf{R}$ *is a (possibly non-primitive) Allen's logic predicate. The* discord *of* m_1 *and* m_2 *with respect to* C *is the* worst *possible primitive predicate (largest according to* $\prec$*) that appears in* $\mathbf{R}$*, i.e.,* $\text{discord}_C(m_1, m_2) = \mathbf{t}$*, where* $\mathbf{t} \in \{\mathbf{p}, \mathbf{p}^{-1}, \mathbf{o}, \mathbf{o}^{-1}, \mathbf{d}, \mathbf{d}^{-1}\}$*,* $\mathbf{t}$ *appears in* $\mathbf{R}$*, and for all* $\mathbf{t}'$ *that appear in* $\mathbf{R}$ *we have* $\mathbf{t}' \preceq \mathbf{t}$*.*

Let us now apply this definition to the six cases that can occur for a pair of messages in a concatenated MSC, as illustrated in Figure 4. In case (a) the messages are in relationship $\mathbf{pp}^{-1}\mathbf{oo}^{-1}\mathbf{dd}^{-1}$. The worst elementary predicate in this formula is $\mathbf{p}^{-1}$, so we conclude that the discord between the messages is $\mathbf{p}^{-1}$. For case (b), there is only one relation $\mathbf{p}$. Similarly, for case (c) the discord is $\mathbf{d}^{-1}$, for case (d) it is $\mathbf{o}$, for (h) it is $\mathbf{o}^{-1}$, and for (j) it is $\mathbf{d}$. We conclude that the value of $\text{discord}_C(m_1, m_2)$ can be *any* elementary Allen's logic predicate.

We now extend the definition of a discord to messages in HMSCs.

Definition 6. *Given an HMSC* $H = (\mathcal{G}, \mathcal{S}, \mathcal{V}_0, \lambda)$ *and a pair of messages* $m_1 \in \lambda(v)$*,* $m_2 \in \lambda(v')$*, let* $\text{discord}_H(m_1, m_2) = \max^{\prec}\{\text{discord}_{\lambda(L)}(m_1, m_2) \mid L = (v, \ldots, v')\}$*, where* $\max^{\prec} \mathcal{A}$ *is the maximum element of the set* $\mathcal{A}$ *with respect to* $\prec$*.*

Consider now the HMSC in Figure 1. For the path (M1, M2), the discord is $\mathbf{p}$, since the maximum event of M1, which is a receive, precedes the minimum event of M2, which is the send of message Approve. On the other hand, for the path (M1, M3, M1), we have that the Report message of M3 corresponds to the Connect message of M1 as in case (h) of Figure 4, which means a discord of $\mathbf{o}^{-1}$. The discord of (M3, M4) is $\mathbf{d}$ due to the relative ordering between Report in M3 and ReqService in M4.

We will now state a simple observation that allows us to compute $\text{discord}_H(m_1, m_2)$.

Claim 1. Consider an HMSC $H = (\mathcal{G}, \mathcal{C}, \mathcal{V}_0, \lambda)$. For any $v, v' \in \mathcal{V}$, $v \neq v'$, and any $m_1 \in \lambda(v), m_2 \in \lambda(v')$, we have $\text{discord}_H(m_1, m_2) = \max^{\prec}\{\text{discord}_{\lambda(L)}(m_1, m_2) \mid L = (v, \ldots, v') \text{ is a simple path}\}$. Also, for two messages $m_1, m_2 \in \lambda(v)$, we have $\text{discord}_H(m_1, m_2) = \max^{\prec}\{\text{discord}_{\lambda(L)}(m_1, m_2) \mid L = (v, \ldots, v) \text{ is a simple cycle}\}$.

Intuitively, this is true because removing a loop from a path from v to v' can only increase the discord between m_1 and m_2. Hence, the path that exhibits the worst-case discord is loop-free.

5 Computing the Discord of a Pair of Messages

For a simple path $L = (v = v_{i_1}, \ldots, v_{i_k} = v')$, computing $\text{discord}_{\lambda(L)}(m_1, m_2)$ for $m_1 \in \lambda(v)$, $m_2 \in \lambda(v')$ is easy. Namely, first we run the transitive closure algorithm to determine the causal relationships between the endpoints of m_1 and m_2. We then identify the corresponding scenario of Figure 4 and apply the case analysis presented after Definition 5. The running time of this algorithm is quadratic in the total number of messages in $\lambda(L)$.

For HMSCs, Definition 6 and Claim 1 suggest a straightforward algorithm for computing the discord: given two messages $m_1 \in \lambda(v)$, $m_2 \in \lambda(v')$, we can consider each simple path from v to v' (or each simple cycle, if $v = v'$), compute the discord along this

path, and output the maximum discord obtained in this way. This naive algorithm runs in exponential time in the input size. In the next subsection, we show that this is perhaps inevitable: we prove that in general the problem of computing $\text{Discord}_H(m_1, m_2)$ is coNP-hard. However, we will now provide an alternative way of verifying whether $\text{Discord}_H(m_1, m_2) = \mathbf{t}$, where $\mathbf{t} \in \{\mathbf{p}, \mathbf{p}^{-1}, \mathbf{o}, \mathbf{o}^{-1}, \mathbf{d}, \mathbf{d}^{-1}\}$. As we will see later, it can be used to construct an efficient algorithm for computing $\text{Discord}_H(m_1, m_2)$ in the important special case when the number of processes is constant.

We will first define a related problem that will be useful for stating our results.

PATH WITH NO CHAIN: Given an HMSC $H = (\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{C}, \mathcal{V}_0, \lambda)$, a pair of nodes $v, v' \in \mathcal{V}$, and a pair of events $e \in \lambda(v)$, $e' \in \lambda(v')$, is there a path L from v to v' in $\mathcal{G}$ such that in the MSC $\lambda(L)$ there is no chain of events from e to e'? We will write $\text{PNC}_H(e, e') = 1$ if such path exists and $\text{PNC}_H(e, e') = 0$ otherwise.

Proposition 2. Given an HMSC $H = (\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{C}, \mathcal{V}_0, \lambda)$, a pair of nodes $v, v' \in \mathcal{V}$, and a pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$, we have

- $\text{discord}_H(m_1, m_2) = \mathbf{p}$ if and only if $\text{PNC}_H(r_1, s_2) = 0$.
- $\text{discord}_H(m_1, m_2) = \mathbf{o}$ if and only if $\text{PNC}_H(r_1, s_2) = 1$, $\text{PNC}_H(s_1, s_2) = 0$, and $\text{PNC}_H(r_1, r_2) = 0$.
- $\text{discord}_H(m_1, m_2) = \mathbf{d}^{-1}$ if and only if $\text{PNC}_H(r_1, r_2) = 1$ and $\text{PNC}_H(s_1, s_2) = 0$.
- $\text{discord}_H(m_1, m_2) = \mathbf{d}$ if and only if $\text{PNC}_H(s_1, s_2) = 1$ and for any path $L = (v, \ldots, v')$ in $\mathcal{G}$, the MSC $\lambda(L)$ contains a chain from s_1 to s_2 or a chain from r_1 to r_2.
- $\text{discord}_H(m_1, m_2) = \mathbf{o}^{-1}$ if and only if there exists a path $L = (v, \ldots, v')$ in $\mathcal{G}$ such that the MSC $\lambda(L)$ contains no chain from s_1 to s_2 and no chain from r_1 to r_2, and $\text{PNC}_H(s_1, r_2) = 0$.
- $\text{discord}_H(m_1, m_2) = \mathbf{p}^{-1}$ if and only if $\text{PNC}_H(s_1, r_2) = 1$.

For the proof of Proposition 2, see the full version of the paper [8].

5.1 Computational Hardness

We will now show that for HMSCs the problem of upper-bounding $\text{discord}_H(m_1, m_2)$ is coNP-complete. Formally, we consider the following problem:

DISCORD$(H, \mathbf{t}, m_1, m_2)$: Given an HMSC H, a predicate $\mathbf{t} \in \{\mathbf{p}, \mathbf{p}^{-1}, \mathbf{o}, \mathbf{o}^{-1}, \mathbf{d}, \mathbf{d}^{-1}\}$, and two messages m_1, m_2 in H, is it the case that $\text{discord}_H(m_1, m_2) \preceq \mathbf{t}$?

Theorem 1. *The problem* DISCORD$(H, \mathbf{t}, m_1, m_2)$ *is coNP-complete.*

Proof. To see that DISCORD$(H, \mathbf{t}, m_1, m_2)$ is in coNP, observe that the complementary problem of checking whether $\text{discord}_H(m_1, m_2) \succ \mathbf{t}$ is in NP: a certificate can be provided by a path L such that $\text{discord}_{\lambda(L)}(m_1, m_2) \succ \mathbf{t}$. In particular, for $\mathbf{t} = \mathbf{p}$ a certificate is a path with no chain from r_1 to s_2, for $\mathbf{t} = \mathbf{o}$ it is a path with no chain from r_1 to r_2, for $\mathbf{t} = \mathbf{d}^{-1}$ it is a path with no chain from s_1 to s_2, for $\mathbf{t} = \mathbf{d}$ it is a path with

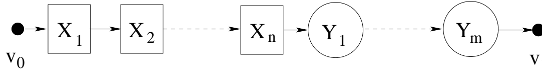


Fig. 5. The high-level structure of the HMSC H used in the proof of Theorem 1

no chain from s_1 to s_2 and no chain from r_1 to r_2, and for $\mathbf{t} = \mathbf{o}^{-1}$ it is a path with no chain from s_1 to r_2.

The coNP-hardness proof is by reduction from 3SAT. Suppose that we are given a 3CNF formula with a set of variables $x_1, \ldots, x_n$ and a set of clauses $c_1, \ldots, c_m$. Let l_j^1, l_j^2, l_j^3 be the literals that appear in the jth clause, i.e., $c_j = l_j^1 \vee l_j^2 \vee l_j^3$, $l_j^k \in \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$. We construct an HMSC H as follows. Set $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_{x_1}, p_{\bar{x}_1}, \ldots, p_{x_n}, p_{\bar{x}_n}, p_{c_1}, \ldots, p_{c_m}\}$. The HMSC H has the following structure. Its underlying graph $\mathcal{G}$ has a source v_0, a sink v_1, n variable gadgets $X_1, \ldots, X_n$ and m clause gadgets $Y_1, \ldots, Y_m$. The variable gadget X_i consists of four vertices u_i^0, u_i^1, u_i^2, u_i^3 and four edges (u_i^0, u_i^1), (u_i^0, u_i^2), (u_i^1, u_i^3), (u_i^2, u_i^3). The clause gadget Y_i consists of five vertices w_i^0, w_i^1, w_i^2, w_i^3, w_i^4 and six edges (w_i^0, w_i^1), (w_i^0, w_i^2), (w_i^0, w_i^3), (w_i^1, w_i^4), (w_i^2, w_i^4), (w_i^3, w_i^4). The source, the vertex gadgets, the clause gadgets, and the sink are all connected in series as depicted in Figure 5. More precisely, there is an edge from v_0 to the vertex u_1^0, for all $i = 1, \ldots, n-1$ there is an edge from u_i^3 to u_{i+1}^0, there is an edge from u_n^3 to w_1^0, for all $i = 1, \ldots, m-1$ there is an edge from w_i^4 to w_{i+1}^0, and finally there is an edge from w_m^4 to v_1.

It remains to define the MSCs that are placed in the vertices of $\mathcal{G}$. The MSC in v_0 consists of a single message (s_1, r_1) from p_1 to p_2. The MSCs in the vertices u_i^0, u_i^3, w_j^0, w_j^4 are empty for all $i = 1, \ldots, n$, $j = 1, \ldots, m$. For $i = 1, \ldots, n$, the MSC in u_i^1 consists of a message from p_2 to p_{x_i}, and the MSC in u_i^2 consists of a message from p_2 to $p_{\bar{x}_i}$. For $j = 1, \ldots, m$, $k = 1, 2, 3$, the MSC in w_j^k contains a message from $p_{l_j^k}$ to p_{c_j}, where l_j^k is the kth literal of c_j. Finally, the MSC in v_1 has $m+1$ messages: a message from each p_{c_j}, $j = 1, \ldots, m$, to p_3, and a message $m_2 = (s_2, r_2)$ from p_3 to p_4 that is sent after all messages from all p_{c_j} are received.

We claim that the original 3CNF formula is satisfiable if and only if the tuple $(H, \mathbf{p}, m_1, m_2)$ constitutes a "no"-instance of DISCORD$(H, \mathbf{p}, m_1, m_2)$, i.e., there is a path L from v_0 to v_1 such that the MSC $\lambda(L)$ contains no chain from r_1 to s_2.

Indeed, suppose that our formula is satisfiable, and let $\mathcal{T} = (t_1, \ldots, t_n)$, $t_i \in \{T, F\}$ be a satisfying assignment for it. Consider a path L that satisfies the following conditions:

- L starts at v_0 and ends at v_1;
- $L \cap X_i = \{u_i^0, u_i^1, u_i^3\}$ if $t_i = F$ and $L \cap X_i = \{u_i^0, u_i^2, u_i^3\}$ if $t_i = T$;
- $L \cap Y_j = \{w_i^0, w_j^k, w_j^4\}$ for some $k \in \{1, 2, 3\}$ such that l_j^k is true under $\mathcal{T}$, i.e., $l_j^k = x_z$ and $t_z = T$ or $l_j^k = \bar{x}_z$ and $t_z = F$. Note that such l_j^k is guaranteed to exist since $\mathcal{T}$ has to satisfy c_j.

First, note that in the corresponding MSC $\lambda(L)$ there is no chain from r_1 to any event of any of the processes p_{c_j}, $j = 1, \ldots, m$. Indeed, the only message received by p_{c_j} in $\lambda(L)$ is from some $p_{l_j^k}$ such that l_j^k is true under $\mathcal{T}$. Since l_j^k is true under $\mathcal{T}$, in $\lambda(L)$

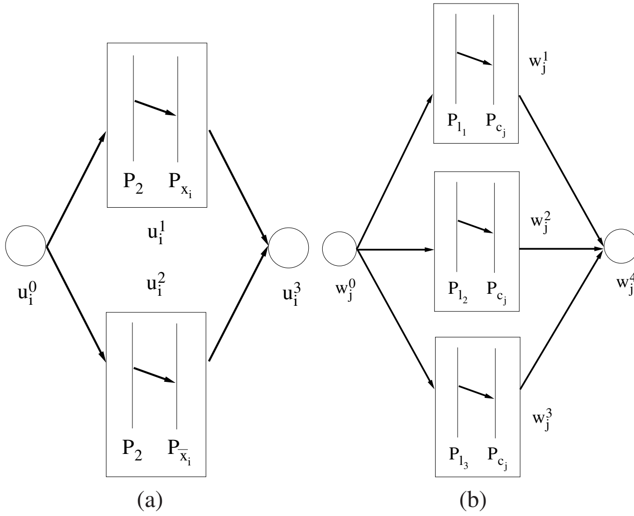


Fig. 6. (a) The gadget X_i; (b) The gadget Y_j

the process $p_{l_j^k}$ receives no messages whatsoever. As p_3 only receives messages from p_{c_j}, $j = 1, \ldots, m$, we conclude that in $\lambda(L)$ there is no chain from r_1 to s_2.

Conversely, suppose that there is a path L such that in the corresponding MSC $\lambda(L)$ there is no chain from r_1 to s_2. Consider a satisfying assignment $\mathcal{T} = (t_1, \ldots, t_n)$ such that $t_i = F$ if $L \cap X_i = \{u_i^0, u_i^1, u_i^3\}$ and $t_i = T$ if $L \cap X_i = \{u_i^0, u_i^2, u_i^3\}$. Note that for any $j = 1, \ldots, m$, if $L \cap Y_j = \{w_i^0, w_j^k, w_j^4\}$ for some $k = 1, 2, 3$, it must be the case that $p_{l_j^k}$ receives no message from p_2 in $\lambda(L)$, because otherwise there would be a chain of messages from r_1 to s_2. Hence, the literal l_j^k is true under $\mathcal{T}$, i.e., c_j is satisfied. As this holds for any $j = 1, \ldots, m$, we have successfully constructed a satisfying assignment for our instance of 3CNF. □

Remark 2. We can consider a weaker version of DISCORD, in which the Allen's logic predicate is not part of the input. Namely, for $\mathbf{t} \in \{\mathbf{p}, \mathbf{p}^{-1}, \mathbf{o}, \mathbf{o}^{-1}, \mathbf{d}, \mathbf{d}^{-1}\}$, we define DISCORD$_\mathbf{t}(H, m_1, m_2)$ as the problem of checking whether $\mathrm{discord}_H(m_1, m_2) \preceq \mathbf{t}$. Obviously, for $\mathbf{t} = \mathbf{p}^{-1}$ this problem is trivially in P: the answer is always "yes". In the full version of the paper [8] we show that this problem is coNP-hard for all $\mathbf{t} \neq \mathbf{p}^{-1}$.

5.2 Polynomial-Time Algorithms for Bounded Number of Processes

In our hardness result, both the size of the graph $\mathcal{G}$ and the number of processes $\mathcal{P}$ are unbounded. It turns out that this is necessary: if either of these parameters is constant, there is an algorithm whose running time is polynomial in the other parameter.

This is easy to see if the size of the graph is constant. In particular, the naive algorithm described in the beginning of this section will run in polynomial time: in a

graph with a constant number of vertices, there is a constant number of simple paths and cycles, and one can compute the discord along a path in polynomial time.

The case when the number of processes is constant is considerably more complicated. Our algorithm for this setting is based on Dijkstra's shortest path algorithm combined with dynamic programming approach. The underlying idea is that given a pair of events $e \in \lambda(v)$, $e' \in \lambda(v')$ and a subset of processes $\mathcal{S}$, we can check if there is a path L from v to v' such that the set of processes reachable from e in $\lambda(L)$ is exactly $\mathcal{S}$. A straightforward generalization of this idea allows us to compute the discord of any pair of messages in an HMSC in polynomial time for any fixed value of $|\mathcal{P}|$. The description of the algorithm and its analysis appear in the full version of the paper [8]. Here, we will state the main result.

Theorem 2. *It is possible to compute* $\mathrm{discord}_H(m_1, m_2)$ *in time* $O(n^2 2^{4|\mathcal{P}|}|H|^2)$.

6 From Pairs of Messages to HMSCs

In some situations, it is convenient to characterize the discord of an HMSC with a single parameter rather than list the discords for all pairs of messages in this HMSC. To this end, we extend the definition of discord from pairs of messages to entire HMSCs by defining the discord of an HMSC H to be the worst discord over all pairs of messages in H. Formally, we set $\mathrm{Discord}(H) = \max^{\prec}\{\mathrm{discord}_H(m_1, m_2) \mid m_1 \in \lambda(v), m_2 \in \lambda(v'), (v, v') \in \mathcal{E}^*\}$, where $\mathcal{E}^*$ is the transitive closure of the edge set $\mathcal{E}$, and $\max^{\prec}\mathcal{A}$ is the maximal element of the set $\mathcal{A}$ with respect to $\prec$.

According to this definition, one can compute $\mathrm{Discord}(H)$ by computing the discords for all pairs of messages in H. However, in general, computing $\mathrm{discord}_H(m_1, m_2)$ is coNP-hard, so this approach is not efficient. Quite surprisingly, it turns out that there exists a different approach that allows us to compute $\mathrm{Discord}(H)$ in polynomial time. It is based on the fact that while it may be hard to check whether there exists a chain between two events, it is easy to prove that there is no chain between two *extremal* events, for a suitable definition of extremality.

In the rest of the section, we describe polynomial-time algorithms for checking that $\mathrm{Discord}(H) = \mathbf{t}$ for $\mathbf{t} \in \{\mathbf{p}, \mathbf{p}^{-1}, \mathbf{o}, \mathbf{o}^{-1}, \mathbf{d}^{-1}\}$. To check whether $\mathrm{Discord}(H) = \mathbf{d}$, we can simply run all these algorithms and return "yes" if all of them return "no". We analyze the efficiency of these algorithms in terms of $n = |\mathcal{V}|$, $|\mathcal{P}|$ and $|H|$; observe that we can assume $n = O(|H|)$, $|\mathcal{P}| = O(|H|)$.

For our analysis, we will use the following calculation. By Proposition 1, we can compute the relation $<^*$ for any MSC C_i that appears in H in time $O(|E_i|^2)$, where E_i is the set of events of C_i. Hence, computing $<^*$ for all MSCs that appear in H can be done in $O(|E_1|^2 + \cdots + |E_n|^2) = O(|H|^2)$ steps.

For the cases $\mathbf{t} \in \{\mathbf{p}, \mathbf{o}, \mathbf{d}^{-1}\}$, we will make use of a set $\mathcal{E}^\star \subset \mathcal{V} \times \mathcal{V}$, constructed as follows: $(v, v') \in \mathcal{E}^\star$ if and only if $(v, v') \in \mathcal{E}$ or there exists a path $(v = v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}}, v_{i_k} = v')$ such that for $j = 2, \ldots, k-1$ the MSC $\lambda(v_{i_j})$ has an empty message set. Note that $\mathcal{E}^\star$ is a subset of the transitive closure of $\mathcal{E}$, i.e., $(v, v') \in \mathcal{E}^\star$ implies that in $\mathcal{G}$ there is a path from v to v'. It is not hard to see that $\mathcal{E}^\star$ can be constructed in time $O(n|H|)$.

Discord(H) = p. We will now show that $\mathrm{Discord}(H) = \mathbf{p}$ if and only if for any $(v, v') \in \mathcal{E}^\star$, and any $m_1 \in \lambda(v')$, $m_2 \in \lambda(v')$ we have $\mathrm{discord}_{(\lambda(v);\lambda(v'))}(m_1, m_2) = \mathbf{p}$. This condition is obviously verifiable in time $O(n^2|H|^2)$. The analysis can be improved to $O(n|H|^2)$, see [8].

Indeed, if for some such m_1, m_2 we have $\mathrm{discord}_{(\lambda(v);\lambda(v'))}(m_1, m_2) \neq \mathbf{p}$, then obviously $\mathrm{Discord}(H) \neq \mathbf{p}$. Conversely, consider any pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ and any path $L = (v = v_{i_1}, \dots, v_{i_k} = v')$. We show by induction on k that if our condition holds then $\mathrm{discord}_{\lambda(L)}(m_1, m_2) = \mathbf{p}$. The proof is based on the fact that for any three time intervals A, B, C, we have $A\mathbf{p}B \wedge B\mathbf{p}C \implies A\mathbf{p}C$. For $k = 2$, the statement is obvious. Now, suppose $k > 2$. If for each $j = 2, \dots, k-1$, the MSC $\lambda(v_{i_j})$ has an empty message set, then we have $\lambda(L) = (\lambda(v); \lambda(v'))$, so $\mathrm{discord}_{\lambda(L)}(m_1, m_2) = \mathbf{p}$. Now suppose that for some $j \in \{2, \dots, k-1\}$ the MSC $\lambda(v_{i_j})$ has non-empty message set and consider some $m = (s, r) \in \lambda(v_{i_j})$. Set $L' = (v_{i_1}, \dots, v_{i_j})$, $L'' = (v_{i_j}, \dots, v_{i_k})$. By induction hypothesis, $\mathrm{discord}_{\lambda(L')}(m_1, m) = \mathbf{p}$, $\mathrm{discord}_{\lambda(L')}(m, m_2) = \mathbf{p}$, so in $\lambda(L')$ there is a chain from r_1 to s, and in $\lambda(L'')$ there is a chain from r to s_2. We conclude that in $\lambda(L)$ there is a chain from r_1 to s_2, i.e., $\mathrm{discord}_{\lambda(L)}(m_1, m_2) = \mathbf{p}$.

Discord(H) = o, d^{-1}. The algorithm and the analysis are similar to the previous case. Namely, $\mathrm{Discord}(H) = \mathbf{o}$ (respectively, $\mathbf{d}^{-1}$) if and only if $\mathrm{Discord}(H) \neq \mathbf{p}$ (respectively, $\mathrm{Discord}(H) \neq \mathbf{p}, \mathbf{o}$), which can be verified in polynomial time as described above, and for any $(v, v') \in \mathcal{E}^\star$ and any $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ we have $\mathrm{discord}_{(\lambda(v);\lambda(v'))}(m_1, m_2) \in \{\mathbf{p}, \mathbf{o}\}$ (respectively, $\mathrm{discord}_{(\lambda(v);\lambda(v'))}(m_1, m_2) \in \{\mathbf{p}, \mathbf{o}, \mathbf{d}^{-1}\}$). The running time of this algorithm is $O(n|H|^2)$, as shown in [8].

The proof is based on the fact that for any three time intervals A, B, C, we have $A\mathbf{po}B \wedge B\mathbf{po}C \implies A\mathbf{po}C$ and $A\mathbf{pod}^{-1}B \wedge B\mathbf{pod}^{-1}C \implies A\mathbf{pod}^{-1}C$.

Discord(H) = p^{-1}. If $\mathrm{Discord}(H) = \mathbf{p}^{-1}$, there exists a pair of nodes $v, v' \in \mathcal{V}$, a pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ and a path $L = (v = v_{i_1}, \dots, v_{i_k} = v')$ such that $\mathrm{discord}_{\lambda(L)}(m_1, m_2) = \mathbf{p}^{-1}$, i.e., in $\lambda(L)$ there is no chain from s_1 to r_2. Let $C = \lambda(v)$, $C' = \lambda(v')$, and $\bar{C} = \lambda(v_{i_2}, \dots, v_{i_{k-1}})$.

Let s be a maximal send event in $(C; \bar{C})$ such there is a chain from s_1 to s, and let r be the corresponding receive. Set $p = P(s)$, $q = P(r)$. It is easy to see that in L there is no chain from s to r_2, or, equivalently, $(s, r)\mathbf{p}^{-1}m_2$. Therefore, without loss of generality we can assume $m_1 = (s, r)$, i.e., s_1 is a maximal send event in $(C; \bar{C})$. This implies that in $(C; \bar{C})$ there are no send events on p that happen after s_1, and there are no send events on q that happen after r_1 (for any such event, there would be a chain from s_1 to this event). Moreover, in C' there is no chain from any event of p or q to r_2.

This suggests the following algorithm. For each pair $v, v' \in \mathcal{V}$, and each pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ do the following. Set $p = P(s_1)$, $q = P(r_1)$. Let $H(v, v', p, q)$ be the HMSC obtained by deleting from H all nodes other than v, v' that have send events on p or q. Output "yes" if all of the following four conditions hold:

(1) in $\lambda(v)$ there are no send events on p after s_1;
(2) in $\lambda(v)$ there are no send events on q after r_1;

(3) in $\lambda(v')$ there is no chain from any event of p or q to r_2 (in particular, $P(r_2) \neq p, q$);
(4) the HMSC $H(v, v', p, q)$ contains a path from v to v'.

If (1) — (4) are all true, then the pair (m_1, m_2) provides a witness that $\mathrm{Discord}(H) = \mathbf{p}^{-1}$. Conversely, by the reasoning above, if $\mathrm{Discord}(H) = \mathbf{p}^{-1}$, then there is a pair (m_1, m_2) that satisfies (1) — (4).

The running time of this algorithm can be bounded by $O(|H|^3)$. To see this, note that there are $O(|H|^2)$ pairs of messages $m_1 \in \lambda(v)$, $m_2 \in \lambda(v')$. For each such pair, conditions (1) — (3) can be verified in time $O(|H|)$ assuming that the relation $<^*$ for $\lambda(v')$ has been precomputed (as argued above, we can precompute $<^*$ for all MSCs that appear in H in time $O(|H|^2)$). Condition (4) corresponds to solving a single instance of reachability problem, so it can be checked in time $O(|H|)$ as well. We can change the order of operations so that the algorithm runs in time $O(|\mathcal{P}|^2|H|^2)$ (see [8]). This is more efficient if $|\mathcal{P}|^2 < |H|$, which is likely to be the case in practice.

$\mathbf{Discord}(\boldsymbol{H}) = \mathbf{o}^{-1}$. Suppose $\mathrm{Discord}(H) = \mathbf{o}^{-1}$. Then there exists a pair of nodes $v, v' \in \mathcal{V}$, a pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ and a path $L = (v = v_{i_1}, \ldots, v_{i_k} = v')$ such that $\mathrm{discord}_{\lambda(L)}(m_1, m_2) = \mathbf{o}^{-1}$, i.e., in $\lambda(L)$ there is a chain from s_1 to r_2, but no chain from s_1 to s_2 and no chain from r_1 to r_2. Let $C = \lambda(v)$, $C' = \lambda(v')$, and $\bar{C} = \lambda(v_{i_2}, \ldots, v_{i_{k-1}})$.

Observe that in $(C; \bar{C})$ there is no chain from r_1 to any send event s. Indeed, suppose such a chain exists, and let r be the receive that corresponds to this send. If in $\lambda(L)$ there is no chain from s to r_2, we would have $(s, r)\mathbf{p}^{-1}(s_1, r_2)$, a contradiction. On the other hand, a chain from r_1 to s together with a chain from s to r_2 gives a chain from r_1 to r_2 in $\lambda(L)$, a contradiction again. By a similar argument, in $(\bar{C}; C')$ there is no chain from any receive event r to s_2.

Set $p = P(r_1)$, $q = P(s_2)$. It follows that in C there are no send events on p after r_1, in C' there are no receive events on q before s_2, and in $\bar{C}$ there are no sends on p and no receives on q. Obviously, in C there is no chain from s_1 to any event of q, and in C' there is no chain from any event of p to r_2. Moreover, it cannot be the case that $p = q$, $q = P(s_1)$ or $p = P(r_2)$.

Consequently, we have the following algorithm for checking whether $\mathrm{Discord}(H) = \mathbf{o}^{-1}$. First check that $\mathrm{Discord}(H) \neq p^{-1}$. Then for each pair $v, v' \in \mathcal{V}$, and each pair of messages $m_1 = (s_1, r_1) \in \lambda(v)$, $m_2 = (s_2, r_2) \in \lambda(v')$ do the following. Set $p = P(r_1)$, $q = P(s_2)$. Let $H(v, v', p, q)$ be the HMSC obtained by deleting from H all nodes other than v and v' that have send events on p or receive events on q. Output "yes" if the following six conditions hold:

(1) we have $p \neq q$, $q \neq P(s_1)$, $p \neq P(r_2)$;
(2) in C there are no send events on p after r_1;
(3) in C' there are no receive events on q before s_2;
(4) in C there is no chain from s_1 to any event of q;
(5) in C' there is no chain from any event of p to r_2;
(6) the HMSC $H(v, v', p, q)$ contains a path from v to v'.

Suppose that for some $v, v' \in \mathcal{V}$, $m_1 \in \lambda(v)$, $m_2 \in \lambda(v')$ (1) — (6) are all true. By (6), there exists a path $L = (v = v_{i_1}, \ldots, v_{i_k} = v')$ in $H(v, v', p, q)$. Set $\lambda(v) = C$,

$\lambda(v') = C'$, $\bar{C} = \lambda(v_{i_2}, \ldots, v_{i_{k-1}})$. Suppose that $\lambda(L)$ contains a chain from s_1 to s_2. As $q \neq P(s_1), p$, this chain must contain a receive event on q. By (3), there is no such event in C', and by construction of $H(v, v', p, q)$, there can be no such event in $\bar{C}$. Finally, by (4) there is no such event in C. Hence, in $\lambda(L)$ there is no chain from s_1 to s_2. Similarly, a chain from r_1 to r_2 must contain a send event on p, and there is no such event in C (by (2)), C' (by (5)), or $\bar{C}$ (by construction of $H(v, v', p, q)$). Hence, the pair (m_1, m_2) provides a witness that $\mathrm{Discord}(H) = \mathbf{o}^{-1}$ Conversely, by the reasoning above, if for some pair (m_1, m_2) we have $\mathrm{discord}_H(m_1, m_2) = \mathbf{o}^{-1}$, then our algorithm will succeed. As in the previous case, this algorithm can be implemented in time $O(|H|^3)$ or, by changing the order of operations, in $O(|\mathcal{P}|^2|H|^2)$.

7 Conclusions

We proposed using Allen's logic for detecting and measuring message order discrepancy in HMSCs. We believe that Allen's logic can be a versatile tool for other message order-related problems in MSCs and HMSCs, such as, e.g., race conditions and message overtake. Allen's logic is very well studied from algorithmic perspective [12]; while in this paper we did not use these results, they may be very useful for other applications of Allen's logic for message order analysis.

We introduced the notion of discord, which measures the difference between the message order in an HMSC and the "ideal" message order for that HMSC. We have shown a coNP-hardness result for computing the discord of a pair of messages in an HMSC, as well as polynomial-time algorithms for restricted versions of this problem. In contrast, we showed how to find the worst-case discord of an HMSC in polynomial time. We believe that the concept of discord will be useful in avoiding design errors in HMSCs. In particular, it can be applied when one wants to partition a large HMSC into smaller components: one should prefer partitions with small discord. Finally, consider an MSC-based programming approach such as the "play-in, play-out" framework of [10], which practically assumes synchronous MSC concatenation. Calculating discords allows one to quantify the potential for relaxing the synchronization assumption and check for possible hazards. This may increase concurrency and efficiency of the implementation and thus can be useful in protocol design.

References

1. Allen, J.F.: Maintaining Knowledge about Temporal Intervals. Communications of ACM 26(11), 832–843 (1983)
2. Alur, R., Holzmann, G., Peled, D.: An Analyzer for Message Sequence Charts. Software — Concepts and Tools 17, 70–77 (1996)
3. Alur, R., Etessami, K., Yannakakis, M.: Realizability and Verification of MSC Graphs. Theoretical Computer Science 331(1), 97–114 (2005)
4. Ben-Abdallah, H., Leue, S.: Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 259–274. Springer, Heidelberg (1997)
5. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. Journal of the ACM 30(2), 323–342 (1983)

6. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
7. Elkind, E., Genest, B., Peled, D.: Detecting Races in Ensembles of Message Sequence Charts. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, Springer, Heidelberg (2007)
8. Elkind, E., Genest, B., Peled, D., Spoletini, P.: Quantifying the Discord: Order Discrepancies in Message Sequence Charts,available from http://perso.crans.org/ğenest/EGPS07.pdf
9. Floyd, R.W.: Algorithm 97 (Shortest Path). Communications of the ACM, 356 (1962)
10. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
11. ITU Z120 standard recommendation (1996)
12. Krokhin, A., Jeavons, P., Jonsson, P.: Reasoning about Temporal Relations: The Tractable Subalgebras of Allen's Interval Algebra. J. ACM 50(5), 591–640 (2003)
13. Lohrey, M., Muscholl, A.: Bounded MSC communication. Information and Computation 189, 160–181 (2004)
14. Muscholl, A., Peled, D.: Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 81–91. Springer, Heidelberg (1999)
15. Peled, D.: Specification and Verification of Message Sequence Charts. In: FORTE'00, IFIP CP 183, pp. 139–154 (2000)
16. Warshall, S.: A Theorem on Boolean Matrices. Journal of the ACM 9(1), 11–12 (1962)

A Formal Methodology to Test Complex Heterogeneous Systems*

Ismael Rodríguez and Manuel Núñez

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
{isrodrig,mn}@sip.ucm.es

Abstract. Complex computational systems may integrate heterogeneous components that may be defined in different ways. In order to test the conformance of an implementation we can use a different testing methodology for each of the (different groups of) components. Still, this approach might overlook details regarding the relation among the parts of the system under test. We present an integrated testing methodology that takes into account the hierarchical dependence of all the parts of a system. Its main peculiarity is that parts of the *implementation under test* (IUT) that have already been tested may partially *define* the behavior of the tests used to check the correctness of other IUT parts.

1 Introduction

The complexity and heterogeneity of systems has reached a level where old solutions to assess reliability turn out to be obsolete. To overcome this problem, *formal testing techniques* [3,5,1,6,7] provide systematic procedures to create and apply tests to implementations. Usually, these techniques require constructing a formal specification to precisely define the expected capabilities of the system. Then, these capabilities are contrasted with those of the implementation. The task of completely specifying the behavior of current computational systems is very difficult due to their complexity. In particular, most real systems are very heterogeneous and include a big amount of components with different natures. Thus, instead of using a unique specification framework to formally define the behavior of systems, it is more adequate to define distinguished parts of the system by using different formalisms. In addition to be composed by several parts, systems often present a *hierarchical* structure. Thus, its definition can be decomposed into different levels of abstraction, consisting each of these levels of several units. In this case, the behavior of each unit will be formally specified by using a (possibly) different formalism. The formal specification of complex heterogeneous systems, by using different formalisms, represents an important challenge for traditional testing methodologies.

The purpose of this paper is to introduce a formal methodology to test this kind of systems. First, we consider that specifications are not described by using

* Research partially supported by the Spanish MEC project WEST/FAST TIN2006-15578-C02-01 and the Marie Curie RTN *TAROT* (MRTN-CT-2003-505121).

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 394–409, 2007.
© Springer-Verlag Berlin Heidelberg 2007

a unique language but that several languages can be used to specify different (sets of) functionalities, that is, different units. Systems have a hierarchical structure (different levels). Each level consists of several units. The first level contains those units that do not depend on other units. For any $i > 1$, the i-th level contains those units that depend only on units belonging to lower levels. Even though the specification of multi-level systems is a stand-alone goal, in this paper we will go one step further. Our objective is to test the conformance of IUTs with respect to these specifications. Obviously, even if a system is specified in different stages, so that the process can be somehow viewed as working with several specifications, we will have a unique integrated implementation, not one implementation for each of the units/levels. In fact, this implementation will be often given in terms of a *black box*, where the internal structure is not observable. Thus, we need tests that can stimulate the IUT by means of basic low level operations that can be understood by it. An obvious mechanism to test these kind of systems can be used if each unit of the IUT can be *disconnected* from those units it depends on. If this is possible then we could test different parts of the IUT in an *isolated* way. Unfortunately, it is not always feasible to disconnect parts of the IUT from other parts they depend on directly. In particular, this process could require some knowledge of the IUT that is not available if the IUT is a black box. Besides, we lose the capability to check the correctness of the *integration* and *assembling* of the parts of the IUT.

We present an integrated testing methodology to test multi-level systems. Since specifications are defined by several levels (and possibly with different formalisms) tests will be so. Different tests to check the correctness of the system at each level will be provided and all levels will be tested bottom-up. The main peculiarity of the methodology is that, once a level has been tested and no failure is found, that IUT level will provide part of the behavior of the tests that will be used to test *upper* levels. Let us consider a system defined in terms of three hierarchical levels. A test devoted to check the intermediate level interacts with the IUT by performing operations that may be defined in terms of other lower level operations. So, in order to stimulate the IUT, this test needs to perform some lower level operations as well. Once all lower level operations have been validated in the IUT by a suitable test suite (consisting of some lowest level tests), intermediate level tests will be able to perform each needed lower level operation by *invoking* the corresponding implementation of this operation in the IUT. That is, part of the behavior of tests will be directly given by the IUT (once the corresponding IUT part has been tested). Besides, the methodology will be refined to deal also with *circular dependencies* between units.

Let us note that our approach contains, as particular cases, standard approaches for the specification and testing of component based systems. Actually, each testing methodology designed to work with a formalism may provide a suitable mechanism to deal with the corresponding level of a multi-level specification. Each of these methodologies will be properly integrated in our methodology. Besides, protocols for testing multi-level systems such as ISO 9646 [2] are also easily included in our methodology. In fact, in [2] systems are *linear* in the sense that

there are n levels where each level contains a unique unit. This is a much simpler conception of systems than the one we present in this paper. Besides, tests do not have the capability of using the IUT as part of their own definition and environments with circular dependencies between units are not considered.

Our testing approach is presented informally and formally in the next section and in Section 3, respectively. In Section 4 we extended it to deal with *circular dependencies* among units. Some conclusions are given in Section 5.

2 Informal Presentation of the Methodology

In this section we informally introduce the main characteristics of our approach. We will review how traditional formal testing works and we will show how it can be adapted/modified/discarded in the case of multi-level testing. In formal testing we usually extract some tests from the considered specification and apply them to the implementation under test (IUT). By comparing the obtained results with the ones expected by the specification we can generate a *diagnosis* about the (in-)correctness of the IUT. These tests usually represent *stimulation plans* of the IUT by providing sequences of events that can be interpreted by the IUT. If the specification language specializes in defining simple communication events (e.g. the messages of a communications protocol) then these events can be straightforwardly reproduced and identified. In this case, it will be easy to apply the tests to the IUT, since the events clearly define how to stimulate the IUT. On the contrary, if the specification language deals with a higher level of abstraction then it will be more complex to decide how the test will apply the foreseen plan of events to the IUT.

Let us consider the following example. We specify an autonomous e-commerce agent by using a language that specializes only in the definition of its highest level of abstraction, that is, the economic objectives of the agent (e.g. [4]). The resulting model will clearly define, for each situation, the set of desirable transactions. Thus, the specification of an agent will indicate its behavior after receiving a given offer, as well as the way in which the agent will propose offers to other agents. Let us suppose that this high-level specification language does not define how an exchange proposal is split into simpler operations. In such a context, one may wonder how a test can indicate to the IUT that a transaction has to be performed. If the specification language deals only with high level behavior then the test should not interact with the implementation by following a fixed communications protocol. This is so because the specification language does not allow to express these details. Thus, if we fix the e-commerce protocol used by the test then we are forcing the IUT to follow that protocol, even though this information is not reflected in the specification. However, it is obvious that the test will be using a specific protocol. Otherwise, it would not be able to propose transactions to the IUT, being these the basic operations to stimulate the IUT. In order to solve the previously stated problem we will impose two conditions.

First, we need a more complete definition of the specification where lower abstraction levels are somehow included. Let us remark that even though a high level language can be used to provide key information about the desired behavior of the agent, the agent must indicate some lower level operations allowing to perform the desired functionalities. Thus, the specification of the agent must be done in several steps and by (potentially) using different specification languages for each level of abstraction. For example, a lower level defining the e-commerce communication protocol must be included in the agent specification.

Second, the definition of tests will be also given by levels/units. The tester has to use the same levels of abstraction as the ones in the specification. In order to test a certain level of abstraction of the IUT the tests will represent activity plans for that level. Nevertheless, each of the interactions between the test and the IUT will be carried out by using auxiliary (possibly lower level) operations. For example, if a test sends a *resource exchange offer* to the IUT then this operation requires to open a channel, codifying the exchange according to some protocol, sending the offer through the channel, etc. Lower level operations have to be tested beforehand with respect to the lower levels of abstraction where they are defined. Besides, lower levels of abstraction are tested by using operations of other lower levels of abstraction, and so on. In general, the definition of a test to check the behavior of the IUT at a level of abstraction needs the definition of all the levels of the test from that level down to the lowest level.

Thus, in order to define tests we will use a bootstrapping approach. The behavior of the units belonging to the lowest level will be tested as usual. That is, tests interact with the IUT by using atomic operations that are understood by both parts and no further definition is required. The difference comes when testing the behavior of a higher unit and, more precisely, when defining tests for these units. If a test is devoted to check a unit belonging to a certain level of the IUT then it needs to use lower level operations to interact with the IUT. However, the test does not have its own definition of how to split operations at level i into operations at level $i-1$. Instead, this definition will be taken from the IUT: The test will invoke IUT operations belonging to lower levels as part of its own definition. Let us note that the purpose is *not* to test these $(i-1)$-level IUT operations, but to *use* them to properly interact with the IUT at level i. That is, part of the test behavior (specifically, that concerning to the use of operations of level $i-1$) will be directly given by the IUT. However, before we use these IUT operations, we need to be confident that they are correctly implemented; otherwise tests would not work as planned. Thus, we require that these IUT operations have already been tested by other (lower level) tests.[1] These lower level tests may need to use IUT operations of other lower levels, and so on. This procedure yields a recursive testing methodology where units belonging to lower levels must be tested before the ones corresponding to higher levels.

[1] Even if these lower IUT operations successfully pass a suitable test suite, their correctness is not guaranteed in general. However, our criterion to assure the correctness of (part of) the tests using them is the *same* as to assure the correctness of IUTs. Hence, if the testing procedure for the IUT is suitable then it will be so for tests.

3 Basic Definitions

In this section we introduce some notation to formally define *multi-level* specifications and tests. Intuitively, a *specification* can be seen as a set of services, that is, functionalities that the system is supposed to provide. Each service is defined by means of an expression in a certain specification language. This expression indicates the operations that take place to perform that service. A given service can depend on other services provided by the specification and/or by other lower level sub-specifications. In particular, a service can be defined by using another service, which in turn is defined in terms of a third service, and so on. In this way, services can define dialogs between the system and the environment. The organization of services in units allows to precisely define how a unit depends on other units. For the sake of notation simplicity, we will assume that some operations over sets can be applied to *tuples* when the order of elements is not relevant. In this case, the operation will transform any tuple $(a_1, \ldots, a_n)$ into a set $\{a_1, \ldots, a_n\}$. For example, we will write expressions such as $e \in (a, b, c) \cup (e, f)$.

Definition 1. A *system* $\mathcal{S} = \{S_1, \ldots, S_n\}$ is a set of *unit specifications* (also called *specifications*) such that, for all $1 \leq i \leq n$, the *specification* S_i is a tuple $(L_i, A_i, \alpha_i, D_i)$ where L_i is the *language* used to define S_i, $A_i = (s_1^i, \ldots, s_{m_i}^i)$ denotes the tuple of *services* of S_i, and $\alpha_i \subseteq \{1, \ldots, n\}$ denotes the set of *indexes of specifications below* S_i. Besides, $D_i = (e_1, \ldots, e_m)$ denotes the tuple of *service definitions* of S_i, that is, each service $s_k \in A_i$ is defined by an expression $e_k ::= f_k(s_1^i, \ldots, s_{m_i}^i, s'_1, \ldots, s'_{m_k})$ where $\{s'_1, \ldots, s'_{m_k}\} \subseteq \bigcup_{j \in \alpha_i} A_j$. This expression is given in language L_i and may depend on any other service of S_i or S_j, with $j \in \alpha_i$. We assume that $j \in \alpha_i$ iff there exists $e_k ::= f_k(s_1^i, \ldots, s_{m_i}^i, s'_1, \ldots, s'_{m_k}) \in D_i$ such that $\{s'_1, \ldots, s'_{m_k}\} \cap A_j \neq \emptyset$. If the service s_k is *atomic* (i.e. it is not defined in terms of other services), we denote it by setting $e_k ::= \llcorner\lrcorner$.

The *set of sub-services* of S_i, denoted by $\texttt{Subservices}(S_i)$, is defined as $\bigcup_{j \in \alpha_i} A_j$. The *specifications of* $\mathcal{S}$ *at level* h, denoted by $\texttt{Level}(\mathcal{S}, h)$, are recursively defined as follows:

$$\texttt{Level}(\mathcal{S}, h) = \begin{cases} \{S_i \mid \alpha_i = \emptyset\} & \text{if } h = 1 \\ \left\{ S_i \,\middle|\, \begin{array}{l} \nexists\, j < h : S_i \in \texttt{Level}(\mathcal{S}, j) \wedge \\ \forall\, v \in \alpha_i \;\exists\, l < h : S_v \in \texttt{Level}(\mathcal{S}, l) \end{array} \right\} & \text{otherwise} \end{cases}$$

If $\alpha_i = \emptyset$ then we say that S_i is *simple*. For a given specification language L, we denote by $\texttt{Units}_L$ the set of all unit specifications in language L. We denote the set of all simple unit specifications in language L by $\texttt{Units}_L^{\emptyset}$. □

Each specification in a system defines a different *unit*. The level of a specification denotes a measure of its dependence on other specifications. Level 1 denotes simple specifications and level 2 denotes specifications depending only on simple specifications. Level 3 denotes specifications depending only on specifications at level 2 as well as those depending on levels 1 and 2, and so on. Let us note that the concept of level applies only if there do not exist *circular dependencies* of

specifications in the system. In this section, we will assume that such dependencies do not appear in our systems. In Section 4 we will remove this restriction. Let us remark that neither individual specifications nor levels correspond with the classical notion of *component*. A unit specification denotes some functionalities of *interaction* of the system with the environment, where each of them may be defined in terms of operations belonging to lower levels. In our framework, we call these functionalities *services*. On the contrary, functionalities provided by a component do no need to interact with the environment. This difference will be relevant for testing purposes. For the sake of simplicity we will assume that service names in a system are *unique*: Given two specifications $S, S' \in \mathcal{S}$ with $S = (L, A, \alpha, D)$ and $S' = (L', A', \alpha', D')$, we have $A \cap A' = \emptyset$.

Example 1. We define a (small) part of the behavior of a client application in a client-server system. Let $\mathcal{S} = \{S_1, S_2, S_3\}$ be a system. The specification S_1 defines how the user makes a request to the server. S_2 defines how to collect from the user the information required by the request, while S_3 defines how to send encrypted/normal messages through a communication channel. These units are defined as follows:

$$
\begin{aligned}
S_1 &= (L_1, (request, click, confirmRequest), \{2,3\}, D_1)\\
S_2 &= (L_2, (userGivesBankData, userGivesName, userGivesAccount,\\
&\qquad askForName, askForBankData), \{3\}, D_2)\\
S_3 &= (L_3, (bankReplies, serverReplies,\\
&\qquad askBank, askServer, sendName, sendEncryptedAccount), \emptyset, D_3)
\end{aligned}
$$

where L_1, L_2, L_3 are (different) specification languages where the sequential execution of operations $a_1, \ldots, a_n$ is expressed as follows: $a_1; \ldots; a_n$. For the sake of simplicity, only sequential executions are considered in this example (e.g., we do not consider *if* statements, *loops*, etc). Besides, D_1, D_2, and D_3 define the services *request*, *userGivesBankData*, and *askBank*, respectively, as follows:

$$
\begin{aligned}
request &::= click; askForName; userGivesName; askForBankData;\\
&\qquad userGivesBankData; askServer; serverReplies;\\
&\qquad confirmRequest\\
userGivesBankData &::= userGivesAccount; askBank; bankReplies\\
askBank &::= sendName; bankAcknowledgesName;\\
&\qquad sendEncryptedAccount
\end{aligned}
$$

The rest of services are atomic, that is, they are defined as $\llcorner\lrcorner$ by the corresponding D_i (e.g., $click, confirmRequest ::= \llcorner\lrcorner \in D_1$). We will assume that the environment can trigger any service by directly *invoking* it.[2] The action of *calling* a service will be denoted by an auxiliary service. For all service x, we assume that the service $\overline{x}$ invokes its execution, that is, we have the expression $\overline{x} ::= x$. For example, we assume that we have $\overline{confirmRequest} ::= confirmRequest \in D_1$.

The subservices of S_1 are all the services of S_2 and S_3, while $\texttt{Subservices}(S_2)$ consists of all the services belonging to S_3. In addition, $\texttt{Level}(\mathcal{S}, i) = \{S_i\}$ for $1 \leq i \leq 3$. The specification S_3 is *simple* but S_1 and S_2 are not. □

[2] Similarly, given the interface of an object in *object oriented programming*, a user can invoke any object method even if the implementation of the object is a black-box.

Next we introduce a general notion of *test suite*. This concept will allow us to abstract the underlying test derivation methodology. We only assume that there exists a fix criteria to construct test suites (see [8] for a survey on coverage criteria). Though the purpose of the following definition is to generalize known test derivation algorithms, where specifications do not have multiple *levels*, tests are defined in such a way that both simple and multi-level tests can be defined. Simple tests, i.e. tests designed to check a specific unit of the specification as if there did not exist any other units, can be composed to create multi-level tests to test multi-level specifications (this will be described in forthcoming Definition 4). Since tests are designed to interact with the IUT, services activated by a test must be services or subservices of the specification. Regarding the latter ones, let us note that tests are not prepared *by their own* to activate subservices. Hence, if a test activates a subservice then we will denote it by defining it as an atomic service of the test (meaning that a the test does not *know* how to produce it) or by using a service taken from a *lower* level of the test (meaning that the test is multi-level). In the next definition, $\mathcal{P}(X)$ denotes the powerset of the set X.

Definition 2. Let L be a specification language and $S = (L, A, \alpha, D) \in \texttt{Units}_L$ be a specification. A *test* for S is a tuple $T = (L', A', \alpha', D') \in \texttt{Units}_{L'}$ such that $A' = A'' \cup \{fail\}$, where $A'' \subseteq \{x, \overline{x} | x \in A \cup \texttt{Subservices}(S)\}$. Besides, for all $x \in A \cap \texttt{Subservices}(S)$ we have $x ::= \llcorner\lrcorner \in D'$. We denote the set of all tests for S by $\texttt{Tests}_S$. We say that the test $T \in \texttt{Tests}_S$ is *simple* if $\alpha' = \emptyset$ (that is, it stimulates only services belonging to A'). We denote the set of simple tests for S by $\texttt{Tests}_S^{\emptyset}$. A *simple test suite* for the specification S is any element in $\mathcal{P}(\texttt{Tests}_S^{\emptyset})$. □

We assume that the special service $fail$ is produced by a test when it detects a failure in the IUT according to some criterion. Besides, let us note that the languages used to define specifications and those used to define tests are not necessarily the same.

Example 2. Let L_1' be a language such that ; denotes a sequential execution and + represents a bifurcation that depends on the next service. We consider the following service, expressed in L_1':

$$\begin{aligned} askBank ::= &\ \overline{askBank}; sendName; bankAcknowledgesName; \\ &\ (sendEncryptedAccount) + (sendName; fail) \end{aligned}$$

This service defines a test case to interact with the IUT, specifically to check whether the $askBank$ service of the IUT behaves as defined by S_1. First, the test produces $\overline{askBank}$. If the IUT is correct, this service will launch the execution of its service $askBank$. Moreover, if this service is implemented by the IUT as S_1 defines, the IUT will reply with $sendName$. Next, the test will stimulate the IUT with $bankAcknowledgesName$. At this point, the test considers two possibilities. If the IUT replies with $sendEncryptedAccount$ then the test finishes correctly because this is the expected behavior. However, if the IUT produces $sendName$ then the IUT behavior does not conform to S_1, thus leading to $fail$. Let us

note that other definitions of $askBank$ would allow to check other parts of the behavior of this IUT service (e.g., the reply after $\overline{askBank}$).

Let T_1 be a test for S_1 where the previous definition of service $askBank$ is used and the rest of services are defined as *atomic*. T_1 is *simple*. Let us note that T_1 does not need to depend on other tests because the services it uses do not need to be further defined: If the IUT is correct then its implementation of S_1 (i.e. the lowest level) will understand these messages.

Now, let us consider the following service definition:

$$\begin{aligned} userGivesBankData ::= {} & \overline{userGivesBankData}; userGivesAccount; \\ & (askBank; bankReplies + askServer; fail) \end{aligned}$$

Let D_2' be a tuple of services definitions including the previous one as well as $askBank ::= \llcorner\lrcorner$, $askServer ::= \llcorner\lrcorner$, and $bankReplies ::= \llcorner\lrcorner$, and let T_2 be a test for S_2 where

$$\begin{aligned} T_2 = (L_2', (& userGivesBankData, \overline{userGivesBankData}, \\ & userGivesAccount, askBank, askServer, bankReplies, fail), \emptyset, D_2') \end{aligned}$$

T_2 is also a simple test. However, T_2 is useless to stimulate the IUT (at least by itself) because it does not *know* how to produce e.g. $bankReplies$ in such a way that it is *understood* by the IUT. □

In the following definition we introduce a general testing framework. As usual in formal testing, we consider that there exists a formal language to construct a precise model to describe the behavior of the IUT.

Definition 3. Let L be a specification language, $S \in \texttt{Units}_L$ be a specification, and $T \in \texttt{Tests}_S$ be a test. If the interaction of S and T may trigger the execution of a service a then we denote it by $\texttt{Produce}(S, T, a)$. Let $S_1 \in \texttt{Units}_L^{\emptyset}$ be a simple specification, $I_1 \in \texttt{Units}_{L'}^{\emptyset}$ be an IUT, and F_1 be a simple test suite for S_1. We say that I_1 *passes* F_1 if, for all $T_1 \in F$, $\texttt{Produce}(I_1, T_1, fail)$ does not hold. □

As we pointed out before, our testing methodology considers that the IUT is a *black box*. So, we cannot assume any internal structure. In particular, when we speak about a given *unit of the IUT*, we mean the implementation in the IUT of some services that are *logically* grouped in the specification as a unit. Similarly, when we talk about *level of the IUT* we mean the *implementation* of the corresponding units in the IUT, that is, a set of sets of services. If the IUT is correct with respect to the specification then these services must be provided by the IUT and they must be correctly implemented, but their physical structure in the IUT is indeed not considered. In order to test the conformance of a given unit with respect to a specification, we will create tests to stimulate the IUT according to some operations used in that unit. However, each of these operations has to be performed according to its specification, which could be defined in a lower unit. Since the IUT is expected to correctly implement all of the units, the test could take and use operations provided in lower levels of the IUT as a way to perform them.

However, the IUT implementation of these units could be *faulty*. Therefore, before we use the operations given in units belonging to a lower level, we will have to *check* their correctness. More precisely, the correctness of the capabilities provided by those units has to be assessed. In order to do that, we will *test* them. Following the same idea, testing the units belonging to a lower level may require to consider the functionalities condensed in an even lower level of the IUT. So, first of all we will have to check the correctness of those units. The same reasoning is repeated until we infer that we need to check the units corresponding to the lowest level. The tests needed to check the correctness of level 1 do not use any lower unit/level. Once we have tested this level of the IUT, we will use its capabilities as part of the activities of the tests that check the units belonging to the immediately higher level, and so on.

Let us remark that using the services provided by a unit of the IUT as part of the activity of a test does not consist in *breaking* this part and connecting it to the test. Since IUTs are black boxes, this cannot be done. Instead, using an IUT unit consists in taking the *whole* IUT and invoking and using *only* some of its services: The services that are logically grouped as the considered unit in the specification. The next definition formalizes this process. We derive a set of *multi-level* tests from a set of *simple* tests. In tests belonging to the latter set, all services are *atomic*. Thus, they do not need any further definition. In order to obtain the set of multi-level tests, we modify the aforementioned simple tests so that all the tests contain the definition of all lower levels. The operations from these units are taken directly from the IUT. To pass a test suite created for checking the i-th level of the IUT (for some $i > 1$) requires that lower units are correct with respect to the units of the specification defining the same services. In order to be confident in this correctness (although it will not be a *proof* of it), we will recursively apply a suitable test suite to the immediately lower units/level of the IUT. If this test suite is passed then we will use the services appearing in these units to construct services of the tests that check the capabilities corresponding to units appearing at level i.

Definition 4. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ and $IUT = \{I_1, \ldots, I_n\}$ be two systems. Let $S = (L_S, A_S, \alpha_S, D_S) \in \mathcal{S}$ and $I = (L_I, A_I, \alpha_I, D_I) \in IUT$. Let $B = \texttt{Subservices}(I)$. Let $F = \{(L_1, A_1, \emptyset, D_1), \ldots, (L_n, A_n, \emptyset, D_n)\}$ be a simple test suite for S. We say that the set $\{(L_1, A_1 \backslash B, \alpha_I, D'_1), \ldots, (L_n, A_n \backslash B, \alpha_I, D'_n)\}$ is a *multi-level test suite* for S and I, where for all $1 \leq i \leq n$ we have that D'_i is constructed by removing any subservice definition (i.e., $e ::= r$ for some $e \in B$ and r) from D_i. Let G be a multi-level test suite for S and I. We say that I *passes* G for S if the following two conditions hold:

(1) For all S_k, with $k \in \alpha_S$, there exists $l \in \alpha_I$ such that I_l passes G' for S_k, where G' is a multi-level test suite for S_k and I_l.
(2) For all $T \in G$ we have that $\texttt{Produce}(I, T, fail)$ does not hold. □

Let us note that the anchor case of the previous recursive definition corresponds to the case when we test a simple specification. In this case, there is no element

to consider in condition (1), so that this case trivially holds and no recursive call is needed. When a non-simple specification is tested, we recursively test each of its sub-specifications. Afterwards, the whole specification is tested. Since tests belonging to the simple test suite are not provided with definitions of lower level operations, these operations are represented by atomic operations at the actual level. However, after these operations are properly provided by the IUT (tests are linked to α_I, which allows them to use any IUT lower operation), these atomic substitutes are removed. The application of the previous definition directly leads to the iterative testing algorithm presented in Figure 1.

Example 3. Let us consider the test T_2 constructed before. In order to turn this simple test into a multi-level test, we must remove the services *askBank*, *askServer*, and *bankReplies* from its list of provided services as well as erasing their definition from D_2' (where they were defined as $\llcorner\lrcorner$), leading to a new tuple of services definitions D_2''. Instead, the new test T_2' will execute these services by *invoking* their respective implementations at the IUT. In this way, the *implementation* of S_1 in the IUT will enable the test to interact with the *implementation* of S_2 in the IUT. In technical terms, a part of the IUT will be a part the new test. Given a system $\mathcal{S} = \{T_2', I_1\}$ where I_1 denotes the implementation of S_1 in the IUT, the new multi-level test T_2' is defined as follows:

$$T_2' = (L_2', \{userGivesBankData, \overline{userGivesBankData}, userGivesAccount, fail\}, \{1\}, D_2'') \qquad \square$$

Next we show that tests use lower units of the IUT as part of their own definition only after these units have already been tested by previous tests.

Lemma 1. Let $\mathcal{S}, \mathcal{I}$ be two systems such that $S \in \mathcal{S}$, $I \in \mathcal{I}$, and $S \in \texttt{Level}(\mathcal{S}, i)$ for some $i \in \mathbb{N}$. Let G be a multi-level test suite for S and I, and let $T \in G$. The algorithm given in Figure 1 applies T only after, for all $S' \in \texttt{Level}(\mathcal{S}, j)$ with $j < i$, a test T' for S' and some $I' \in \mathcal{I}$ has already been applied. $\square$

4 Specifications with Circular Dependencies

The methodology presented in the previous section is based on the idea that each unit of the specification uses other units that are located *below* it. Hence, we assume the existence of some *minimal* units whose definition does not depend on other units. These units play the role of the *anchor case* in our recursive methodology. This fact makes our methodology to be well-formed. However, one can imagine specifications with *circular* dependencies between units. For instance, let us consider the specification of a distributed system where each part uses services that are provided by all the other parts. In this case, the methodology presented in the previous section might not work because the existence of an anchor case is not guaranteed.

In order to tackle this problem we could consider that all the units containing mutual circular dependencies are part of a *single* logical unit. Hence, the *logical* division of the specification for testing purposes would be free of circular dependencies. However, this method would partially break the modularity of the

Input: A specification S with units grouped in n levels and an implementation I.
Output: A diagnosis result: **true** or **false**.

$i := 1$;
error := **false**;
while $i \leq n$ **and not** error **do**
  Let $\mathcal{S}_i$ be the set of all sub-specifications of S at level i;
  while $\mathcal{S}_i \neq \emptyset$ **and not** error **do**
    Choose $S' \in \mathcal{S}_i$;
    Let I' be the implementation of S' in IUT;
    $\mathcal{S}_i := \mathcal{S}_i \backslash \{S'\}$;
    Generate a test suite G for S';
    if $i \geq 2$ **then**
      forall test $T \in G$ **do**
        use all subparts of I' for implementing level $i-1$ of T
      od
    fi;
    forall test $T \in G$ **do** apply T to I' **od**;
    if *fail* is obtained **then** *error* := **true fi**
  od
  $i := i + 1$;
od
return (**not** error);

Fig. 1. Multi-level testing algorithm

testing procedure. If it were possible, we could also disconnect those units with circular dependencies from the rest of units. Then, we could test each of them in an isolated fashion. Let us note that this requires to *physically* break a part of the IUT. Thus, this alternative is not feasible in a black-box testing approach because we cannot separately access the different parts of the system under test.

In the rest of the section we show another alternative whose main idea is to consider that the order to test units is governed by the *services* instead of by the own units. In fact, there exist several situations where we may have a circular dependence between units but we can still find a sequence of services without a circular dependence between them. If such a condition holds then the order of application of our methodology will be governed by these sequences.

Example 4. A given system consists of two concurrent processes (specified by S_1 and S_2) that can perform dialogs to exchange some information. The service of sending data from one of the processes to the other implies using the service of reception of the last one. We have:

$$\begin{aligned} S_1 = (&L_1, (sendingTo2, receivingFrom2), \{2\}, \\ &(sendingTo2 ::= receivingFrom1, receivingFrom2 ::= \llcorner\lrcorner)) \\ S_2 = (&L_2, (sendingTo1, receivingFrom1), \{1\}, \\ &(sendingTo1 ::= receivingFrom2, receivingFrom1 ::= \llcorner\lrcorner)) \end{aligned}$$

The definition of each of these units depends on the other one, and we have a circular dependence between units. However, the services of reception of data do not depend on any other service. In addition, the services to send data depend on the services of reception, but the dependence of services finishes there. So, there is an order to test services avoiding any circular dependence between them: We test both reception services, and next we check both sending services. That is, in spite of the existence of a circular dependence between units, there is no circular dependence between services. This makes possible to apply our methodology by considering a structure *based on services* instead of one *based on levels*. □

Let us formalize this alternative methodology. First of all, we need to identify all the services conforming a multi-level specification, regardless of the level where the units using them are located.[3] Given a unit, the next recursive function finds all services (as well as their respective definitions) from that unit down through any number of dependence links. The function uses an auxiliary set Q to contain all the services cumulated in previous recursive calls. We will use this set to avoid performing a call over a unit whose services have already been included in the set. By doing so we avoid that a circular dependence produces an infinite sequence of recursive calls to this function. Similarly, we also define the full set of specifications that can be reached from a given specification, through dependency links. In this case, R denotes the set of indexes of specifications already cumulated in previous recursive calls.

Definition 5. Let $\mathcal{S} = \{S_1, \dots, S_n\}$ be a system and $S = (L, A, \alpha, D) \in \mathcal{S}$ be a specification where $A = (s_1, \dots, s_b)$ and $D = (e_1, \dots, e_b)$. Besides, let $P_S = \{(s_1, e_1), \dots, (s_b, e_b)\}$. We define the *full set of defined services* of S, denoted by $\mathtt{Full}(S)$, as $\mathtt{Full'}(S, \emptyset)$, where

$$\mathtt{Full'}(S,Q) = P_S \cup \left\{ (s,e) \in \mathtt{Full'}(S_i, Q \cup A) \;\middle|\; \begin{array}{c} S_i = (L_i, A_i, \alpha_i, D_i) \in \mathcal{S} \\ \wedge\, i \in \alpha \;\wedge\; A_i \not\subseteq Q \end{array} \right\}$$

Let $\mathtt{Full}(S) = \{(s_1, e_1), \dots, (s_m, e_m)\}$. Then, the *full set of services* of S, denoted by $\mathtt{FullServ}(S)$, is the set $\{s_1, \dots, s_m\}$. Let $S_p = (L_p, A_p, \alpha_p, D_p) \in \mathcal{S}$. We define the *set of specifications* below S_p, denoted by $\mathtt{Dependents}(S)$, as $\mathtt{Dep'}(S_p, \{p\})$, where

$$\mathtt{Dep'}(S_p, R) = \{S_k | k \in \alpha_p\} \cup \left\{ S_j \in \mathtt{Dep'}(S_i, \alpha_p \cup R) \;\middle|\; \begin{array}{c} S_i = (L_i, A_i, \alpha_i, D_i) \in \mathcal{S} \\ \wedge\, i \in \alpha_p \;\wedge\; \alpha_i \not\subseteq R \end{array} \right\}$$

□

Let us recall that services are defined as an *expression* that may depend on other services. In turn, these services may belong either to the same unit in which the service is defined or to another unit that is directly referred from it.

Example 5. We have $\mathtt{FullServ}(S_1) = \mathtt{FullServ}(S_2)$, which in turn are equal to $\{sendingTo1, receivingFrom1, sendingTo2, receivingFrom2\}$. We also have $\mathtt{Dependents}(S_1) = \mathtt{Dependents}(S_2) = \{1, 2\}$. □

[3] The notion of *level* is partially lost in systems with circular dependencies because the maximal distance from some unit to simple units could be *infinite*.

In the next definition we give a condition to construct a sequence containing all the services used in a multi-level specification in such a way that no service is defined in terms of other services that appear *before* in the sequence.

Definition 6. Let S be a specification, $\texttt{Full}(S) = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ be the full set of defined services of S, and $[(s_{i_1}, e_{i_1}), \ldots, (s_{i_n}, e_{i_n})]$ be a permutation of $\texttt{Full}(S)$. The sequence of services $[s_{i_1}, \ldots, s_{i_n}]$ is a *non-circular sequence of services* for S if for all $1 \leq j < k \leq n$ we have that either

- there exists a specification $S' = (L', A', \alpha', D') \in \texttt{Dependents}(S)$ such that $s_{i_j}, s_{i_k} \in A'$, or
- e_{i_k} is not defined in terms of s_{i_j}.

If there exists such a sequence for S then we say that S is *service-structured.* □

If a specification is service-structured then we can develop a methodology where all the services of the specification are tested in a given order: The order given by a list of services fulfilling the condition of the previous definition.

Example 6. $C = [sendingTo1, sendingTo2, receivingFrom1, receivingFrom2]$ is a non-circular sequence of services for S_1 as well as for S_2. □

In the previous section we introduced a testing methodology where test suites for checking the functionalities of each unit were considered. Now we consider tests suited to check a *single* service belonging to the corresponding unit. In order to check the correctness of this service, tests may use only those IUT lower services that appear *further* in the non-circular sequence of services. As we will see, this is valid because these services will be tested *before*. In fact, any other lower service will not appear in the tests constructed to check that service.

Definition 7. Let $S = (L, A, \alpha, D) \in \texttt{Units}_L$ where $A = (s_1, \ldots, s_b)$ and $D = (e_1, \ldots, e_b)$. Let $s = s_k$ for some $1 \leq k \leq b$, and let $e_k ::= f_k(s'_1, \ldots, s'_g, s''_1, \ldots, s''_h)$ where $\{s'_1, \ldots, s'_g\} \subseteq A$ and $\{s''_1, \ldots, s''_h\} \cap A = \emptyset$. Finally, let us consider that $C = [s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n]$ is a non-circular sequence of services of S where $\{s''_1, \ldots, s''_h\} \subseteq \{s_{i+1}, \ldots, s_n\}$, and let $\overline{A} = \{\overline{x} | x \in A\}$. A *test for the specification* S, *the service* s, *and* C is a tuple $T = (L', A', \alpha', D') \in \texttt{Units}_{L'}$ such that

(a) $s \in A'$ *(i.e., s is checked by T)*,
(b) $A' = A'' \cup \{fail\}$ with $A'' \subseteq \{x, \overline{x} | x \in A \cup \texttt{Subservices}(S)\}$, and for all $x \in A \cap \texttt{Subservices}(S)$ we have $x ::= \llcorner\lrcorner \in D'$ *(i.e., T checks only services and subservices of S)*,
(c) $\{fail, s, \overline{s}, s'_1, \ldots, s'_g, s''_1, \ldots, s''_h\} \subseteq \texttt{FullServ}(T)$ *(i.e., T or its sub-levels tackle the full definition of s)* and $\texttt{FullServ}(T) \subseteq \{fail, s\} \cup A \cup \overline{A} \cup \{s_{i+1}, \ldots, s_n\}$ *(i.e., T and its sub-levels do not refer to services before s in the sequence C)*.

We denote the set of all tests for S, s, and C by $\texttt{Tests}_{S,s,C}$. A test $T = (L', A', \alpha', D') \in \texttt{Tests}_{S,s,C}$ is *simple* if $\alpha' = \emptyset$. The set of simple tests for S

is denoted by $\texttt{Tests}^{\emptyset}_{S,s,C}$. A *simple test suite* for the specification S, service s, and list C is any element in $\mathcal{P}(\texttt{Tests}^{\emptyset}_{S,s,C})$. □

Example 7. Next we show a simple test for S_1, $sendingTo2$, and C:

$$T_1 = (L'_1, (sendingTo2, receivingFrom2, recevingFrom1, fail), \emptyset,\\ sendingTo2 ::= \overline{sendingTo2}; (receivingFrom1) + (receivingFrom2; fail),\\ receivingFrom2 ::= \llcorner\lrcorner, receivingFrom1 ::= \llcorner\lrcorner)$$ □

We have the needed machinery to define our *services-oriented* testing methodology. Basically, we will traverse a non-circular sequence of services, checking the correctness of the IUT for each service in the sequence one after each other. For any sequence of services, testing the first service s of the list requires to test before all the services remaining in the sequence. This is done by performing a recursive call where the remaining sequence of services is the parameter.

We will use the services of the IUT already checked in the tests to check other services. We will do it similarly to the way we did it for checking units. In order to check a service used in a certain unit, we create tests that may use lower services, which are provided by lower units of the IUT. However, in the current setting we check a unique service in each step. Hence, it may happen that some of the units that we use as part of the tests are not *completely* tested. Nevertheless, let us note that those services such that the service we are testing depends on them are tested in previous steps indeed. This is so because the order in the sequence of services properly keeps the dependencies between services. Hence, there is no risk to include in a test some functionality that will *actually* be used and has not been checked yet. Next we assume that $[c|C]$ denotes a sequence where the first element c is followed by the sequence C.

Definition 8. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ and $IUT = \{I_1, \ldots, I_n\}$ be two systems. Let $S = (L_S, A_S, \alpha_S, D_S) \in \mathcal{S}$ and $I = (L_I, A_I, \alpha_I, D_I) \in IUT$. Let $B = \texttt{Subservices}(I)$. Let C be a non-circular sequence of services of S. Let $F = \{(L_1, A_1, \emptyset, D_1), \ldots, (L_n, A_n, \emptyset, D_n)\}$ be a simple test suite for S, s, and C. We say that the set $\{(L_1, A_1 \backslash B, \alpha_I, D'_1), \ldots, (L_n, A_n \backslash B, \alpha_I, D'_n)\}$ is a *multi-level test set* for S, I, s, and C, where for all $1 \leq i \leq n$ we have that D'_i is constructed by removing any subservice definition (i.e., $e ::= r$ for some $e \in B$ and r) from D_i.

Let $C = [s|C']$. Besides, let $I' = (L'_I, A'_I, \alpha'_I, D'_I)$ be the unique element in $\texttt{Dependents}(I)$ such that $s \in A'_I$, and $S' = (L'_S, A'_S, \alpha'_S, D'_S)$ be the unique element in $\texttt{Dependents}(S)$ such that $s \in A'_S$. Let G be a multi-level test suite for S', I', s, C. We say that I *passes* C for S if the following conditions hold:

(1) I passes C' for S, and
(2) for all $T' \in G$ we have that $\texttt{Produce}(I', T', fail)$ does not hold.

Besides, we consider that I *passes* [ ] for S, that is, the empty sequence of services is always successfully passed. We say that I *passes* S if I passes C for S, where C is a non-circular sequence of services for S. □

Example 8. Let us suppose that I_1 and I_2 represent the implementation of S_1 and S_2 in the IUT, respectively. Given the system $\mathcal{S} = \{T_1', I_2\}$, T_1' is the multi-level test for S_1, I_1, $sendingTo2$, and C defined as follows:

$$\begin{array}{l} T_1' = (L_1', (sendingTo2, receivingFrom2, fail), \{2\}, \\ \quad sendingTo2 ::= \overline{sendingTo2}; (receivingFrom1) + (receivingFrom2; fail), \\ \quad receivingFrom2 ::= \sqcup\!\sqcup) \end{array}$$ □

Next we show that tests do not use IUT services as part of their own definition until these services have already been tested by other tests.

Lemma 2. Let $\mathcal{S}, \mathcal{I}$ be two systems such that $S = (L, A, \alpha, D) \in \mathcal{S}$, and let $I \in \mathcal{I}$. Let $C = [s_k|C']$ be a non-circular sequence of services of S such that $s_k \in A$ is a service defined by the expression $e_k ::= f_k(s_1', \ldots, s_g', s_1'', \ldots, s_h'') \in D$, where $s_1', \ldots, s_g' \in A$ and $s_1'', \ldots, s_h'' \notin A$. Let G be a multi-level test suite for S, I, s_k, and C, and let $T \in G$. The testing method given in Definition 8 applies T only after, for all $1 \leq j \leq h$, a test T' for some $S' \in \mathcal{S}$, $I' \in \mathcal{I}$, s_j'', and a suffix of C has already been applied. □

5 Conclusions and Future Work

In this paper we have presented a testing methodology that is suitable for testing complex and heterogeneous systems. In these systems, each component can be specified by using a different specification language. These languages can be, in general, very different. The proposed methodology is defined in a recursive way and is based on the idea of testing at first lower levels and by continuing with higher levels, up to the highest one. Testing the correctness of the functionalities of each unit of the IUT allows us to *use* these operations as part of the tests that will check the behavior of units located in higher levels of the IUT. We propose a second methodology allowing to test systems presenting circular dependencies.

References

1. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
2. ISO/IEC 9646-1: 1994. Information technology – Open Systems Interconnection – Conformance testing methodology and framework. Part 1: General concepts (1994)
3. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
4. Núñez, M., Rodríguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005)
5. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)

6. Rodríguez, I., Merayo, M.G., Núñez, M.: A logic for assessing sets of heterogeneous testing hypotheses. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 39–54. Springer, Heidelberg (2006)
7. Rodríguez, I., Merayo, M.G., Núñez, M.: HOTL: Hypotheses and Observations Testing Logic. Journal of Logic and Algebraic Programming (in press, 2007), Available at http://dx.doi.org/10.1016/j.jlap.2007.01.001
8. Zhu, H., Hall, P.A.V, May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surverys 29(4), 366–427 (1997)

A New Approach to Bounded Model Checking for Branching Time Logics

Rotem Oshman and Orna Grumberg

Technion – Israel Institute of Technology
Department of Computer Science

Abstract. Bounded model checking (BMC) is a technique for overcoming the state explosion problem which has gained wide industrial acceptance. Bounded model checking is typically applied only for linear-time properties, with a few exceptions, which search for a counter-example in the form of a tree-like structure with a pre-determined shape. We suggest a new approach to bounded model checking for universal branching-time logic, in which we encode an arbitrary graph and allow the SAT solver to choose both the states and edges of the graph. This significantly reduces the size of the counter-example produced by BMC.

A dynamic completeness criterion is presented which can be used to halt the bounded model checking when it becomes clear that no counter-example can exist. Thus, verification of the checked property can also be achieved. Experiments show that our approach outperforms another recent encoding for μ-calculus on complex ACTL properties.

1 Introduction

Bounded model-checking (BMC) is a model-checking method that has gained popularity due to the inability of BDD-based symbolic model-checkers to handle large designs. In classical BMC [3], one tries to find a bug of bounded length k. If a bug is not found, the bound is increased until either a bug is found or a pre-determined *completeness threshold* [4] is reached. If the threshold has been reached but no bug has been found, it is concluded that the formula holds in the model. In practice, the threshold is rarely reached, but recent works (e.g., [6]) also describe techniques for SAT-based temporal induction which can be used to prove formulas without reaching the completeness threshold.

BMC has mostly been restricted to linear-time specifications, with a few exceptions ([14], [16]). Most encodings for linear-time logic have a common form, a conjunction of two formulas: one encodes a path starting from an initial state of the model, and the second is property-dependent and constrains the path to be a counter-example to the property being checked.

Bounded model-checking for branching-time logic is a somewhat thornier problem, because it is usually not known in advance what exact shape the counter-example will take. The works that extended the BMC paradigm to universal branching-time logic dealt with this problem in different ways: [14] encodes a property-dependent number of bounded paths, either lasso-shaped or finite, and constrains them to represent a counter-example; [16] encodes a bounded

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 410–424, 2007.
© Springer-Verlag Berlin Heidelberg 2007

proof-tree for the negation of the formula, using the local proof rules of [15] for μ-calculus. Both approaches assume a worst-case scenario in the construction of the tree-like structure they encode, with the result that many more states may be encoded than are necessary for the counter-example. For example, to disprove a formula of the form $\psi_1 \vee \psi_2$, both approaches will encode two separate tree-like structures — one for disproving ψ_1 and one for disproving ψ_2. In practice, there may exist counter-examples for both ψ_1 and ψ_2 which share many model states. The counter-examples returned in [14] and [16] are therefore not minimal in the number of states they contain.

In this paper we suggest a new approach to bounded model-checking for universal branching-time logic wherein we encode exactly the states that are necessary for the counter-example. Unlike [14] and [16], we make no assumptions about the structure of the counter-example; we encode k states, where k is the bound, and allow the SAT solver to choose both the states and the edges of the model that will comprise the counter-example. We use the *local constraints* of Namjoshi's proof system for μ-calculus [13] to ensure that the structure represented by the states is a counter-example to the formula being checked. Our approach ensures a *minimal* counter-example, and it avoids representing the same model-state more than once.

We present an encoding for proving existential μ-calculus properties (or falsifying universal properties), using alternating parity tree automata as the specification mechanism. We also present a simplified variation of the encoding for alternation-free existential μ-calculus. The encoding for full existential μ-calculus uses roughly $O(|Q| \cdot k \log k)$ variables, where k is the bound and $|Q|$ is the number of automaton states. The simpler encoding for alternation-free μ-calculus is less compact, requiring $O(|Q| \cdot k^2)$ variables, but it is more explicit and performs better than the more general encoding. The simplified encoding can be extended to handle fairness constraints while still requiring roughly $O(|Q| \cdot k^2)$ variables.

We also describe a dynamic termination criterion which can be used to halt the bounded model-checking by determining that no counter-example comprising $k' > k$ states can exist, where k is the current bound. The idea is similar to the criterion suggested in [16]: we attempt to identify situations where the structure encoded cannot be extended by adding new states (that is, increasing the bound). However, our implementation is quite different, due to the difference between the encodings. Using the termination criterion it is possible to prove and disprove both existential and universal formulas. As is typical for bounded model-checking, the algorithm performs better when proving existential formulas or disproving universal ones than when proving universal formulas or disproving existential ones.

Finally, we present experimental results for the branching-time logic ACTL. Our experiments show that our approach is a good complement to the encoding of [16], especially for complex formulas with a large nesting depth, where our encodings can often disprove formulas that cannot be disproven by the encoding of [16]. Deeply-nested formulas are generated during automatic translation to ACTL from a high-level specification language, e.g., PSL [1], where complex regular expressions translate into deeply-nested ACTL formulas.

2 Preliminaries

2.1 Alternating Parity Tree Automata

A *normal-form alternating parity tree automaton* [17] is a tuple $A = (AP, Q, q_0, \delta, \Omega)$, where AP is a set of atomic propositions, Q is the set of automaton states, and q_0 is the initial state; δ is an alternating transition relation, assigning to each state $q \in Q$ a transition of the form $q_1 \vee q_2$, $q_1 \wedge q_2$, $\Diamond q_1$, $\Box q_1$, p or $\neg p$, where $q_1, q_2 \in Q$ and $p \in AP$; and finally, $\Omega : Q \to \mathbb{N}$ is a partial priority function which represents a *parity acceptance condition* (in [13], the acceptance condition is represented as a partition of Q instead of a priority function). For an automaton state $q \in Q$, $\Omega(q)$ will be called the *priority* of q. The automata we will deal with contain no cycles of priorityless states. We will say that an infinite sequence $\pi = q_0 q_1 \ldots \in Q^\omega$ *satisfies* Ω if the lowest priority $\Omega(q)$ of a state q that has a priority and appears infinitely often in π is even. (An alternative definition, e.g. in [17], requires that the infinite sequence also have an infinite number of states for which the priority is defined. However, automata constructed using the standard translation from μ-calculus have at least one state that has a priority on every cycle in the automaton; all infinite sequences contain an infinite number of states that have a priority. We assume this property in the automaton representing the specification we check.)

Universal μ-calculus properties [11] can be expressed by automata that do not have $\Diamond$-transitions. We will refer to such an automaton as a $\Box$-automaton. Similarly, existential μ-calculus properties can be expressed by $\Diamond$-automata, which have no $\Box$-transitions.

Tree automata run over labeled trees. A *labeled tree* is a pair $T = (N, L)$ where $N \subseteq \mathbb{N}^+$ is a prefix-closed set of tree nodes and $L : N \to 2^{AP}$ is a labeling function. The node ε (the empty word) is the tree root, and there is an edge from node n_1 to node n_2 iff $n_2 = n_1 \cdot i$ for some $i \in \mathbb{N}$. We will use $Succ(n)$ to denote the targets of edges outgoing from n; that is, $Succ(n) = \{n \cdot i \mid i \in \mathbb{N}\} \cap N$.

The acceptance of a tree by an automaton is defined in terms of a two-player infinite game. The game positions are $N \times Q$, and the initial position is (ε, q_0). The player who owns the position (n, q) and the moves available to that player are determined according to δ: player I owns positions (n, q) such that $\delta(q) = q_1 \vee q_2$ or $\delta(q) = \Diamond q_1$; player II owns positions (n, q) such that $\delta(q) = q_1 \wedge q_2$ or $\delta(q) = \Box q_1$. If $\delta(q) = q_1 \vee q_2$ or $\delta(q) = q_1 \wedge q_2$, the available moves are (n, q_1) and (n, q_2); if $\delta(q) = \Diamond q_1$ or $\delta(q) = \Box q_1$, the available moves are (m, q_1) for all $m \in Succ(n)$. A position (n, q) is winning for player I if $\delta(q) = p$ and $p \in L(n)$ or $\delta(q) = \neg p$ and $p \notin L(n)$, and winning for player II if $\delta(q) = p$ and $p \notin L(n)$ or $\delta(q) = \neg p$ and $p \in L(n)$. A play is winning for player I if it is finite and ends in a position that is winning for player I, or if it is infinite and satisfies Ω; otherwise, the play is winning for player II. Strategies are defined as usual: a *strategy* for player x is a partial function mapping finite sequences of configurations to a choice of the next configuration at every position owned by player x. A play is said to be *according to* a strategy for player x if every choice made by player x in the play conforms to the strategy. A strategy is *winning* for player x if player x wins any play she plays according to the strategy. We will say that an automaton A *accepts* a tree T iff player I has a winning strategy for the game thus described.

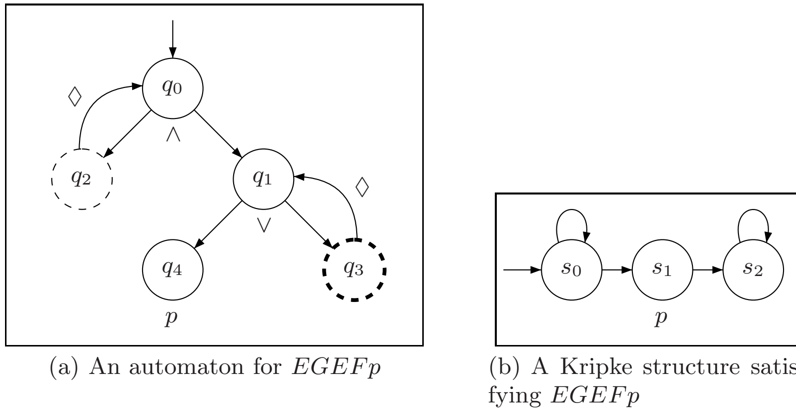
2.2 Kripke Structures

To represent finite-state programs, we use *Kripke structures*. A Kripke structure is a tuple $M = (S, s_0, R, L)$, where S is the set of states, s_0 is the initial state, $R \subseteq S \times S$ is a total transition relation and $L : S \to 2^{AP}$ is a labeling function.

Given a Kripke structure M and an alternating parity tree automaton A, we will say that M *satisfies* A iff the computation tree of M is accepted by A. We will say that a model state $s \in S$ satisfies an automaton state q if the computation tree starting from s is accepted by the automaton A' which is identical to A except that q is the initial state of A'.

Theorem 1 ([9], [17]). *For every mu-calculus formula φ there exists an alternating parity tree automaton A_φ such that for every Kripke structure M, M satisfies A_φ iff $M \models \varphi$.*

Example 1. Consider the automaton $A = (\{p\}, \{q_0, q_1, q_2, q_3, q_4\}, q_0, \delta, \Omega)$ shown in Fig. 1, where the type of the transition is indicated below each state or on the relevant edge, and Ω is defined only for q_2 and q_3 (shown dashed), which have $\Omega(q_2) = 2$ and $\Omega(q_3) = 1$.



(a) An automaton for $EGEFp$

(b) A Kripke structure satisfying $EGEFp$

Fig. 1. An example automaton and Kripke structure

The automaton is equivalent to the property "there exists a path on which from every state, p is reachable", expressed as $EGEFp$ in the temporal logic ECTL. State q_0 stands for $EGEFp$, and state q_2 stands for $EXEGEFp$, "there exists a successor that satisfies $EGEFp$". Similarly, q_1 stands for EFp and q_3 for $EXEFp$, and state q_4 stands for p. The odd priority for q_3 requires that a winning play only pass through q_3 a finite number of times (since there is no state with a lower even priority), so that eventually the play must transition to q_4, which requires that p be satisfied at the current model state.

2.3 Namjoshi-Style Temporal Proofs

Several proof systems have been suggested for the model checking problem of μ-calculus. Here we focus on the proof system presented by Namjoshi in [13]. The feature which makes it useful for our purposes is that its conditions are local: to verify that a proof is valid, one need only check a series of *local* conditions, which refer at most to a state's immediate successors in the Kripke structure. Other proof systems, such as Stirling's proof rules [15], keep track of states visited along the current proof branch; in Namjoshi's system such "book-keeping" is not required, and ranks are used instead.

In [13], the proof system is presented for automata where the priority function is full. We will present the system from [13] and then explain how it can be extended to the case where the priority function is a partial function.

Let $M = (S, s_0, R, L)$ be a Kripke structure, and let $A = (AP, Q, q_0, \delta, \Omega)$ be a normal-form alternating parity tree automaton with Ω defined for all $q \in Q$. To show that M satisfies A, one must exhibit: (i) for each automaton state $q \in Q$, a predicate I_q, which, intuitively, characterises the set of model states which satisfy q; (ii) non-empty, well-founded sets $W_1, \ldots, W_m$, where m is the number of odd priorities assigned by Ω to states from Q, and pre-orders $\prec_1, \ldots, \prec_m$; (iii) for each automaton state $q \in Q$, a partial rank function $\rho_q : S \to (W, \prec)$, where $W = W_1 \times \ldots \times W_m$ and $\prec$ is the lexicographic order induced by $\prec_1, \ldots, \prec_m$ on W. In this paper, we will assume without loss of generality that $W = \mathbb{N}^k$, with $\prec_i$ the standard order $<$ over $\mathbb{N}$. We will henceforth omit W and simply write $\Pi = (I, \rho)$, where $I = \{I_q \mid q \in Q\}$ is the set of invariants and $\rho = \{\rho_q \mid q \in Q\}$ is the set of rank functions.

We use Invariance and Progress obligations to ensure that player I has a winning strategy for the game induced by A on the computation tree of M: the obligation for automaton states q with $\vee$- or $\Diamond$-transitions represents the move player I must make in positions (s, q) owned by her. Obligations for states q with $\wedge$- or $\Box$-transitions ensure that no matter which move player II makes from a position (n, q), player I will have a winning strategy from the resulting position. In the case of an infinite play, we use ranks to ensure that the play satisfies Ω.

Intuitively, the rank $\rho_q(s)$ represents a commitment regarding the number of times we may pass through states with each odd priority before passing through a state with lower priority in a play from position (n, q), where n is a tree node corresponding to model state s. For example, coordinate 0 of the rank counts the number of times we may pass through states with priority 1 before passing through a state with priority 0. Each time we pass through a state with priority $2i + 1$, coordinates 0 through i decrease lexicographically, and can only increase again when passing through a state q with $\Omega(q) < 2i+1$. A play according to the strategy induced by the invariants can only pass through a state with an odd priority $2i+1$ a finite number of times before coordinates $0, \ldots, i$ of the rank reach zero, and then we must pass through a state with lower priority. The lowest priority occurring infinitely often in the play must be even, and player I wins.

This notion is captured by an order $\lhd_q$ over $\mathbb{N}^k$, defined for each $q \in Q$ as follows: $(x_0, \ldots, x_{m-1}) \lhd_q (y_0, \ldots, y_{m-1})$ iff $\Omega(q) = 0$, or $\Omega(q) = 2i, i > 0$

and $(x_0, \ldots, x_i, 0, \ldots, 0) \preceq (y_0, \ldots, y_i, 0, \ldots, 0)$, or $\Omega(q) = 2i+1$ and $(x_0, \ldots, x_i, 0, \ldots, 0) \prec (y_0, \ldots, y_i, 0, \ldots, 0)$. Note that coordinates $i, \ldots, m-1$ are unconstrained when passing through a state with priority $\Omega(q) < 2i+1$, but when passing through states with $\Omega(q) \geq 2i+1$, coordinate i may not increase. When passing through a state with priority $2i+1$, coordinates $0, \ldots, i$ must decrease in lexicographic order.

A valid proof must satisfy the following requirements.

- Consistency: for each $q \in Q$ and $s \in I_q$, $\rho_q(s)$ is defined.
- Initiality: $s_0 \in I_{q_0}$.
- Invariance and Progress: for each $q \in Q$ and $s \in I_q$:
  - If $\delta(q) = p$ then $p \in L(s)$.
  - If $\delta(q) = \neg p$ then $p \notin L(s)$.
  - If $\delta(q) = q_1 \vee q_2$, then either $s \in I_{q_1}$ and $\rho_{q_1}(s) \lhd_q \rho_q(s)$, or $s \in I_{q_2}$ and $\rho_{q_2}(s) \lhd_q \rho_q(s)$.
  - If $\delta(q) = q_1 \wedge q_2$, then $s \in I_{q_1}$ and $\rho_{q_1}(s) \lhd_q \rho_q(s)$, and also $s \in I_{q_2}$ and $\rho_{q_2}(s) \lhd_q \rho_q(s)$.
  - If $\delta(q) = \Diamond q_1$, then there exists $t \in S$ such that $(s,t) \in R$ and $t \in I_{q_1}$ and $\rho_{q_1}(t) \lhd_q \rho_q(s)$.
  - If $\delta(q) = \Box q_1$, then for all $t \in S$ such that $(s,t) \in R$, $t \in I_{q_1}$ and $\rho_{q_1}(t) \lhd_q \rho_q(s)$.

Theorem 2 ([13]). *For every Kripke structure M and automaton A with a full priority function, M satisfies A iff there exists a Namjoshi-style proof showing that M satisfies A.*

Automata resulting from the standard translation for μ-calculus have a *partial* priority function, with infinitely many priorities on every infinite path. For such automata, we would still like the $\lhd_q$ relation to enforce the parity acceptance condition, which now concerns only states that have a priority. Define an extension $\unlhd_q$ as follows: $x \unlhd_q y$ iff $\Omega(q)$ is defined and $x \lhd_q y$, or $\Omega(q)$ is undefined and $x = y$. The idea is that priorityless states should simply preserve the rank, keeping it unchanged until the next time we pass through a state with a priority.

Lemma 1. *The proof system obtained by replacing $\lhd_q$ with $\unlhd_q$ is sound and complete for all automata with no cycles of priorityless states.*

Example 2. Consider the automaton and structure shown in Fig. 1. A proof showing that M satisfies A is given by $\Pi = (I, \rho)$, where $I_{q_0} = I_{q_2} = \{s_0\}$ (states that satisfy $EGEFp$ and $EXEGEFp$), $I_{q_1} = I_{q_3} = \{s_0, s_1\}$ (states that satisfy EFp and $EXEFp$), and $I_{q_4} = \{s_1\}$ (the only state labeled with p). The ranks in Π have a single coordinate, representing the length of a path to a state satisfying p. The relation $\unlhd_q$ is $=$ for all $q \neq q_3$, and for q_3, $\unlhd_{q_3}$ is $<$. An assignment of ranks that satisfies the proof obligations is $\rho_q(s_0) = 1$, $\rho_q(s_1) = 0$ and $\rho_q(s_2)$ undefined for all $q \in Q$. Note that since we attached the decrease in rank to an automaton state with a $\Diamond$-transition, all the automaton states agree on the rank assigned to each model state. This can be done for all ECTL formulas.

2.4 Notation and Terminology

We will let $Q_\Diamond$ denote the set of automaton states q with a transition $\delta(q) = \Diamond q_1$. For automaton states $q \in Q_\Diamond$ and model states $s \in I_q$, it will sometimes be useful to identify the model state (or one of the model states) $t \in S$ which serves to satisfy the Invariance and Progress obligation for s and q in a proof Π. We will refer to t as a *proof successor* for s as required by q.

3 The Encodings

We present two encodings to SAT for model checking existential alternating parity tree automata (with no $\Box$-transitions). In both encodings, we search for a counter-example of bounded size k in the Kripke structure, where k is the number of states in the counter-example; the counter-example is represented as an arbitrary graph of unknown structure, and each state in the graph must satisfy certain local obligations to ensure that the graph constitutes a counter-example for the formula in question. The structure of the graph is determined by the local obligations of each state.

3.1 Encoding Namjoshi-Style Proof Obligations

The first encoding we present is a direct translation of the proof obligations of a Namjoshi-style temporal proof to Boolean constraints.

Let $M = (S, s_0, R, L)$ be a Kripke structure. We assume that the initial state and the transition relations are given in the form of propositional formulas I and R respectively, and that the state space S is represented by $\{0,1\}^n$. Also, for each atomic proposition $p \in AP$, we assume a propositional formula $\mathsf{L_p}$ which is true exactly for states $s \in S$ such that $p \in L(s)$. Let $A = (AP, Q, q_0, \delta, \Omega)$ be a $\Diamond$-automaton. To encode the requirements on ranks, we use a set of propositional formulas $\mathsf{LT_q}$ for all $q \in Q$, such that $\mathsf{LT}_q(\sigma_1, \sigma_2)$ holds iff $\sigma_1 \otimes_q \sigma_2$.

The encoding uses the following variables.

- $u_0, \ldots, u_{k-1}$: vectors representing model states. Each vector comprises n bits.
- x_i^q for each $i = 0, \ldots, k-1$ and $q \in Q$: an indicator variable for the fact that the state assigned to u_i satisfies q.
- ρ_i^q for each $i = 0, \ldots, k-1$: a vector representing the rank $\rho_q(s)$ assigned to u_i by q.

Each rank vector ρ_i^q has m coordinates, where m is the number of odd priorities assigned by Ω to automaton states, and each coordinate j comprises $\log|Q|k$ bits. This is sufficient because if there exists an infinite winning play for player I, then there exists a play that does not pass through an odd-priority state twice before passing through a state with a lower priority. The total number of variables used in the encoding is $O(nk + k|Q| + |Q|mk \log |Q|k)$.

The obligations for a state u_i and an automaton state q are encoded as a Boolean formula of the form $x_i^q \rightarrow \langle\langle \delta(q) \rangle\rangle_i$, where $\langle\langle \delta(q) \rangle\rangle_i$ is defined as follows.

$$\begin{aligned}
&\langle\langle p \rangle\rangle_i = \mathsf{L_p}(u_i) \\
&\langle\langle \neg p \rangle\rangle_i = \neg \mathsf{L_p}(u_i) \\
&\langle\langle q_1 \wedge q_2 \rangle\rangle_i = x_i^{q_1} \wedge \mathsf{LT_q}(\rho_i^{q_1}, \rho_i^q) \wedge x_i^{q_2} \wedge \mathsf{LT_q}(\rho_i^{q_2}, \rho_i^q) \\
&\langle\langle q_1 \vee q_2 \rangle\rangle_i = (x_i^{q_1} \wedge \mathsf{LT_q}(\rho_i^{q_1}, \rho_i^q) \vee x_i^{q_2}) \wedge (\mathsf{LT_q}(\rho_i^{q_2}, \rho_i^q)) \\
&\langle\langle \Diamond q_1 \rangle\rangle_i = \bigvee_{j=0}^{k-1} \left(\mathsf{R}(u_i, u_j) \wedge x_j^{q_1} \wedge \mathsf{LT_q}(\rho_j^{q_1}, \rho_i^q) \right)
\end{aligned}$$

It is possible to eliminate the indicators x_i^q when $\delta(q) = q_1 \wedge q_2$ or $\delta(q) = q_1 \vee q_2$ by substituting the constraints generated for these formulas anywhere that the indicator appears.

To represent the Initiality requirement, we add the constraint $\mathsf{I}(u_0) \wedge x_0^{q_0}$. The resulting formula is given by

$$\mathsf{PRF}^1_{M,A,k} = \mathsf{I}(u_0) \wedge x_0^{q_0} \wedge \bigwedge_{i=0}^{k-1} \bigwedge_{q \in Q} x_i^q \rightarrow \langle\langle \delta(q) \rangle\rangle_i$$

Optimizating the Encoding. The encoding presented above is naive, and can be improved in several ways.

First, the encoding suffers from symmetry, which has an adverse effect on the performance of most SAT solvers; the model states $u_1, \ldots, u_{k-1}$ are interchangeable, and the SAT solver is forced to consider many equivalent permutations of the same counter-example before eliminating it. We have found that performance is greatly improved when we break the symmetry by ordering the states $u_1, \ldots, u_{k-1}$, obtaining the formula

$$\mathsf{PRF}^{1'}_{M,A,k} = \mathsf{PRF}^1_{M,A,k} \wedge \bigwedge_{i=1}^{k-2} u_i < u_{i+1}$$

where "<" is implemented as the lexicographic order on binary vectors. (u_0 is excluded from the ordering as it is the only state that serves a "special" role: it must be an initial state, and we cannot require that it be smaller than all the other states.)

The way ranks are handled in the encoding can also be improved. By analyzing the structure of the automaton, we can identify sets of automaton states that can share the same rank vectors. For example, if $\delta(q) = q_1 \wedge q_2$ and q does not have a priority, then in a valid proof, $\rho_q(s) = \rho_{q_1}(s) = \rho_{q_2}(s)$ for any model state s. There is no need to encode the rank separately, and instead of having three vectors ρ_i^q, $\rho_i^{q_1}$ and $\rho_i^{q_2}$ all three automaton states can "share" a vector $\rho_i^{\{q,q_1,q_2\}}$. This also simplifies the constraint $\langle\langle \delta(q) \rangle\rangle_i$, because now we can remove the $\mathsf{LT_q}$ constraint; it is implicit in using the same rank vector.

In particular, for ECTL formulas it is possible to construct automata where *all* the automaton states share a single rank vector ρ_i^Q, greatly simplifying the encoding. For lack of space, we do not elaborate.

Encoding Successor States Explicitly. In the constraints generated for states $q \in Q_\Diamond$, the transition relation appears k times each. Since the transition relation is often complicated, it is desirable to decrease the number of times it appears. We can do so at the cost of increasing the number of variables, by encoding proof-successors explicitly. For each $q \in Q_\Diamond$ and $i = 0, \ldots, k-1$, we will assign a vector t_i^q to represent the successor required by q if the state assigned to u_i is in I_q. Since we are searching for a proof of size k, t_i^q will be constrained to be one of the states $u_0, \ldots, u_{k-1}$; also, if $t_i^q = u_j$, then we require u_j to be in the appropriate invariant I_{q_1}, where $\delta(q) = \Diamond q_1$, and its rank must behave appropriately. The constraint can now be written as

$$\langle\!\langle \Diamond q_1 \rangle\!\rangle_i = \mathsf{R}(u_i, t_i^q) \wedge \bigvee_{j=0}^{k-1} \left(t_i^q = u_j \wedge x_j^{q_1} \wedge \mathsf{LT_q}(\rho_j^{q_1}, \rho_i^q) \right)$$

In the new encoding, the transition relation appears $k \cdot |Q_\Diamond|$ times instead of $k^2 \cdot |Q_\Diamond|$ times as before. We will also have further use for the information we gain by explicitly encoding proof successors in constructing a dynamic completeness criterion (Section 4).

3.2 Eliminating the Use of Ranks

Although the previous encoding uses a rather small number of variables, which increases as $O(k \log k)$ with the bound k, the use of ranks can be SAT-unfriendly. The second encoding we present is similar to the first, but using ideas from [10] and [8], we eliminate the use of ranks. The idea is that instead of directly encoding the rank $\rho_q(s)$ for states $s \in I_q$, we will store a subset I_q^σ for each rank σ, containing states $s \in I_q$ that have $\rho_q(s) \leq \sigma$. In the encoding, we will unroll the proof obligations once for each such invariant; when a decrease in rank is called for, we will use the subset that represents the lower rank. This encoding becomes inefficient when the ranks have more than one coordinate, and we will restrict attention to automata that only assign the priorities $1, 2$. Such automata require a single coordinate in the rank, and includes the alternation-free fragment of μ-calculus.

The encoding will use the following variables.

- $u_0, \ldots, u_{k-1}$: vectors representing model states.
- $x_i^{q,t}$ for each $i = 0, \ldots, k-1$, $q \in Q$ and $t = 0, \ldots, mk$ where m is the number of odd-priority automaton states: an indicator variable for the fact that the state assigned to u_i satisfies q and has rank no greater than t.

Our obligations will now take the form $x_i^{q,t} \rightarrow \langle\!\langle \delta(q) \rangle\!\rangle_i^t$ for $t > 0$, where $\langle\!\langle \delta(q) \rangle\!\rangle_i^t$ is defined by

$$\langle\!\langle p \rangle\!\rangle_i^t = \mathsf{L_p}(u_i)$$
$$\langle\!\langle \neg p \rangle\!\rangle_i^t = \neg \mathsf{L_p}(u_i)$$
$$\langle\!\langle q_1 \wedge q_2 \rangle\!\rangle_i^t = x_i^{q_1, r_q(t)} \wedge x_i^{q_2, r_q(t)}$$

$$\langle\!\langle q_1 \vee q_2 \rangle\!\rangle_i^t = x_i^{q_1, r_q(t)} \vee x_i^{q_2, r_q(t)}$$

$$\langle\!\langle \Diamond q_1 \rangle\!\rangle_i^t = \bigvee_{j=0}^{k-1} \left(\mathsf{R}(u_i, u_j) \wedge x_j^{q_1, r_q(t)} \right)$$

where $r_q(t) = t$ if $\Omega(q) = 2$ or $\Omega(q)$ is not defined, and $r_q(t) = t - 1$ if $\Omega(q) = 1$. For $t = 0$, we will constrain $x_i^{q,0} \to$ **false** (or just substitute **false** where the indicator appears).

The optimizations for the previous encoding can be applied here as well. The encoding can be further optimized for ECTL by exploiting the weak structure of the automata along the lines of [8]. Also, although ECTL formulas with fairness cannot always be described by automata that only assign the priorities 1 and 2, it is not difficult to extend the encoding for ECTL to handle fairness, by imitating the way a symbolic model-checker for ECTL handles fairness constraints.

Theorem 3. *Given a Kripke structure M and a $\Diamond$-automaton A, the formulas generated by the encodings of Sections 3.1 and 3.2 are satisfiable iff there exists a proof $\Pi = (I, \rho)$ showing that M satisfies A that contains k states; that is, $\left|\bigcup_{q \in Q} I_q\right| = k$.*

4 A Dynamic Completeness Criterion

Both encodings presented in the previous section provide a way to determine when a Kripke structure does *not* satisfy a $\Box$-automaton A: construct the complement $A_\neg$ for A, choose a bound k, and if a SAT solver returns a satisfying assignment for $\mathsf{PRF}_{M,A_\neg,k}$ then M does not satisfy A. For some formulas there is a known *completeness threshold*, which is a bound on the number of states (usually the length of a path) necessary to disprove the formula. However, the completeness threshold usually depends on both the formula and the model, and in practice it is difficult to compute. Following [16], we are interested in a *dynamic completeness criterion*: a formula $\mathsf{CMP}_{M,A,k}$ that is satisfiable while there is still hope of finding a counter-example, and that becomes unsatisfiable when there is none. Essentially, $\mathsf{CMP}_{M,A,k}$ should encode the fact that it is possible to arrange k states so that they form a "beginning" of a proof that might be extended into a valid proof by adding more states.

To see how $\mathsf{CMP}_{M,A,k}$ should be constructed, consider the situation where we have not found a proof when the bound is k, but there exists a proof Π with $k' > k$ states. Now let Π' be an invalid proof constructed by taking k states of Π, including s_0, and using the invariants and ranks of Π restricted to these k states. It is easy to see that in Π', the Initiality and Consistency obligations are satisfied. For automaton states $q \in Q \setminus Q_\Diamond$, the Invariance and Progress requirements are satisfied as well. However, Π' must violate some proof obligations, because there is no valid proof of size k. The obligations that are violated are Invariance and Progress obligations for states $q \in Q_\Diamond$ that have $\Diamond$-transitions, and the reason they are violated in Π' is that they rely on some of the states that appear in Π but not in Π'.

We would like to identify situations where no proof fragment Π' matching this description exists, and therefore $\mathsf{CMP}_{M,A,k}$ will encode these requirements:

- Initiality.
- For automaton states $q \in Q \setminus Q_\Diamond$, Invariance and Progress.
- For automaton states $q \in Q_\Diamond$, a weakened version of Invariance and Progress, where the successor required by the obligation for q with $\delta(q) = \Diamond q_1$ is not required to be in the invariant for q_1 unless it is one of the states $u_0, \ldots, u_{k-1}$. This allows the successor to be a new unconstrained state, required only to be distinct from the regular proof states.

The weakened Invariance and Progress requirement identifies cases where adding more states to the proof may yield a valid proof. Assuming the encoding of Section 3.1 with proof successors encoded explicitly, the weakened requirement is given by

$$\langle\!\langle \Diamond q_1 \rangle\!\rangle_i^W = \mathsf{R}(u_i, t_i^q) \wedge \bigwedge_{j=0}^{k-1} \left(t_i^q = u_j \rightarrow \left(x_j^{q_1} \wedge \mathsf{LT_q}(\rho_j^{q_1}, \rho_i^q)\right)\right)$$

where t_i^q is the successor required by $q \in Q_\Diamond$ that has $\delta(q) = \Diamond q_1$ for u_i. Violated Invariance and Progress requirements for states that do not have a $\Diamond$-transition cannot be satisfied directly by adding more states, so they are left unchanged: $\langle\!\langle \delta(q) \rangle\!\rangle_i^W = \langle\!\langle \delta(q) \rangle\!\rangle_i$ for all $q \in Q \setminus Q_\Diamond$.

Additionally, we would like to constrain the SAT solver to use the k proof states "constructively", otherwise a satisfying assignment might encode, e.g., k unreachable states, and satisfy all their proof obligations by adding new and unconstrained states. To this end we will require that all k proof states be distinct from each other, and also that each state except s_0 (represented by u_0) be a proof-successor for some state. This constrains satisfying assignments to encode only "proof-reachable" states — states that are reachable from s_0 by a path in which each state is a proof-successor for the state preceding it.

The formula that encodes all these requirements is given by

$$\mathsf{CMP}^1_{M,A,k} = \mathsf{I}(u_0) \wedge x_0^{q_0} \wedge \bigwedge_{i=0}^{k-1} \bigwedge_{q \in Q} x_i^q \rightarrow \langle\!\langle \delta(q) \rangle\!\rangle_i^W \wedge \bigwedge_{i \neq j} u_i \neq u_j \wedge$$
$$\wedge \bigwedge_{i=1}^{k-1} \left(\bigvee_{j=0}^{k-1} \bigvee_{q \in Q_\Diamond} u_i = t_j^q \wedge x_j^q \right)$$

Theorem 4. *If there exists a proof for the fact that M satisfies A with $k' \geq k$ states, then $\mathsf{CMP}^1_{M,A,k}$ is satisfiable.*

The scheme for using the dynamic completeness criterion is shown in Alg. 1, which takes as input a $\Diamond$-automaton A and a structure M. For a $\Box$-automaton one constructs the complement, calls the algorithm and returns the opposite answer.

Algorithm 1. BMC using the dynamic completeness criterion

for $k = 1$ to $|S|$ **do**
  if $\mathsf{PRF}^{1'}_{M,A,k}$ is satisfiable **then**
    return $M \models A$
  else if $\mathsf{CMP}^{1}_{M,A,k}$ is not satisfiable **then**
    return $M \not\models A$
  end if
end for
return $M \not\models A$

5 Related Work

Many BMC encodings have been suggested for linear-time logics of increasing complexity, among them [7], which handles LTL with past, and [10] and [8], which handle all ω-regular properties. In particular, [10] and [8] apply ideas from the world of symbolic model-checking for branching-time logic to BMC for linear-time. Here we apply similar ideas in their original context of model-checking for branching-time logic.

The first BMC scheme for a branching-time logic, ACTL, was suggested in [14]. This scheme works by explicitly encoding a computation tree of depth k which does not satisfy the formula. Instead of encoding a full tree, it bounds the number of paths needed, based on the structure of the formula. The number of paths is exponential in the nesting depth of the formula (in the worst case).

A more general approach was suggested by Wang in [16], in which the existence of a bounded-depth proof for the negation of a universal μ-calculus property is encoded as a SAT problem. The approach is somewhat similar to ours, but [16] relies on Stirling's local proof rules [15], which keep track of all the states visited along each proof branch, and require directly that the branch be acyclic (for a least fixpoint) or cyclic (for a greatest fixpoint). Different branches of the proof share no information, and a single model state can appear many times in the proof tree.

The encoding of [16] is simple and elegant, but encoding the proof as a tree-like structure allows no sharing of information between different branches of the proof. The disadvantage becomes more acute for complicated formulas, where the proof tree contains many nested subgoals, each of which needs to be justified separately. For disjunction, the encoding of [16] unrolls two separate subproofs, even though only one of the goals needs to be satisfied; for example, to show that $EFp \vee EFq$ holds ("there exists a path on which we eventually reach p or a path on which we eventually reach q"), two separate paths will be unrolled. In contrast, our encoding enables maximal sharing of information: a model state need never be encoded more than once, and information about the automaton states it satisfies can be used to justify many different subgoals.

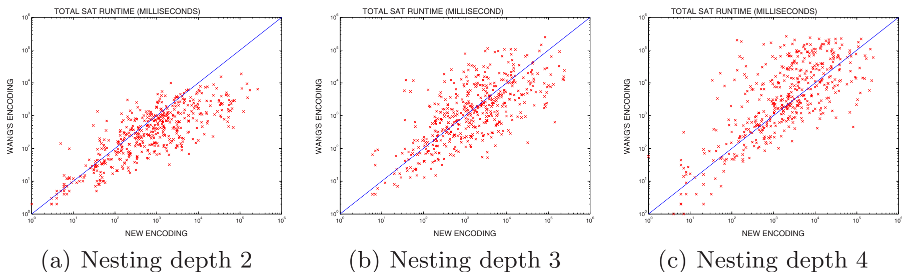
The number of variables used in the encoding of [16] increases exponentially with the bound in the worst case for universal μ-calculus properties, and polynomially for ACTL formulas, with the exponent being the nesting depth of temporal operators in the formula; when the bound increases by one, a full layer

is added to the proof tree. Our encoding allows finer control. Although in the worst case it may need to encode the same number of states as the encoding of [16], it can often terminate with a smaller bound and smaller counter-examples. It is not possible to obtain a minimal counter-example from a satisfying assignment to the formula generated in [16], and it is also not possible to determine which parts of the structure returned are relevant to the proof (for example, in the case of conjunction, it is not possible to tell which conjunct was disproved). The bounds used in our encoding and in [16] are incomparable: our bound represents the exact number of states in the counter-example, while the bound in [16] represents the depth of the proof tree. Either method may terminate with a smaller bound than the other.

In [12] it is shown how to solve parity games through a reduction to SAT. This work is closely related to our own, since parity games are equivalent to μ-calculus and to alternating parity tree automata; [12] also uses ranks in a manner similar to ours. However, the encoding of [12] assumes that an explicit representation of the gameboard, which is difficult to compute for the model checking problem, since it means computing the product of the model and the automaton. Also, the encoding represents the entire gameboard at once, and therefore it does not lend itself immediately to bounded model checking.

6 Experimental Results

We implemented an ACTL version of our encodings and Wang's encoding from [16] in the NuSMV2 framework [2], and tested their performance on a 3GHz Pentium 4 with 4GB memory, using the ZChaff SAT solver. We used random Kripke structures with 100 states each, and random formulas that were not satisfied in the models. The formulas were of nesting depth 2 – 5 of the temporal operators AF, AG and AU. We used a maximal bound of 20, which was never reached in our experiments, and a timeout of 5 minutes. The simplified encoding from Section 3.2 greatly outperformed the general encoding of Section 3.1, probably owing to our naive implementation of ranks, and we present results only for the simplified encoding. Our results for 500 formulas for each nesting depth from 2 to 4 are summarized in Fig. 2 and Table 1.



(a) Nesting depth 2 (b) Nesting depth 3 (c) Nesting depth 4

Fig. 2. Total SAT runtimes for disproving ACTL formulas of nesting depth 2 – 4

Table 1. Success rate in disproving ACTL formulas of nesting depth 2 – 5

	Nesting depth 2	Nesting depth 3	Nesting depth 4	Nesting depth 5
Wang's encoding	**100%**	**99%**	84%	76%
New encoding	98%	97%	**95%**	**94%**

For nesting depth 2, the encoding of [16] performs better than our encoding (Fig. 2(a)). For nesting depth 3 (Fig. 2(b)) the encodings perform roughly the same, and for nesting depth 4 or greater our encoding performs better than the encoding of [16] (Fig. 2(c)). The counter-examples found by our encoding were generally very small (10 states or less), and the depth of the proof tree encoded in Wang's encoding was often larger. The counter-examples returned by Wang's encoding were larger by an order of magnitude than the examples returned by our encoding for all nesting depths.

7 Conclusion

We have presented a novel approach to bounded model-checking for branching-time logics. We showed two encodings, together with a dynamic termination criterion, which can be used to both prove and disprove specifications in universal or existential branching-time logic. Our experimental results show that for ACTL formulas with a large nesting depth, our encodings perform better than the previous encoding of Wang. We believe that these results will extend to ACTL with fairness and to general μ-calculus formulas.

The approach presented here is applicable to many logics, from ACTL to μ-calculus, and can be extended to use different types of automata as specifications, using ranking functions to represent different acceptance conditions. The use of ranks can also be applied to BMC in linear-time logics, for example by modifying the encoding of [8] for weak alternating Büchi word automata, resulting in encodings that use fewer variables; however, it is not clear that performance will be improved.

The formulas generated in our encodings, and particularly the dynamic completeness criterion, share most of their constraints with the formulas generated in previous iterations. This makes them suitable for incremental SAT, where conflict clauses learned in previous calls to the SAT solver are re-used to help solve the next instance. Performance may also be improved by using encodings of ranks optimized for SAT, developed in the context of termination checking (e.g., [5]).

References

1. Accellera. PSL Reference Manual v1.1 (2004)
2. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)

3. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19(1), 7–34 (2001)
4. Clarke, E., Kroening, D., Strichman, O., Ouaknine, J.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
5. Codish, M., Lagoon, V., Stuckey, P.J.: Solving partial order constraints for LPO termination. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 4–18. Springer, Heidelberg (2006)
6. Eén, N., Sörensson, N.: Temporal induction by incremental sat solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
7. Heljanko, K., Junttila, T., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
8. Heljanko, K., Junttila, T.A., Keinänen, M., Lange, M., Latvala, T.: Bounded model checking for weak alternating büchi automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 95–108. Springer, Heidelberg (2006)
9. Janin, D., Walukiewicz, I.: Automata for the mu-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)
10. Jehle, M., Johannsen, J., Lange, M., Rachinsky, N.: Bounded model checking for all regular properties. In: Biere, A., Strichman, O. (eds.) BMC 2005. Proc. 3rd Int. Workshop on Bounded Model Checking. Electr. Notes in Theor. Comp. Sc, vol. 144, pp. 3–18. Elsevier, Amsterdam (2005)
11. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)
12. Lange, M.: Solving parity games by a reduction to SAT. In: Majumdar, R., Jurdziński, M. (eds.) GDV 2005 (2005)
13. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
14. Penczek, W., Wozna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. Fundam. Inf. 51(1), 135–156 (2002)
15. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theor. Comput. Sci. 89(1), 161–177 (1991)
16. Wang, B.Y.: Proving $\forall\mu$-calculus properties with SAT-based model checking. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 113–127. Springer, Heidelberg (2005)
17. Wilke, T.: Alternating tree automata, parity games, and modal μ-calculus. Bull. Soc. Math. Belg. 8(2) (2001)

Exact State Set Representations in the Verification of Linear Hybrid Systems with Large Discrete State Space*

Werner Damm[2,3], Stefan Disch[1], Hardi Hungar[3], Swen Jacobs[4], Jun Pang[2], Florian Pigorsch[1], Christoph Scholl[1], Uwe Waldmann[4], and Boris Wirtz[2]

[1] Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg, Germany
[2] Carl von Ossietzky Universität Oldenburg
Ammerländer Heerstraße 114-118, 26111 Oldenburg, Germany
[3] OFFIS e.V., Escherweg 2, 26121 Oldenburg, Germany
[4] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany

Abstract. We propose algorithms significantly extending the limits for maintaining exact representations in the verification of linear hybrid systems with large discrete state spaces. We use AND-Inverter Graphs (AIGs) extended with linear constraints (LinAIGs) as symbolic representation of the hybrid state space, and show how methods for maintaining compactness of AIGs can be lifted to support model-checking of linear hybrid systems with large discrete state spaces. This builds on a novel approach for eliminating sets of redundant constraints in such rich hybrid state representations by a suitable exploitation of the capabilities of SMT solvers, which is of independent value beyond the application context studied in this paper. We used a benchmark derived from an Airbus flap control system (containing 2^{20} discrete states) to demonstrate the relevance of the approach.

1 Introduction

We target the verification of safety properties for embedded control applications in the transportation domain. Typical for such applications is a ratio of between 1:5 to 1:10 between the core control algorithms and diagnostic and fault-tolerance measures integrated into the controller, leading to a blow up of the discrete state space against pure control applications often reaching some 10^6 discrete states. As an example, we analyze a model derived from an Airbus flap controller [13], which on top of its control-loop for flap extraction and retraction is performing envelope protection to prevent loads on flaps possibly causing physical ruptures, and offers extensive monitoring of the health of its sub-systems to e. g. react on

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org/`).

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 425–440, 2007.
© Springer-Verlag Berlin Heidelberg 2007

loss of hydraulic pressure, rupture of the transmission shaft, or hardware failures. To prove safety of such controllers, we must combine methods for analyzing the pure control part (typically using linear dynamics for design models serving as reference for subsequent implementation steps) with state-space exploration methods dealing with large discrete state spaces. Such applications are out of reach for existing hybrid verification tools such as CheckMate [26], PHAVer [11], HyTech [15], `d/dt` [5]: while their strength rests in being able to address complex dynamics, they do not scale in the discrete dimension, since modes – the only discrete states considered – are represented explicitly when performing reachability analysis. On the other hand, hardware verification tools such as SMV [20] and VIS [27] scale to extremely large discrete systems, but clearly fail to be applicable to systems with continuous dynamics. To achieve a compact representation of such hybrid state-spaces, we enrich AND-Inverter Graphs (AIGs) with linear constraints. Previous work [23] demonstrated advantages of AIGs over BDDs for representing large discrete state-spaces compactly, due to their higher robustness in handling broad classes of Boolean functions in exhaustive state-space exploration. We lift methods such as test vector generation and SAT checking to detect equivalent (and thus redundant) nodes to the LinAIG level, providing a suite of heuristics including precise checks for equivalent LinAIG nodes using the SMT solver HySAT [10]. Moreover, we provide efficient methods for detecting and eliminating redundant linear constraints from LinAIGs, which are basically arbitrary boolean combinations of boolean variables and linear constraints. This extends results for eliminating redundant linear constraints from convex polyhedra used by Wang [28] and Frehse [11]. Our approach can be applied to perform backward reachability both for discrete time models (such as reference models for embedded controller implementation) and linear hybrid automata enriched with large discrete state spaces. In the latter case, we exploit the fact that the number of modes of a single controller is typically small (in the order of tens of modes) – this allows us to co-factor the LinAIG representation along modes. For each mode, we use the Loos-Weispfenning quantifier elimination technique for backward evaluation of the symbolic state-space representation along continuous flows. We counteract a worst-case quadratic blow up of linear constraints by tightly integrating redundancy elimination into the quantifier elimination process. Jointly, the presented techniques allow to achieve preciseness while maintaining sufficiently compact representations for the targeted application class.

This paper significantly extends our previous work [7] in adding quantifier elimination and redundancy elimination. The introduction of quantifier elimination was originally motivated by the wish to reduce the diameter of discrete time models. In allowing to fold the effect of large sequences of discretized flows into a single substitution, we accelerate hybrid system verification. This is different from the acceleration by folding hybrid control loops as in [6] which is performed in the world of few discrete states.

The presented methods are orthogonal to and may in the future be combined with abstraction techniques (such as bounding the degree of precision or loosening constraints as in [11]), incorporating robustness [12,8] or slackness [1,2]

in models allowing precise abstractions by finite grids under robustness respectively slackness assumptions, counter-example guided abstraction refinement as in [24,17,25] and techniques such as hybridization [4] for approximate linearization of richer dynamics.

The paper is organized as follows: Sections 2 and 3 give the formal mathematical model and present the backward-reachability algorithm. Sections 4 and 5 are dedicated to flow extrapolation and redundancy elimination. Evaluation results on the flap controller case study are presented in Section 6.

2 System Model

2.1 An Informal Description

This section elaborates on the characteristics of the systems to be analyzed, and motivates particular choices incorporated in the formal definition given in the following section. Our definition of hybrid systems can be seen as an extension of linear hybrid automata (LHA) [14] with a set of discrete variables. The state space is spanned by three classes of variables:

- *Continuous variables* represent sensor values, actuator values, plant states, and other real-valued variables used for the modeling of control-laws and plant dynamics.
- *Mode variables* represent a finite (small) set of *modes*, corresponding to the discrete states of an LHA; each mode is uniquely associated with a constant slope for each of the continuous variables, determining how the continuous valuation evolves over time as long as the system is in the given mode.
- *Discrete variables* code states from state-machines, switches, counters, sanity bits of sensor values, etc., and appear in modeling tools typically as bits, range types, or integer sub-ranges. In this paper we will assume some Boolean encoding of these variables. There are additional discrete input variables to our system.

Our models are closed-loop models without continuous input variables, combining controller and its controlled plant, hence sensors and actuators are internal continuous variables. Interactions of the environment are only possible through discrete input variables, allowing e. g. to select set-points, and to react to protocol messages. Non-deterministic choices are also modeled using discrete input variables. We remark that employing the construction from [3] permits us to extend our procedure to cope with slope sets bounded by constants which allow for non-determinism in plant dynamics, though we will not provide technical details of this extension in this paper.

The system evolves in alternating between continuous flows, in which time passes and only continuous variables are changed according to their slopes associated with the currently active mode of the system, and sequences of discrete transitions, which happen in zero time. Such discrete transitions update both discrete and continuous variables, and finally select the next active mode. All

(discrete) transitions are urgent, eliminating the need to associate state invariants with modes, as in other models of hybrid systems. Discrete inputs enter only in assignments to other discrete variables, i. e. they are disregarded during continuous evolutions. To allow e. g. for periodic sampling of discrete inputs, one can explicitly encode a (continuous) clock within one mode, and test for expiration of the clock-cycle within a transition guard.

2.2 Formal Model

We assume disjoint sets of variables C, D and I. The elements of C are continuous variables, which are interpreted over the reals $\mathbb{R}$. The elements of D and I are discrete variables, where I will be used for inputs. For simplicity, we assume that they are of type boolean and range over the domain $\mathbb{B} = \{0, 1\}$. In the same way we assume that modes are encoded by a set $\mathbf{M} \subseteq \{0, 1\}^l$ of boolean vectors of some fixed length l, leading to a set M of l (boolean) mode variables. We denote a valuation of (a subset of) these variables by $(\mathbf{d}, \mathbf{i}, \mathbf{c}, \mathbf{m})$.

A set of valuations (or states) can be represented symbolically using a suitable (quantifier-free) logic formula over $D \cup I \cup C \cup M$. We denote by $\mathcal{B}(D \cup I)$ the set of boolean expressions over $D \cup I$ and by $\mathcal{B}(M)$ the set of boolean expressions over M. Here we restrict terms over C to the class of linear terms of the form $\sum \alpha_i c_i + \alpha_0$ with rational constants α_i and $c_i \in C$. Predicates are given by the set $\mathcal{L}(C)$ of linear constraints, they have the form $t \sim 0$, where $\sim \in \{=, <, \leq\}$ and t is a linear term. Finally, $\mathcal{P}(D, C)$ is the set of all boolean combinations of variables from D and linear constraints over C.

In the following we use ξ for formulas in $\mathcal{P}(D, C)$, θ for boolean expressions from $\mathcal{B}(M)$, g for boolean expressions from $\mathcal{B}(D \cup I)$, t for linear terms over C, and ℓ for linear constraints over C.

Definition 1 (Syntax of CTHSs). *A continuous-time hybrid system CTHS contains six components:*

- $D = \{d_1, \ldots, d_n\}$ *is a finite set of discrete variables,* $I = \{d_{n+1}, \ldots, d_p\}$, *($p \geq n$) is a finite set of discrete inputs.*
- $C = \{c_1, \ldots, c_f\}$ *is a finite set of continuous variables.*
- $M = \{m_1, \ldots, m_l\}$ *is a finite set of mode variables,* $\mathbf{M} = \{\mathbf{m}_1, \ldots, \mathbf{m}_k\} \subseteq \{0, 1\}^l$ *is a finite set of modes, each value* $\mathbf{m}_i$ *is associated with a vector* $\mathbf{v}_i \in \mathbb{R}^f$ *of slopes for the variables in* C.
- *GC is a global constraint in the form* $g_{gc}(D) \wedge \bigwedge_i \ell_i$ *with linear constraints* ℓ_i.
- *Init is a set of initial states, given in the form of* $\xi_0 \wedge \theta_0$, *where* $\xi_0 \in \mathcal{P}(D, C)$ *and* $\theta_0 \in \mathcal{B}(M)$.
- *DTrs is the set of discrete transitions; each discrete transition is given as a guarded assignment* ga_i *($i = 1, \ldots, u$ and $u \geq 1$) in the form*

$$\begin{aligned} \xi_i \wedge \theta_i \rightarrow {} & (d_1, \ldots, d_n) := (g_{i,1}, \ldots, g_{i,n}); \\ & (c_1, \ldots, c_f) := (t_{i,1}, \ldots, t_{i,f}); \\ & (m_1, \ldots, m_l) := \mathbf{m}_{j_i}. \end{aligned}$$

The typical usage of GC is to specify lower and upper bounds for continuous variables in runs to be considered. For simplicity, we assume that discrete inputs appear only on the right-hand side of assignments, but not in conditions.[1]

We add the following derived notions and restrictions to the CTHSs we consider:

Definition 2 (Restrictions on CTHSs)

- *The guards of the discrete transitions must be mutually exclusive, i. e.* $(\xi_i \wedge \theta_i) \Rightarrow \neg(\xi_j \wedge \theta_j)$ *for* $i \neq j$.
- *For each mode* $\mathbf{m}_i$ *its* boundary condition β_i *is given by the cofactor of the disjunction of all discrete transition guards wrt.* $\mathbf{m}_i$.[2] *The boundary conditions have to form closed subsets of* $\mathbb{R}^f$ *for each valuation of variables in* D.

Definition 3 (Semantics of CTHSs)

- *A* state *of a CTHS is a valuation* $s = (\mathbf{d}, \mathbf{c}, \mathbf{m})$ *of* D, C *and* M.
- *A discrete transition* ga_i *relates two states* $s \rightarrow_i s'$ *iff the guard* $\xi_i \wedge \theta_i$ *is true in* s *and the values in* s' *result from executing the assignments for some valuation* $\mathbf{i}$ *of the input variables.*
- *A state* $s = (\mathbf{d}, \mathbf{c}, \mathbf{m}_i)$ *evolves in time* $\lambda \in \mathbb{R}_{>0}$ *into* $s' = (\mathbf{d}, \mathbf{c} + \lambda \mathbf{v}_i, \mathbf{m}_i)$, *written as* $s \rightsquigarrow^{\lambda} s'$. s' *is a* λ*-time successor of* s ($s \rightarrow^{\lambda} s'$), *if* $s \rightsquigarrow^{\lambda} s'$ *and for all* s'' *with* $s = s''$ *or* $s \rightsquigarrow^{\lambda''} s''$ *for some* $\lambda'' < \lambda$, *we have* $s'' \models GC$ *and* $s'' \not\models \beta_i$ *(i. e. neither we violate the global constraints nor hit a discrete transition guard along the way).*
- $\rightarrow =_{\mathrm{df}} \left(\bigcup_{i=1}^{u} \rightarrow_i\right) \cup \left(\bigcup_{\lambda>0} \rightarrow^{\lambda}\right)$ *is the transition relation of the CTHS. A* trajectory *is a finite or infinite sequence of states* $(s^j)_{j \geq 0}$ *with* $s^0 \in Init$, *all* $s^j \models GC$, *and* $s^{j-1} \rightarrow s^j$ *for each* $j > 0$. *A state is* reachable *if there is a trajectory ending in that state.*

Note that the definition of a time successor makes the discrete transitions *urgent*: they fire once they become enabled. This explains why we do not need invariants of modes while on the other hand we have to require closed sets for boundary conditions.

3 Approach

In this section, we describe the main structure of our algorithm. We recall the ingredients which it shares with its predecessor from [7] and point to the new constituents which are detailed in the ensuing sections.

Overview. Our algorithm checks whether all reachable states are within a given set of (safe) states S_0. To establish this, a backwards fixpoint computation is performed. Starting with the set S_0 enriched by all states violating the global

---

[1] The usual solution is to check discrete inputs only when a timer has expired.

[2] The cofactor is the partial evaluation of the disjunction wrt. $(m_1, \ldots, m_l) = \mathbf{m}_i$. It does not depend on M anymore.

constraints, repeatedly the (safe) pre-image is computed until a fixpoint is reached or some initial state is removed from the fixpoint approximant. In the latter case, a state outside of S_0 is reachable (while observing the global constraints). So we employ repeatedly

$$Safepre(S) =_{\mathrm{df}} \{ s \in S \mid \forall s'.\, s \to s' \Rightarrow s' \in S \},$$

which corresponds to the temporal operator **AX**. We have chosen the backwards direction, because for discrete transitions the pre-image is expressed essentially by a substitution (see Hoare's program logic [16]).

Step computation. We split the computation of *Safepre* into a discrete ($Safepre_D$) and a continuous ($Safepre_C$) part. The computation of $Safepre_D$ using boolean operations, substitutions (both for boolean and real variables), and boolean quantification has been already described in [7]. We will explain our new method to cope with continuous-time evolutions (which did not occur in the discrete-time models of the precursor paper) in detail in Sect. 4.

Termination. Since the equivalence of state sets (we deal with boolean combinations of linear constraints, as detailed in the following) is decidable, termination of the algorithms enables us to answer the reachability question. However, it should be noted that termination is not guaranteed – otherwise our algorithm would constitute a solution to an undecidable problem[3]. We expect that the algorithm terminates – in theory – for the great majority of problems coming from applications. We consider complexity the much more relevant challenge in practice. Let us also remark that the implemented fixpoint computation is more elaborated in detail than the somewhat simplified version described here (due to lack of space).

Representation of state sets. Our algorithm operates on a specific data structure efficiently implementing formulas from $\mathcal{P}(D, C) \cup \mathcal{B}(M)$. These can be seen as boolean combinations over D, M and linear constraints $\mathcal{L}(C)$. We use a set of new (boolean) *constraint variables* Q as encodings for the linear constraints, where each occurring $\ell \in \mathcal{L}(C)$ is encoded by some $q_\ell \in Q$. An important characteristic of our procedure is that the set of constraint variables may grow as the step computation continues, so that new variables are introduced continuously.

For the boolean structure we employ Functionally Reduced AND-Inverter Graphs (FRAIGs) [21,23]. These are a semi-canonical variant of AND-Inverter Graphs (AIGs) [22,18]. Basically, they are boolean circuits consisting only of AND gates and inverters. Semi-canonical means that no two nodes represent the same boolean function. In the presence of atoms encoding linear constraints, we call them linear constraint AIGs, or shortly LinAIGs. Their structure is illustrated in Fig. 1.

[3] Even if the global constraints define a bounded region, one can straightforwardly encode arithmetic on integers represented as fractions $1/2^n$ of continuous values. This is a common integer representation used in the literature for showing undecidabilities in related domains.

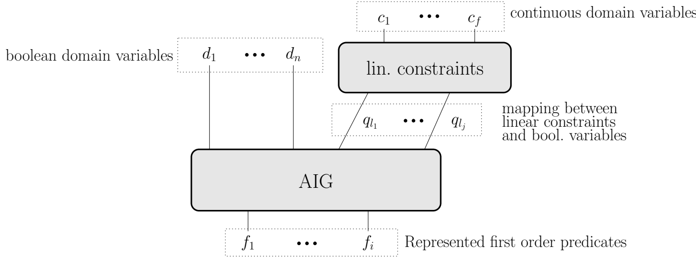


Fig. 1. The LinAIG structure

Efficiency measures. We have put much effort into the efficiency of our implementation, in particular into the time efficiency of the routines which keep the representations as small as possible. We briefly summarize some techniques described in more detail in [7], while Sec. 5 presents important improvements. Basically, functional reducedness (generalized from FRAIGs to LinAIGs) can be achieved by checking all pairs of nodes for equivalence, taking the interpretation of constraint variables q_ℓ by the corresponding linear constraints ℓ into account. This task can be performed by an SMT (SAT modulo theories) solver such as HySAT [10], which combines DPLL with linear programming as a decision procedure. However, it would be much too costly to call HySAT every time a new node is introduced. Instead, a hierarchy of approximate techniques is used to factor out "easy" problem instances. In first steps purely boolean approximations are employed: If two nodes represent equivalent boolean formulas, we do not need to refer to the definition of the constraint variables. Here we make use of capabilities of FRAIGs, which include local boolean normalization rules, simulation, and SAT checks. Additionally, boolean reasoning is supported by (approximate) knowledge on linear constraints such as implications between constraints. For identifying non-equivalent LinAIG nodes we use test vectors with valuations $\mathbf{c} \in \mathbb{R}^f$, and it proved to be worthwhile to use not only randomly generated test vectors, but also test vectors extracted from failed exact checks done by HySAT (*learning* test vectors). All of these techniques are arranged in a carefully designed and tested strategy of when to apply which technique.

4 Flow Extrapolation

Continuous transitions. In our system model, the time steps only concern the evolutions of continuous variables and leave the discrete part unchanged. For each mode, the continuous safe pre-image $Safepre_C$ can be expressed as a formula with one quantified real variable (time). We will show how to eliminate this quantifier to arrive at a formula which can again be represented by a LinAIG.

Let $\phi(D, M, Q)$ be a representation of a state set. Each valuation $\mathbf{m}_i$ of the mode variables in M encodes a concrete mode with an associated evolution $\mathbf{v}_i$ of C and boundary condition β_i. Let ϕ_i be the cofactor of ϕ w. r. t. mode $\mathbf{m}_i$. Thus we have $\phi \Leftrightarrow \bigvee_{i=1}^{k} \phi_i \wedge (m_1, \ldots, m_l) = \mathbf{m}_i$, where each ϕ_i is a boolean

formula over D and Q. For each mode $\mathbf{m}_i$, we must now determine the set of all valuations for which every (arbitrarily long) evolution along $\mathbf{v}_i$ remains in the set of valuations satisfying ϕ_i, either forever or until it meets a point that satisfies the boundary condition β_i or violates the global constraints GC. We denote this set by $\mathit{Safepre}_C(\phi_i, \mathbf{v}_i, \beta_i)$. Logically, it can be described by the formula

$$\forall\lambda.\,\big(\lambda < 0 \lor \phi_i(\mathbf{c}{+}\lambda\mathbf{v}_i) \lor \neg GC(\mathbf{c}{+}\lambda\mathbf{v}_i) \\ \lor \exists\lambda'.\,(\lambda' \geq 0 \land \lambda' < \lambda \land (\beta_i(\mathbf{c}{+}\lambda'\mathbf{v}_i) \lor \neg GC(\mathbf{c}{+}\lambda'\mathbf{v}_i)))\big).$$

Under the assumption that the set described by GC is convex, and using the fact that we are only interested in states satisfying GC, this formula can be simplified (modulo GC) to

$$\forall\lambda.\,\big(\lambda < 0 \lor \phi_i(\mathbf{c}{+}\lambda\mathbf{v}_i) \lor \neg GC(\mathbf{c}{+}\lambda\mathbf{v}_i) \lor \exists\lambda'.\,(\lambda' \geq 0 \land \lambda' < \lambda \land \beta_i(\mathbf{c}{+}\lambda'\mathbf{v}_i))\big).$$

Our task is now to convert this formula over λ, λ', C, and D into an equivalent formula over the original variables in C and D. If the variables in C occur in ϕ and β only within linear constraints, then this amounts to variable elimination for linear real arithmetic.[4]

Test points. The Loos-Weispfenning test point method [19,9] eliminates universal quantifiers by converting them into finite conjunctions (and dually, existential quantifiers into finite disjunctions). The method is based on the following observation: Assume that a formula $\psi(x, \vec{y})$ is written as a positive boolean combination of linear constraints $x \sim_i t_i(\vec{y})$ and $0 \sim'_j t'_j(\vec{y})$, where $\sim_i, \sim'_j \in \{=, \neq, <, \leq, >, \geq\}$. Let us keep the values of $\vec{y}$ fixed for a moment. If the set of all x such that $\psi(x, \vec{y})$ does not hold is non-empty, then it can be written as a finite union of (possibly unbounded) intervals, whose boundaries are among the $t_i(\vec{y})$. To check whether $\forall x.\,\psi(x, \vec{y})$ holds, it is therefore sufficient to test $\psi(x, \vec{y})$ for either all upper or all lower boundaries of these intervals. The test values may include $+\infty$, $-\infty$, or a positive infinitesimal ε, but these can easily be eliminated from the substituted formula. For instance, if x is substituted by $t_j(\vec{y}) - \varepsilon$, then both the linear constraints $x \leq t_i(\vec{y})$ and $x < t_i(\vec{y})$ are turned into $t_j(\vec{y}) \leq t_i(\vec{y})$, and both $x \geq t_i(\vec{y})$ and $x > t_i(\vec{y})$ are turned into $t_j(\vec{y}) > t_i(\vec{y})$.

There are two possible sets of test points, depending on whether we consider upper or lower boundaries:

$$TP_1 = \{+\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{\neq, >\}\,\} \cup \{\, t_i(\vec{y}) - \varepsilon \mid \sim_i \in \{=, \geq\}\,\}$$
$$TP_2 = \{-\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{\neq, <\}\,\} \cup \{\, t_i(\vec{y}) + \varepsilon \mid \sim_i \in \{=, \leq\}\,\}.$$

Let TP be the smaller one of the two sets and let T be the set of all symbolic substitutions x/t for $t \in TP$. Then the formula $\forall x.\,\psi(x, \vec{y})$ can be replaced by an equivalent finite conjunction $\bigwedge_{\sigma \in T} \psi(x, \vec{y})\sigma$. The size of TP is in general linear in

[4] The variables in D are assumed to remain constant during mode $\mathbf{m}_i$, so boolean expressions over D behave like propositional variables. For simplicity, we will ignore them in the rest of this section.

the size of ψ, so the size of the resulting formula is quadratic in the size of ψ. This is independent of the boolean structure of ψ – conversion to CNF is not required. On the other hand, if ψ is a conjunction $\bigwedge \psi_i$, then the test point method can also be applied to each of the formulas ψ_i individually, leading to a smaller number of test points. Moreover, when the test point method transforms each ψ_i into a finite conjunction $\bigwedge \psi_i^j$, then each ψ_i^j contains at most as many linear constraints as the original ψ_i, and only the length of the outer conjunction increases.

Applying the test point method to flow extrapolation. We have demonstrated above that the safe pre-image $Safepre_C(\phi_i, \mathbf{v}_i, \beta_i)$ of the formula ϕ_i is

$$\forall \lambda. \big(\lambda < 0 \vee \phi_i(\mathbf{c}{+}\lambda \mathbf{v}_i) \vee \neg GC(\mathbf{c}{+}\lambda \mathbf{v}_i) \vee \exists \lambda'. (\lambda' \geq 0 \wedge \lambda' < \lambda \wedge \beta_i(\mathbf{c}{+}\lambda' \mathbf{v}_i))\big).$$

Assuming that ϕ_i equals $\bigwedge_k \phi_{ik}$ and that β_i equals $\bigvee_j \beta_{ij}$, we obtain

$$\textstyle\bigwedge_k \forall \lambda. \big(\lambda < 0 \vee \phi'_{ik}(\mathbf{c}{+}\lambda \mathbf{v}_i) \vee \bigvee_j \exists \lambda'. (\lambda' \geq 0 \wedge \lambda' < \lambda \wedge \beta_{ij}(\mathbf{c}{+}\lambda' \mathbf{v}_i))\big).$$

where ϕ'_{ik} abbreviates $\phi_{ik} \vee \neg GC$. Applying the test point method, we replace the universal and the existential quantifier by a finite conjunction or disjunction using a set of symbolic substitutions T'_j for λ' (which depends on β_{ij} and $\mathbf{v}_i$) and a set of symbolic substitutions T_k for λ (which depends on ϕ_{ik}, the β_{ij}, and $\mathbf{v}_i$):

$$\begin{aligned} Safepre_C(\phi_i, \mathbf{v}_i, \beta_i) \;=\; & \textstyle\bigwedge_k \bigwedge_{\sigma \in T_k} \big((\lambda < 0 \vee \phi'_{ik}(\mathbf{c} + \lambda \mathbf{v}_i))\sigma \\ & \textstyle\vee \bigvee_j \bigvee_{\tau \in T'_j} (\lambda' \geq 0 \wedge \lambda' < \lambda \wedge \beta_{ij}(\mathbf{c} + \lambda' \mathbf{v}_i))\tau\sigma\big). \end{aligned}$$

Note that the test point method can work directly on the internal formula representation of LinAIGs – in contrast to the classic Fourier-Motzkin algorithm, there is no need for a costly CNF or DNF conversion before eliminating quantifiers. Moreover, the resulting formulas preserve most of the boolean structure of the original ones: the method behaves largely like a generalized substitution.

Convexity. It should be noted that some of the complexity of the general case disappears automatically if the complement of the boundary conditions is convex, that is, if every β_{ij} is a single linear inequation. Consider the formula $\bigvee_{\tau \in T'_j} (\lambda' \geq 0 \wedge \lambda' < \lambda \wedge \beta_{ij}(\mathbf{c} + \lambda' \mathbf{v}_i))\tau$. If β_{ij} is a single linear inequation, then two test points are always sufficient:[5] (a) If $\beta_{ij}(\mathbf{c} + \lambda' \mathbf{v}_i)$ has the form $\lambda' \leq t(\mathbf{c})$ or $\lambda' < t(\mathbf{c})$, then the test points are $-\infty$ and 0, (b) otherwise, if $\beta_{ij}(\mathbf{c} + \lambda' \mathbf{v}_i)$ has the form $\lambda' \geq t(\mathbf{c})$ or $\lambda' > t(\mathbf{c})$, or if λ' is cancelled out completely in $\beta_{ij}(\mathbf{c} + \lambda' \mathbf{v}_i)$, then the test points are $+\infty$ and $\lambda - \varepsilon$. Moreover, if $+\infty$ or $-\infty$ is substituted for λ', the conjunction becomes trivially false, so the whole formula is reduced to $0 < \lambda \wedge \beta_{ij}(\mathbf{c})$ in case (a) and to $\lambda > 0 \wedge \beta_{ij}(\mathbf{c} + (\lambda - \varepsilon)\mathbf{v}_i)$ in case (b).

[5] Since we want to eliminate an existential quantifier, we have to use the dual form of the method described above.

5 Redundancy Elimination

Our earlier experiments demonstrated that LinAIGs form an efficient data structure for boolean combinations of boolean variables and linear constraints over real variables [7]. However, in connection with flow extrapolation using Loos-Weispfennig quantifier elimination, one observes that the number of "redundant" linear constraints grows rapidly during the fixpoint iteration of the model checker. For illustration see Fig. 2 and 3, which show a typical example from a model checking run representing a small state set based on two real variables: Lines in Figures 2 and 3 represent linear constraints, and the gray shaded area represents the space defined by some boolean combination of these constraints. While the representation depicted in Fig. 2 contains 24 linear constraints, a closer analysis shows that an optimized representation can be found using only 15 linear constraints as depicted in Fig. 3.

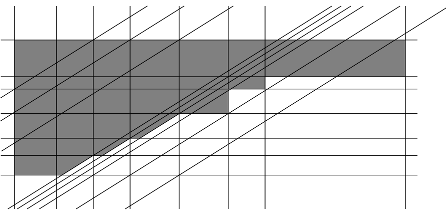


Fig. 2. Before redundancy removal

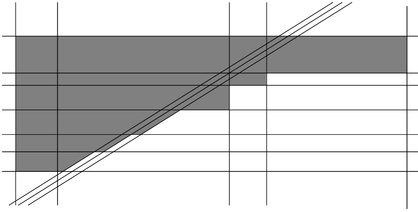


Fig. 3. After redundancy removal

Removing redundant constraints from our representations turned out to be a crucial task for the success of our methods. It should be noted that, since we represent arbitrary boolean combinations of linear constraints (and boolean variables), this task is not as straightforward as for other approaches such as [14,11] which represent sets of convex polyhedra, i. e., sets of conjunctions $\ell_1 \wedge \ldots \wedge \ell_n$ of linear constraints. If one is restricted to convex polyhedra, the question whether a linear constraint ℓ_1 is redundant in the representation reduces to the question whether $\ell_2 \wedge \ldots \wedge \ell_n$ represents the same polyhedron as $\ell_1 \wedge \ldots \wedge \ell_n$, or equivalently, whether $\overline{\ell_1} \wedge \ell_2 \wedge \ldots \wedge \ell_n$ represents the empty set. This question can simply be answered by a linear constraint solver.

For redundancy elimination in our context consider a predicate $F(b_1, \ldots, b_k, \ell_1, \ldots, \ell_n)$ (represented by a LinAIG) where $b_1, \ldots, b_k$ are boolean variables, $\ell_1, \ldots, \ell_n$ are linear constraints over C, and F is a boolean function.

Definition 4 (Redundancy of linear constraints). *The linear constraints $\ell_1, \ldots, \ell_r$ ($1 \leq r \leq n$) are called redundant in the representation of $F(b_1, \ldots, b_k, \ell_1, \ldots, \ell_n)$ iff there is a boolean function G with the property that $F(b_1, \ldots, b_k, \ell_1, \ldots, \ell_n)$ and $G(b_1, \ldots, b_k, \ell_{r+1}, \ldots, \ell_n)$ represent the same predicates.*

In order to be able to check for redundancy, we need a disjoint copy $C' = \{c'_1, \ldots, c'_f\}$ of the continuous variables $C = \{c_1, \ldots, c_f\}$. Moreover, for each

linear constraint ℓ_i $(1 \le i \le n)$ we introduce a corresponding linear constraint ℓ_i' which coincides with ℓ_i up to replacement of variables $c_j \in C$ by variables $c_j' \in C'$. Our check for redundancy is based on the following theorem:

Theorem 5 (Redundancy check). *The linear constraints $\ell_1, \dots, \ell_r$ $(1 \le r \le n)$ are redundant in the representation of $F(b_1, \dots, b_k, \ell_1, \dots, \ell_n)$ iff the predicate*

$$F(b_1, \dots, b_k, \ell_1, \dots, \ell_n) \oplus F(b_1, \dots, b_k, \ell_1', \dots, \ell_n') \wedge \bigwedge_{i=r+1}^{n} (\ell_i \equiv \ell_i') \qquad (1)$$

(where $\oplus$ denotes exclusive-or) is not satisfiable by any assignment of boolean values to $b_1, \dots, b_k$ and real values to the variables $c_1, \dots, c_f$, $c_1', \dots, c_f'$.

Note that the check from Thm. 5 can be performed by an SMT solver such as HySAT [10]. By lack of space we just give a sketch of the intuition behind Thm. 5.

According to Def. 4 linear constraints $\ell_1, \dots, \ell_n$ are redundant iff there is a boolean function G such that $G(b_1, \dots, b_k, \ell_{r+1}, \dots, \ell_n)$ and $F(b_1, \dots, b_k, \ell_1, \dots, \ell_n)$ represent the same predicates. Now let us look at $F(b_1, \dots, b_k, \ell_1, \dots, \ell_n)$ as a boolean function $F(b_1, \dots, b_k, q_{\ell_1}, \dots, q_{\ell_n})$ with (new) boolean constraint variables $q_{\ell_1}, \dots, q_{\ell_n}$ and a mapping connecting q_{ℓ_i} to ℓ_i (just as in our definition of LinAIGs). In comparison to F the required boolean function G must depend only on variables $b_1, \dots, b_k, q_{\ell_{r+1}}, \dots, q_{\ell_n}$.

If formula (1) is satisfied by some assignment $\mathbf{d} \in \{0,1\}^k$ to the boolean variables $b_1, \dots, b_k$, $\mathbf{c} \in \mathbb{R}^f$ to the real variables $c_1, \dots, c_f$ (which are inputs of linear constraints ℓ_i), and $\mathbf{c}' \in \mathbb{R}^f$ to the copied real variables $c_1', \dots, c_f'$ (which are inputs of copied linear constraints ℓ_i'), then the first part of formula (1), i. e. $F(b_1, \dots, b_k, \ell_1, \dots, \ell_n) \oplus F(b_1, \dots, b_k, \ell_1', \dots, \ell_n')$ enforces that the predicate F changes its value if input $\mathbf{c}$ is replaced by input $\mathbf{c}'$ in the corresponding linear constraints. On the other hand, the second part $\bigwedge_{i=r+1}^{n} (\ell_i \equiv \ell_i')$ enforces that the truth assignment to linear constraints $\ell_{r+1}, \dots, \ell_n$ does not change when replacing $\mathbf{c}$ by $\mathbf{c}'$. However, since G only depends on variables $b_1, \dots, b_k$, $q_{\ell_{r+1}}, \dots, q_{\ell_n}$ (whose truth assignments are not changed), function G "cannot see" the effect of changing $\mathbf{c}$ to $\mathbf{c}'$. Thus G is not able to change its value like F when replacing $\mathbf{c}$ by $\mathbf{c}'$ and therefore it is not able to represent the same predicate as F.

Conversely, it can be seen that an appropriate function G can be constructed, when formula (1) is unsatisfiable. When constructing G, we use the notion of the *don't care set DC induced by linear constraints $\ell_1, \dots, \ell_n$*: This don't care set $DC := \{(v_{b_1}, \dots v_{b_k}, v_{\ell_1}, \dots, v_{\ell_n}) \mid \nexists (v_{c_1}, \dots v_{c_f}) \in \mathbb{R}^f \text{ with } \ell_i(v_{c_1}, \dots, v_{c_f}) = v_{\ell_i} \forall 1 \le i \le n\}$ contains all boolean combinations that can not occur due to inconsistent assignments to boolean constraint variables. While for all $(\mathbf{d}, \mathbf{c}) \in \overline{DC} := \{0,1\}^{k+n} \setminus DC$ we have to postulate $G(\mathbf{d}, \mathbf{c}) = F(\mathbf{d}, \mathbf{c})$, the value of G may be chosen arbitrarily for all $(\mathbf{d}, \mathbf{c}) \in DC$, since these values can not occur due to inconsistencies between linear constraints. A closer analysis shows that – under assumption of unsatisfiability of formula (1) – it is indeed possible to define the function values of $G(\mathbf{d}, \mathbf{c})$ for $(\mathbf{d}, \mathbf{c}) \in DC$ in such a way that G will not depend on variables $q_{\ell_1}, \dots, q_{\ell_r}$. This proves that linear constraints $\ell_1, \dots, \ell_r$ are then redundant.

A straightforward realization of this approach would need a (compact) representation of the don't care set DC in order to compute an appropriate boolean

function G. However, two interesting observations turn the basic idea into a feasible approach:

1. In general, we do not need the complete set DC for the definition of the boolean function G.
2. A representation of a subset of DC which is needed for removing the redundant constraints $\ell_1, \ldots, \ell_r$ is already computed by an SMT solver when checking satisfiability of formula (1).

Again, more details on how the SMT solver internally computes a representation of a sufficient subset of DC and on the method for actually removing redundant constraints from our representations are omitted due to lack of space. Our ideas for redundancy detection and removal have been implemented based on the SMT solver HySAT. Experiments given in Section 6 show that integrating redundancy removal is crucial for the success of our methods.

6 Experimental Results

Our sample application is derived from a case study for Airbus, a controller for the flaps of an aircraft [13]. The flaps are extended during take-off and landing to generate more lift at low velocity. They are not robust enough for high velocity, so they must be retracted for other periods. It is the controller's task to correct the pilot's commands if he endangers the flaps. Additionally, there is also an extensive monitoring of the health of its sub-systems, checking for instance for hardware failures. The health monitoring system interacts with the flap control by enforcing a more conservative behavior of the control when errors are supposed to be in the system.

The benchmark used here is a simplified version of the full system including the flap controller and a health monitoring system, which is triggered by a timer. The model has three continuous variables: the velocity, the flap angle, and the timer value. Discrete states of the controller and of the health monitoring system contribute to the discrete state space. The discrete state space contains 2^{20} discrete states. This size is clearly out of reach for hybrid verification tools known from the literature, which do not scale in the discrete dimension, since modes – the only discrete states considered – are represented *explicitly* when performing reachability analysis.

The safety property to be established for our model is "For the current flap setting, the aircraft's velocity shall not exceed the nominal velocity (w. r. t. the flap position) plus 7 knots". Whether this requirement holds for our model depends on a "race" between flap retraction and speed increase. The controller is correct, if it initiates flap retraction (by correcting the pilot) early enough.

Based on the ideas presented in the previous sections we implemented a prototype model checker using LinAIGs for representing sets of states. Our experiments were run on an AMD Opteron with 2.6 GHz and 16 GB RAM.

Our model checker was able to prove the given safety invariant for the case study in 888.6 CPU seconds. The LinAIG representation had a maximum number

of 30887 nodes and a maximum number of 80 linear constraints. The number of flow extrapolation steps using Loos-Weispfennig quantifier elimination was 6, the number of discrete image computation steps performed until reaching the fixpoint was 20. This result clearly demonstrates that our approach is able to successfully verify hybrid systems including discrete parts with state spaces of considerable sizes.

In the following we analyze how the individual ingredients of our method contribute to its overall success. Redundancy elimination turned out to be absolutely necessary to make flow extrapolation using Loos-Weispfennig quantifier elimination feasible. Fig. 4 illustrates the difference between the model checking runs for our case study with and without redundancy removal by plotting the numbers of linear constraints used during the model checking run. Without redundancy removal (dotted line), the number of linear constraints is rapidly increasing up to a number of 1000 linear constraints and 150000 LinAIG nodes in the fourth flow extrapolation.[6] On the other hand, redundancy elimination detects many of the linear constraints to be redundant in our LinAIG representations. Having a closer look at the solid line in Fig. 4 one can identify six groups of three peaks in the number of linear constraints corresponding to six flow extrapolations for three modes, respectively. One notices that redundancy elimination is able to keep the numbers of linear constraints small after Loos-Weispfennig quantifier elimination, so that the number of linear constraints does not exceed 80 during the model checking run. Redundancy elimination removes redundant constraints early and has thus the additional effect that the number of constraints does not blow up due to a series of further substitutions into the removed constraints in following flow extrapolation steps.

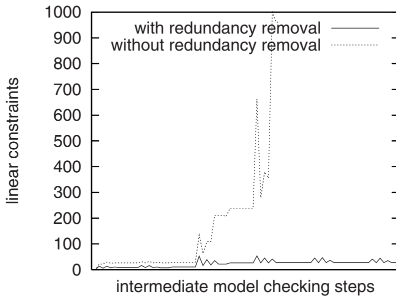


Fig. 4. Comparision of the LinAIG evolution with and without redundancy removal

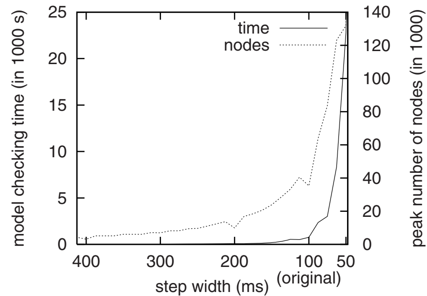


Fig. 5. Discrete time example with different time steps

Finally, we want to compare the results for our current system model, which includes continuous evolution of variable length in one operation based on flow

[6] Without redundancy removal the remaining two flow extrapolations could not be performed within our timeout of 24 hours.

extrapolation, to results for a corresponding model with discrete time semantics as presented in [7]. The system model from [7] has no continuous evolution, and discrete steps take fixed time δ. We emphasize that, although time discretized models are widely used in practical applications, they have the problem that unsafe states may be reachable from the initial state, but reachability of these states is not observed due to the time discrete nature of steps. Reducing the width of the discrete time steps can alleviate this problem, but it comes at the cost of a larger number of steps for fixpoint iterations and a larger number of LinAIG nodes for representing sets of states. Our continuous time approach does not show this problem. An analysis of this issue (here done for a flap controller without health monitoring system) is given in Fig. 5. It shows the peak numbers of LinAIG nodes (dotted line) and the run times (solid line) for the example. Here the width of the discrete time step varies between 400 ms and 50 ms. Our analysis clearly shows that run times in the discrete time model largely depend on the width of the time discretization step. At a time step of 400 ms the fixpoint iteration took 5.4 CPU seconds for 9 steps, at 100 ms 763.8 CPU seconds for 33 steps, and at 50 ms 22497 CPU seconds for 65 steps.

This demonstrates a dilemma of the time discretized version: We have to keep the time step small both to be sure not to miss relevant reachable states and to be able to model the system correctly (of course, with discrete time steps of 400 ms we are not able to model realistic controllers sampling every 100 ms). However, decreasing the time step too much may turn the model checking problem intractable. In contrast, in our novel approach we do not work with time discretizations, but we are able to compute continuous evolutions of variable lengths in one operation based on flow extrapolation. Sequences of discrete steps of the previous version [7] where no mode switches are triggered are collapsed into a single symbolic substitution in this way. Note that in the example without health monitoring system only five flow extrapolation steps are needed to reach the fixpoint within a runtime of 27.7 s (whereas for the discrete time model with a time step of 50 ms, e. g., the number of steps amounts to 65 with a run time of 22497 s).

7 Conclusion

We consider the tight integration of LinAIGs and HySAT in backward reachability analysis a core technology to address scalability of hybrid system verification methods with large discrete state spaces, and have demonstrated the relevance of the approach using a benchmark derived from an Airbus flap controller. The redundancy elimination technique presented in Section 5 is of independent value and could be integrated in other hybrid verification tools. Next imminent extensions of our approach cover differential inclusions and continuous inputs. We will experiment with incorporating orthogonal extensions to our approach such as exploiting robustness, over-approximation, and counterexample guided abstraction refinement to address richer dynamics and achieve further scalability.

References

1. Agrawal, M., Thiagarajan, P.S.: Lazy rectangular hybrid automata. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 1–15. Springer, Heidelberg (2004)
2. Agrawal, M., Thiagarajan, P.S.: The discrete time behavior of lazy linear hybrid automata. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 55–69. Springer, Heidelberg (2005)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
4. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of non-linear systems. Acta Informatica 43(7), 451–476 (2007)
5. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of the hybrid systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 365–370. Springer, Heidelberg (2002)
6. Boigelot, B., Herbreteau, F.: The power of hybrid acceleration. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 438–451. Springer, Heidelberg (2006)
7. Damm, W., Disch, S., Hungar, H., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Automatic verification of hybrid systems with large discrete state space. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 276–291. Springer, Heidelberg (2006)
8. Damm, W., Pinto, G., Ratschan, S.: Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. Journal of Foundations of Computer Science 18(1), 63–86 (2007)
9. Dolzmann, A.: Algorithmic Strategies for Applicable Real Qunantifier Elimination. PhD thesis, Universität Passau (2000)
10. Fränzle, M., Herde, C.: HySAT: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design 30(3), 179–198 (2007)
11. Frehse, G.: Compositional Verification of Hybrid Systems using Simulation Relations. PhD thesis, Radboud Universiteit Nijmegen (2005)
12. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. IEEE Transactions on Automatic Control 52(5), 782–798 (2007)
13. H3 FOMC Team. The flap controller description. `http://www.avacs.org/Benchmarks/flapcontroller.pdf`
14. Henzinger, T.A.: The theory of hybrid automata. In: 11th IEEE Symposium on Logic in Computer Science, pp. 278–292. IEEE Press, Los Alamitos (1996)
15. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer 1(1–2), 110–122 (1997)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Communication of the ACM 12, 576–583 (1969)
17. Jha, S., Brady, B., Seshia, S.: Symbolic reachability analysis of lazy linear hybrid automata. Technical report, EECS Dept. UC Berkeley (2007)
18. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Transactions on Computer-Aided Design 21(12), 1377–1394 (2002)
19. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
20. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)

21. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept. UC Berkeley (2005)
22. Paruthi, V., Kuehlmann, A.: Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In: 18th IEEE Conference on Computer Design, pp. 459–464. IEEE Press, Los Alamitos (2000)
23. Pigorsch, F., Scholl, C., Disch, S.: Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In: 6th Conference on Formal Methods in Computer Aided Design, pp. 89–96. IEEE Press, Los Alamitos (2006)
24. Platzer, A., Clarke, E.: The image computation problem in hybrid systems model checking. In: 10th Workshop on Hybrid Systems: Computation and Control. LNCS, vol. 4416, pp. 473–486. Springer, Heidelberg (2007)
25. Segelken, M.: Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007)
26. Silva, B.I., Richeson, K., Krogh, B.H., Chutinan, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: 4th Conference on Automation of Mixed Processes (2000)
27. The VIS Group. VIS: A system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)
28. Wang, F.: Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. IEEE Transactions on Software Engineering 31(1), 38–52 (2005)

A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains*

Hichem Boudali[1], Pepijn Crouzen[2,**], and Mariëlle Stoelinga[1]

[1] Department of Computer Science, University of Twente,
P.O. Box 217, 7500AE Enschede, The Netherlands
[2] Saarland University, Department of Computer Science,
D-66123 Saarbrücken, Germany
{hboudali@cs,p.crouzen@alumnus,marielle@cs}.utwente.nl

Abstract. Dynamic fault trees (DFTs) are a versatile and common formalism to model and analyze the reliability of computer-based systems. This paper presents a formal semantics of DFTs in terms of input/output interactive Markov chains (I/O-IMCs), which extend continuous-time Markov chains with discrete input, output and internal actions. This semantics provides a rigorous basis for the analysis of DFTs. Our semantics is fully compositional, that is, the semantics of a DFT is expressed in terms of the semantics of its elements (i.e. basic events and gates). This enables an efficient analysis of DFTs through compositional aggregation, which helps to alleviate the state-space explosion problem by incrementally building the DFT state space. We have implemented our methodology by developing a tool, and showed, through four case studies, the feasibility of our approach and its effectiveness in reducing the state space to be analyzed.

Fault trees (FTs) [20], also called static FTs, provide a high-level, graphical formalism to model and analyze system failures. An FT is made up of basic events, usually modeling the failure of physical components, and of logical gates, such as AND and OR gates, modeling how the component failures induce the system failure. Dynamic fault trees (DFTs) [11] extend (static) fault trees by allowing the modeling of more complex behaviors and interactions between components: Whereas FTs only take into consideration the combination of failures, DFTs also take into account the order in which they occur. A DFT is typically analyzed by first converting it to a continuous time Markov chain (CTMC) and by then analyzing this CTMC.

This paper formally describes the DFT syntax and semantics, thus providing a rigorous basis for DFT analysis and tool development. The DFT syntax is given in terms of a directed acyclic graph (DAG) and its semantics in terms

* This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.505 (MOQS); the EU under grant number IST-004527 (ARTIST2); and by the DFG/NWO bilateral cooperation programme under project number DN 62-600 (VOSS2).

** The majority of this work was done while the author was at the University of Twente.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 441–456, 2007.
© Springer-Verlag Berlin Heidelberg 2007

of an input/output interactive Markov chain (I/O-IMC). Our semantics is fully compositional. That is, we present the semantics of each DFT element (i.e. gate or basic event) as an I/O-IMC; the semantics of a DFT is then obtained by parallel composing the I/O-IMC semantics of all its elements. Compositionality is a fundamental and highly desirable property of a semantics: it enables compositional reasoning, i.e. analyzing complex systems by breaking them down into their constituting parts. In our case, it enables compositional aggregation to combat the state-space explosion problem existing in DFTs. Moreover, a compositional semantics is comprehensible, since one can focus on one construct at a time, and readily extensible. As elaborated in [4], we can easily add new DFT gates or concepts such as repair policies.

Earlier work on formalizing DFTs can be found in [10], where a semantics is described using the Z specification language. This work revealed a number of ambiguities in the DFT framework. Most notably, in some instances of DFTs non-determinism has arisen. But, since non-determinism was not intended in the original formulation of DFTs and every DFT had to be mapped into a CTMC, this non-determinism was resolved by transformation into a deterministic or probabilistic choice (see [4] for further details on non-determinism in DFTs). The semantics in [10] is, however, not compositional and hence is, in our opinion, not easy to understand. Formalizing the DFT syntax and semantics in a compositional way turned out to be a non-trivial task.

Interactive Markov chains (IMCs) [14] are an extension of CTMCs with discrete actions and have proven to be a powerful formalism for a variety of applications. IMCs come with efficient algorithms for aggregating equivalent (i.e. weakly bisimilar) states and operators for parallel composing IMCs and for hiding (i.e. making internal) certain discrete actions. For our purposes, we needed IMCs that distinguish between input and output actions. Hence, we introduce I/O-IMCs, combining IMCs with features from the I/O automaton model in [17]. We also present a notion of weak bisimilarity for I/O-IMCs. To aggregate as many states as possible, our weak bisimulation disregards Markovian transitions from a state s into its own equivalence class. Thus, we do not only generalize the usual IMC weak bisimulation to I/O-IMCs, but also extend it along the lines of [8]. Furthermore, we show that weak bisimulation is a congruence w.r.t. parallel composition and hiding.

The conversion into a CTMC and resolution of a DFT has been first introduced by Dugan et al. in the so called DIFTree methodology [12]. This method suffers from the well-known state-space explosion problem. In fact, the size of the CTMC grows exponentially with the number of basic events in the DFT. Recently, there have been attempts in dealing with the state-space explosion problem by avoiding the CTMC generation [6,1]. In [6], the authors propose a Bayesian network approach and provide an approximate DFT solution. In [1], the authors present a method to identify submodules in a DFT where the CTMC generation is not needed.

We use a compositional aggregation approach to build the I/O-IMC of the whole DFT: We start with the interpretation of a single DFT element as an

I/O-IMC. Then we repeatedly take the parallel composition with the interpretation of another element, while aggregating equivalent states. We keep repeating these two steps until we are left with a single aggregated I/O-IMC. This compositional aggregation approach is crucial in alleviating the state-space explosion problem. To summarize, this paper makes the following contributions:

1. We derive, based on IMCs and I/O automata, the I/O-IMC formalism and introduce a notion of weak bisimilarity for I/O-IMCs, which we show to be a congruence w.r.t. parallel composition and hiding.
2. We formally define the DFT syntax and semantics in terms of respectively a DAG description and I/O-IMCs.
3. We report on a tool and show the feasibility of our approach on four case studies.

The remainder of the paper is organized as follows: Section 1 introduces DFTs and Section 2 treats the formalism of I/O-IMCs. In Section 3, we present the syntax and the semantics of DFTs, and in Section 4 we illustrate the compositional aggregation technique. Finally, Section 5 provides some case studies and presents the prototype tool, and we conclude the paper in Section 6.

1 Dynamic Fault Trees

An FT is a tree (or rather, a DAG) in which the leaves are called *basic events* (BEs) and the other elements are *gates*. BEs model the failure of physical components and are depicted by circles. The failure of a BE is governed by an exponential distribution. That is, the probability that the BE fails within t time units equals $1 - e^{-\lambda t}$, where λ is the *failure rate* of the BE. The non-leaf elements are gates, modeling how the component failures induce a system failure. Static fault trees have three type of (static) gates: the AND gate, the OR gate and the K/M (or called VOTING) gate, depicted in Figure 1.a, 1.b, and 1.c, respectively. These gates fail if respectively all, at least one, or at least K (called the threshold) out of M of their inputs fail.

Dynamic fault trees [11] extend (static) FTs with three novel types of gates[1]: The priority AND gate (PAND); the spare gate (SPARE), modeling the management and allocation of spare components; and the functional dependency gate (FDEP). These gates (depicted in Figure 1.d, 1.e, and 1.f) are described below.

PAND Gate. The priority AND (PAND) gate models a failure sequence dependency. It fails if all of its inputs fail from left to right order in the gate's depiction. If the inputs fail in a different order, the gate does not fail.

SPARE Gate. The SPARE gate has one primary input and zero (which is a degenerated case) or more alternate inputs called *spares*. All inputs are BEs.

[1] A fourth gate called 'Sequence Enforcing' (SEQ) gate has also been defined in [11], but it turns out that this gate is expressible in terms of the cold spare gate.

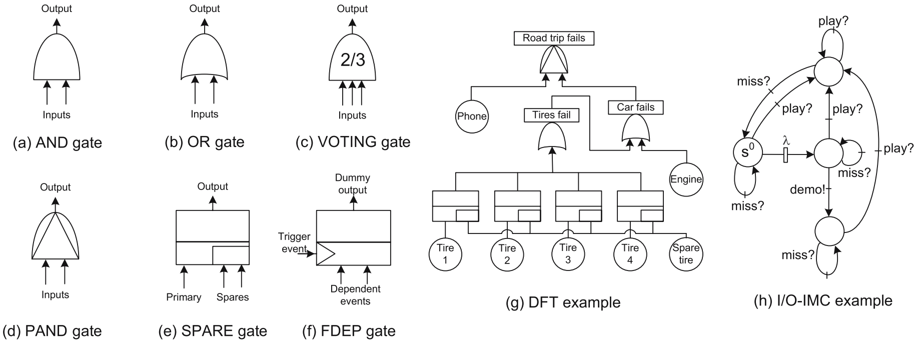


Fig. 1. DFT gates, DFT example, and I/O-IMC example

The primary input of a SPARE gate is initially powered on and the alternate inputs are in standby mode. When the primary fails, it is replaced by the first available alternate input (which then switches from the standby mode to the active mode). In turn, when this alternate input fails, it is replaced by the next available alternate input, etc.

In standby (or dormant) mode, the BE failure rate λ is reduced by a *dormancy factor* $\alpha \in [0, 1]$. Thus, the BE failure rate in standby mode is $\mu = \alpha\lambda$. In active mode, the failure rate switches back to λ. Two special cases arise if $\alpha = 0$ or $\alpha = 1$. If $\alpha = 0$, the spare is called a *cold spare* and can by definition not fail before the primary. When $\alpha = 1$, the spare is called a *hot spare* and its failure rate is the same whether in standby or in active mode. If $0 < \alpha < 1$, the spare is called a *warm spare*. The SPARE gate fails when the primary and all its spares have failed.

Multiple spare gates can share a pool of spares. When the primary unit of any of the spare gates fails, it is replaced by the first available (i.e. not failed or not already taken by another spare gate) spare unit; which becomes, in turn, the active unit for that spare gate.

FDEP Gate. The functional dependency gate consists of a trigger event (i.e. a failure) and a set of dependent events (or components). When the trigger event occurs, it causes all the dependent components to become inaccessible or unusable (the dependent components can of course also still fail by themselves). Dependent events need to be BE's. All dependent events and the trigger event are considered to be inputs to the FDEP gate. The FDEP gate's output is a 'dummy' output (i.e. it is not taken into account during the calculation of the system failure probability).

Example 1. Figure 1.g shows a DFT modeling a road trip. Looking at the top PAND gate, we see that the road trip fails (i.e. we are stuck on the road) if the car fails after the mobile phone has failed; if the car fails first, then we can call the road services to tow the car and continue our journey. The car fails if either the engine fails or the tire subsystem fails, as modeled by the OR gate

labeled 'car fails'. The car is equipped with a spare tire, which can be used to replace any of the primary tires. When a second tire fails, the tire subsystem fails, causing in turn a car failure. Thus, we model the tire subsystem by four spare gates, each having a primary tire (BEs 'Tire 1', 'Tire 2','Tire 3', and 'Tire 4') and all sharing a spare tire (BE 'Spare tire'). The spare tire is a cold spare, i.e. it is initially in standby mode with failure rate 0.

2 Input/Output Interactive Markov Chains

The formalism. This section introduces the formalism of input/output interactive Markov chains (I/O-IMCs), which are based on IMCs [14]. IMCs combine continuous-time Markov chains with discrete actions (also called signals). State changes in IMCs can occur either because a discrete action is taken, or after a delay, which is governed by an exponential distribution. Thus, IMCs have two types of transitions: discrete transitions (denoted $\xrightarrow{a}$ and $\dashrightarrow$ in figures) labeled with a discrete action a and Markovian transitions (denoted $\xrightarrow{\lambda}_M$ and $\Rightarrow$ in figures) labeled with the rate λ of an exponential distribution.

I/O-IMCs are a variant of IMCs that partition the set of discrete actions into input actions, output actions and internal actions (inspired by the I/O variant of automata introduced in [17]). Input actions, being under the control of the environment of the I/O-IMC, are delayable, while output actions must be taken immediately and cannot be delayed. This partition is natural in the DFT context where elements have input and output signals and where – rather than being a handshake – communication is always initiated by the failing (or activating) component. Moreover, in contrast with IMCs where all observable actions are delayable, in I/O-IMCs only input actions are delayable. Internal actions are not visible to the environment and are also immediate.

Definition 1. *An* input/output interactive Markov chain $\mathcal{P}$ *is a tuple* $\langle S, s^0, A, \rightarrow, \rightarrow_M \rangle$*, where*

- S *is a set of states,*
- $s^0 \in S$ *is the initial state.*
- A *is a set of discrete actions (or signals), where* $A = (A^I, A^O, A^{int})$ *is partitioned into a set of input actions* A^I*, output actions* A^O *and internal actions* A^{int}*. We write* $A^V = A^I \cup A^O$ *for the set of visible actions of* $\mathcal{P}$*. We suffix input actions with a question mark (e.g.* $a?$*), output actions with an exclamation mark (e.g.* $a!$*) and internal actions with a semi-colon (e.g.* $a;$*).*
- $\rightarrow \subseteq S \times A \times S$ *is a set of interactive transitions. We write* $s \xrightarrow{a} s'$ *for* $(s, a, s') \in \rightarrow$*. We require that I/O-IMCs are input-enabled:* $\forall s \in S, a? \in A^I, \exists s' \in S \cdot s \xrightarrow{a?} s'$*.*
- $\rightarrow_M \subseteq S \times \mathbb{R}_{>0} \times S$ *is a set of Markovian transitions. We write* $s \xrightarrow{\lambda}_M s'$ *for* $(s, \lambda, s') \in \rightarrow_M$*.*

We denote the components of $\mathcal{P}$ *by* $S_\mathcal{P}$*,* $s^0_\mathcal{P}$*,* $A_\mathcal{P}$*,* $\rightarrow_\mathcal{P}$*,* $\rightarrow^M_\mathcal{P}$ *and omit the subscript* $\mathcal{P}$ *whenever clear from the context. The* action signature *of an I/O-IMC is the partitioning* (A^I, A^O, A^{int}) *of* A*. We denote the class of all I/O-IMCs by* IOIMC*.*

Example 2. Figure 1.h shows an example I/O-IMC modeling a video game. If a user presses the play button (input action *play*?), the game starts and continues until the user makes a mistake (*miss*?), which brings the game back to the initial state s^0. If after some delay d, no *play*? signal has been received, then the system runs a game demonstration (output *demo*!) until a *play*? signal is received. The delay d (in hours) is exponentially distributed with rate 12. This means that, on average, a demo is played after $\frac{1}{12}$ hours (= 5 minutes).

Note that the system is input enabled, i.e. each state enables the input actions *play*? and *miss*?.

I/O-IMCs can be built from smaller I/O-IMCs through parallel composition. If two I/O-IMCs $\mathcal{P}$ and $\mathcal{Q}$ are composable, then their composition $\mathcal{P}\|\mathcal{Q}$ is the I/O-IMC representing their joint behavior. As in the I/O automaton framework, the components $\mathcal{P}$ and $\mathcal{Q}$ synchronize on shared actions and evolve independently on actions that are internal or not shared. The hiding operator *hide* B *in* $\mathcal{P}$ makes all actions in a set B of visible actions internal.

Definition 2. *Let $\mathcal{P}$ and $\mathcal{Q}$ be I/O-IMCs.*

1. *$\mathcal{P}$ and $\mathcal{Q}$ are* composable *if $A^O_{\mathcal{P}} \cap A^O_{\mathcal{Q}} = A^{int}_{\mathcal{P}} \cap A_{\mathcal{Q}} = A_{\mathcal{P}} \cap A^{int}_{\mathcal{Q}} = \emptyset$.*
2. *If $\mathcal{P}$ and $\mathcal{Q}$ are composable I/O-IMCs, their* composition *$\mathcal{P}\|\mathcal{Q}$ is the I/O-IMC $(S_{\mathcal{P}} \times S_{\mathcal{Q}}, (s^0_{\mathcal{P}}, s^0_{\mathcal{Q}}), ((A^I_P \cup A^I_Q) \setminus (A^O_P \cup A^O_Q), (A^O_P \cup A^O_Q), (A^{int}_P \cup A^{int}_Q)), \rightarrow_{\mathcal{P}\|\mathcal{Q}}, \rightarrow^M_{\mathcal{P}\|\mathcal{Q}})$, where*

$$\begin{aligned}
\rightarrow_{\mathcal{P}\|\mathcal{Q}} = {} & \{(s,t)\xrightarrow{a}_{\mathcal{P}\|\mathcal{Q}}(s',t) \mid s\xrightarrow{a}_{\mathcal{P}}s' \wedge a \in A_{\mathcal{P}} \setminus A_{\mathcal{Q}}\} \\
& \cup\{(s,t)\xrightarrow{a}_{\mathcal{P}\|\mathcal{Q}}(s,t') \mid t\xrightarrow{a}_{\mathcal{Q}}t' \wedge a \in A_{\mathcal{Q}} \setminus A_{\mathcal{P}}\} \\
& \cup\{(s,t)\xrightarrow{a}_{\mathcal{P}\|\mathcal{Q}}(s',t') \mid s\xrightarrow{a}_{\mathcal{P}}s' \wedge t\xrightarrow{a}_{\mathcal{Q}}t' \wedge a \in A_{\mathcal{P}} \cap A_{\mathcal{Q}}\} \\
\rightarrow^M_{\mathcal{P}\|\mathcal{Q}} = {} & \{(s,t)\xrightarrow{\lambda}{}^M(s',t) \mid s\xrightarrow{\lambda}{}^M_{\mathcal{P}}s'\} \cup \{(s,t)\xrightarrow{\lambda}{}^M(s,t') \mid t\xrightarrow{\lambda}{}^M_{\mathcal{Q}}t'\}
\end{aligned}$$

   Given a set $\{\mathcal{P}_1, \mathcal{P}_2 \dots \mathcal{P}_n\}$ of I/O-IMCs, we write $\|\mathcal{P}_i$ for $\mathcal{P}_1\|\mathcal{P}_2\| \cdots \mathcal{P}_n$.
3. *Let $B \subseteq A^V$ be a set of visible actions. We define the I/O-IMC hide B in $\mathcal{P}$ by $(S_{\mathcal{P}}, s^0_{\mathcal{P}}, (A^I_{\mathcal{P}} \setminus B, A^O_{\mathcal{P}} \setminus B, A^{int}_{\mathcal{P}} \cup B), \rightarrow_{\mathcal{P}}, \rightarrow^M_{\mathcal{P}})$.*

Bisimilarity. Bisimilation relations are equivalences on the state-space that identify states with the same step-wise behavior. Our notion of weak bisimilarity is based on bisimulation for IMCs [14]. The key differences are we distinguish between input and output transitions, and ignore Markovian self-loops (as in [8]); i.e. Markovian transition from a state in an equivalence class to a state in the same equivalence class.

Let $\mathcal{P}$ be an I/O-IMC and let $s, s' \in S$ be states in $\mathcal{P}$. We write $s\overset{\varepsilon}{\Longrightarrow}s'$ if there exists a sequence (possibly of length zero, i.e. $s = s'$) $s\xrightarrow{a_1}s_1\xrightarrow{a_2}s_2 \dots \xrightarrow{a_n}s_n = s'$ of transitions with $a_i \in A^{int}$. For $a \in A^V$ we write $s\overset{a}{\Longrightarrow}s'$ if there exists states s_1, s_2 such that $s\overset{\varepsilon}{\Longrightarrow}s_1\xrightarrow{a}s_2\overset{\varepsilon}{\Longrightarrow}s'$. Also, $\gamma_M(s, C) = \sum\{|\lambda \mid s\xrightarrow{\lambda}{}^M s' \wedge s' \in C|\}$, with $\{| \dots |\}$ denoting a multiset of transition rates, is the sum of the rates of all Markovian transitions from s into C. We write $C^{int} = \{s \mid \exists s' \in C \cdot s\overset{\varepsilon}{\Longrightarrow}s'\}$ for the set of all states which can reach some element in C via internal transitions.

Finally, we say that s is *stable* if s has no outgoing immediate (i.e. internal or output) transition.

Definition 3 (Weak bisimulation). *Let P be an I/O-IMC. A* weak bisimulation *for P is an equivalence relation R on S such that for all $(s,t) \in R$, $a \in A \cup \{\varepsilon\}$*

1. *$s \overset{a}{\Longrightarrow} s'$ implies that there is a weak transition $t \overset{a}{\Longrightarrow} t'$ with $(s',t') \in R$.*
2. *$s \overset{\varepsilon}{\Longrightarrow} s'$ and s' stable imply that there is a t' such that $t \overset{\varepsilon}{\Longrightarrow} t'$ and t' stable and $\gamma_M(s', C^{int}) = \gamma_M(t', C^{int})$, for all equivalence classes C of R, except for $C = [s']_R$, the equivalence class of s'.*

States s and t in P are weakly bisimilar*, notation $s \approx t$, if there exists a weak bisimulation R with $(s,t) \in R$.*

Our notion of weak bisimilarity satisfies the usual properties: $\approx$ is the largest weak bisimulation and it is a congruence with respect to parallel composition and hiding. To compute $\approx$, one can use an algorithm similar to the one in [14], which runs in time $O(n^3)$, where n is the number of states in the I/O-IMC. We refer the reader to [5] for more details.

As most bisimulation relations, $\approx$ can be used to aggregate (also referred to as lump, minimize or reduce) an I/O-IMC $\mathcal{P}$: By grouping together equivalent (i.e. weakly bisimilar) states in $\mathcal{P}$, we obtain an equivalent I/O-IMC that is (usually) smaller. The mentioned properties enable an efficient aggregation algorithm that works in a compositional way, cf. Section 4.

3 Formalizing DFTs

3.1 DFT Syntax

To formalize the syntax of a DFT, we first define the set $\mathcal{E}$, characterizing each DFT element by its type, number of inputs and possibly some other parameters. We use the following notation. Given a set X, we denote by $\mathcal{P}(X)$ the power set over X and by X^* the set of all sequences over X. For a sequence $x \in X^*$, we denote by $|x|$ the length of the sequence (also called list), and by $(x)_i$ the ith element in x.

Definition 4. *The set $\mathcal{E}$ of DFT elements consists of the following tuples. Here, $k, n \in \mathbb{N}$ are natural numbers with $1 \leq k \leq n$ and $\lambda, \mu \in \mathbb{R}^{\geq 0}$ are rates.*

- *(OR, n), (AND, n), $(PAND, n)$ represent respectively OR, AND and PAND gates with n inputs.*
- *(VOT, n, k) represent a voting gate with n inputs and threshold k.*
- *$(SPARE, n)$ represent a SPARE gate with one primary and $n-1$ spares. By convention, the first input to the SPARE gate is the primary component.*
- *$(FDEP, n)$ represents an FDEP gate with 1 trigger input event and $n-1$ dependent input events. By convention, the first input to the FDEP gate is the trigger event.*

– $(BE, 0, \lambda, \mu)$, *represents BE, which has no inputs (i.e. $n = 0$), an active failure rate λ and a dormant failure rate μ.*

Given a tuple $e \in \mathcal{E}$, we write type(e) for the first item in e, and arity(e) for the second.

A DFT is a directed acyclic graph, where each vertex v is labeled with a DFT element $l(v) \in \mathcal{E}$. An edge from v to w means that the output of $l(v)$ is an input to $l(w)$. Since the order of inputs to a gate matters (e.g. for a PAND gate), the inputs to v are given as a list *preds*(v), rather than as a set.

Definition 5. *A dynamic fault tree is a quadruple $\mathcal{D} = (V, preds, l)$, where*

– *V is a set of vertices,*
– *$l : V \to \mathcal{E}$ is a labeling function, that assigns to each vertex a DFT element.*
– *$preds : V \to V^*$ is a function that assigns to each vertex a list of inputs.*

The set of *edges* E is the set $\{(v, w) \in V^2 | \exists i \,.\, v = (preds(w))_i\}$ of all pairs (v, w) such that v appears as a predecessor of w. We write *type*(v) for *type*$(l(v))$ and *arity*(v) for *arity*$(l(v))$. For $\mathcal{D}$ to be a well-formed DFT, the following restrictions have to be met.

– The set (V, E) is a directed acyclic graph.
– All inputs to a DFT element must be connected to some node in $\mathcal{D}$, i.e. for all $v \in V$, we have $arity(v) = |preds(v)|$.
– Since we do not include the dummy output of an FDEP gate in $\mathcal{D}$, FDEP gates have no outgoing edges: if $(v, w) \in E$, then $type(v) \neq FDEP$.
– There is a unique top element in $\mathcal{D}$, i.e. a non-FDEP element whose output is not connected. That is, there exists a unique $v \in V$, $type(v) \neq FDEP$ such that for all $w \in V \,.\, (v, w) \notin V$. This unique v is denoted by $T_{\mathcal{D}}$; or by T if $\mathcal{D}$ is clear from the context.
– The first input of a SPARE gate can not be an input to another SPARE gate (i.e. primary components can not be shared): If $v = (preds(w))_1 = (preds(w'))_1$ with $type(w) = type(w') = SPARE$, then $w = w'$.
– Inputs (primary and spare components) of a SPARE gate must be BEs: if $type(w) = SPARE$, then $type((preds(w))_i) = BE$, for all $1 \leq i \leq |preds(w)|$.
– The dependent inputs (i.e. inputs number 2 and higher) of an FDEP gate must be BEs: if $type(w) = FDEP$, then $type((preds(w))_i) = BE$, for all $2 \leq i \leq |preds(w))|$.
– An output can not be twice or more the input of the same gate: for all $1 \leq i, j \leq |preds(w)|$ with $(preds(w))_i = (preds(w))_j$, we have $i = j$.

3.2 DFT Element Semantics

This section provides the I/O-IMC semantics $[\![e]\!]_{\mathsf{ELT}}$ for each DFT element $e \in \mathcal{E}$. The I/O-IMC is parametric in its input and output signals. (These parameters are instantiated in Section 3.3, so that output signals in the semantics of a child element correspond to input signals in the semantics of its parents.) Thus,

formally, $[\![e]\!]_{\mathsf{ELT}}$ is a function that, depending on the type of e, takes as arguments a number of actions and returns an I/O-IMC. Each of these I/O-IMCs has an initial operational state, some intermediate operational states, a *firing* (or failed) state, and an absorbing *fired* state. The firing and fired states are drawn as gray circles and double circles respectively. For the sake of clarity, all self-loops $(s, a?, s)$ labeled by input actions are omitted from the figures.

Basic Events I/O-IMC Model. As pointed out in Section 1, a BE has a different failing behavior depending on its dormancy factor. Figure 2 shows the (parameterized) I/O-IMCs associated to a cold, warm, and hot BE[2], i.e. it shows the functions $[\![(BE, 0, \lambda, \mu)]\!]_{\mathsf{ELT}} : A^2 \rightarrow \mathsf{IOIMC}$ taking as arguments an activation signal $a?$ and a firing signal $f!$.

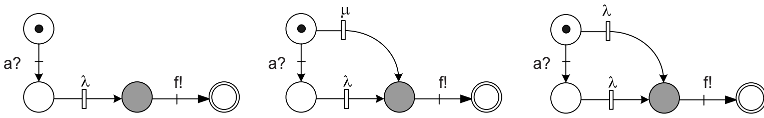


Fig. 2. The I/O-IMCs $[\![(BE, 0, \lambda, 0)]\!]_{\mathsf{ELT}}(a, f)$, $[\![(BE, 0, \lambda, \mu)]\!]_{\mathsf{ELT}}(a, f)$, and $[\![(BE, 0, \lambda, \lambda)]\!]_{\mathsf{ELT}}(a, f)$, modeling the semantics of a cold, warm and hot BE

AND Gate I/O-IMC Model. Figure 3.a shows the semantics of the $(AND, 2)$ gate, i.e. the function $[\![(AND, 2)]\!]_{\mathsf{ELT}} : A^3 \rightarrow \mathsf{IOIMC}$, taking as arguments the output and two inputs signals of the AND gate.[3] This I/O-IMC models that the AND gate fires (action f_1) after it receives firing signals from both its inputs (actions f_2 and f_3). Note that the AND gate does not have an activation signal as this element does not exhibit a dormant or active behavior as such. The semantics of the OR and VOTING gates are similar.

PAND Gate I/O-IMC Model. Figure 3.b shows the semantics $[\![(PAND, 2]\!]_{\mathsf{ELT}}$: $A^3 \rightarrow \mathsf{IOIMC}$ of a PAND gate with two inputs. The PAND gate fires after all its inputs fire from left to right order. If the inputs fire in the wrong order, the PAND gate moves to an operational absorbing state (denoted with an **X** in Figure 3.b).

FDEP Gate I/O-IMC Model. An FDEP gate does not have semantics itself, but instead is used in combination with the semantics of its dependent BEs. To model a functional dependency, we define the *firing auxiliary* function FA : $A^2 \times \mathcal{P}(A) \rightarrow \mathsf{IOIMC}$. This (parametric) I/O-IMC ensures that a dependent BE fires either when the BE fails by itself, or when its failure is triggered by the FDEP gate trigger: Figure 3.c shows the FA to be applied in combination with a BE that is functionally dependent on n triggers. Signal f_2 corresponds to the failure of the dependent event by itself; signals $f_3, f_4, \ldots, f_{n+2}$ correspond to the

[2] The hot BE I/O-IMC can be reduced to: $\odot \xrightarrow{\lambda} \bigcirc \xrightarrow{f!} \bigcirc$.

[3] The semantics $[\![(AND, n)]\!]_{\mathsf{ELT}} : A^{n+1} \rightarrow \mathsf{IOIMC}$ of the AND gate with n inputs can be constructed in a similar fashion, cf [5].

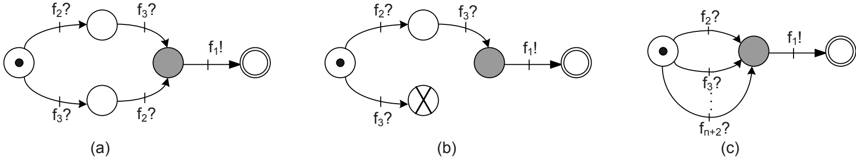


Fig. 3. The semantics (a) $[\![(AND, 2)]\!]_{\mathsf{ELT}}(f_1, f_2, f_3)$, (b) $[\![(PAND, 2)]\!]_{\mathsf{ELT}}(f_1, f_2, f_3)$, and (c) $FA(f_1, f_2, \{f_3, f_4 \ldots, f_{n+2}\})$

failures of any of the triggers; and f_1 corresponds to the failure of the dependent event when also considering its functional dependency upon the triggers. Hence, f_1 is emitted as soon as any signal from $\{f_2, f_3, \ldots, f_{n+2}\}$ occurs. Thus, the *FA* takes as arguments two firing signals and a set of firing signals (corresponding to all triggers of the dependent BE).

SPARE Gate I/O-IMC Model. Figure 4 shows the I/O-IMC of a spare gate sharing a spare with another spare gate. The SPARE gate behaves, to a certain extent, similarly to the AND gate. That is, for the spare gate to fail, both its primary has to fail and its spare has to be unavailable (fail or be taken by the other spare gate). The state reached after the primary fails is of particular interest (i.e. the state reached from the initial state after transition $f_2?$ is taken). In this state, a non-deterministic situation arises where the spare can be activated by either of the spare gates (signals $a_{1,1}!$ and $a_{1,2}?$). We could of course also get signal $f_3?$ (i.e. failure of the spare) immediately after signal $f_2?$. The signals $a_{1,1}$ and $a_{1,2}$ are signals between the two spare gates notifying each other about the activation (and thus the acquisition) of the shared spare. These signals are also sent to the spare to activate it. The semantics of a spare gate having n spares is a function $A^2 \times (A^2 \times \mathcal{P}(A))^n \rightarrow \mathsf{IOIMC}$ that takes as arguments the firing signal of the spare gate, the firing signal of its primary and n spare-tuples containing, for each spare, its firing signal, its activation signal (by the spare gate) and a set of activation signals of the other spare gates sharing that spare. Figure 4.b shows the semantics $[\![(SPARE, 2)]\!]_{\mathsf{ELT}}(f_1, f_2, (f_3, a_{1,1}, \{a_{1,2}\}))$ of the spare gate in Figure 4.a. Generalizing the SPARE gate I/O-IMC model to handle the case

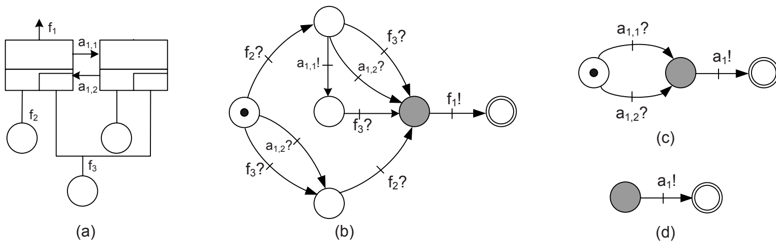


Fig. 4. (a) A DFT, (b) semantics $[\![(SPARE, 2)]\!]_{\mathsf{ELT}}(f_1, f_2, (f_3, a_{1,1}, \{a_{1,2}\}))$ of (left) SPARE gate, (c) $AA(a_1, \{a_{1,1}, a_{1,2}\})$, (d) $AA(a_1, \emptyset)$

where multiple spare gates share multiple spares turned out to be a non-trivial task [4,5].

3.3 DFT Semantics

This section shows how to get the semantics of a DFT from the semantics of its elements. First, we define the node semantics $[\![v]\!]$ of a DFT node $v \in V$ by instantiating the parameters of $[\![l(v)]\!]_{\mathsf{ELT}}$ appropriately, using the following main actions: The firing signal f_X of element $X \in \mathcal{E}$ denotes the failure of X and the activation signal a_X denotes the activation of a BE X, i.e. the switching from dormant to active mode. When used as a spare, a BE is activated by its SPARE gates; and $a_{S,G}$ denotes the activation of spare S by SPARE gate G. Otherwise, the BE is activated from the start.

Given a node v in a DFT $\mathcal{D}$, we define the node semantics $[\![v]\!]$ as follows.

OR, AND, VOT, and PAND. If v is labeled as an OR, AND, VOT, or PAND gate, then $[\![v]\!]$ is obtained from $[\![l(v)]\!]_{\mathsf{ELT}}$ by instantiating its parameters in such a way that the input signals of v connect to output signals of v's children (i.e. nodes $w \in \mathit{preds}(v)$). Thus, for $\mathit{type}(v) = \mathit{OR}, \mathit{AND}, \mathit{VOT}, \mathit{PAND}$, with $\mathit{preds}(v) = w_1 w_2 \dots w_n$, we have

$$[\![v]\!] = [\![l(v)]\!]_{\mathsf{ELT}}(f_v, f_{w_1}, f_{w_2}, \dots f_{w_n})$$

BE. If v is labeled as a basic event, two steps need to be carried out. First, we have to check if the BE is a dependent event of some FDEP gate. If so, we use the firing auxiliary so that a failure $f_v!$ is emitted whenever either the BE fails (via f_v^*) or any of the triggers fails (via $f_t \in T_v$). As an intermediate step, let

$$[\![v]\!]_1 \;=\; [\![l(v)]\!]_{\mathsf{ELT}}(a_v, f_v^*) \parallel \mathit{FA}(f_v, f_v^*, T_v)$$

Here, $T_v = \{f_t \mid \exists w \in V \,.\, (v,w) \in E \wedge l(w) = \mathit{FDEP} \wedge t = (\mathit{preds}(w))_1\}$ is the set of trigger signals of FDEP gates on which $l(v)$ is dependent.[4]

Second, we need to activate the BE if it is used as a spare. This is done by composing $[\![v]\!]_1$ in parallel with an activation auxiliary (see Figure 4.c and Figure 4.d), where the latter outputs the activation signal a_v of $l(v)$. Thus we have

$$[\![v]\!] \;=\; [\![v]\!]_1 \parallel \mathit{AA}(a_v, \mathit{Atv}_v)$$

Here, $\mathit{Atv}_v = \{a_{v,w} \mid v \in \mathit{preds}(w) \wedge \mathit{type}(w) = \mathit{SPARE}\}$ is the set of activation signals emitted by all SPARE gates sharing $l(v)$.

SPARE. If v is labeled as a SPARE gate, with $\mathit{preds}(v) = w_1 w_2 \dots w_n$, then w_1 is its primary BE and $w_2, \dots, w_n$ are its $n-1$ spare BEs. As with the other gates, $[\![v]\!]$ is obtained from $[\![l(v)]\!]_{\mathsf{ELT}}$ by instantiating its parameters in such a way that the input signals of v connect to output signals of v's primary and

[4] If v is not a dependent event, the firing auxiliary can be omitted and we have $[\![v]\!]_1 = [\![l(v)]\!]_{\mathsf{ELT}}(a_v, f_v)$.

spare BEs. In addition, we need to find all the other SPARE gates that share any of v's spare BEs. Following the SPARE gate semantics in Section 3.2, we have

$$[\![v]\!] = [\![l(v)]\!]_{\mathsf{ELT}}(f_v, f_{w_1}, (f_{w_2}, a_{w_2,v}, P_{w_2}), \ldots, (f_{w_n}, a_{w_n,v}, P_{w_n}))$$

where $P_{w_i} = \{a_{w_i,g} \mid (w_i, g) \in E \wedge g \neq v \wedge type(g) = SPARE\}$ is the set of activation signals emitted by all other SPARE gates sharing spare $l(w_i)$.

We do not define node semantics for nodes labeled by an FDEP gate, since these are already incorporated in the semantics of their dependent BEs. Now, the semantics of a DFT is obtained by parallel composing the semantics of all (non-FDEP) nodes.

Definition 6. *The semantics of a DFT* $\mathcal{D} = (V, preds, l)$ *is the I/O-IMC* $[\![\mathcal{D}]\!] = \|_{v \in V \mid type(v) \neq FDEP} [\![v]\!]$.

To compute the reliability of $\mathcal{D}$, we are only interested in the failure of the top node T. Hence, we hide all signals except f_T, i.e. we compute $M_{\mathcal{D}} = hide\ A_{\mathcal{D}} \setminus f_T\ in\ [\![\mathcal{D}]\!]$; recall that $A_{\mathcal{D}}$ denotes the set of all actions in $\mathcal{D}$. The compositional aggregation technique described in the following section is an efficient way to derive $M_{\mathcal{D}}$.

Example 3. Figure 5 shows the I/O-IMC semantics of a DFT consisting of a SPARE gate A having a primary B and a spare C. The I/O-IMC of the DFT is obtained by parallel composing the seven I/O-IMCs shown on the figure:

$[\![A]\!] = [\![(SPARE, 2)]\!]_{\mathsf{ELT}}(f_A, f_B, (f_C, a_{C,A}, \emptyset))$
$[\![B]\!]_1 = [\![(BE, 0, \lambda, 0)]\!]_{\mathsf{ELT}}(a_B, f_B*) \| FA(f_B, f_B*, \emptyset)$ $[\![B]\!] = [\![B]\!]_1 \| AA(a_B, \emptyset)$
$[\![C]\!]_1 = [\![(BE, 0, \lambda, \mu)]\!]_{\mathsf{ELT}}(a_C, f_C*) \| FA(f_C, f_C*, \emptyset)$ $[\![C]\!] = [\![C]\!]_1 \| AA(a_C, \{a_{C,A}\})$

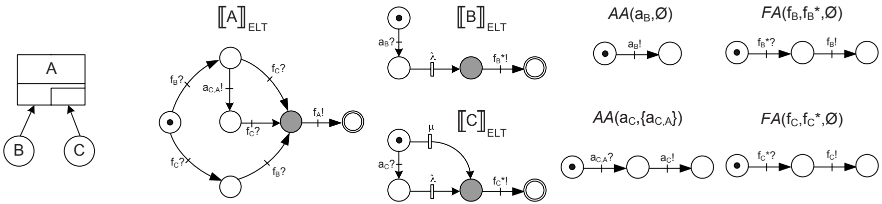


Fig. 5. A DFT and the seven I/O-IMCs that model its behavior

4 Compositional Aggregation Approach

Our compositional semantics allows one to build the I/O-IMC associated to a DFT in a component-wise fashion, leading to a significant state-space reduction. This kind of compositional aggregation approach has been previously successfully used, most notably in [16]. The compositional aggregation approach is to be contrasted with a more classical approach of model generation, such as the one

used by DIFTree, where the CTMC model of a dynamic system is generated at once and as a whole and then possibly aggregated at the end. We propose the following conversion algorithm to transform a DFT into an I/O-IMC.

1. Translate each DFT element to its corresponding (aggregated) I/O-IMC.
2. Pick two I/O-IMCs and parallel compose them (Definition 2).
3. Hide (Definition 2) output signals that will not be subsequently used (i.e. synchronized on).
4. Aggregate, using weak bisimulation (Definition 3), the I/O-IMC obtained in step 3.
5. Go to step 2 if more than one I/O-IMC is left, otherwise stop.

The choice of I/O-IMCs we make in step 2 is important as this has an impact on the size of the generated state space during the intermediate steps. In the case studies (see Section 5) we have used intuitive heuristics based on the level of interaction between models to decide the composition order. In the absence of simultaneous failures [4] in the DFT model, the algorithm results in an aggregated CTMC. However, in cases with simultaneous failures the result can be a continuous-time Markov decision process, which can be analyzed by computing bounds on the performance measure of interest [2,15].

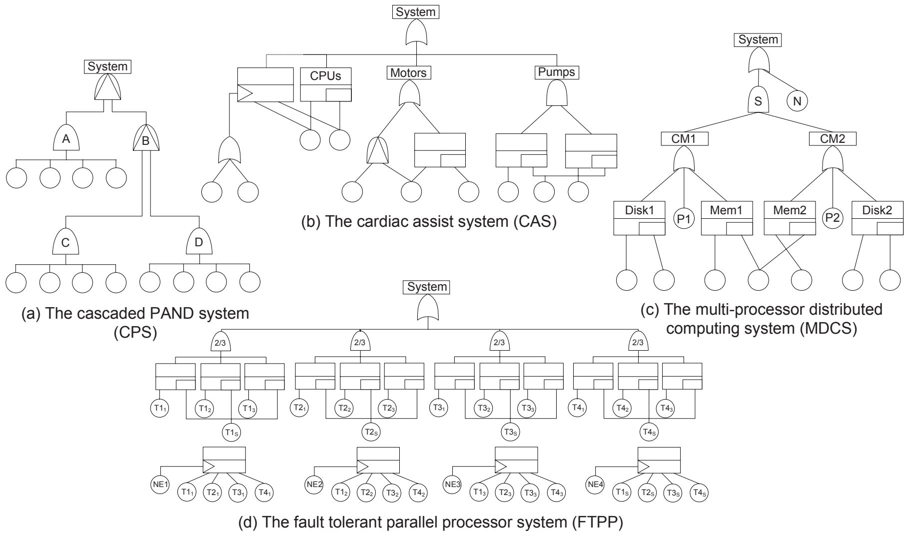
Note that originally non-determinism was not intended to be present in DFT models. Our algorithm also yields a well-specified check for DFTs: By seeing whether the I/O-IMC translation yields a CTMC or a CTMDP, one can decide if any unintended non-determinism is present in a DFT.

5 Tool Support and Case Studies

We have developed a tool named *CORAL* [3] (COmpositional Reliability and Availability anaLysis) that takes as input a DFT specified in the Galileo DFT format and computes, if there is no non-determinism in the resulting I/O-IMC, the unreliability of the DFT for given mission times. CORAL is integrated with the CADP tool set [9], which provides tool support for IMCs [13].

The tool consists of three parts:

1. The *dft2bcg* tool which uses as input the DFT file in Galileo format. This tool translates the elements of the DFT into their I/O-IMC counterparts.
2. The *composer* tool which uses as input the I/O-IMC models created by the dft2bcg tool and a composition order. The composer tool applies compositional aggregation to the I/O-IMCs according to the composition order to generate a single I/O-IMC representing the DFT's behaviour. The composition order must be supplied by the user (see Section 4).
3. The *dft_eval* tool with as its input the I/O-IMC generated by the composer tool and a number of mission-times. The dft_eval tool calculates the unreliability of the system modeled by the original DFT for the given mission-times.



(a) The cascaded PAND system (CPS)

(b) The cardiac assist system (CAS)

(c) The multi-processor distributed computing system (MDCS)

(d) The fault tolerant parallel processor system (FTPP)

Fig. 6. The DFT representations of the case studies

The composer tool uses the CADP tool set to compose, abstract (i.e. hide signals) and aggregate I/O-IMCs. In particular we have used a version of the bcg_min tool, which was adapted to aggregate I/O-IMCs using weak bisimulation (see Definition 3).

To compare the compositional aggregation (Comp-Aggr) approach with the traditional DIFtree method, we have conducted four case studies (none having non-determinism). Figure 6 shows the cascaded PAND system [6,7] (CPS), the cardiac assist system [7] (CAS), the multi-processor distributed computing system [18] (MDCS) and the fault-tolerant parallel processor [11] (FTPP).

The results of the case studies are given in Figure 7. The size of the largest model (with regard to the number of states) appearing during analysis is given for each experiment.

Case study	Analysis method	Maximum number of states	Maximum number of transitions	Unreliability (Mission-time = 1)
CPS	DIFTree	4113	24608	0.00135668
CPS	Comp-Aggr	132	426	0.00135668
CAS	DIFtree	8	10	0.657900
CAS	Comp-Aggr	36	119	0.657900
MDCS	DIFtree	253	1383	$2.00025 \cdot 10^{-9}$
MDCS	Comp-Aggr	157	756	$2.00025 \cdot 10^{-9}$
FTPP	DIFtree	32757	426826	$2.56114 \cdot 10^{-11}$
FTPP	Comp-Aggr	1325	14153	$2.56114 \cdot 10^{-11}$

Fig. 7. The results of the case studies

From these experiments one can conclude that the compositional aggregation approach to analyzing DFTs is very promising and we expect it to combat the state-space explosion effectively in many cases. The relative performance of the DIFtree and compositional aggregation approaches vary greatly for different DFTs. The DIFtree method seems to perform better for DFTs with few basic events and (possibly) many interconnections (e.g. each of the three modules in the CAS). The compositional aggregation approach seems to perform better in DFTs with symmetries (such as in the CPS and FTPP examples) and DFTs with a large number of elements and few connections (and highly modular). More research is needed to further investigate which method is best to apply under which circumstances.

6 Conclusions and Future Work

In this paper, we have formalized the semantics of DFTs using I/O-IMCs (a variant of IMCs that we have defined) and showed how a compositional aggregation approach is used to analyze DFTs. Being a first step, we have restricted our attention to basic events with exponential failure distributions, but the same approach could be taken for other probability distributions using phase-type distributions or a different underlying formalism, e.g. the one in [8]. Future work includes experimenting with other case studies and improving on the heuristic used in the order of the I/O-IMCs composition (for instance by adapting and implementing the heuristics proposed in [19]).

Acknowledgment. We thank Hong Xu from the University of Virginia for running various experiments with the Galileo tool.

References

1. Amari, S., Dill, G., Howald, E.: A new approach to solve dynamic fault trees. In: Annual Reliability and Maintainability Symposium, pp. 374–379 (January 2003)
2. Baier, C., Hermanns, H., Katoen, J.P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time markov decision processes. Theor. Comput. Sci. 345(1), 2–26 (2005)
3. Boudali, H., Crouzen, P., Stoelinga, M.I.A.: CORAL - a tool for compositional reliability and availability analysis. In: ARTIST workshop. Presented at the 19th international conference on Computer Aided Verification
4. Boudali, H., Crouzen, P., Stoelinga, M.I.A.: Dynamic fault tree analysis using input/output interactive markov chains. In: Proc. of Dependable Systems and Networks conference, UK, pp. 708–717. IEEE Computer Society, Los Alamitos (2007)
5. Boudali, H., Crouzen, P., Stoelinga, M.I.A.: Compositional analysis of dynamic fault trees. Technical report, University of Twente (to appear)
6. Boudali, H., Dugan, J.B.: A discrete-time Bayesian network reliability modeling and analysis framework. Reliability Engineering and System Safety 87(3), 337–349 (2005)

7. Boudali, H., Dugan, J.B.: A new Bayesian network approach to solve dynamic fault trees. In: Proc. of Reliability and Maintainability Symposium, pp. 451–456. IEEE, Los Alamitos (2005)
8. Bravetti, M., Gorrieri, R.: The theory of interactive generalized semi-markov processes. Theoretical Computer Science 282(1), 5–32 (2002)
9. Construction and Analysis of Distributed Processes (CADP) software tool. http://www.inrialpes.fr/vasy/cadp/
10. Coppit, D., Sullivan, K.J., Dugan, J.B.: Formal semantics of models for computational engineering: A case study on dynamic fault trees. In: Proc. of the Inter. Symp. on Software Reliability Engineering, pp. 270–282. IEEE, Los Alamitos (2000)
11. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. on Reliability 41(3), 363–377 (1992)
12. Dugan, J.B., Venkataraman, B., Gulati, R.: DIFTree: a software package for the analysis of dynamic fault tree models. In: Reliability and Maintainability Symposium, pp. 64–70 (January 1997)
13. Garavel, H., Hermanns, H.: On combining functional verification and performance evaluation using cadp. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 410–429. Springer, Heidelberg (2002)
14. Hermanns, H.: Interactive Markov Chains. LNCS, vol. 2428. Springer, Heidelberg (2002)
15. Hermanns, H., Johr, S.: Uniformity by construction in the analysis of nondeterministic stochastic systems. In: Proc. of Dependable Systems and Networks conference, UK, pp. 718–728. IEEE Computer Society, Los Alamitos (2007)
16. Hermanns, H., Katoen, J.P.: Automated compositional Markov chain generation for a plain-old telephone system. Sci. of Comp. Programming 36(1), 97–127 (2000)
17. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
18. Malhotra, M., Trivedi, K.S.: Dependability modeling using petri-nets. IEEE Transactions on Reliability 44(3), 428–440 (1995)
19. Tai, K.-C., Koppol, P.V.: An incremental approach to reachability analysis of distributed programs. In: Software Specifications & Design workshop, IEEE Computer Society Press, Los Alamitos (1993)
20. Veseley, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault tree handbook, NUREG-0492. Technical report, NASA (1981)

3-Valued Circuit SAT for STE with Automatic Refinement

Orna Grumberg, Assaf Schuster, and Avi Yadgar

Computer Science Department, Technion, Haifa, Israel

Abstract. Symbolic Trajectory Evaluation (STE) is a powerful technique for hardware model checking. It is based on a 3-valued symbolic simulation, using 0,1 and X ("unknown"), where the X is used to abstract away values of the circuit nodes.

Most STE tools are BDD-based and use a dual rail representation for the three possible values of circuit nodes. SAT-based STE tools typically use two variables for each circuit node, to comply with the dual rail representation.

In this work we present a novel 3-valued Circuit SAT-based algorithm for STE. The STE problem is translated into a Circuit SAT instance. A solution for this instance implies a contradiction between the circuit and the STE assertion. An unSAT instance implies either that the assertion holds, or that the model is too abstract to be verified. In case of a too abstract model, we propose a refinement automatically.

We implemented our 3-Valued Circuit SAT-based STE algorithm and applied it successfully to several STE examples.

1 Introduction

Symbolic Trajectory Evaluation (STE) [18] is a powerful model checking technique for hardware verification, which combines symbolic simulation with 3-valued abstraction. Consider a circuit M, described as a Directed Acyclic Graph (*DAG*) of nodes that represent gates and latches. For such a circuit, an STE assertion is of the form $A \rightarrow C$, where the *Antecedent A* imposes constraints over nodes of M at different times, and the *Consequent C* imposes requirements on M's nodes at different times.

The antecedent may introduce symbolic Boolean variables, and the assertions it imposes on M depends on them. For each node n and time t, STE computes the symbolic representation of (n, t), according to the constraints imposed by A, and the behavior of M. The nodes that are not restricted by A are initialized by STE to the value X ("unknown"), and thus an $abstraction$ of the checked model is obtained.

For an assertion $A \rightarrow C$ and a circuit M, STE may return "pass", "fail", or "unknown" (X) result. If the computed values of all nodes (n, t) comply with the requirements of C for these nodes, then the assertion passes. If, for some requirement of C on (n, t), STE computes a definite value (0 or 1) which contradicts the requirement, then "fail" is returned, together with a counterexample. If, on the other hand, STE computes X for (n, t), though C contains requirements for (n, t), then an "unknown" result is returned. The latter case means that the abstraction induced by A is too coarse, and requires some *refinement*.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 457–473, 2007.
© Springer-Verlag Berlin Heidelberg 2007

STE is successfully used in the hardware industry for verifying very large models with wide data paths [19,17,22]. The common method for performing STE is by representing the values of each node in the circuit by *Binary Decision Diagrams (BDDs)* that depend on the symbolic variables [19]. In this method, the *dual rail* representation is used, where two BDDs represent the three possible values of a node. The main drawback of this method is the unpredictability of the BDDs' sizes, and their tendency to explode when a large number of symbolic variables is used. Another limitation in common STE methods is the need for manual refinement, which is time consuming and requires close familiarity with the checked circuit.

For general model checking problems, it has been recognized for quite some time that SAT-based algorithms can often handle much larger models than BDD-based ones. It is therefore very appealing to try and implement SAT-based algorithms for STE as well. However, only a few works took this direction. In [21], *non-canonical Boolean expressions* are used instead of BDDs during the simulation, and a SAT solver is used to check if the resulting expressions meet the requirements of the STE assertion. The Boolean expressions used in this method might be too large to handle, and might require a theorem prover for reducing their size. In [2] and [4], the *dual rail* encoding is used to create a CNF formula for STE. This representation uses two Boolean variables for each node in the circuit, which we avoid in our algorithm. In [15], a 3-valued SAT solver was suggested, which did not perform well. Additionally in [15], an approximation for a 3-valued SAT solver is computed. This approximation, however, does not completely correspond to the semantics of STE. None of the methods discussed above performs automatic refinement. We further elaborate on these works in Section 7.

Particularly interesting for hardware verification is the Circuit-SAT method [8,11,10], which gets its input in the form of a circuit rather than a CNF formula. A circuit SAT solver is based on *justification* of nodes, as described in [7]. For a node n in a circuit, and a Boolean value d, it searches for a *justification* for $[n, d]$. That is, it looks for a (partial) assignment to some of the circuit inputs, under which n evaluates to d.

Our contribution is a novel framework for STE, which is based on a 3-valued justification algorithm. Our algorithm exploits the abstraction induced by using X values, without using the dual rail encoding. It is far less sensitive to the number of symbolic variables than BDD methods. Furthermore, it provides automatic refinement.

For a circuit M and an STE assertion $A \rightarrow C$, we create a circuit that represents $M \wedge A \wedge \neg C$. A justification to the value 1 at the output of the circuit represents a run of M that agrees with the constraints of A, and does not satisfy the requirements of C. This implies that the STE assertion does not hold on M. If no such justification exists, it implies either that $A \rightarrow C$ holds on M, or that the abstraction implied by A is too coarse for verifying $A \rightarrow C$. If no justification is found, our algorithm produces a core for the proof of un-justifiability. If this proof does not depend on variables whose values are X, then we conclude that $A \rightarrow C$ holds. Otherwise, the core indicates which variables should be refined.

Our algorithm uses a hybrid representation of the problem: as a set of constraints in CNF, and as the *DAG* of the circuit. The CNF representation is used for efficient *Boolean Constraint Propagation* and for learning, as in common SAT solvers [13,24]. The *DAG* representation is a higher level description of the circuit than the CNF

representation. It is used for branching as in [8,10,11], for propagating X values, and for deciding termination.

We exploit the fact that for each variable, a Boolean solver holds three possible values, $true$, $false$ and $unspecified$. Thus, we can represent each circuit node by a single variable in the CNF formula. Additional information is used to distinguish between the case the variable has the value X and the case it is unspecified. An X value at a specific node is marked so in the *DAG*. Additionally, it is represented by special constraints added to the CNF formula. New X values can be learnt both on the *DAG* and on the CNF formula. They are used to avoid traversal of abstracted parts of the circuit, thus reducing the amount of work.

We implemented our 3-valued justification algorithm on top of zChaff [13], which is a state of the art CNF SAT solver, and of [9]. We employed our tool for solving several STE problems, and compared it to other methods. Using our algorithm, we managed to solve problems that could not be solved by BDDs, and in most cases it outperformed other SAT based methods. A characterization of such problems is given in Section 6.

The rest of this paper is organized as follows: In Section 2 we present preliminaries. In sections 3 and 4 we describe our justification algorithm, and show how to use it for STE. In Section 5 we explain how automatic refinement can be performed. In Section 6 we present our experimental results, and in Sections 7 and 8 we discuss related work, conclusions, and future research.

2 Preliminaries

A hardware model M is a *circuit*, represented by a directed graph. The graph's nodes $\mathcal{N}$ are input and internal nodes, where internal nodes are latches and combinational gates. A combinational gate represents a Boolean operator. The graph of M may contain circles, but not combinational circles. Given a directed edge (n_1, n_2), we say that n_1 is an *input* of n_2. We denote by (n, t) the value of node n at time t. The value of a gate (n, t) is the result of applying its operator on the inputs of n at time t. The value of a latch (n, t) is determined by the value of its input at time $t - 1$.

2.1 Symbolic Trajectory Evaluation (STE)

In STE, a node can get a value in a quaternary domain $\mathcal{Q} = \{0, 1, X, \bot\}$. X ("unknown") is given to a node whose value cannot be determined by its inputs. $\bot$ is used to describe an over constrained node. This might occur when there is a contradiction between an external assumption on the circuit and its actual behavior.

A *state* s in M is an assignment of values from $\mathcal{Q}$ to every node, $s : \mathcal{N} \to \mathcal{Q}$.

A *trajectory* π is an infinite series of states, describing a run of M. We denote by $\pi(i), i \in \mathbb{N}$, the state at time i in π, and by $\pi(i)(n), i \in \mathbb{N}, n \in \mathcal{N}$, the value of node n in the state $\pi(i)$. $\pi^i, i \in \mathbb{N}$, denotes the suffix of π starting at time i.

AND	X	0	1	$\bot$
X	X	0	X	$\bot$
0	0	0	0	$\bot$
1	X	0	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

OR	X	0	1	$\bot$
X	X	X	1	$\bot$
0	X	0	1	$\bot$
1	1	1	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

NOT	
X	X
0	1
1	0
$\bot$	$\bot$

Fig. 1. Quaternary Operations

Let $\mathcal{V}$ be a set of *symbolic Boolean variables* over the domain $\{0,1\}$. A *symbolic expression* over $\mathcal{V}$ is an expression consisting of quaternary operations, applied to $\mathcal{V} \cup \mathcal{Q}$. The truth tables of the quaternary operators is given in Figure 1. A *symbolic state* over $\mathcal{V}$ is a mapping from each node of M to a symbolic expression. A symbolic state represents a set of states, one for each assignment to $\mathcal{V}$. A *symbolic trajectory* over $\mathcal{V}$ is a series of symbolic states, compatible with the circuit. It represents a set of trajectories, one for each assignment to $\mathcal{V}$. Given a symbolic trajectory π and an assignment ϕ to $\mathcal{V}$, $\phi(\pi)$ denotes the trajectory that is received by applying ϕ to all of the symbolic expressions in π.

A *Trajectory Evaluation Logic* (TEL) formula is defined recursively over $\mathcal{V}$ as follows: $f ::= n\ is\ p \mid f_1 \wedge f_2 \mid p \rightarrow f \mid \mathbf{N}f$, where $n \in \mathcal{N}$, p is a Boolean expression over $\mathcal{V}$, and $\mathbf{N}$ is the next time operator. The *maximal depth* of a TEL formula f is the maximal time t for which a constraint exists in f on some node n, plus 1.

Given a TEL formula f over $\mathcal{V}$, a symbolic trajectory π over $\mathcal{V}$, and an assignment ϕ to $\mathcal{V}$, we define the satisfaction of f as defined in [20]:

$[\phi, \pi \models f] = \perp \ \leftrightarrow\ \exists i \geq 0, n \in \mathcal{N} : \phi(\pi)(i)(n) = \perp$. Otherwise:
$[\phi, \pi \models n \text{ is } p] = 1 \ \leftrightarrow \phi(\pi)(0)(n) = \phi(p)$
$[\phi, \pi \models n \text{ is } p] = 0 \ \leftrightarrow\ \phi(\pi)(0)(n) \neq \phi(p)$ and $\phi(\pi)(0)(n) \in \{0,1\}$
$[\phi, \pi \models n \text{ is } p] = X \ \leftrightarrow\ \phi(\pi)(0)(n) = X$ $\qquad \phi, \pi \models p \rightarrow f \equiv \neg\phi(p) \vee \phi, \pi \models f$
$\phi, \pi \models f_1 \wedge f_2 \equiv (\phi, \pi \models f_1 \wedge \phi, \pi \models f_2)$ $\qquad \phi, \pi \models \mathbf{N}f \equiv \phi, \pi^1 \models f$

Note that given an assignment ϕ to $\mathcal{V}$, $\phi(p)$ is a constant (0 or 1).
We define the truth value of $\pi \models f$ as follows:

$[\pi \models f] = 0 \ \leftrightarrow\ \exists\phi : [\phi, \pi \models f] = 0$
$[\pi \models f] = X \ \leftrightarrow\ \forall\phi : [\phi, \pi \models f] \neq 0$ and $\exists\phi : [\phi, \pi \models f] = X$
$[\pi \models f] = 1 \ \leftrightarrow\ \forall\phi : [\phi, \pi \models f] \notin \{0, X\}$ and $\exists\phi : [\phi, \pi \models f] = 1$
$[\pi \models f] = \perp \ \leftrightarrow\ \forall\phi : [\phi, \pi \models f] = \perp$

This definition creates an order of importance between 0 and X. If there exists an assignment such that $[\phi, \pi \models f] = 0$, the truth value of $\pi \models f$ is 0, even if there are other assignments such that $[\phi, \pi \models f] = X$.

t	n_1	n_2	n_3	n_4	n_5	n_6
1	X	X	X	0	X	0
2	X	X	0	X	1	X

Fig. 2. Symbolic Simulation

STE assertions are of the form $A \rightarrow C$, where A (the antecedent) and C (the consequent) are TEL formulae. A expresses constraints on circuit nodes at specific times, and C expresses requirements that should hold on circuit nodes at specific times. $M \models (A \rightarrow C)$ iff for all trajectories π of M and assignments ϕ to $\mathcal{V}$, $[\phi, \pi \models A] = 1$ implies that $[\phi, \pi \models C] = 1$. When applying A to M, if a node n is evaluated to X, but is also constrained to a Boolean value 0 or 1 by A, then n is assigned with the value imposed by A. If n is evaluated to $0(1)$ and A constraints it to $1(0)$, then n is assigned $\perp$. As in [20], an *antecedent failure* is a case where for every trajectory π and every assignment ϕ to the symbolic variables, there is a node n and time t such that (n, t) is over constrained by π, ϕ and A. Consider the circuit in Figure 3(a), and the STE assertion $A \rightarrow C$, where $A = (n_4, 1)\ is\ 0$ and $C = (n_5, 2)\ is\ 1$. The table in Figure 2 corresponds to a symbolic simulation of this assertion. $(n_5, 1)$ is evaluated to 1, and thus the assertion holds.

Most STE implementations use the *dual rail* encoding in order to represent the 4 values. In this encoding, the value of each node (n, t) is determined by the evaluations of two Boolean functions $f^1_{n,t}, f^2_{n,t} : \mathcal{V} \to \{0, 1\}$ over the set of symbolic variables $\mathcal{V}$.

2.2 The SAT Problem

The *Boolean satisfiability problem* (SAT) is the problem of finding an assignment ϕ to a set of Boolean variables V such that a Boolean formula $\varphi(V)$ will have the value '1' under this assignment. ϕ is called a *satisfying assignment* for φ.

We discuss formulae presented in the conjunctive normal form (CNF). That is, φ is a conjunction of clauses, where each clause is a disjunction of literals over V. A literal l is an instance of a variable or its negation: $l \in \{v, \neg v \mid v \in V\}$. We shall consider a clause as a set of literals, and a formula as a set of clauses.

A clause cl is satisfied under an assignment ϕ iff $\exists l \in cl, \phi(l) = 1$. For a formula φ given in CNF, an assignment satisfies φ iff it satisfies all of its clauses. Hence, if, under an assignment ϕ (or a partial assignment), all of the literals of some clause in φ are 0, than ϕ does not satisfy φ. We call this situation a *conflict*.

For an unsatisfiable formula $\varphi = \mathcal{C}$, where $\mathcal{C}$ is a set of clauses, an *unSAT core* $\mathcal{C}'$ is a set of clauses $\mathcal{C}' \subseteq \mathcal{C}$ such that $\mathcal{C}'$ is unSAT.

For two clauses $cl_1 = (w_1, v_1 \ldots v_n)$ and $cl_2 = (\neg w_1, z_1 \ldots z_m)$ ($(v_1 \ldots v_n)$ and $(z_1 \ldots z_m)$ are not necessarily disjoint), their *resolvent* is $cl_{res} = (v_1 \ldots v_n) \cup (z_1 \ldots z_m)$. It is easy to show that $cl_1 \wedge cl_2 \wedge cl_{res} \equiv cl_1 \wedge cl_2$. For an unSAT formula, there exists a series of resolutions that leads to the empty clause. This series is the proof of the formula's unsatisfiability. This series is called *resolution tree*, where the root is the empty clause, and the rest of the nodes are the clauses in the series that led to it. The antecedents of a node are the clauses that were involved in the resolution that create it. The leaves are a subset of the original clauses of the formula. This subset of clauses is an unSAT core.

Davis-Putnam-Logemann-Loveland Backtrack Search (DPLL). We begin by describing the *Boolean Constraint Propagation* (*bcp*). Given a partial assignment ϕ and a clause cl, if there is one literal $l \in cl$ with no value, while the rest of the literals are all 0, then in order to avoid a conflict, ϕ must be extended such that $\phi(l) = 1$. cl is called a *unit clause*, and the assignment to l is called an *implication*. *bcp* computes all possible implications at a given moment. This procedure is efficiently implemented in [13,23,12].

The DPLL algorithm [6,5] iteratively chooses an assignment to some variable, and computes its implications. If no conflict occurs, a new assignment is chosen, and so on. If a conflict occurs, the algorithm invalidates the last chosen assignment, and tries another one instead. Choosing an assignment to a variable is called *branching*, and invalidating a decision is called *backtracking*. DPLL terminates in the following cases: If all of the variables are assigned without causing a conflict, φ is satisfiable, and the current assignment to the variables is a satisfying assignment. On the other hand, if a conflict occurs but there are no decisions to invalidate, it is concluded that φ is unsatisfiable.

Optimizing DPLL

Modern SAT solvers apply several optimization on the basic DPLL backtrack search. Such optimizations are conflict based learning, conflict driven backtracking, restarts and more. These optimizations result in a significant speedup of the SAT solving tools.

Learning is performed upon the occurrence of a conflict. At this point, two clauses implicate conflicting values to the same variable. The resolution of the clauses describes the cause to the conflict. Thus, it is added to the formula, and prevents the conflict from reoccurring. Different learning strategies yield different conflict clauses. *1UIP* is a common and very efficient strategy[24].

2.3 Bounded Model Checking

We shall briefly describe Bounded Model Checking (BMC)[1] for a model M and a property P, which is a commonly used model checking technique. In BMC, the transition relation of M is described as a Boolean formula $R(\overline{x}, \overline{x}')$, where $\overline{x}$ and $\overline{x}'$ are the current and next state variables, respectively. The property P is also described as a Boolean formula $P(\overline{x})$. Additionally, the set of initial states of M is described by a boolean formula $I_0(\overline{x})$.

BMC is an iterative algorithm. At iteration k, the formula $\varphi_k = I_0(\overline{x_0}) \bigwedge_{i=0}^{k-1} R(\overline{x_i}, \overline{x_{i+1}}) \wedge \neg P(\overline{x_k})$ is constructed, and is given to a SAT solver. A solution for φ_k represents a path of length k from an initial state in M, along which the property P does not hold. Thus, a solution represents a bug in the model. If φ_k is unSAT, then no such path of length k exists, and the algorithm continues to the next iteration.

If P describes only finite paths, BMC terminates when k reaches the length of the longest path in P. Otherwise, BMC terminates when k reaches the diameter of M. In practice, the diameter of the model is not reached, and BMC stops due to memory limits or timeouts.

2.4 Circuit SAT Solvers

Justification of Assignments. For a circuit node n and value d, we say that $[n, d]$ is *justified* by the inputs to n if d is implied by them according to the semantics of n. On that case, we say that n is justified by its inputs. For example, consider a node n, associated with an "AND" operator, and its inputs $in_1 \ldots in_m$. $[n, 0]$ is justified iff $\exists\ i\ s.t.\ in_i = 0$, regardless of the values of the rest of the inputs. $[n, 1]$ is justified iff $\forall\ i,\ in_i = 1$. We generalize this definition for the set of nodes in the graph that may effect the value of n. When given a (partial) assignment to the inputs of a circuit, we say that $[n, d]$ is justified if d is implied by those inputs. An input is thus trivially justified. Throughout the rest of this paper, an *assignment* is considered a *partial assignment*.

Circuit SAT. A *Circuit SAT Solver* [8,11,10] is a solver that uses a graph representation of the circuit instead of a CNF formula. Given a circuit, a node n and a value d. A circuit SAT solver returns a justification for $[n, d]$ if one exists, or "unjustifiable" otherwise. Branching, *bcp*, learning and other procedures are performed over the graph.

3 3-Valued Justification

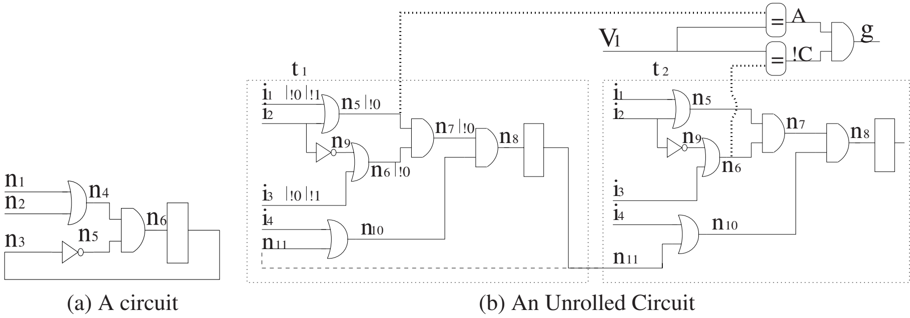
In this section we describe our 3-valued algorithm for justifying a value of a node in a circuit. Our algorithm uses a dual representation of the circuit. The first is a graph G of the circuit's gates and latches, and the other is a CNF description of it, denoted φ. φ is built as described in [1]. ψ^1_{and} in Figure 6 is an example for a CNF description of an "AND" gate n, with inputs in_1 and in_2. There is a 1-1 mapping between the variables of φ and the nodes of G. Thus, we can refer to a node by its corresponding variable and vice versa. The graph and the CNF representations are maintained throughout the computation in order to keep the correlation between them. The pseudo code of our algorithm is given in Figure 4. Throughout this Section we refer to the example of circuit t_1 in Figure 3(b).

3.1 *not-0* and *not-1* Variables

When working in a 3-valued domain, a variable being *not-1* does not imply being 0, and vice versa. Therefore, we introduce the notions of *not-0* and *not-1*. A variables is *not-0* or *not-1* if it is not allowed to be assigned 0 or 1, respectively. Consequently, a node which is both *not-0* and *not-1* can only be assigned X. Such restrictions can be derived from external constraints, or learned during the search. We denote *not-0* and *not-1* by $|_{!0}$ and $|_{!1}$, respectively.

In the graph representation G we have a mechanism for marking $|_{!0}$ and $|_{!1}$ nodes. We need a mechanism for marking and manipulating $|_{!0}$ and $|_{!1}$ variables in φ. Therefore, we do not consider the clauses to be sets of literals, as defined in Section 2.2. Instead, we consider the clauses to be *multi-sets* of literals. The definition of a conflict and constraint propagation remain as in Section 2.2. $|_{!0}$ and $|_{!1}$ variables are marked by adding the clauses $(n \vee n)$ and $(\neg n \vee \neg n)$, respectively. When applying constraint propagation, each of these clauses causes a conflict if we try to assign n with a value 0 or 1, respectively. However, since they never become *unit clauses*, neither of the clauses forces any value on n. In the Boolean domain, the propositional formula $(n \vee n) \wedge (\neg n \vee \neg n)$ is not *satisfiable*. In contrast, in our algorithm, these clauses correctly represent X values, since our algorithm does not necessarily *satisfy* all the clauses, as we describe in Section 3.2. In particular, our algorithm does not assign a value to a variable that is both $|_{!0}$ and $|_{!1}$. Note that though a variable may have multiple instances in a clause, we only have to distinguish between single and multiple instances. Thus, if a variable has more than one instance in a clause, we only keep two instances.

$|_{!0}$ and $|_{!1}$ restrictions are propagated on G. Consider n_5 in the example. i_1 is $|_{!0}$. Therefore, n_5 cannot be assigned 0, and is also $|_{!0}$. Similarly, n_6 is also $|_{!0}$. In addition, since all the inputs to n_7 are $|_{!0}$, n_7 is also $|_{!0}$. We do not propagate the restrictions directly on φ. However, when propagating them on G, we also create the appropriate clauses for the implied restrictions, and add them to φ. For example, i_1 is $|_{!0}$ implies that n_5 is $|_{!0}$. Thus we add the clause (n_5, n_5). Additionally, i_1 being $|_{!1}$ changes the relation between i_2 and n_5: Since i_1 is $|_{!1}$, $n_5 = 1$ implies $i_2 = 1$. This new relation is expressed by the clause $(i_2, \neg n_5, \neg n_5)$. Note that i_2 is only one of the inputs to an "OR" gate, and therefore $i_2 = 0$ *should not* imply $n_5 = 0$. The two instances of $\neg n_5$ prevent that. Similarly, the clause $(n_9, \neg n_6, \neg n_6)$ is created.



(a) A circuit

(b) An Unrolled Circuit

Fig. 3. Circuits

In section 2.2 we defined the *resolution tree* for clauses that are created by resolution. In our context, clauses can be created by propagating $|_{!0}$ and $|_{!1}$ on G. The propagation on G corresponds to the semantics of the nodes, which is also expressed by the clauses of the nodes in φ. Thus, the generated clauses are considered as the result of applying resolution on the relevant clauses in φ. In the example, the clause (n_5, n_5) can be created by applying resolution on the clauses (i_1, i_1) and $(\neg i_1, n_5)$, and on their resolvent and $(\neg i_1, n_5)$ again. The definition of the resolution tree thus remains unchanged.

3.2 3-Valued Justification Algorithm

Given a *DAG* G of a circuit, a CNF description of it φ, a node $r \in G$, and a Boolean value d, our 3-valued justification algorithm (3VJA) returns a justifying assignment for $[r, d]$, or *unjustifiable* if $[r, d]$ is not justifiable. We call r the *root* of G. 3VJA performs an iterative backtrack search over G. The information in G about the structure of the model is used for branching during the search, and allows propagation of $|_{!0}$ and $|_{!1}$ restrictions. It is also used for correct termination of the algorithm. The CNF representation φ is used for efficient constraint propagation, detection of conflicts and for learning. X values are described by using clauses representing the $|_{!0}$ and $|_{!1}$ constraints, and can be learned during the solving process. Next we describe and explain it. We refer to the pseudo-code given in Figure 4.

3VJA (G,φ,n,d)
1) while true
2) if ($\neg$branch on G)
3)   return justification
4) if (*bcp on* φ $\Rightarrow$*conflict*){
5)   *learn conflict clause*
6)   if learned X clause {
7)   mark X on G
8)   propagate X on G
9)   *add clauses to* φ
10)   }
11)   if possible
12)   backtrack
13)   else
14)   return unjustifiable
15) }

Fig. 4. 3VJA. Lines 2, 7 and 8 are executed on G. Lines 4, 5 and 9 are executed on φ.

We begin by describing the branching procedure used in line 2 of 3VJA. This is a 3-valued variation of the justification procedure described in [7]. Our branching procedure traverses G, assigning the nodes with values in a pre-order manner, starting from the root. For each node it chooses values only to its inputs that are needed in order to justify it. The rest of the input nodes are not assigned and are not traversed. The branching procedure does not

assign $|_{!0}$ and $|_{!1}$ nodes with the values 0 and 1, respectively. In the example, justification of $[n_8, 0]$ will not be done by assigning $n_7 = 0$. If it is impossible to justify a node with any of its inputs, 3VJA invalidates the last branching and tries another path. The root of G is assigned either 1 or 0. Therefore, we never justify an X value, nor do we have to assign a node with the value X for justification of a node.

After each branching, the assigned value has to be propagated through the variables (line 4). We exploit the fact that a value of a variable in a Boolean SAT solver can be either 1, 0, or $unassigned$ in order to represent 3 values in a Boolean context. Thus, we use φ to propagate the branching assignment through the circuit. The propagation and the definition of a *conflict* remain as defined for Boolean SAT. If the propagation does not cause a conflict, 3VJA continues to the next iteration. If a conflict occurs, 3VJA learns a new conflict clause, and backtracks accordingly.

When a conflict occurs (line 5), the resolvent of the clauses that were involved in the conflict is added to the problem. In the 3-valued context, we define the resolvent of clauses $cl_1 = (w_1, v_1 \dots v_n)$ and $cl_2 = (\neg w_1, z_1 \dots z_m)$ to be $cl^3_{res} = (v_1 \dots v_n, z_1 \dots z_m)$. Note that the clauses are considered to be *multi-sets*, and clauses may have multiple instances of a variable. For example, the resolvent of (v_1, v_2, v_3) and $(\neg v_1, v_3, v_4)$ is (v_2, v_3, v_3, v_4). Adding a resolvent as defined above to φ does not change the set of justifying assignments to $[r, d]$. Due to space limitations, we omit the proof of this claim. This resolution is similar to the resolution described in [15]. We elaborate on this in Section 7.

It is possible to learn conflict clauses such as (n, n) and $(\neg n, \neg n)$. As described in lines 6-9, when learning such clauses, we mark the corresponding nodes in G as $|_{!0}$ and $|_{!1}$, respectively. We propagate this information on G, thus extracting additional information from the learned conflict clause. We then generate the appropriate clauses, as described in Section 3.1, and thus maintain the correlation between G and φ.

By learning (n, n) and $(\neg n, \neg n)$ clauses we can conclude that a node is forced to X even if such a conclusion can not be explicitly derived from G. This is an important result, because it prevents the branching procedure from trying to use the constrained node for justification in the future. It also helps detecting conflicts earlier. We demonstrate this on our example. Assume that the branching procedure assigned $n_8 = 1$. A possible series of implications from this assignment is $n_7 = 1, n_5 = 1, n_6 = 1, n_9 = 1, i_2 = 0$. Other series could be computed, depending on the order of computing the implications. The result of these implication is that all the literals in the clause $(i_2, \neg n_5, \neg n_5)$ are 0. That is, a conflict has occurred. We show the series of resolutions that is performed upon this occurrence in Figure 5. The learned conflict clause is $(\neg n_7, \neg n_7)$, and it is added to φ. We also mark that n_7 is $|_{!1}$ in G and propagate it, implying n_8 is $|_{!1}$ and n_{11} is $|_{!1}$. These implications are also added as the clauses $(\neg n_8, \neg n_8)$ and $(\neg n_{11}, \neg n_{11})$ to φ. The result is that n_7 is assigned X, and n_8 and n_{11} are $|_{!1}$.

Note that unlike implications computed by constraint propagation, nodes that are assigned X remain X throughout the solving process, and are not effected by backtracking. This is because the conclusion about X nodes is derived from the problem itself, regardless of the current partial assignment. Therefore, a mechanism for invalidating X assignments is not required.

1. $(i_2, \neg n_5, \neg n_5)$	5. $(\neg n_7, n_5)$	9. $(8 \uplus 4) \uplus 4 = (\neg n_7, \neg n_7, \neg n_5, \neg n_5)$
2. $(\neg n_9, \neg i_2)$	6. $(\neg n_8, n_7)$	10. $(9 \uplus 5) \uplus 5 = (\neg n_7, \neg n_7)$
3. $(n_9, \neg n_6, \neg n_6)$	7. $1 \uplus 2 = (\neg n_9, \neg n_5, \neg n_5)$	11. $(\neg n_8, \neg n_8)$
4. $(\neg n_7, n_6)$	8. $7 \uplus 3 = (\neg n_5, \neg n_5, \neg n_6, \neg n_6)$	12. $(\neg n_{11}, \neg n_{11})$

Fig. 5. Learning X clauses. $\uplus$ denotes the *resolution* operation. Refer to the circuit t_1 in Figure 3(b). $n_8 = 1$, implies $n_7 = 1, n_5 = 1, n_6 = 1, n_9 = 1, i_2 = 0$, by using clauses $1 - 6$. Clauses $1, 3$ originate from propagating $|_{!1}$ for i_1 and i_3, respectively, as described in Section 3.1. Clause 2 is a part of the description of the "NOT" gate. Clauses $4, 5$ and 6 are the relevant clauses of the "AND" and "OR" gates. Clauses $7-10$ are created by applying *resolution* on the original clauses. Clause 10 is the conflict clause derived by the *1UIP* strategy. Having n_7 is $|_{!1}$ on G implies that n_8 and n_{11} are $|_{!1}$, and thus clauses 11 and 12 are created.

A justifying assignment for $[r, d]$ is found when we complete the traversal of G (line 3). This traversal does not necessarily include all the nodes in G, but rather only the nodes that were required for this justification. Alternatively, if the traversal can not be completed, we conclude that $[r, d]$ can not be justified.

4 STE with 3-Valued Justification

In this section we show how to employ our 3-valued justification algorithm for STE. We start by describing the construction of circuits to represent an STE problem, and then show how to use the algorithm from Section 3 for solving it.

4.1 Constructing Circuits for STE Assertions

Consider a model circuit M, and an STE assertion $A \rightarrow C$. A and C are given in TEL, as described in Section 2.1. In order to prove or falsify the assertion, M has to be simulated k times, where k is the *maximal depth* of A and C.

We create a new graph by unrolling M k times. Each node $n \in M$ has k instances in the new graph. The ith instance of node n represents node n at time i. In the new graph, the connectivity of the input and gate nodes remains the same. The latches are connected such that the input to a latch at time t are the nodes at time $t-1$, and the latch is an input to nodes at time t. Due to the new connectivity of the latches, and since M does not have combinational circles, the unrolled graph is a *DAG*. The inputs to the new graph are k instances of each of the inputs to the circuit. In Figure 3(b), an unrolling of a circuit is presented. t_1 and t_2 are two instances of the circuit. The inputs to the latch n_{11} in t_2 are the nodes of t_1, thus eliminating the circle in t_1. The inputs to the new circuit are the two instances of $i_1 - i_4$. From herein we denote by M the unrolled graph of the circuit.

As mentioned before, A and C are given in TEL. Therefore, we can construct combinational circuits that represent them. The inputs to these circuits are nodes in M, and new constructed nodes that represent the symbolic variables of the STE assertion. The output of each circuit, denoted the *root* of the circuit, equals to the evaluation of the corresponding TEL formula. For example, for a TEL formula $A = (n, i)\ is\ V_1$, the input

to the circuit of A is the ith instance of the circuit node n in M, and a node associated with the symbolic variable V_1. The root of the circuit is 1 if the input values are equal, and 0 otherwise. The construction of circuits for $n \ is \ p$, $f_1 \wedge f_2$ and $p \rightarrow f$ are trivial. The circuit for $f = Nf'$ is derived by constructing the circuit for f', and replacing each of its input nodes (n, t) by the node $(n, t+1)$. Note that each symbolic variable has only one instance. Also note, that the constructed circuits for A and C are also *DAG*s. From herein we denote by A and C the corresponding circuits, respectively. Also, we refer to a node n at time i by the name of the ith instance of n in M, instead of by (n, i).

We construct a circuit for $M \wedge A$ by connecting the relevant nodes in M to the inputs of A. The inputs to $M \wedge A$ are the k instances of the inputs to the hardware model, and the symbolic variables defined in A. The assertions that are imposed by A are in fact assumptions on the circuit. As defined in Section 2.1, a node n is assigned the Boolean value imposed on it by A, even if its evaluation on the circuit is X. In our algorithm, this means that the values of n do not have to be justified, and should not propagate from n to its inputs. We mark asserted nodes in the graph, such that X values do not propagate through them, and the branching procedure considers them justified, not trying to assign their inputs. Additionally, we construct the CNF clauses for an asserted node such that they allow forward propagation only. This is demonstrated in Figure 6. For a node $n = in_1 \wedge in_2$, we create ψ^2_{and} instead of ψ^1_{and}. For example, if n is assigned the value 1 by A, none of the clauses propagates this value to in_1 and in_2. On the other hand, forward propagation is still implied.

$$\psi^1_{and} = (n, \neg in_1, \neg in_2) \wedge (\neg n, in_1) \wedge (\neg n, in_2)$$
$$\psi^2_{and} = (n, \neg in_1, \neg in_1, \neg in_2, \neg in_2) \wedge (\neg n, in_1, in_1) \wedge (\neg n, in_2, in_2)$$

Fig. 6. A CNF representation of an "AND" gate $n = in_1 \wedge in_2$. ψ^1_{and} propagates implications in both directions. ψ^2_{and} propagates implications only forwards.

We construct the circuit $\Gamma = M \wedge A \wedge \neg C$ by connecting the relevant nodes in $M \wedge A$ to the inputs of C. As with $M \wedge A$, the inputs to the new circuit are the inputs to M and the symbolic variables. We create a new "AND" node such that its inputs are the roots of A and $\neg C$. This node is considered the root of Γ. An example for such a construction is given in Figure 3(b). The node associated with "=" represents a combinational circuit that evaluate to 1 if the values in the inputs are equal, and 0 otherwise. Consider an assertion $A \rightarrow C$ such that $A = (n_5, 1) \ is \ V_1$, and $C = (n_6, 2) \ is \ \neg V_1$. t_1 and t_2 are the unrolled circuit. The node A is the root of the circuit that corresponds to A. The inputs to this circuit are $(n_5, 1)$ and V_1. $!C$ is the root of the circuit that corresponds to $\neg C$. The inputs to this circuit are $(n_6, 2)$ and V_1. The node g evaluates to Γ.

4.2 Running STE

We first verify that A does not cause an *antecedent failure* with M. Therefore, we have to verify that there is at least one run of the model that does not conflict with the assertions from A. Consider the circuit $M \wedge A$, described in the previous section. We apply 3VJA for justifying $[a, 1]$, where a is the root of A. A justifying assignment for this

problem represents a run of M that satisfies the constraints imposed by A. Therefore, if such an assignment is found, we conclude that there is no antecedent failure. If the problem is unjustifiable, then no such run exists, which means an antecedent failure.

Assuming no antecedent failure was found,we apply 3VJA on $[\gamma, 1]$, where γ is the root of Γ, defined in the previous section.

If a justifying assignment is found, it represents an assignment to the inputs of Γ that makes γ evaluate to 1. This assignment represents a run of M that satisfies the constraints imposed by A, but contradicts the requirements of C. Such an assignment means that the STE assertion $A \rightarrow C$ does not hold in M.

If $[\gamma, 1]$ is unjustifiable, an empty clause was learned. We extract the unSAT core from the resolution tree of the empty clause, and check if it contains clauses for $|_{!0}$ or $|_{!1}$ nodes, that originate from A. If there are no such clauses in the core, then no X value has participated in proving the unjustifiability of $[\gamma, 1]$. Therefore, we conclude that there is no run of M that complies with the restrictions of A, but does not satisfy the requirements of C. That is, the STE assertion $A \rightarrow C$ holds in M. On the other hand, if the unSAT core includes clauses for $|_{!0}$ or $|_{!1}$ nodes that originate from A, then the proof for unjustifiability depends on X values. In that case, it might be that we did not find a counter example for $A \rightarrow C$ due to a too coarse abstraction. Therefore, we have to refine the model in order to prove or falsify the STE assertion.

Note that in case of an unjustifiability proof that depends on X values from A, another proof that does not depend on X values might exist. Therefore, it might be possible to prove the STE assertion without refining the model. We could avoid this by changing the justification and traversing larger portions of the circuit. We then have a trade off between light-weight justification with more refinements, and heavy-weight justification. In our current algorithm, we choose to perform the light-weight justification and refine the model if needed. We discuss refinement in Section 5.

5 Refinement

A major strength of STE is the use of abstraction. The abstraction is determined by assigning nodes in the model M with the value X by A, the antecedent of the STE assertion. However, if the abstraction is too coarse, there is not enough information for proving or falsifying the STE assertion. We present a "CEGAR" [3] approach for refining such assertions.

For an unjustifiable instance given to 3VJA, the resolution tree, derived for it, is the proof that the instance is unjustifiable. We define a *spurious* proof to be a resolution tree such that the unSAT core defined by it includes clauses for X nodes, that originate from the antecedent A of the STE assertion.

In our STE method, a too coarse abstraction results in an unjustifiable instance with a spurious proof of unjustifiability. By associating the X nodes in the unSAT core with symbolic variables, we refine the model and invalidate the current spurious proof.

Refining only X nodes in the unSAT core, only the variables needed to eliminate the spurious proof are refined. This means that for an X node n, we only add variables for X nodes that took part in implying X on n, rather than all the X nodes in the cone of influence of n. Refer to the example in Figure 3(b), and consider A that assigns, at t_1, X to i_1, i_3 and i_4, and 1 to n_{11}. This implies $n_{10} = 1$, and n_8 is $|_{!0}$. When trying to justify

$n_8 = 1$, as seen in Section 3.2, we learn that $n_7 = X$, and n_8 is $|_{!1}$. Therefore, $n_8 = X$. Note that the conclusion that n_8 is $|_{!1}$ is independent of i_2, i_4 and n_{10}. If $n_8 = X$ takes part in the proof that the whole circuit is unjustifiable, i_1 and i_3 will be in the unSAT core, while i_2 and i_4 will not. Thus, when refining, we will not add a variables for i_2 and i_4.

Our refinement eliminates the proof of unjustifiability that was found. Running the justification algorithm again, we either find another spurious proof that has to be refined, a concrete proof of unjustifiability, or proof of justifiability (a justifying assignment).

6 Experimental Results

For evaluating our justification algorithm 3VJA, presented in Section 3.2, we implemented it on top of zChaff [13], a state of the art SAT solver, and [9], and used it for STE, as described in Section 4. For comparison, we used the dual rail encoding for solving SAT based STE [15], and *Forte*, a BDD based STE tool by Intel [19]. Additionally, we used BMC for solving the benchmarks, considering the STE assertions as an LTL formulae. For the SAT based STE and for BMC, we used the same SAT solver zChaff, on top of which we implemented our algorithm. All experiments use dedicated computers with 3.2Ghz Intel Pentium CPU, and 3GB RAM, running Linux operating system. Time out was set to one hour.

For our experiments we used the *Memory* and *CAM* circuits from Intel's GSTE tutorial, which are large enough to demonstrate various characteristics of the algorithm. The *Content Addressable Memory* (CAM) has 16 entries, 64 bits data width, and 8 bits tag width. The memory circuit has a 6 bits address width and 128 bits data width.

The results of our experiments are presented in Table 1. We verified the *associative read* property of the CAM by using "full", "plain" and "cam" symbolic indexing schemes, as defined in [14]. Additionally, we checked the CAM and the memory against series of multiple write and read operations. Each assertion has a different set of symbolic variables and a different depth. Assertions $1-14$ were verified, whereas assertions $15-25$ were falsified. Columns 3V, BDD, DR and BMC present the solving time of our 3VJA based STE, BDD based STE, Dual Rail SAT based STE, and BMC, respectively.

3VJA has outperformed the *BDD based algorithm* on most of the assertions, especially the harder ones. Compared to the BDD algorithm, 3VJA is far less sensitive to the number of symbolic variables. Consider assertions $1-3$ and $4-6$. These assertions are different encodings for the associative read operation of CAM, defined for depth 2 and 6, respectively. Each encoding of the assertion requires a different number of symbolic variables. On both depths, the BDD algorithm timed out for "full" and "plain" encodings, while 3VJA solved the problems in seconds. On the other hand, 3VJA is more sensitive to the number of nodes in the circuit, and thus to the depth of the assertions, than BDDs. This is also a characteristic of the other SAT based algorithms, and is demonstrated by assertions 4 and $10-11$, relatively to 1 and 9, respectively. In each of these cases, a similar assertion is checked to different depths. The number of symbolic variables is about the same, but the number of nodes in the circuit grows. This affects the SAT based algorithms more than it affects the BDD based algorithm.

Table 1. Experimental Results. D is the depth of the STE assertion, #Vars is the number of symbolic variables, #N is the number of circuit nodes in thousands, and 3V, BDD, DR and BMC are the times required by 3VJA, BDD STE, Dual Rail SAT STE, and BMC, respectively.

Verification					Time (s)			
	Assertion	D	# vars	#N x10^3	3V	BDD	DR	BMC
1	CAM cam	2	124	5	4	0.5	5	1
2	CAM plain	2	204	5	2	T.O	1	1
3	CAM full	2	1160	5	1	T.O	1	1
4	CAM cam	6	128	15	31	1	94	87
5	CAM plain	6	208	15	15	T.O	27	30
6	CAM full	6	1164	15	14	T.O	26	34
7	CAM 1	10	152	25	349	5	513	493
8	CAM 2	10	242	25	45	T.O	537	473
9	Mem 1	2	86	110	5	1	9	2
10	Mem 1	5	104	260	773	3	413	320
11	Mem 1	11	164	550	T.O.	9	T.O	T.O
13	Mem 2	5	304	260	54	455	72	52
14	Mem 2	11	334	550	77	523	142	81

Falsification					Time (s)			
	Assertion	D	# Vars	#N x10^3	3V	BDD	DR	BMC
15	CAM 3	4	320	10	10	437	5	1
16	CAM 4	4	260	10	14	209	19	13
16	CAM 5	5	72	10	32	3	12	3
17	Mem 3	2	134	110	280	282	832	327
18	Mem 3	5	134	260	536	436	T.O	2753
19	Mem 3	10	134	550	1943	641	T.O	T.O
20	Mem 3	15	134	770	T.O	943	T.O	T.O
21	Mem 4	5	168	260	536	T.O	343	2854
22	Mem 4	10	168	550	1765	T.O	2248	3004
23	Mem 4	15	168	770	2064	T.O	3440	T.O
24	Mem 5	10	670	550	3276	T.O	3555	T.O
25	Mem 5	15	670	770	T.O	T.O	T.O	T.O

Note, however, that in case of a failed assertion with many symbolic variables, the BDD method may fail due to the need to compute the values of all nodes up to the depth of the contradiction, while a SAT based algorithm only has to find one erroneous path. This is demonstrated by assertions $15, 16$ and $21 - 25$.

We see that *BMC* outperforms the *dual rail* method in most of the cases, especially for verification. The dual rail representation uses two Boolean variables to represent each node. The result is a very large SAT instance, which is harder to solve. This result matches the results in [15].

3VJA outperforms BMC in most cases, especially in falsification. While not very sensitive to the number of symbolic variables, BMC does not use X values, and thus does not use an abstraction. This makes BMC more sensitive to the width of data paths, and the depth of the assertions. For verification, we expected 3VJA to return "unjustifiable" faster than BMC, since the justification is constrained by the X nodes. However, in a few cases, such as 10, we had to refine the model multiple times until a concrete proof for unjustifiability was found. In these experiments, refinement was performed manually. In 11, we could not find such a proof within the time limit. For falsification, we see a clear advantage to 3VJA. This can be explained by the fact that 3VJA does not try to assign values to X nodes, and thus does not traverse large portions of the circuits. This advantage increases with the number of nodes that are abstracted out by the STE assertion, and is demonstrated by assertions $17 - 25$.

7 Related Work

SAT based methods for STE were previously suggested in [2], [21] [4], and [15].

In [21], *non-canonical Boolean expressions* are used to represent the symbolic expressions of the circuit's nodes during the simulation. At the end of the simulation, a SAT solver is used to check if the resulting expressions meet the requirements of the STE assertion. In this method, the expressions associated with the nodes may grow very

large, and even become too large to handle. In such cases, a theorem prover has to be used in order to simplify them. This method is inherently different than 3VJA.

In [2], the *dual rail* encoding is used to create a CNF formula for STE. This construction is referred to in [4] as *simulation based SAT STE*. In [4], a different construction is suggested, and is referred to as *constraint based STE*. The *constraint based* construction is equivalent to the construction presented in [1], that we used in our work. This construction forces propagation of Boolean values through the gates of the circuit. The *simulation based* construction forces propagation of X values as well, and results in much larger CNF formulae. In [4], it is shown that the *constraint based* construction outperforms the *simulation based* construction. We compared 3VJA to the *constraint based* construction in Section 6.

In [15], the constraint based construction is solved by a 3-valued SAT solver. In this work, Boolean variables of a SAT solver represent 3 values, considering an "unassigned" variable as X. The definition of satisfiability is changed respectively. In this work, clauses are regarded as multi-sets, and the definition of the resolution is also changed. Note that in our work we do not change the definition of the satisfiability of a formula. Instead, our algorithm does not *satisfy* the formula, but rather *justifies* the root of the graph. Moreover, in our work we distinguish between unassigned nodes and nodes assigned with X. This distinction allows us to propagate X values, and to suggest an automatic refinement for too coarse abstractions. Additionally, while the 3-valued resolution defined in our paper is similar to the resolution defined in [15], the reasons for their correctness are different. As described in [15], modifying the SAT solver to fit the new definition of satisfaction and resolution did not yield good performance.

Additionally in [15], an approximation to 3-valued SAT is computed. This algorithm corresponds to a different semantics than the STE semantics, and an assertion that holds by this algorithm might not hold in STE semantics. This algorithm is also not suitable for refining STE assertions.

An automatic refinement scheme was suggested in [20]. This refinement scheme chooses a nodes that is assigned with X, and tries to choose a small set of inputs such that this node will evaluate to 0 or 1. In [16], a method for assisting manual refinement is presented. Our refinement scheme eliminates a spurious proof of unjustifiability of the circuit in each iteration, and is inherently different than these methods.

8 Conclusions and Future Work

We have presented a 3-valued justification algorithm, 3VJA, that uses a *DAG* and a CNF descriptions of a circuit, and finds a 3-valued justification for the value at the root. We showed how to use 3VJA for STE.

We implemented 3VJA and compared it to other STE tools. It is our opinion that 3VJA is a valuable complement to BDD based STE, especially for falsification, as is the case in other model checking problems. 3VJA is far less sensitive to the number of symbolic variables than BDD methods. Moreover, for falsification, 3VJA may find an erroneous path quickly, while a BDD engine has to compute the values of all the nodes in all the iterations prior to the contradiction.

We compared 3VJA to other SAT based algorithms and in many cases showed a significant speedup. This is the result of introducing the notion of X into the Boolean context, without doubling the number of variables that are used, by propagating X values over a graph representation of the circuit, and by learning X values trough 3-valued resolution. While BMC is a powerful model checking method, it is considered useful mainly for falsification of "shallow" bugs. Exploiting the abstraction used in STE, 3VJA may extend the capabilities of BMC as well.

Last, we showed that 3VJA can be used for an automatic refinement scheme of STE assertions. This scheme takes a "CEGAR" approach, where the spurious counter examples are proofs of unjustifiability of the problem, that depend on X values. The refinement adds symbolic variables to the nodes that are needed in order to eliminate the proof. We intend to research heuristics for minimizing the set of variables that have to be refined for eliminating spurious counter examples.

3VJA also allows us to address the problem of vacuity in STE. Given an STE assertion $A \rightarrow C$, it might hold vacuously if A may never occur in the model. We believe that by applying our justification algorithm, this problem can be solved efficiently.

Acknowledgements. We thank Yakir Vizel and Gala Yadgar for their help with the benchmarks.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC, IEEE Computer Society Press, Los Alamitos (1999)
2. Bjesse, P., Leonard, T., Mokkedem, A.: Finding bugs in an alpha microprocessor using satisfiability solvers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 454–464. Springer, Heidelberg (2001)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. Journal of the ACM 50(5), 752–794 (2003)
4. Classen, K., Roorda, J.-W.: A new SAT-based algorithm for symbolic trajectory evaluation. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, Springer, Heidelberg (2005)
5. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. CACM 5(7) (July 1962)
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. JACM 7(3), 201–215 (1960)
7. Fujiwara, H., Shimono, T.: On the acceleration of test generation algorithms. IEEE Trans. Computers 32(12), 1137–1144 (1983)
8. Ganai, M.K., Ashar, P., Gupta, A., Zhang, L., Malik, S.: Combining Strengths of Circuit-Based and CNF-Based Algorithms for a High-Performance SAT Solver. In: DAC (2002)
9. Grumberg, O., Schuster, A., Yadgar, A.: Hybrid BDD and all-sat method for model checking and other application. Technical report, Technion, CS-2007-08 (2007)
10. Jin, H., Awedh, M., Somenzi, F.: CirCUs: A Satisfiability Solver Geared towards Bounded Model Checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, Springer, Heidelberg (2004)

11. Lu, F., Wang, L.C., Cheng, K.-T., Huang, R.C.Y: A Circuit SAT Solver With Signal Correlation Guided Learning. In: DATE 2003, p. 10892. IEEE Computer Society Press, Washington (2003)
12. Marques-Silva, J.P., Sakallah, K.A.: Conflict analysis in search algorithms for propositional satisfiability. In: IEEE ICTAI (1996)
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC 2001. 39th Design Aotomation Conference (2001)
14. Pandey, M., Raimi, R., Bryant, R.E., Abadir, M.S.: Formal verification of content addressable memories using symbolic trajectory evaluation. *dac*, 00, 167 (1997)
15. Roorda, J.-W.: Symbolic trajectory evaluation using a satisfiability solver. Licentiate Thesis (2005)
16. Roorda, J.-W., Claessen, K.: Sat-based assistance in abstraction refinement for symbolic trajectory evaluation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 175–189. Springer, Heidelberg (2006)
17. Schubert, T.: High level formal verification of next-generation microprocessors. In: DAC (2003)
18. Seger, C.-J.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods in System Design 6(2) (1995)
19. Seger, C.-J.H., Jones, R.B., O'Leary, J.W., Melham, T.F., Aagaard, M., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 24(9) (2005)
20. Tzoref, R., Grumberg, O.: Automatic refinement and vacuity detection for symbolic trajectory evaluation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 190–204. Springer, Heidelberg (2006)
21. Yang, J., Gil, R., Singerman, E.: satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In: DCC (2004)
22. Yang, J., Goel, A.: GSTE through a case study. In: ICCAD (2002)
23. Zhang, H.: SATO: An efficient propositional prover. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, Springer, Heidelberg (1997)
24. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD (2001)

Bounded Synthesis*

Sven Schewe and Bernd Finkbeiner

Universität des Saarlandes, 66123 Saarbrücken, Germany

Abstract. The bounded synthesis problem is to construct an implementation that satisfies a given temporal specification and a given bound on the number of states. We present a solution to the bounded synthesis problem for linear-time temporal logic (LTL), based on a novel emptiness-preserving translation from LTL to safety tree automata. For distributed architectures, where standard unbounded synthesis is in general undecidable, we show that bounded synthesis can be reduced to a SAT problem. As a result, we obtain an effective algorithm for the bounded synthesis from LTL specifications in arbitrary architectures. By iteratively increasing the bound, our construction can also be used as a semi-decision procedure for the unbounded synthesis problem.

1 Introduction

Verification and synthesis both provide a formal guarantee that a system is implemented correctly. The difference between the two approaches is that while verification proves that a *given* implementation satisfies the specification, synthesis automatically *derives* one such implementation. Synthesis thus has the obvious advantage that it completely eliminates the need for manually writing and debugging code.

Unfortunately, the synthesis problem is undecidable even for simple distributed architectures. Consider, for example, the typical 2-process arbiter architecture shown in Figure 1b: the environment (env) sends requests (r_1, r_2) for access to a critical resource to two processes p_1 and p_2, which react by sending out grants (g_1, g_2). As shown by Pnueli and Rosner [1], the synthesis problem is undecidable for this architecture, because both p_1 and p_2 have access to information (r_1 and r_2, respectively) that is hidden from the other process. For system architectures without such *information forks* [2], like pipeline architectures (Figure 1a shows a pipeline of length 3), the synthesis problem is decidable, but has nonelementary complexity.

The high complexity of synthesis is explained by the fact that, as pointed out by Rosner [3], a small LTL formula of size n which refers to m different processes already suffices to specify a system that cannot be implemented with less than m-exp(n) states. From a practical point of view, however, it is questionable

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS).

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 474–488, 2007.
© Springer-Verlag Berlin Heidelberg 2007

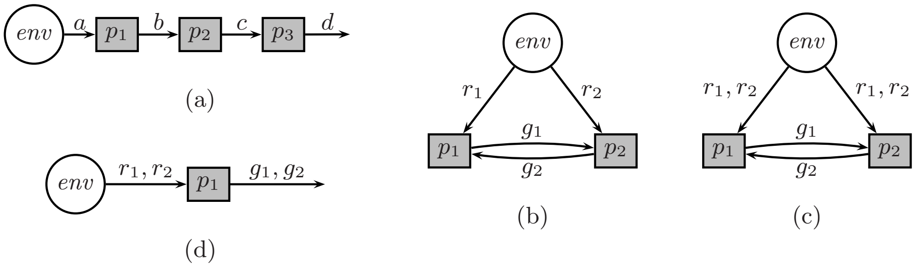


Fig. 1. Distributed architectures: (a) pipeline architecture, (b) 2-process arbiter architecture, (c) 2-process arbiter architecture with complete information, (d) single-process architecture

whether such huge implementations should be considered by the synthesis algorithm, because they are likely to violate other design considerations (such as the available memory). In this paper, we therefore study a variation of the synthesis problem, which we call the *bounded synthesis problem*, where an upper limit on the size of the implementation is set in advance. The bound may either be an explicit design constraint or the result of iteratively increasing the limit in the search for a solution of minimal size.

Our starting point is the representation of the LTL specification as a universal co-Büchi tree automaton. We show that the acceptance of a finite-state transition system by a universal co-Büchi automaton can be characterized by the existence of an annotation that maps each pair of a state of the automaton and a state of the transition system to a natural number. The advantage of this characterization is that the acceptance condition can be simplified to a simple safety condition: we show that the universal co-Büchi automaton can be translated to an (emptiness-equivalent) deterministic *safety automaton* that implicitly builds a valid annotation. The emptiness of the safety automaton can then be determined in a simple two-player game, where player *accept* represents the system implementation and wins the game if the specification is satisfied; the opponent, player *reject*, wins the game if the specification is violated.

If the system architecture consists of a single process, as in Figure 1d, then a victory for player *accept* means that the specification is realizable. Any winning strategy for player *accept* immediately defines a correct implementation for the process. If the architecture consists of more than one process, as in the arbiter architecture of Figure 1b, then a victory for player *accept* only means that the specification can be implemented in the slightly modified architecture (shown for the arbiter example in Figure 1c), where all processes have the same information. An implementation for the architecture with incompletely informed processes must additionally satisfy a consistency requirement: if a process cannot distinguish between two different computation paths, it must react in the same way.

Inspired by the success of bounded model checking [4,5], we show that the bounded synthesis problem for *distributed architectures* can be effectively

reduced to a SAT problem. We define a constraint system that describes the existence of a valid annotation and, additionally, ensures that the resulting implementation is consistent with the limited information available to the distributed processes. For this purpose, we introduce a mapping that decomposes the states of the safety game into the states of the individual processes: because the reaction of a process only depends on its local state, the process is forced to give the same reaction whenever it cannot distinguish between two paths in the safety game. The satisfiability of the constraint system can be checked using standard SAT solvers [6,7]. As a result, we obtain an effective algorithm for the bounded synthesis from LTL specifications in arbitrary distributed architectures. By iteratively increasing the bound, our construction can also be used as a semi-decision procedure for the standard (unbounded) synthesis problem.

Related work. The *synthesis of distributed reactive systems* was pioneered by Pnueli and Rosner [1], who showed that the synthesis problem is undecidable in general and has nonelementary complexity for pipeline architectures. An automata-based synthesis algorithm for pipeline and ring architectures is due to Kupferman and Vardi [8]; Walukiewicz and Mohalik provided an alternative game-based construction [9]. We recently showed that the synthesis problem is decidable if and only if the architecture does not contain an information fork [2]. Madhusudan and Thiagarajan [10] consider the special case of *local* specifications (each property refers only to the variables of a single process). Among the class of acyclic architectures (without broadcast) this synthesis problem is decidable for exactly the doubly-flanked pipelines. Castellani, Mukund and Thiagarajan [11] consider *transition systems* as the specification language: an implementation is correct if the product of the processes is bisimilar to the specification. In this case, the synthesis problem is decidable independently of the architecture.

Our translation of LTL formulas to tree automata is based on Kupferman and Vardi's *Safraless decision procedures* [12]. We use their idea of avoiding Safra's determinization using universal co-Büchi automata. Our construction improves on [12] in that it produces deterministic safety automata instead of nondeterministic Büchi automata.

2 Preliminaries

We consider the synthesis of distributed reactive systems that are specified in linear-time temporal logic (LTL). Given an architecture A and an LTL formula φ, we decide whether there is an implementation for each system process in A, such that the composition of the implementations satisfies φ.

Architectures. An *architecture* A is a tuple (P, env, V, I, O), where P is a set of processes consisting of a designated environment process $env \in P$ and a set of system processes $P^- = P \setminus \{env\}$. V is a set of boolean system variables (which also serve as atomic propositions), $I = \{I_p \subseteq V \mid p \in P^-\}$ assigns a set I_p of input variables to each system process $p \in P^-$, and $O = \{O_p \subseteq V \mid p \in P\}$

assigns a set O_p of output variables to each process $p \in P$ such that $\bigcup_{p \in P} O_p = V$. While the same variable $v \in V$ may occur in multiple sets in I to indicate broadcasting, the sets in O are assumed to be pairwise disjoint. If $O_{env} \subseteq I_p$ for every system process $p \in P^-$, we say the architecture is *fully informed*. Since every process in a fully informed architecture has enough information to simulate every other process, we can assume w.l.o.g. that a fully informed architecture contains only a single system process p, and that the input variables of p are the output variables of the environment process $I_p = O_{env}$.

Implementations. We represent implementations as labeled transition systems. For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-*transition system* is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a set of states T, an initial state $t_0 \in T$, a transition function $\tau : T \times \Upsilon \to T$, and a labeling function $o : T \to \Sigma$. $\mathcal{T}$ is a *finite-state* transition system iff T is finite.

Each system process $p \in P^-$ is implemented as a 2^{O_p}-labeled 2^{I_p}-transition system $\mathcal{T}_p = (T_p, t_p, \tau_p, o_p)$. The specification φ refers to the composition of the system processes, which is the 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T}_A = (T, t_0, \tau, o)$, defined as follows: the set $T = \bigotimes_{p \in P^-} T_p \times 2^{O_{env}}$ of states is formed by the product of the states of the process transition systems and the possible values of the output variables of the environment. The initial state t_0 is formed by the initial states t_p of the process transition systems and a designated *root direction* $\subseteq O_{env}$. The transition function updates, for each system process p, the T_p part of the state in accordance with the transition function τ_p, using (the projection of) o as input, and updates the $2^{O_{env}}$ part of the state with the output of the environment process. The labeling function o labels each state with the union of its $2^{O_{env}}$ part with the labels of its T_p parts.

With respect to the system processes, the combined transition system thus simulates the behavior of all process transition systems; with respect to the environment process, it is *input-preserving*, i.e., in every state, the label accurately reflects the input received from the environment.

Synthesis. A specification φ is (finite-state) *realizable* in an architecture $A = (P, V, I, O)$ iff there exists a family of (finite-state) implementations $\{\mathcal{T}_p \mid p \in P^-\}$ of the system processes, such that their composition $\mathcal{T}_A$ satisfies φ.

Bounded Synthesis. We introduce bounds on the size of the process implementations and on the size of the composition. Given an architecture $A = (P, V, I, O)$, a specification φ is *bounded realizable* with respect to a family of bounds $\{b_p \in \mathbb{N} \mid p \in P^-\}$ on the size of the system processes and a bound $b_A \in \mathbb{N}$ on the size of the composition T_A, if there exists a family of implementations $\{\mathcal{T}_p \mid p \in P^-\}$, where, for each process $p \in P$, $\mathcal{T}_p$ has at most b_p states, such that the composition $\mathcal{T}_A$ satisfies φ and has at most b_A states.

Alternating Automata. An *alternating parity tree automaton* is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$, where Σ denotes a finite set of labels, Υ denotes a finite set of directions, Q denotes a finite set of states, $q_0 \in Q$ denotes a designated

initial state, δ denotes a transition function, and $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive boolean combination of states and directions. In our setting, the automaton runs on Σ-labeled Υ-transition systems. The acceptance mechanism is defined in terms of run graphs.

A *run graph* of an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ on a Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is a minimal directed graph $\mathcal{G} = (G, E)$ that satisfies the following constraints:

- The vertices $G \subseteq Q \times T$ form a subset of the product of Q and T.
- The pair of initial states $(q_0, t_0) \in G$ is a vertex of $\mathcal{G}$.
- For each vertex $(q, t) \in G$, the set $\{(q', \upsilon) \in Q \times \Upsilon \mid ((q, t), (q', \tau(t, \upsilon))) \in E\}$ satisfies $\delta(q, o(t))$.

A run graph is *accepting* if every infinite path $g_0 g_1 g_2 \ldots \in G^\omega$ in the run graph satisfies the *parity condition*, which requires that the highest number occurring infinitely often in the sequence $\alpha_0 \alpha_1 \alpha_2 \in \mathbb{N}$ with $\alpha_i = \alpha(q_i)$ and $g_i = (q_i, t_i)$ is even. A transition system is accepted if it has an accepting run graph.

The set of transition systems accepted by an automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty iff its language is empty.

The acceptance of a transition system can also be viewed as the outcome of a game, where player *accept* chooses, for a pair $(q, t) \in Q \times T$, a set of atoms satisfying $\delta(q, o(t))$, and player *reject* chooses one of these atoms, which is executed. The transition system is accepted iff player *accept* has a strategy enforcing a path that fulfills the parity condition.

A *nondeterministic* automaton is a special alternating automaton, where the image of δ consists only of such formulas that, when rewritten in disjunctive normal form, contain exactly one element of $Q \times \{\upsilon\}$ for all $\upsilon \in \Upsilon$ in every disjunct. The emptiness of a nondeterministic automaton can be checked with a variation of the acceptance game called the *emptiness game*, where, in each step, player *accept* additionally chooses the label from Σ. A nondeterministic automaton is empty iff the emptiness game is won by player *reject.*

An alternating automaton is called *universal* if, for all states q and input letters σ, $\delta(q, \sigma)$ is a conjunction. A universal and nondeterministic automaton is called *deterministic.*

A parity automaton is called a *Büchi* automaton if the image of α is contained in $\{1, 2\}$, a *co-Büchi* automaton iff the image of α is contained in $\{0, 1\}$, and a *safety* automaton if the image of α is $\{0\}$. Büchi and co-Büchi automata are denoted by $(\Sigma, \Upsilon, Q, q_0, \delta, F)$, where $F \subseteq Q$ denotes the states with the higher color. Safety automata are denoted by $(\Sigma, \Upsilon, Q, q_0, \delta)$. A run graph of a Büchi automaton is thus accepting if, on every infinite path, there are infinitely many visits to F; a run graph of a co-Büchi automaton is accepting if, on every path, there are only finitely many visits to F. For safety automata, every run graph is accepting.

3 Annotated Transition Systems

In this section, we introduce an annotation function for transition systems. The annotation function has the useful property that a finite-state transition system satisfies the specification if and only if it has a valid annotation.

Our starting point is a representation of the specification as a universal co-Büchi automaton. Since the automaton is universal, every transition system in the language of the automaton has a unique run graph. The annotation assigns to each pair (q, t) of a state q of the automaton and a state t of the transition system either a natural number or a blank sign. The natural number indicates the maximal number of rejecting states that occur on some path to (q, t) in the run graph.

We show that the finite-state transition systems accepted by the automaton are exactly those transition systems for which there is an annotation that assigns only natural numbers to the vertices of the run graph. We call such annotations *valid*.

3.1 Universal Co-Büchi Automata

We translate a given LTL specification φ into an equivalent universal co-Büchi automaton $\mathcal{U}_\varphi$. This can be done with a single exponential blow-up by first negating φ, then translating $\neg\varphi$ into an equivalent nondeterministic Büchi word automaton, and then constructing a universal co-Büchi automaton that simulates the Büchi automaton along each path: if each path is co-Büchi accepting (i.e., it violates the Büchi condition), then the specification φ must hold along every path.

Theorem 1. *[12] Given an LTL formula φ, we can construct a universal co-Büchi automaton $\mathcal{U}_\varphi$ with $2^{O(|\varphi|)}$ states that accepts a transition system $\mathcal{T}$ iff $\mathcal{T}$ satisfies φ.* □

3.2 Bounded Annotations

An *annotation* of a transition system $\mathcal{T} = (T, t_0, \tau, o)$ on a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ is a function $\lambda : Q \times T \to \{_\} \cup \mathbb{N}$. We call an annotation *c-bounded* if its mapping is contained in $\{_\} \cup \{0, \ldots, c\}$, and *bounded* if it is c-bounded for some $c \in \mathbb{N}$. An annotation is *valid* if it satisfies the following conditions:

- the pair (q_0, t_0) of initial states is annotated with a natural number $(\lambda(q_0, t_0) \neq _)$, and
- if a pair (q, t) is annotated with a natural number $(\lambda(q, t) = n \neq _)$ and $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $(q', \tau(t, \upsilon))$ is annotated with a greater number, which needs to be strictly greater if $q' \in F$ is rejecting. That is, $\lambda(q', \tau(t, \upsilon)) \rhd_{q'} n$ where $\rhd_{q'}$ is $>$ for $q' \in F$ and $\geq$ otherwise.

Theorem 2. *A finite-state Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is accepted by a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ iff it has a valid $(|T| \cdot |F|)$-bounded annotation.*

Proof. Since $\mathcal{U}$ is universal, $\mathcal{U}$ has a unique run graph $\mathcal{G} = (G, E)$ on $\mathcal{T}$. Since $\mathcal{T}$ and $\mathcal{U}$ are finite, $\mathcal{G}$ is finite, too.

If $\mathcal{G}$ contains a lasso with a rejecting state in its loop, i.e., a path $(q_0, t_0)(q_1, t_1)\dots(q_n, t_n) = (q'_0, t'_0)$ and a path $(q'_0, t'_0)(q'_1, t'_1)\dots(q'_m, t'_m) = (q'_0, t'_0)$ such that q'_i is rejecting for some $i \in \{1, \dots, m\}$, then, by induction, any valid annotation λ satisfies $\lambda(q_j, t_j) \in \mathbb{N}$ for all $j \in \{0, \dots, n\}$, $\lambda(q'_j, t'_j) \in \mathbb{N}$ for all $j \in \{0, \dots, m\}$, $\lambda(q'_{j-1}, t'_{j-1}) \leq \lambda(q'_j, t'_j)$ for all $j \in \{1, \dots, m\}$, and $\lambda(q'_{i-1}, t'_{i-1}) < \lambda(q'_i, t'_i)$. $\lightning$

If, on the other hand, $\mathcal{G}$ does not contain a lasso with a rejecting state in its loop, we can easily infer a valid $(|T| \cdot |F|)$-bounded annotation by assigning to each vertex $(q, t) \in G$ of the run graph the highest number of rejecting states occurring on some path $(q_0, t_0)(q_1, t_1)\dots(q, t)$, and by assigning $_$ to every pair of states $(q, t) \notin G$ not in $\mathcal{G}$. $\square$

3.3 Estimating the Bound

Since the distributed synthesis problem is undecidable, it is in general not possible to estimate a sufficient bound c that guarantees that a transition system with a valid c-bounded annotation exists if the specification is realizable.

For fully informed architectures, however, such an estimate is possible. If a universal co-Büchi automaton is non-empty, then the size of a smallest accepted transition system can be estimated by the size of an equivalent deterministic parity automaton.

Theorem 3. [13] *Given a universal co-Büchi automaton $\mathcal{U}$ with n states, we can construct an equivalent deterministic parity automaton $\mathcal{P}$ with n^{2n+2} states and $2n$ colors.* $\square$

A solution to the synthesis problem is required to be input-preserving, i.e., in every state, the label must accurately reflect the input. Input preservation can be checked with a deterministic safety automaton $\mathcal{D}_\mathcal{I}$, whose states are formed by the possible inputs $\mathcal{I} = 2^{O_{env}}$. In every state $i \in \mathcal{I}$, $\mathcal{D}_\mathcal{I}$ checks if the label agrees with the input i, and sends the successor state $i' \in \mathcal{I}$ into the direction i'. If $\mathcal{U}$ accepts an input-preserving transition system, then we can construct a *finite* input-preserving transition system, which is accepted by $\mathcal{U}$, by evaluating the emptiness game of the product automaton of $\mathcal{P}$ and $\mathcal{D}_\mathcal{I}$. The minimal size of such an input-preserving transition system can be estimated by the size of $\mathcal{P}$ and $\mathcal{I}$.

Corollary 1. *If a universal co-Büchi automaton $\mathcal{U}$ with n states and m rejecting states accepts an input-preserving transition system, then $\mathcal{U}$ accepts a finite input-preserving transition system $\mathcal{T}$ with $n^{2n+2} \cdot |\mathcal{I}|$ states, where $\mathcal{I} = 2^{O_{env}}$. $\mathcal{T}$ has a valid $m \cdot n^{2n+2} \cdot |\mathcal{I}|$-bounded annotation for $\mathcal{U}$.* $\square$

4 Automata-Theoretic Bounded Synthesis

Using the annotation function, we can reduce the synthesis problem for fully informed architectures to a simple emptiness check on safety automata. The following theorem shows that there is a deterministic safety automaton that, for a given parameter value c, accepts a transition system iff it has a valid c-bounded annotation. This leads to the following automata-theoretic synthesis procedure for fully informed architectures:

Given a specification, represented as a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$, we construct a sequence of safety automata that check for valid bounded annotations up to the bound $c = |F| \cdot b$, where b is either the predefined bound b_A on the size of the transition system, or the sufficient bound $n^{2n+2} \cdot |\mathcal{I}|$ from Corollary 1. If the intersection of $\mathcal{D}_\mathcal{I}$ with one of these automata is non-empty, then the specification is realizable; if the intersection with the safety automaton for the largest parameter value c is empty, then the specification is unrealizable. The emptiness of the automata can be checked by solving their emptiness games.

Theorem 4. *Given a universal co-Büchi automaton* $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$*, we can construct a family of deterministic safety automata* $\{\mathcal{D}_c = (\Sigma, \Upsilon, S_c, s_0, \delta_c) \mid c \in \mathbb{N}\}$ *such that* $\mathcal{D}_c$ *accepts a transition system iff it has a valid c-bounded annotation.*

Construction: We choose the functions from Q to the union of $\mathbb{N}$ and a blank sign ($S = Q \to \{_\} \cup \mathbb{N}$) as the state space of an abstract deterministic safety automaton $\mathcal{D} = (\Sigma, \Upsilon, S, s_0, \delta_\infty)$. Each state of $\mathcal{D}$ indicates how many times a rejecting state may have been visited in some trace of the run graph that passes the current position in the transition system. The initial state of $\mathcal{D}$ maps the initial state of $\mathcal{U}$ to 0 ($s_0(q_0) = 0$) and all other states of $\mathcal{U}$ to blank ($\forall q \in Q \smallsetminus \{q_0\}.\, s_0(q) = _$).

Let $\delta^+_\infty(s,\sigma) = \{((q', s(q') + f(q')), \upsilon) \mid q, q' \in Q,\, s(q) \neq _$, and $(q', \upsilon) \in \delta(q,\sigma)\}$, where $f(q) = 1\, \forall q \in F$, and $f(q) = 0\, \forall q \notin F$, be the function that collects the transitions of $\mathcal{U}$. The transition function δ_∞ is defined as follows: $\delta_\infty(s,\sigma) = \bigwedge_{\upsilon \in \Upsilon} (s_\upsilon, \upsilon)$ with $s_\upsilon(q) = max\{n \in \mathbb{N} \mid ((q,n), \upsilon \in \delta^+_\infty(s,\sigma)\}$ (where $max\{\emptyset\} = _$). $\mathcal{D}_c$ is formed by restricting the states of $\mathcal{D}$ to $S_c = Q \to \{_\} \cup \{0, \ldots, c\}$.

Proof. Let λ be a valid c-bounded annotation of $\mathcal{T} = (T, t_0, \tau, o)$ for $\mathcal{U}$, and let λ_t denote the function with $\lambda_t(q) = \lambda(q,t)$. For two functions $s, s' : Q \to \{_\} \cup \mathbb{N}$, we write $s \leq s'$ if $s(q) \leq s'(q)$ holds for all $q \in Q$, where $_$ is the minimal element ($_ < n$ for all $n \in \mathbb{N}$). We show by induction that $\mathcal{D}_c$ has a run graph $\mathcal{G} = (G, E)$ for $\mathcal{T}$, such that $s \leq \lambda_t$ holds true for all vertices $(s,t) \in G$ of the run graph. For the induction basis, $s_0 \leq \lambda_{t_0}$ holds by definition. For the induction step, let $(s,t) \in G$ be a vertex of $\mathcal{G}$. By induction hypothesis, we have $s \leq \lambda_t$. With the definition of δ^+_∞ and the validity of λ, we can conclude that $((q',n),\upsilon) \in \delta^+_\infty(s, o(t))$ implies $n \leq \lambda_{\tau(t,\upsilon)}(q')$, which immediately implies $s' \leq \lambda_{t'}$ for all successors (s',t') of (s,t) in $\mathcal{G}$.

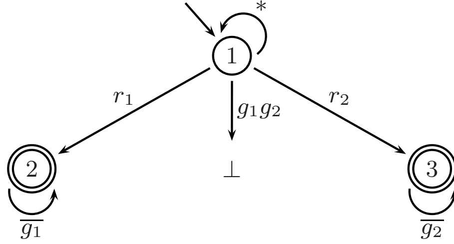


Fig. 2. Specification of a simple arbiter, represented as a universal co-Büchi automaton. The states depicted as double circles (2 and 3) are the rejecting states in F.

Let now $\mathcal{G} = (G, E)$ be an accepting run graph of $\mathcal{D}_c$ for $\mathcal{T}$, and let $\lambda(q,t) = max\{s(q) \mid (s,t) \in G\}$. Then λ is obviously a c-bounded annotation. For the validity of λ, $\lambda(q_0, t_0) \in \mathbb{N}$ holds true since $s_0(q_0) \in \mathbb{N}$ is a natural number and $(s_0, t_0) \in G$ is a vertex of $\mathcal{G}$. Also, if a pair (q, t) is annotated with a natural number $\lambda(q,t) = n \neq _$, then there is a vertex $(s,t) \in G$ with $s(q) = n$. If now $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $((q', n + f(q')), \upsilon) \in \delta^{+}_{\infty}(s, o(t))$ holds true, and the υ-successor $(s', \tau(t, \upsilon))$ of (s,t) satisfies $s'(q') \triangleright_{q'} n$. The validity of λ now follows with $\lambda(q', \tau(t, \upsilon) \geq s'(q')$. $\square$

Remark 1. Since $\mathcal{U}$ may accept transition systems where the number of rejecting states occurring on a path is unbounded, the union of the languages of all $\mathcal{D}_c$ is, in general, a strict subset of the language of $\mathcal{U}$. Every finite-state transition system in the language of $\mathcal{U}$, however, is accepted by almost all $\mathcal{D}_c$.

Example. Consider the specification of a simple arbiter, depicted as a universal co-Büchi automaton in Figure 2. The specification requires that globally (1) at most one process has a grant and (2) each request is eventually followed by a grant. The emptiness game for $\mathcal{D}_1$ intersected with $\mathcal{D}_{\mathcal{I}}$ is depicted in Figure 3.

5 Constraint-Based Bounded Synthesis

We now develop an alternative synthesis method for fully informed architectures that uses a SAT solver to determine an input-preserving transition system with a valid annotation. The constraint system defined in this section will provide the foundation for the synthesis method for general distributed architectures in Section 6.

We represent the (unknown) transition system and its annotation by uninterpreted functions. The existence of a valid annotation is thus reduced to the satisfiability of a constraint system in first-order logic modulo finite integer arithmetic. The advantage of this representation is that the size of the constraint system is small (bilinear in the size of $\mathcal{U}$ and the number of directions).

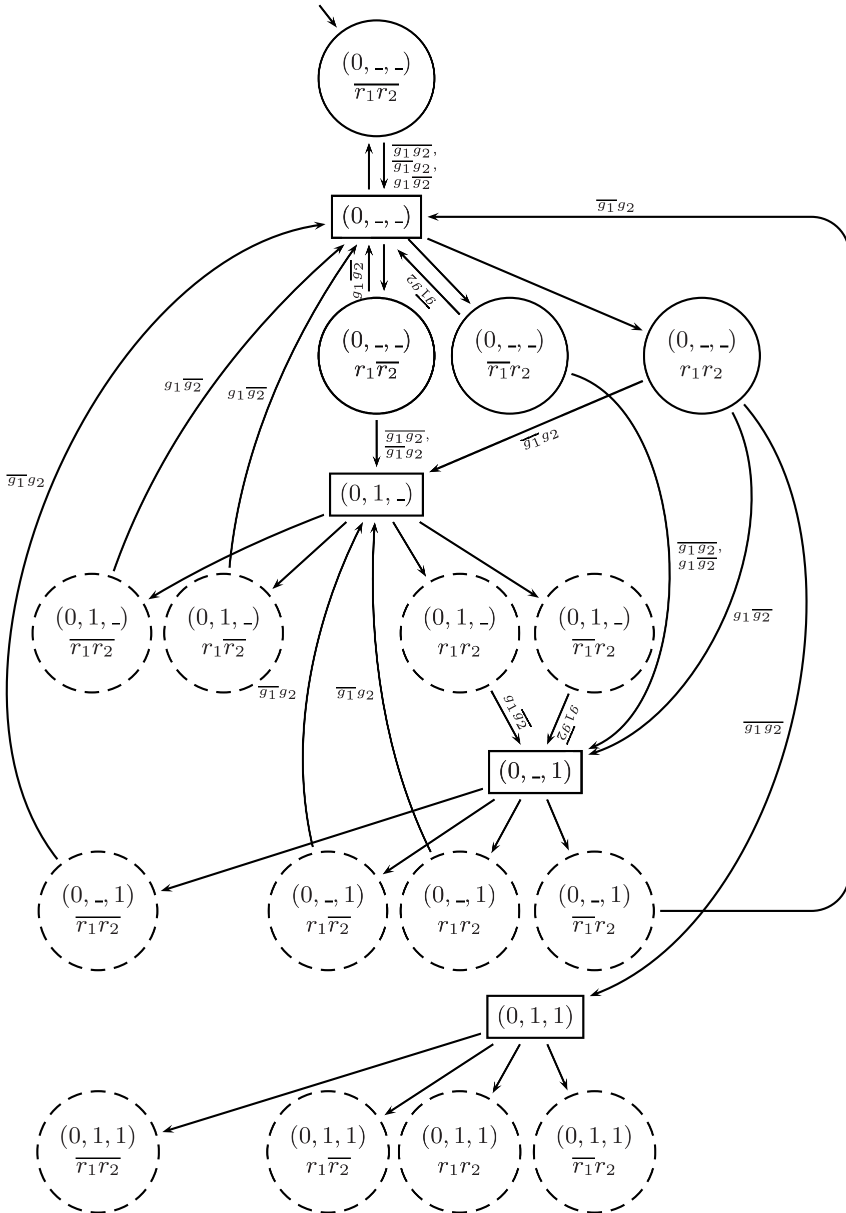


Fig. 3. Example of a safety game for synthesis in a fully informed architecture. The figure shows the emptiness game for the intersection of $\mathcal{D}_1$ and $\mathcal{D}_\mathcal{I}$ in the arbiter example (Figure 2). Circles denote game positions for player *accept*, rectangles denote game positions for player *reject*. Game positions that are not completely expanded (i.e., that have more successors if the parameter is increased) are dashed. The starting position specifies $\overline{r_1 r_2}$ as the (arbitrarily chosen) root direction. Player *accept* wins the game by avoiding the move to $(0, 1, 1)$.

Furthermore, the additional constraints needed for distributed synthesis, which will be defined in Section 6, have a compact representation as well (logarithmic in the number of directions of the individual processes).

The constraint system specifies the existence of a finite input-preserving 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T} = (T, t_0, \tau, o)$ that is accepted by the universal co-Büchi automaton $\mathcal{U}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ and has a valid annotation λ.

To encode the transition function τ, we introduce a unary function symbol τ_υ for every output $\upsilon \subseteq O_{env}$ of the environment. Intuitively, τ_υ maps a state t of the transition system $\mathcal{T}$ to its υ-successor $\tau_\upsilon(t) = \tau(t, \upsilon)$.

To encode the labeling function o, we introduce a unary predicate symbol a for every variable $a \in V$. Intuitively, a maps a state t of the transition system $\mathcal{T}$ to *true* iff it is part of the label $o(t) \ni a$ of $\mathcal{T}$ in t.

To encode the annotation, we introduce, for each state q of the universal co-Büchi automaton $\mathcal{U}$, a unary predicate symbol $\lambda_q^{\mathbb{B}}$ and a unary function symbol $\lambda_q^{\#}$. Intuitively, $\lambda_q^{\mathbb{B}}$ maps a state t of the transition system $\mathcal{T}$ to *true* iff $\lambda(q, t)$ is a natural number, and $\lambda_q^{\#}$ maps a state t of the transition system $\mathcal{T}$ to $\lambda(q, t)$ if $\lambda(q, t)$ is a natural number and is unconstrained if $\lambda(q, t) = _$.

We can now formalize that the annotation of the transition system is valid by the following first order constraints (modulo finite integer arithmetic):
$\forall t.\, \lambda_q^{\mathbb{B}}(t) \wedge \underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))} \rightarrow \lambda_{q'}^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_{q'}^{\#}(\tau_\upsilon(t)) \rhd_q \lambda_q^{\#}(t)$, where $\overrightarrow{a}(t)$ represents the label $o(t)$, $\underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))}$ represents the corresponding propositional formula, and $\rhd_q$ stands for $\rhd_q \equiv >$ if $q \in F$ and $\rhd_q \equiv \geq$ otherwise. Additionally, we require $\lambda_{q_0}^{\mathbb{B}}(0)$, i.e., we require the pair of initial states to be labeled by a natural number (w.l.o.g. $t_0 = 0$).

To guarantee that the resulting transition system is input-preserving, we add, for each $a \in O_{env}$ and each $\upsilon \subseteq O_{env}$, a constraint $\forall t.\, a(\tau_\upsilon(t))$ if $a \in \upsilon$, and a constraint $\forall t.\, \neg a(\tau_\upsilon(t))$ if $a \notin \upsilon$. Additionally, we require that the initial state is labeled with the root direction.

As an obvious implication of Theorem 2, this constraint system is satisfiable if and only if $\mathcal{U}$ accepts a finite input-preserving transition system.

Theorem 5. *For fully informed architectures, the constraint system inferred from the specification, represented as the universal co-Büchi automaton $\mathcal{U}$, is satisfiable modulo finite integer arithmetic iff the specification is finite-state realizable.* □

Lemma 1. *For a specification represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, the inferred constraint system has size $O(|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|)$.* □

The main parameter of the constraint system is the bound b_A on the size of the transition system $\mathcal{T}_A$. If we use b_A to unravel the constraint system completely (i.e., if we resolve the universal quantification explicitly), the size of the resulting constraint system is linear in b_A.

$$
\begin{array}{ll}
1. & \forall t.\, r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \overline{r}_2}(t)) \\
& \quad \wedge \neg r_1(\tau_{\overline{r}_1 r_2}(t)) \wedge r_2(\tau_{\overline{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{\overline{r}_1 \overline{r}_2}(t)) \\
2. & \lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0) \\
3. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t) \\
& \qquad\qquad \wedge\ \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 r_2}(t)) \geq \lambda_1^{\#}(t) \\
& \qquad\qquad \wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t) \\
& \qquad\qquad \wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}(t) \\
4. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t) \\
5. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t) \\
6. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
& \qquad\qquad\qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t) \\
7. & \forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t) \\
8. & \forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
& \qquad\qquad\qquad\quad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)
\end{array}
$$

Fig. 4. Example of a constraint system for synthesis in a fully informed architecture. The figure shows the constraint system for the arbiter example (Figure 2). The arbiter is to be implemented in the fully informed architecture shown in Figure 1d.

Theorem 6. *For a specification, represented as a universal co-Büchi automaton* $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, *and a given bound* b_A *on the size of the transition system* $\mathcal{T}_A$, *the unraveled constraint system has size* $O(b_A \cdot (|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|))$. *It is satisfiable if and only if the specification is bounded realizable in the fully informed architecture* $(\{env, p\}, V, \{I_p = O_{env}\}, \{O_{env}, O_p = V \smallsetminus O_{env}\})$ *with bound* b_A*.* □

Example. Figure 4 shows the constraint system, resulting from the specification of an arbiter by the universal co-Büchi automaton depicted in Figure 2, implemented on the single process architecture of Figure 1d (or, likewise, on the distributed but fully informed architecture of Figure 1c).

The first constraint represents the requirement that the resulting transition system must be input-preserving, the second requirement represents the initialization (where $\neg r_1(0) \wedge \neg r_2(0)$ represents an arbitrarily chosen root direction), and the requirements 3 through 8 each encode one transition of the universal automaton of Figure 2. Following the notation of Figure 2, r_1 and r_2 represent the requests and g_1 and g_2 represent the grants.

6 Distributed Synthesis

To solve the distributed synthesis problem for a given architecture $A = (P, V, I, O)$, we need to find a family of (finite-state) transition systems $\{\mathcal{T}_p = (T_p, t_0^p, \tau_p, o_p) \mid p \in P^-\}$ such that their composition to $\mathcal{T}_A$ satisfies the specification. The constraint system developed in the previous section can be adapted to distributed synthesis by explicitly decomposing the global state space of the combined transition system $\mathcal{T}_A$: we introduce a unary function symbol d_p for each process $p \in P^-$, which, intuitively, maps a state $t \in T_A$ of the product state space to its p-component $t_p \in T_p$.

The value of an output variable $a \in O_p$ may only depend on the state of the process transition system $\mathcal{T}_p$. We therefore replace every occurrence of $a(t)$ in the constraint system of the previous section by $a(d_p(t))$. Additionally, we require that every process p acts consistently on any two histories that it cannot distinguish. The update of the state of $\mathcal{T}_p$ may thus only depend on the state of $\mathcal{T}_p$ and the input visible to p. This is formalized by the following constraints:

1. $\forall t.\, d_p(\tau_\upsilon(t)) = d_p(\tau_{\upsilon'}(t))$ for all decisions $\upsilon, \upsilon' \subseteq O_{env}$ of the environment that are indistinguishable for p (i.e., $\upsilon \cap I_p = \upsilon' \cap I_p$).
2. $\forall t, u.\, d_p(t) = d_p(u) \wedge \bigwedge_{a \in I_p \smallsetminus O_{env}} \big(a(d_{p_a}(t)) \leftrightarrow a(d_{p_a}(t))\big) \rightarrow d_p(\tau_\upsilon(t)) = d_p(\tau_\upsilon(u))$ for all decisions $\upsilon \subseteq O_{env} \cap I_p$ (picking one representative for each class of environment decisions that p can distinguish). $p_a \in P^-$ denotes the process controlling the output variable $a \in O_{p_a}$.

Since the combined transition system $\mathcal{T}_A$ is finite-state, the satisfiability of this constraint system modulo finite integer arithmetic is equivalent to the distributed synthesis problem.

Theorem 7. *The constraint system inferred from the specification, represented as the universal co-Büchi automaton $\mathcal{U}$, and the architecture A is satisfiable modulo finite integer arithmetic iff the specification is finite-state realizable in the architecture A.* □

Lemma 2. *For a specification, represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, and an architecture A, the inferred constraint system for distributed synthesis has size $O(|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}| + \sum_{p \in P^-} |I_p \smallsetminus O_{env}|)$.* □

The main parameters of the constraint system for distributed synthesis are the bound b_A on the size of the transition system $\mathcal{T}_A$ and the family $\{b_p \mid p \in P^-\}$ of bounds on the process transition systems $\{\mathcal{T}_p \mid p \in P^-\}$. If we use these parameters to unravel the constraint system completely (i.e., if we resolve the universal quantification explicitly), the resulting transition system is linear in b_A, and quadratic in b_p.

Theorem 8. *For a given specification, represented as a universal co-Büchi automaton $\mathcal{U} = (2^V, 2^{O_{env}}, Q, q_0, \delta, F)$, an architecture $A = (P, V, I, O)$, a bound b_A*

$$
\begin{array}{rl}
4. & \forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_1(t)) \vee \neg g_2(d_2(t)) \\
7. & \forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_1(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_2^{\#}(t) \\
 & \qquad \wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t) \\
8. & \forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_2(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_3^{\#}(t) \\
 & \qquad \wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t) \\
9. & \forall t.\ d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1\overline{r}_2}(t)) \wedge d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1\overline{r}_2}(t)) \\
 & \quad \wedge\ d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{\overline{r}_1 r_2}(t)) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t)) \\
10. & \forall t, u.\, d_1(t) = d_1(u) \wedge \big(g_2(d_2(t)) \leftrightarrow g_2(d_2(u))\big) \rightarrow d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1 r_2}(u)) \\
 & \qquad \wedge\ d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1 r_2}(u)) \\
11. & \forall t, u.\, d_2(t) = d_2(u) \wedge \big(g_1(d_1(t)) \leftrightarrow g_1(d_1(u))\big) \rightarrow d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{r_1 r_2}(u)) \\
 & \qquad \wedge\ d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{r_1\overline{r}_2}(u))
\end{array}
$$

Fig. 5. Example of a constraint system for distributed synthesis. The figure shows modifications and extensions to the constraint system from Figure 4 for the arbiter example (Figure 2) in order to implement the arbiter in the distributed architecture shown in Figure 1b.

on the size of the input-preserving transition system $\mathcal{T}_A$, and a family $\{b_p \mid p \in P^-\}$ of bounds on the process transition systems $\{\mathcal{T}_p \mid p \in P^-\}$, the unraveled constraint system has size $O(b_A \cdot (|\delta| \cdot |V| + |O_{env}| \cdot |2^{O_{env}}|) + \sum_{p \in P^-} {b_p}^2 |I_p \setminus O_{env}|))$. It is satisfiable if and only if the specification is bounded realizable in A for the bounds b_A and $\{b_p \mid p \in P^-\}$. □

Example. As an example for the reduction of the distributed synthesis problem to SAT, we consider the problem of finding a distributed implementation to the arbiter specified by the universal automaton of Figure 2 in the architecture of Figure 1b. The functions d_1 and d_2 are the mappings to the processes p_1 and p_2, which receive requests r_1 and r_2 and provide grants g_1 and g_2, respectively. Figure 5 shows the resulting constraint system. Constraints 1–3, 5, and 6 are the same as in the fully informed case (Figure 4). The consistency constraints 9–11 guarantee that processes p_1 and p_2 show the same behavior on all input histories they cannot distinguish.

7 Conclusions

Despite its obvious advantages, synthesis has been less popular than verification. While the complexity of verification is determined by the size of the implementation under analysis, standard synthesis algorithms [1,8,9,2] suffer from the daunting complexity determined by the theoretical upper bound on the smallest implementation, which, as shown by Rosner [3], increases by an extra exponent with each additional process in the architecture.

By introducing a bound on the size of the implementation, we have levelled the playing field for synthesis and verification. We have shown that the bounded synthesis problem can be solved effectively with a reduction to SAT.

Our solution for the bounded synthesis problem can be extended to the standard (unbounded) synthesis problem by iteratively increasing the bound. The advantage of this approach is that the complexity is determined by the size of the smallest actual implementation. Typically, this is far less than the exploding upper bound.

References

1. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. FOCS, pp. 746–757. IEEE Computer Society Press, Los Alamitos (1990)
2. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proc. LICS, pp. 321–330. IEEE Computer Society Press, Los Alamitos (2005)
3. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science, Rehovot, Israel (1992)
4. Copty, F., Fix, L., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
5. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
6. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the satisfiability (SAT) Problem: A survey. In: Du, D.Z., Gu, J., Pardalos, P. (eds.) Satisfiability Problem: Theory and applications. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 19–152. American Mathematical Society (1997)
7. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001. Proceedings of the 38th Design Automation Conference (2001)
8. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: Proc. LICS, pp. 389–398. IEEE Computer Society Press, Los Alamitos (2001)
9. Walukiewicz, I., Mohalik, S.: Distributed games. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 338–351. Springer, Heidelberg (2003)
10. Madhusudan, P., Thiagarajan, P.S.: Distributed controller synthesis for local specifications. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001)
11. Castellani, I., Mukund, M., Thiagarajan, P.S.: Synthesizing distributed transition systems from global specification. In: Pandu Rangan, C., Raman, V., Ramanujam, R. (eds.) FSTTCS 1999. LNCS, vol. 1738, pp. 219–231. Springer, Heidelberg (1999)
12. Kupferman, O., Vardi, M.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, Pittsburgh, pp. 531–540 (2005)
13. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proc. LICS, pp. 255–264. IEEE Computer Society Press, Los Alamitos (2006)

Formal Modeling and Verification of High-Availability Protocol for Network Security Appliances

Moonzoo Kim

CS Dept. Korea Advanced Institute of Science and Technology
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr

Abstract. One of the prerequisites for information society is *secure* and *reliable* communication among computing systems. Accordingly, network security appliances become key components of infrastructure, not only as security guardians, but also as reliable network components. Thus, for both fault tolerance and high network throughput, multiple security appliances are often deployed together in a group and managed via *High-Availability (HA)* protocol.

In this paper, we present our experience of formally modeling and verifying the HA protocol used for commercial network security appliances through model checking. In addition, we applied a new debugging technique to detect *multiple* bugs *without* modifying/fixing the HA model by analyzing all counter examples. Throughout these formal analysis, we could effectively detect several design flaws.

1 Introduction

As more computing systems are deployed in wide functions of our society such as mobile banking, tele-conferencing, and online stock trading systems, communication between remote systems becomes essential for ubiquitous computing society. Internet provides such communication services for a large number of applications, but at the cost of *security* and *reliability*. Therefore, more and more security appliances such as firewall, VPN, and IDS/IPS are deployed in small to giant size networks. As security appliances become key components of network, their reliability, not only as security guardians, but also as network components, becomes a critical issue. For high-traffic networks, it is convention to deploy multiple security appliances grouped together for both fault tolerance and high network throughput. Most high-end security appliances achieve these two goals via *High-Availability (HA) protocol* among the appliances.

Despite the importance of HA protocol, HA protocol often causes failures to network security appliances for the following reasons. First, HA protocol, which is a fault-tolerant distributed network protocol, is notorious for its high complexity. It is a challenging task to consider all possible communication/coordination scenarios among the network security appliances in a group. Furthermore, failure and recovery of each machine in the group should also be considered, which

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 489–500, 2007.
© Springer-Verlag Berlin Heidelberg 2007

adds complexity further. Second, testing high-end network security appliances requires great efforts due to complex network/machine configurations and a large number of test scenarios. Also, it is hard to determine whether misbehavior is due to errors of the HA protocol or due to other factors such as OS/HW failure and/or misconfiguration of networks, etc. Finally, manufacturers often concentrate on developing functions of each security appliance without considering how these machines should communicate each other through HA protocol. Thus, HA protocol is often designed and implemented in an ad-hoc way at the last stage of development, which often creates unexpected behaviors. As a result, it is often observed that a group of network security appliances exhibits abnormal behaviors such as decreased network throughput or dropped normal packets while a single security appliance works well without a problem. Therefore, it is highly desirable to *formally* model and verify HA protocol design as formal method techniques have been actively applied to enhance reliability of network applications [7,10,11,9].

This paper presents our experience of formally modeling and verifying the HA protocol implemented in a commercial network security appliance NXG2000 [1]. We built a HA protocol model of a moderate size and verified the model using the Spin model checker [2] to check the absence of deadlock in the HA protocol. In this project, we could overcome the limitation of traditional debugging by detecting multiple bugs from the all counter examples without modifying/fixing the HA model as described in Sect. 4.

2 Overview of the HA Protocol of NXG2000

NXG2000 [1] is an integrated network security appliance consisting of firewall, VPN, and IDS targeted for gigabit networks. NXG2000 provides upto 1 million concurrent sessions (maximum 2 Gbps throughput) via six gigabit ports. In addition, NXG2000 has a 100 Mbps HA port dedicated to the HA protocol.

Network equipments located at the gateway must achieve high reliability as well as fast recovery lest the whole internal network cannot be operational. Thus, multiple network security appliances are deployed in a group for both fault tolerance and increased network throughput. For this purpose, machines of a group cooperate with each other to perform several tasks such as session synchronization and group management through the HA protocol.

In order to manage a group of network security appliances, one security appliance in the group is designated as a *master* to manage the other *slaves*. Initially, a master is statically designated by a network administrator. Although a master performs various jobs such as synchronizing sessions, configuring network, and creating event logs, we focus on the core management tasks of a master as follows.

1. *Addition of slaves* (see Fig. 1.a))
   When a slave becomes operational, the slave broadcasts `join_request` messages every second until it receives a `join_permit` message from a master.

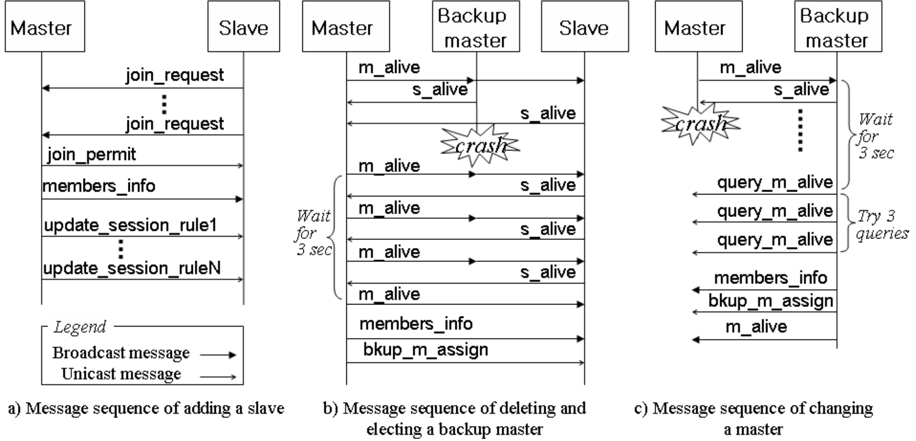


Fig. 1. Message sequences regarding the HA activities

Once the master allows the slave to join the group by sending a `join_permit` message to the slave, the master broadcasts new information to the group. Following this, the master sends all session information to the slave.

2. *Deletion of slaves* (see Fig. 1.b))
   A master constantly checks the status of slaves by receiving `s_alive` from every slave each second. If a master does not receive `s_alive` from a slave for three seconds, the master erases the slave from the group. If the erased slave is a backup master, the master elects another slave as a backup master and sends `bkup_m_assign` to the slave.
3. *Assignment of a backup master* (see Fig. 1.b))
   To prepare for a case in which a master crashes, the master assigns a slave as a backup master that will become a master when the master crashes. For backup master assignment, a master sends an assignment message `bkup_m_assign` to a slave that is elected as a backup master.

A backup master constantly checks whether or not a master is operational by receiving `m_alive`, which is broadcasted by a master every second (see Fig. 1.c)). If a backup master does not receive `m_alive` for three seconds, the backup master sends `query_m_alive` three times to the master. If the backup master does not receive a response from the master, the backup master becomes a master and broadcasts its new status. The backup master then assigns another slave as a new backup master by sending `bkup_m_assign` and starts broadcasting `m_alive` messages. A security appliance starts working as a slave when it recovers from failure. An *exception* is that machine 0, which is statically designated as a master by a network administrator, will work as a master if there is no master when it recovers from a failure.

3 The HA protocol Model

We model the HA protocol in Promela [2]. Each machine, regardless of whether it is a master or a slave, is modeled as a process. The overall execution of each machine is depicted in Fig. 2.

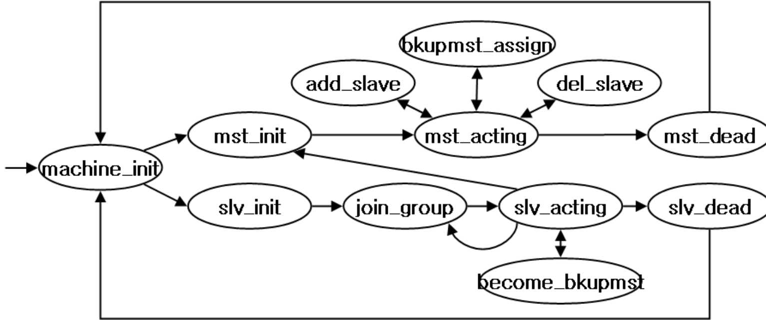


Fig. 2. Overview of the HA protocol model

Each machine starts from `machine_init` state (located at the left end of Fig. 2). Initially, machine 0 (whose process id is 0) is statically designated as a master and the machine moves to `mst_init` state to become a master. Then, the machine is working at `mst_acting` state that is the core of the master procedure. A master performs the following tasks at `mst_acting`.

- To add a slave to the group (`add_slave` state)
- To assign a slave as a backup master (`bkupmst_assign` state)
- To delete a slave from the group if the slave is found dead (`del_slave` state)
- To exhibit a crash (`mst_dead` state)

Note that there exists only one crash point for master in this model; a master can fail/crash only at `mst_dead` state. Thus, this model does not exhibit failure while a master is adding/deleting a slave or assigning a slave as a backup master. This simplified failure model abstracts out the need of cleanup procedures in a case of failure, which reduces complexity of the HA model significantly.

Once a machine is determined as a slave, the machine moves to `slv_init` state to initialize settings to become a slave. Then, the slave moves to `join_group` state where the slave requests a permission to join the group from a master. Once the slave receives the permission from the master, the slave moves to `slv_acting` state performing the following tasks.

- To become a backup master (`become_bkupmst` state)
- To become a master if it is a backup master and there exists no master (a transition to `mst_init` state)
- To exhibit a crash (`slv_dead` state)

4 A New Debugging Technique to Detect Multiple Bugs

Model checking techniques are effectively used as a means to improve the reliability of computing systems by detecting bugs of formal system models [8,15]. The traditional way of debugging a formal model is as follows: First, a human engineer identifies a bug in a counter example. The bug is then fixed by modifying the model. Once the bug is fixed, the modified model is verified again in order to detect the next bug, if one exists, in a new counter example. This debugging process is repeated until there no more bugs are found. This approach toward debugging has the following limitations:

- There are cases where it is not feasible to fix a bug for several reasons. In such cases, no further debugging progress can be made.
- Fixing a bug may introduce other bugs so that the traditional debugging iterations may continue indefinitely, or never terminate in the worst cases.
- When a model is modified to fix one bug, all requirement properties must be verified once again. Considering that real-world applications often have several hundred properties to check, these repeated fix-and-verify trials consume considerable project time.

Therefore, we propose a new debugging technique to identify multiple bugs *without* modification of a model by analyzing all counter examples generated by model checker. There have been researches on analysis of counter examples with various goals such as model refinement and localization of bugs [6,16,3,4]. Our focus, which is orthogonal to these related works, is to provide an automated process to detect as many bugs as possible by analyzing multiple counter examples without modification of a model.

4.1 An Automated Process to Detect Multiple Bugs

First, we describe an automated process that detects multiple bugs that violate the requirement property ϕ without modification to the target model. A key point of the process is to construct a set of formulas ψ_i's each of which captures/describes a bug b_i revealed in a subset of counter example traces.[1] Following this, the traces that satisfy/conform to ψ_i's are automatically detected, i.e., those that violate ϕ due to ψ_i. For this automatic trace analysis, it is necessary to formally specify ψ_i's in a formal specification language such as Meta Event Definition Language (MEDL)(see Sect. 4.3). Notations are defined before describing this debugging process formally.

- T_ϕ is the set of all counter example traces of a requirement property ϕ such that $T_\phi = \{t_i |\ t_i$ is a counter example of $\phi\}$.
- B_ϕ is a set of formulas of the bugs that violate ϕ, i.e., $B_\phi = \{\psi_i |\ \psi_i$ is a formula of a bug that violates $\phi\}$.

[1] A bug b_i is identified through manual analysis as in the traditional debugging.

– $t \models_\phi \psi$ where $t \in T_\phi$ and $\psi \in B_\phi$ signifies that a counter example trace t satisfies ψ that is a cause of the violation of ϕ (i.e., t violates ϕ due to ψ).
– $t_\phi : B_\phi \rightarrow \mathcal{P}(T_\phi)$ is a function such that $t_\phi(\psi) = \{t_i \in T_\phi | \ t_i \models_\phi \psi\}$.

An algorithm that detects multiple bugs without modifying the target model is described in Fig. 3. This algorithm is guaranteed to terminate if evaluation of a trace at Step 3 is decidable (which is true in most practical cases) as T_ϕ is finite in a finite state model. All steps of the algorithm can be automated except Step 2, which still requires human ingenuity to identify a bug and specify the bug as ψ in a formal specification language.

1. Set T with T_ϕ.
2. Select the smallest trace $t_{init} \in T$. Then a user analyzes t_{init} to identify a bug b that violates ϕ and specifies the bug b as ψ.
3. Obtain $t_\phi(\psi)$ by checking all traces of T with ψ.
4. Set a new set of traces T' with $T - t_\phi(\psi)$ and select the new smallest trace $t'_{init} \in T'$.
5. Set T with T' and t_{init} with t'_{init}, then repeat from Step 2 until T becomes $\emptyset$.

Fig. 3. An algorithm to detect multiple bugs that violate ϕ

Fig. 4 illustrates the algorithm. Initially, the smallest trace t_0 that violates ϕ is manually analyzed and the bug b_0 revealed in t_0 is described as ψ_0. Following this, $t_\phi(\psi_0)$ is obtained, and the smallest trace $t_1 \in (T_\phi - t_\phi(\psi_0))$ is found. This process is repeated until ψ_0, ψ_1, and ψ_2 that cover T_ϕ completely are found (i.e., $t_\phi(\psi_0) \cup t_\phi(\psi_1) \cup t_\phi(\psi_2) = T_\phi$). Note that the algorithm of Fig. 3 does not strictly require a collection of $t_\phi(\psi_i)$ to be pairwise disjoint. It is possible that $t_\phi(\psi_i)$ overlaps $t_\phi(\psi_j)$, which, however, does not affect a result of the algorithm.

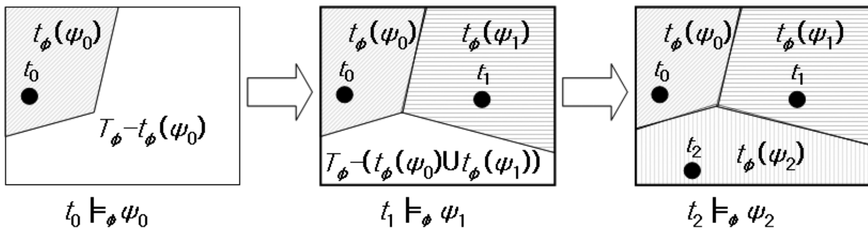


Fig. 4. A process of detecting bugs that violate a requirement property ϕ

4.2 Overview of the MacDebugger Framework

The MacDebugger framework [13] (see Fig. 5), which is an extension of the MaC framework [12], is a general framework for analyzing a large volume of counter

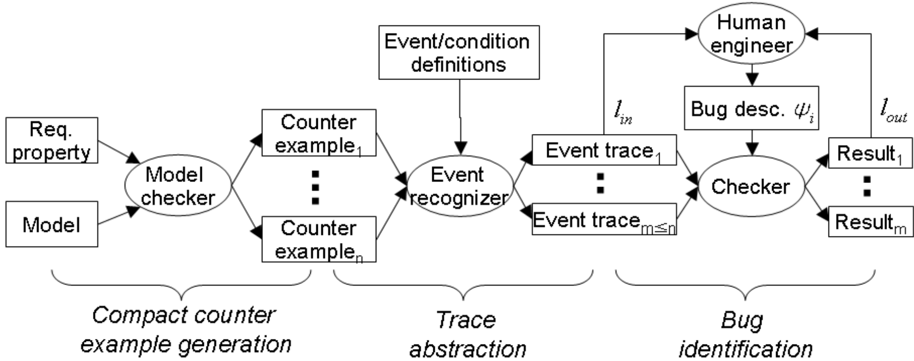


Fig. 5. Overview of the MacDebugger framework

example traces. MacDebugger is designed to work with any model checker that can generate multiple counter examples. As a prototype, however, it was implemented to work with the Spin model checker [8]. MacDebugger consists of the following three components - a *model checker*, an *event recognizer*, and a *checker*.

MacDebugger aims to analyze a large number of counter examples efficiently. Thus, performance of storing and analyzing counter examples is a critical issue, as even a simple model can generate hundreds of gigabytes of counter examples. For that purpose, we modified the Spin model checker to generate counter examples in a compact format. An event recognizer (an oval in the middle of Fig. 5) extracts sequences of *primitive events* and *conditions* from counter examples generated from a model checker and generates event traces which contain these sequences. A checker (an oval in the right of Fig. 5) receives a list of the event traces to analyze, for example l_{in}, and a bug description written in MEDL, in this example ψ_i, as its inputs. The checker analyzes all event traces in l_{in} with respect to ψ_i and returns a list of event traces, in this example l_{out}, which do not satisfy ψ_i. Following this, a human engineer investigates the shortest event trace in l_{out} and identifies a new bug ψ_{i+1} from the trace. The checker then repeats this debugging process using l_{out} and ψ_{i+1} as new inputs until all event traces/counter examples are covered by $\psi_0 ... \psi_n$ as described in Sect. 4.1.

4.3 Meta Event Definition Language

MEDL is based on an extension of linear temporal logic with auxiliary variables to record history of the event trace. MEDL distinguishes between two kinds of data that make up the trace of an execution - *events* and *conditions*. Events occur instantaneously during the system execution, whereas conditions represent information that holds for a duration of time [5]. A checker assumes that truth values of all conditions remain unchanged between updates from the event recognizer. For events, a checker makes the dual assumption, namely, that no events (of interest) happen between updates. Based on this distinction between

Table 1. The syntax of conditions, events, and guards

E ::= e | `start`(C) | `end`(C) | E&&E | E||E | E `when` C
C ::= c | `defined`(C) | [E,E) | !C | C&&C | C||C | C⇒C
G ::= E → {*statements*}

events and conditions, we have a simple two-sorted logic that constitutes MEDL. The syntax of events (E), conditions (C), and guards (G) is given in Table 1.

Here e refers to primitive events that are reported in the trace by the event recognizer; c is either a primitive condition reported in the trace or it is a boolean condition defined on the auxiliary variables. Guards (G) are used to update auxiliary variables. The semantics for boolean operations over conditions and events is defined naturally. There are some natural events associated with conditions, namely, the instant when the condition becomes *true* (`start`(c)), and the instant when the condition becomes $false$ (`end`(c)). Also, any pair of events define an interval of time, so forms a condition $[e_1, e_2)$ that is *true* from event e_1 until event e_2. The event (e `when` c) is present if e occurs at a time when condition c is *true*. Finally, a guard $e \rightarrow \{statements\}$ updates auxiliary variables according to the assignments given in *statements* when e happens.

A MEDL script defines a requirement property as a special event, called *alarm*. To check whether an alarm occurs or not, a checker evaluates the events and conditions defined in the script whenever it reads an element from the trace. For more detail on the formal semantics of MEDL, see [12].

5 Verification of the HA protocol

The full state space of the model is generated without stopping at violations. Statistics on the model with a different number of machines in a group are illustrated in Table 2. N/A indicates that the state space failed to be generated due to a lack of memory. A Pentium IV 3Ghz computer equipped with 2 GB of memory, and 80GB of hard disk running Spin 4.2.6 on Fedora Linux 4 was used. A maximum search depth was set as 5×10^6 and the estimated state space as 10^8, of which the hash table and DFS stack took 227 Mb. The HA protocol model in Promela is approximately 200 lines long. We found that all HA models with $N \geq 2$ had deadlock and all counter examples causing deadlock were generated. Table 3 shows the statistics on the counter examples.

Immediate Cause of the Deadlock. Firstly, an immediate cause of the deadlock at $N = 2$ was identified. The shortest counter example was analyzed and it was found that deadlock occurred when all machines in the group were slaves, in other words when no master existed to admit slaves to join the group. In this situation, no progress could be made unless machine 0 crashed and revived as a master, which is clearly beyond the control of the HA protocol. Fig. 6 shows this fault that immediately causes deadlock formulated in MEDL. `deadlock` in line 2

Table 2. Statistics on the HA protocol model with a different number of machines

Number of machines in a group (N)	2	3	4	5	6
States	246	17489	551052	1.40×10^7	N/A
Transitions	409	43419	1.75×10^6	5.24×10^7	N/A
Memory usage(in Mb)	228	229	264	1321	N/A
Time to generate state space (in sec)	1.0	1.1	3.2	86.9	N/A

Table 3. Statistics on the counter examples showing deadlock

Number of machines (N)	2	3	4	5
# of counter examples	4	156	4440	123360
Size of total counter examples (in bytes)	0.3K	628K	53M	36G
Avg. length of counter example (in steps)	36	1271	2.8×10^4	8×10^5
Time to generate all counter examples	0.1 sec	0.3 sec	65 sec	11 hour

is a primitive event representing deadlock. The event recognizer recognizes this event by detecting the end of a counter example. `m0_slave` in line 3 is a primitive condition indicating whether or not machine 0 is a slave. `m1_slave` is similarly defined. Thus, if deadlock occurs when all machines are slaves, the `all_slaves` alarm in line 4 is triggered.

```
01:ReqSpec DeadlockDetector
02:    import event deadlock;
03:    import condition m0_slave, m1_slave;
04:    alarm all_slaves = deadlock when (m0_slave && m1_slave);
05:End
```

Fig. 6. MEDL specification of the fault causing deadlock ($N = 2$)

All counter examples of the models were checked with $N \geq 2$. It was found that all traces raised the `all_slave` alarm, indicating that the immediate cause of the deadlock was incorrect master election process.

Identification of Design Flaws. First, the shortest counter example in the smallest model ($N = 2$) was analyzed further and we found the following faulty scenario.

> f_1: *A master (machine 1) died immediately after a backup master (machine 0) had died and revived as a slave. Machine 1 then revived as a slave and all machines became slaves.*

f_1 was formulated as `f1` in line 6 of Fig. 7. `mst_died` and `bkupmst_died` indicate crashes of the corresponding machines. `becomes_mst` occurs when a backup master becomes a master. `bkupmst_elected` indicates that a new backup master is elected, and `m0_alive` indicates that machine 0 is alive. `m0_working` indicates that machine 0 has joined the group and is cooperating with the other machines in the group. `f1` is triggered when a master dies without a backup master (line 6) with additional conditions for machine 0 (line 7) satisfied.

It is important to note that a bug triggering f_1 is hard to fix because fixing the HA protocol to work correctly with this scenario requires major redesign of the HA protocol, which was not feasible due to limited project resources. Thus, other bugs could not be detected if we used the traditional debugging method. We could, however, continue debugging process to detect other remaining bugs as explained below by using the new debugging technique.

```
01:ReqSpec f1Detector
02:  import event mst_died, bkupmst_died, becomes_mst, bkupmst_elected;
03:  import condition m0_working, m0_alive;
04:
05:  condition restriction = !m0_working && m0_alive;
06:  alarm f1 =mst_died when ([bkupmst_died||becomes_mst,bkupmst_elected)
07:          && value(mst_died,0) != 0 && restriction );
08:end
```

Fig. 7. Specification of f_1 in MEDL

It was found that 4 out of 4 ($N = 2$), 90 out of 156 ($N = 3$), 2703 out of 4440 ($N = 4$), as well as 70042 out of 123360 counter examples ($N = 5$) raised the `f1` alarm (see Table 4). This indicates that other faults exist, as f_1 does not cover all counter examples. The smallest counter example trace that did not raise the alarm in $N = 3$ was analyzed, and the following faulty scenario in the trace was found.

> *f_2: A master elected a machine that was dead, as a backup master without recognizing that the machine was dead. The master then died and it happened that there existed no master.*

This fault is caused by a bug in which a master assigns a slave as a backup master by only sending `bkup_m_assign` to the slave and not requiring acknowledgment from the slave. It was found that 62 ($N = 3$), 1560 ($N = 4$) and 51200 counter examples ($N = 5$) raised f_2 (see Table 4). f_1 and f_2, however, still do not cover all counter examples, indicating that there still are other faults.

In a similar manner, f_3 was detected and formulated to specify that a backup master died immediately after a master has died, making all machines slaves. Finally, f_1, f_2, and f_3 covered all counter examples, indicating that all of the bugs that cause deadlock had been founded. These analysis results are shown in

Table 4. Analysis results of counter examples due to f_1, f_2, and f_3

Number of machines (N)	2	3	4	5
Total # of counter examples	4	156	4440	123360
# of event traces due to f_1	4	90	2703	70042
# of event traces due to f_2	0	62	1560	51200
# of event traces due to f_3	0	4	177	2118

Table 4. It takes less than one minute to analyze all counter examples for $N \leq 4$ and takes around 7 hours to analyze all counter examples to check each of f_1, f_2, and f_3 for $N = 5$.

6 Conclusion

In this paper, we present results of formal modeling and verification of the HA protocol of NXG2000. In this study, we could find several bugs in the HA protocol through analyzing counter examples generated by model checking. These bugs had not been noticed by the company before, and the company decided to adopt a distributed master election process [14] in the next version of NXG2000. We are convinced that the new debugging technique in this paper is effective to verify systems of industrial strength, which often have hard-to-fix bugs. We plan to develop this debugging technique further by adopting other works on counter example analysis and apply the technique to formally analyze more industrial systems.

As a future study, we plan to work to add backward analysis capability to MacDebugger. It was noticed that a root cause of the violation of a safety property most often exists at the end of a counter example. Thus, if it is possible to analyze a counter example backward (from the end to the start of the counter example), this may decrease the analysis time significantly. In addition, we will formulate identified bugs using Promela `never` claim and run model checker on the original model with the bug descriptions to know if there still exist unrevealed bugs or not. This approach eliminates the overhead of analyzing a large volume of counter examples at the cost of increased model size. The comparison between these two different approaches on analysis performance and convenience of formulating bugs can be an interesting research topic.

References

1. High-availability technique in NXG 2000. Technical report, http://www.secui.com/product/nxg/pdf/NXG_technique_03.pdf
2. The Spin Model Checker Home Page, http://www.spinroot.com
3. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–136. Springer, Heidelberg (2003)

4. Basu, S., Saha, D., Smolka, S.A.: Localizing programs errors for cimple debugging. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 79–96. Springer, Heidelberg (2004)
5. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: Scr*: A toolset for specifying and analyzing requirements. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) COMPASS 1995. LNCS, vol. 1130, Springer, Heidelberg (1996)
6. Pasareanu, C.S., Dwyer, M.B., Visser, W.: Finding feasible counter-examples when model checking java programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 284–298. Springer, Heidelberg (2001)
7. Holzmann, G.J., Smith, M.H.: Automating software feature verification. Bell Labs Technical Journal 5(2), 72–87 (2000)
8. Holzmann, G.J.: The Spin Model Checker. Wiley, New York (2003)
9. Zakiuddin, I., Goldsmith, M., Whittaker, O., Gardiner, P.: A methodology for model-checking ad-hoc networks. In: Ball, T., Rajamani, S.K. (eds.) SPIN Workshop. LNCS, vol. 2648, Springer, Heidelberg (2003)
10. Bhargavan, K., Gunter, C.A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal Analysis of Network Simulations. IEEE Transaction on Software Engineering 8(2) (2002)
11. Bhargavan, K., Obradovic, D., Gunter, C.: Formal verification of standards for distance vector routing protocols. Journal of the ACM 49(4), 538–576 (2002)
12. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design (2004)
13. Kim, M.: MacDebugger: A Monitoring and Checking (MaC) based Debugger for Formal Models, Technical Report CS-TR-2007-270, CS Dept. KAIST (2007)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1997)
15. Ruys, T.C., Holzmann, G.J.: Advanced spin tutorial. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 304–305. Springer, Heidelberg (2004)
16. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. Principles of Programming Languages (2003)

A Brief Introduction to $\mathcal{THOTL}$*

Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
mgmerayo@fdi.ucm.es, mn@sip.ucm.es, isrodrig@sip.ucm.es

Abstract. In this paper we extend $\mathcal{HOTL}$ (Hypotheses and Observations Testing Logic) to provide a formal framework to test timed systems. The main idea underlying $\mathcal{HOTL}$ is to infer whether a set of *observations* (i.e., results of test applications) allows to claim that the IUT conforms to the specification *if* a specific set of hypotheses is assumed. In this paper we adapt $\mathcal{HOTL}$ to cope with the inclusion of time issues. In addition, we show the soundness and completeness of the new framework, that we call $\mathcal{THOTL}$, with respect to a general notion of timed conformance.

1 Introduction

The main goal of this paper is to extend $\mathcal{HOTL}$ [2] (*Hypotheses and Observations Testing Logic*) to deal with timed systems. The *correctness* of an implementation with respect to a given specification can be stated by using a notion of *conformance*: An implementation *conforms* to a specification if the former shows a behavior *similar* to that of the latter. In this line, we may use formal testing techniques to extract tests from the specification, each test representing a desirable behavior that the implementation under test (in the following IUT) must fulfill. In order to limit the (possibly infinite) time devoted to testing, testers add some reasonable assumptions about the structure of the IUT. However, a framework of hypotheses established in advance is very strict and limits the applicability of a specific testing methodology. The logical framework $\mathcal{HOTL}$ was introduced to cope with the rigidity of other frameworks. $\mathcal{HOTL}$ aims to assess whether a given set of observations implies the correctness of the IUT under the assumption of a given set of hypotheses.

The first decision to define the new framework, that we call $\mathcal{THOTL}$, is to consider a formal language to represent timed systems. Since $\mathcal{HOTL}$ is oriented to deal with a language with a strict alternation between inputs and outputs, we decided to consider a timed extension of finite state machines in order to reuse, as much as possible, the definition of the predicates and rules. Regarding the time domain, we decided to choose a simple approach but richer than singles values: Time intervals.

* This research was partially supported by the Spanish MEC project WEST/FAST TIN2006-15578-C02-01 and the Marie Curie project MRTN-CT-2003-505121/ TAROT. A longer version of this paper can be found in [1].

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 501–510, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Once the language and a notion of timed conformance is fixed, we have to work on how to *adapt* $\mathcal{HOTL}$ to the new setting. First, we have to adapt the notion of *observation* to take into account not only assumptions about the possible time interval governing transitions but also to record the observed time values. Next, we have to modify the existing rules.

The rest of the paper is organized as follows. In Section 2 we briefly review the main concepts underlying the definition of $\mathcal{HOTL}$. This section represents a (very small) summary of [2]. In Section 3 we introduce our extension of finite state machines to model timed systems and define two implementation relations. In Section 4, representing the bulk of the paper, we define the new logical framework. Finally, in Section 5 we present our conclusions and some directions for further research.

2 A Short Summary of $\mathcal{HOTL}$

The goal of $\mathcal{HOTL}$ is to decide whether a given finite set of observations, extracted by applying a test suite to an IUT, is *complete* in the case that the considered hypotheses hold. In other words, we assess whether obtaining these observations from the IUT implies that the IUT conforms to the specification *if* the hypotheses hold. Our logic assumes that specifications and implementations can be represented by using a classical formalism, finite state machines.

The behavior of the IUT observed during the application of tests is represented by means of *observations*, that is, a sequence of inputs and outputs denoting the test and the response produced by the IUT, respectively. Moreover, observations include *attributes* that allow to represent hypothesis concerning specific states of the IUT. Once the observations have been processed, the deduction *rules* of the logic will allow to infer whether we can claim that the IUT conforms to the specification.

Next, we will review the basic elements that are part of the original $\mathcal{HOTL}$: *Observations, predicates, and rules*. During the rest of the paper $\mathtt{Obs}$ denotes the multiset of observations collected during the preliminary interaction with the IUT while $\mathtt{Hyp}$ denotes the set of *hypotheses* the tester has assumed. In this latter set, we will not consider the hypotheses that are implicitly introduced by means of observations.

2.1 Observations

Observations follow the form $ob = (a_1, i_1/o_1, a_2, \ldots, a_n, i_n/o_n, a_{n+1}) \in \mathtt{Obs}$, where ob is a unique identifier. This observation denotes that when the sequence of inputs $i_1, \ldots, i_n$ was proposed to the implementation, the sequence $o_1, \ldots, o_n$ was obtained as response. In addition, for all $1 \leq j \leq n+1$, a_j represents a set of *special attributes* concerning the state of the implementation reached after performing $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ in *this* observation. Attributes denote our assumptions about this state. For all $1 \leq j \leq n+1$ the attributes in the set a_j are of the form $\mathtt{imp}(q)$ or $\mathtt{det}$, where $\mathtt{imp}(q)$ denotes that the implementation state

reached after $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ is associated to a *state identifier name* q and `det` denotes that the implementation state reached after $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ in this observation is deterministic. State identifier names are used to match equal states. The set of all state identifier names will be denoted by $\mathcal{Q}$. In addition, attributes belonging to a_{n+1} can also be of the form `spec`(s) denoting that the implementation state reached after $i_1/o_1, \ldots, i_n/o_n$ is such that the subgraph that can be reached from it is isomorphic to the subgraph that can be reached from the state s of the specification.

2.2 Predicates

A *model predicate* denotes our knowledge about the implementation. Models will be constructed according to the observations and hypotheses we consider. We denote model predicates by `model`(m), where $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{O})$ is a *model*. The meaning of the different components of the tuple is:

- $\mathcal{S}$ (*states*): The set of states that appear in the model.
- $\mathcal{T}$ (*transitions*): Set of transitions appearing in the graph of the model.
- $\mathcal{I}$ (*initial states*): Set of states that are initial in the model.
- $\mathcal{A}$ (*accounting*): The set of *accounting registers* of the model. An accounting register is a tuple $(s, i, outs, f, n)$ denoting that in state $s \in \mathcal{S}$ the input i has been offered n times and we have obtained the outputs belonging to the set $outs$. Besides, for each transition departing from state s and labeled with input i, the function $f : \mathcal{T} \longrightarrow \mathbb{N}$ returns the number of times the transition has been observed. If, due to the hypotheses that we consider, we infer that the number of times we observed an input is high enough to believe that the implementation cannot react to that input either with an output that was not produced before or leading to a state that was not taken before, then the value n is set to $\top$.
- $\mathcal{E}$ (*equality relations*): Set of equalities relating states belonging to $\mathcal{S}$. Equalities have the form s `is` q, where $s \in \mathcal{S}$ is a state and $q \in \mathcal{Q}$ is a state identifier name.
- $\mathcal{D}$ (*deterministic states*): Set of states that are deterministic.
- $\mathcal{O}$ (*used observations*): Set of observations we have used so far for the construction of this model.

Depending on the form of m, a `model`(m) predicate may denote some additional information about m. Models can be labeled by some *tags* to denote special characteristics. A different predicate, `allModelsCorrect`, represents a set of correct models. This predicate is the *goal* of the logic: If it holds then all the IUTs that could produce the observations in `Obs` and meet all the requirements in `Hyp` conform to the specification.

In general, several models can be constructed from a set of observations and hypotheses. Hence, our logic will deal with *sets* of models. If $\mathcal{M}$ is a set of models then the `models`$(\mathcal{M})$ predicate denotes that, according to the observations and hypotheses considered, $\mathcal{M}$ contains all the models that are valid candidates to properly describe the implementation.

2.3 The $\mathcal{HOTL}$ Rules

Once all the observations have been considered a second phase, to add the hypotheses, starts. All rules of the second phase will include the requirement $\mathcal{O} = \texttt{Obs}$.

We present a rule to construct a model from a simple observation. Given a predicate denoting that an observation was collected, the rule deduces some details about the behavior of the implementation.

$$(obser)\frac{ob = (a_1, i_1/o_1, a_2, \ldots, a_n, i_n/o_n, a_{n+1}) \in \texttt{Obs} \ \wedge \ s_1, \ldots, s_{n+1} \text{ are fresh states}}{\texttt{model}\left(\begin{array}{c} \{s_1, \ldots, s_{n+1}\} \cup \mathcal{S}', \\ \{s_1 \stackrel{i_1/o_1}{\longrightarrow} s_2, \ldots, s_n \stackrel{i_n/o_n}{\longrightarrow} s_{n+1}\} \cup \mathcal{T}', \{s_1, \beta\}, \\ \{(s_j, i_j, \{o_j\}, f_{s_j}, 1) \mid 1 \leq j \leq n\} \cup \mathcal{A}', \\ \{s_j \ \texttt{is} \ q_j | 1 \leq j \leq n+1 \ \wedge \ \texttt{imp}(q_j) \in a_j\}, \\ \{s_j \mid 1 \leq j \leq n+1 \ \wedge \ \texttt{det} \in a_j\} \cup \mathcal{D}', \{ob\} \end{array}\right)}$$

where $f_{s_j}(tr) = 1$ if $tr = s_j \stackrel{i_j/o_j}{\longrightarrow} s_{j+1}$ and $f_{s_j}(tr) = 0$ otherwise. Intuitively, the sets $\mathcal{S}'$, $\mathcal{T}'$, $\mathcal{A}'$, and $\mathcal{D}'$, denote the additions due to the possible occurrence of an attribute of the form $\texttt{spec}(s)$. The formal definition of these sets can be found in [2].

We will be able to join different models, created from different observations, into a single model by means of the *(fusion)* rule. The components of the new model will be the union of the components of each model.

$$(fusion)\frac{\begin{array}{c}\texttt{model}\,(\mathcal{S}_1, \mathcal{T}_1, \mathcal{I}_1, \mathcal{A}_1, \mathcal{E}_1, \mathcal{D}_1, \mathcal{O}_1) \ \wedge \\ \texttt{model}\,(\mathcal{S}_2, \mathcal{T}_2, \mathcal{I}_2, \mathcal{A}_2, \mathcal{E}_2, \mathcal{D}_2, \mathcal{O}_2) \ \wedge \ \mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset\end{array}}{\texttt{model}\left(\mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{T}_1 \cup \mathcal{T}_2, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{O}_1 \cup \mathcal{O}_2\right)}$$

By iteratively applying these two rules, we will eventually obtain a model where $\mathcal{O}$ includes all the observations belonging to the set $\texttt{Obs}$. At this point, the inclusion of those hypotheses that are not covered by observations will begin. Our logic considers several possible hypotheses about the IUT. The complete repertory of hypotheses appearing in $\mathcal{HOTL}$ appears in [2].

Since a model is a (probably incomplete) representation of the IUT, in order to check whether a model conforms to the specification, two aspects must be taken into account. First, only the conformance of consistent models will be considered. Second, we will check the conformance of a consistent model by considering the *worst* instance of the model, that is, if this instance conforms to the specification then any other instance extracted from the model does so. This worst instance will conform to the specification only if the unspecified parts of the model are not relevant for the correctness of the IUT it represents. $\mathcal{HOTL}$ provides us with an additional rule that allows to deduce the correctness of a model if the model is consistent and the worst instance of the model conforms to the specification.

3 Timed FSMs and Timed Implementations Relations

In this section we present our timed extension of the classical finite state machine model. We also introduce an *implementation relation* to formally define what is a *correct* implementation. Time intervals will be used to express time constraints associated with the performance of actions. We need to introduce notation, related to time intervals and multisets, that we will use during the rest of the paper.

Definition 1. We say that $\hat{a} = [a_1, a_2]$ is a *time interval* if $a_1 \in \mathbb{R}_+$, $a_2 \in \mathbb{R}_+ \cup \{\infty\}$, and $a_1 \leq a_2$. We assume that for all $r \in \mathbb{R}_+$ we have $r < \infty$ and $r + \infty = \infty$. We consider that $\mathcal{I}_{\mathbb{R}_+}$ denotes the set of time intervals. Let $\hat{a} = [a_1, a_2]$ and $\hat{b} = [b_1, b_2]$ be time intervals. We write $\hat{a} \subseteq \hat{b}$ if we have both $b_1 \leq a_1$ and $a_2 \leq b_2$. In addition, $\hat{a} + \hat{b}$ denotes the interval $[a_1 + b_1, a_2 + b_2]$ and $\pi_i(\hat{a})$, for $i \in \{1, 2\}$, denotes the value a_i.

We will use the delimiters $\{\!|$ and $|\!\}$ to denote multisets. We denote by $\wp(\mathbb{R}^+)$ the multisets of elements belonging to $\mathbb{R}^+$. □

Let us note that in the case of $[t_1, \infty]$ we are abusing the notation since this interval is in fact a half-closed interval, that is, it represents the interval $[t_1, \infty)$.

Definition 2. A *Timed Finite State Machine*, in the following TFSM, is a tuple $M = (\mathcal{S}, \texttt{inputs}, \texttt{outputs}, \mathcal{I}, \mathcal{T})$ where $\mathcal{S}$ is a finite set of states, inputs is the set of input actions, outputs is the set of output actions, $\mathcal{T}$ is the set of transitions, and $\mathcal{I}$ is the set of initial states.

A transition belonging to $\mathcal{T}$ is a tuple $(s, s', i, o, \hat{d})$ where $s, s' \in \mathcal{S}$ are the initial and final states of the transition, respectively, $i \in \texttt{inputs}$ and $o \in \texttt{outputs}$ are the input and output actions, respectively, and $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$ denotes the possible time values the transition needs to be completed. We usually denote transitions by $s \xrightarrow{i/o}_{\hat{d}} s'$.

We say that $(s, s', (i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a *timed trace*, or simply *trace*, of M if there exist $(s, s_1, i_1, o_1, \hat{d}_1)$,..., $(s_{r-1}, s', i_r, o_r, \hat{d}_r) \in \mathcal{T}$, such that $\hat{d} = \sum \hat{d}_i$. We say that $((i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a *timed evolution* of M if there exists $s_{in} \in \mathcal{I}$ such that $(s_{in}, s', (i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a trace of M. We denote by $\texttt{TEvol}(M)$ the set of timed evolutions of M. In addition, we say that $(i_1/o_1, \ldots, i_r/o_r)$ is a *non-timed evolution*, or simply *evolution*, of M and we denote by $\texttt{NTEvol}(M)$ the set of non-timed evolutions of M.

Finally, we say that $s \in \mathcal{S}$ is *deterministic* if there do not exist $(s, s', i, o', \hat{d})$, $(s, s'', i, o'', \hat{d}') \in \mathcal{T}$ such that $o' \neq o''$ or $s' \neq s''$. □

As usual, we assume that both implementations and specifications can be represented by appropriate TFSMs.

During the rest of the paper we will assume that a generic specification is given by $spec = (\mathcal{S}_{spec}, \texttt{inputs}_{spec}, \texttt{outputs}_{spec}, \mathcal{I}_{spec}, \mathcal{T}_{spec})$.

Next we present the basic conformance relation that we consider in our framework. Intuitively, an IUT is conforming if it does not *invent* behaviors for those traces that can be executed by the specification.

Definition 3. Let S and I be TFSMs. We say that I *conforms to* S, denoted by $I \ \mathtt{conf}\ S$, if for all $\rho_1 = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o_n) \in \mathtt{NTEvol}(S)$, with $n \geq 1$, we have $\rho_2 = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o'_n) \in \mathtt{NTEvol}(I)$ implies $\rho_2 \in \mathtt{NTEvol}(S)$. □

In addition to the non-timed conformance of the implementation, we require some time conditions to hold. Specifically, we will check that the observed time values (from the implementation) belong to the time interval indicated in the specification.

Definition 4. Let I be a TFSM. We say that $((i_1/o_1, \ldots, i_n/o_n), t)$ is an *observed timed execution* of I, or simply *timed execution*, if the observation of I shows that the sequence $(i_1/o_1, \ldots, i_n/o_n)$ is performed in time t.

Let $\Phi = \{e_1, \ldots, e_m\}$ be a set of input/output sequences and let us consider a multiset of timed executions $H = \{\!\!\{(e'_1, t_1), \ldots, (e'_n, t_n)\}\!\!\}$. We say that $\mathtt{Sampling}_{(H,\Phi)} : \Phi \longrightarrow \wp(\mathbb{R}^+)$ is a *sampling application* of H for Φ if for all $e \in \Phi$ we have $\mathtt{Sampling}_{(H,\Phi)}(e) = \{\!\!\{t \mid (e,t) \in H\}\!\!\}$. □

Timed executions are input/output sequences together with the time that it took to perform the sequence. Regarding sampling applications, we just associate with each evolution the multiset of observed execution time values.

Definition 5. Let I and S be TFSMs, H be a multiset of timed executions of I, and $\Phi = \{e \mid \exists\, t : (e,t) \in H\} \cap \mathtt{NTEvol}(S)$. For all $e \in \Phi$ we define the *sample interval* of e in H as

$$\widehat{S}_{(H,e)} = [\mathtt{min}(\mathtt{Sampling}_{(H,\Phi)}(e)), \mathtt{max}(\mathtt{Sampling}_{(H,\Phi)}(e))]$$

We say that I $H-$*timely conforms* to S, denoted by $I \ \mathtt{conf}^H_{int}\ S$, if $I \ \mathtt{conf}\ S$ and for all $e \in \Phi$ we have that for all time interval $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$ we have that if $(e, \hat{d}) \in \mathtt{TEvol}(S)$ then $\widehat{S}_{(H,e)} \subseteq \hat{d}$ holds. □

4 Timed Extension of $\mathcal{HOTL}$: $\mathcal{THOTL}$

In this section we show how $\mathcal{HOTL}$ has to be adapted and extended to cope with time issues. While some of the rules dealing with the internal *functional* structure of the implementation remain the same, the inclusion of time strongly complicates the framework, constituting $\mathcal{THOTL}$ almost a complete *new* formalism. First, we need to redefine most components of the logic to consider temporal aspects. Observations will include the time values that the IUT takes to emit an output since an input is received. Additionally, the model will be extended to take into account the different time values appearing in the observations for each input/output outgoing from a state. Finally, we will modify the deduction rules.

4.1 Temporal Observations

Temporal observation are an extension of the observations introduced in $\mathcal{HOTL}$. They follow the format $ob = (a_1, i_1/o_1/t_1, a_2, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \texttt{Obs}$. It denotes that when the sequence of inputs $i_1, \ldots, i_n$ was proposed from an initial state of the implementation, the sequence $o_1, \ldots, o_n$ was obtained as response in $t_1, \ldots, t_n$ time units, respectively.

In addition to the attributes presented in Section 2, temporal observations may include a new type of attribute. For all $1 < j \leq n$, the attributes in the set a_j can be also of the form $\texttt{int}(\hat{d})$, with $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$. Such an attribute denotes that once the implementation has performed $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$, the time that it takes to emit the output o_j, after the input i_j is received, belongs to the interval $\hat{d}$. We assume that this attribute cannot appear in the set a_1 since the implementation is in an initial state. Thus, at this stage, no actions have taken place yet.

4.2 New Model Predicates

Temporal observations will allow to create *model predicates* that denote our knowledge about the implementation. A model predicate is denoted by $\texttt{model}\,(m)$, where $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{O})$. The only change we need to introduce affects the accounting component $\mathcal{A}$. Now, each register will be a tuple $(s, i, outs, f, \delta, n)$ where the new function $\delta : \mathcal{T} \longrightarrow \mathcal{I}_{\mathbb{R}_+} \times \wp(\mathbb{R}^+)$ computes for each transition departing from state s with input i and output $o \in outs$ the time interval, according with our knowledge up to now, in which the transition could be performed. In addition, it also returns the set of time values the implementation took to perform the transition. If no assumptions about the interval are made by means of temporal observations, the interval will be set to $[0, \infty]$. In the case of transitions not fulfilling the required conditions about s, i, and $outs$, an arbitrary value is returned.

4.3 Changing the Existing Rules

First, we will include a new rule to construct a model from a temporal observation. This rule plays a similar role to the original $(obser)$ rule of $\mathcal{HOTL}$ that allows to build a model from a simple non-temporal observation.

$$(tobser)\,\frac{ob = (a_1, i_1/o_1/t_1, a_2, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \texttt{Obs} \ \wedge\ s_1, \ldots, s_{n+1} \text{ fresh states}}{\texttt{model}\left(\begin{array}{c} \{s_1, \ldots, s_{n+1}\} \cup \mathcal{S}', \\ \{s_1 \xrightarrow{i_1/o_1} s_2, \ldots, s_n \xrightarrow{i_n/o_n} s_{n+1}\} \cup \mathcal{T}', \{s_1, \beta\}, \\ \{(s_j, i_j, \{o_j\}, f_{s_j}\delta_{s_j}, 1) \mid 1 \leq j \leq n\} \cup \mathcal{A}', \\ \{s_j \texttt{ is } q_j | 1 \leq j \leq n+1 \ \wedge\ \texttt{imp}(q_j) \in a_j\}, \\ \{s_j \mid 1 \leq j \leq n+1 \ \wedge\ \texttt{det} \in a_j\} \cup \mathcal{D}', \{ob\} \end{array}\right)}$$

where $f_{s_j}(tr)$ is equal to 1 if $tr = s_j \xrightarrow{i_j/o_j} s_{j+1}$ and equal to 0 otherwise; and

$$\delta_{s_j}(tr) = \begin{cases} (\hat{d}, \{\!\{t_j\}\!\}) & \texttt{if } tr = s_j \xrightarrow{i_j/o_j} s_{j+1} \wedge \texttt{int}(\hat{d}) \in a_{j+1} \\ ([0,\infty], \{\!\{t_j\}\!\}) & \texttt{if } tr = s_j \xrightarrow{i_j/o_j} s_{j+1} \wedge \texttt{int}(\hat{d}) \notin a_{j+1} \\ ([0,\infty], \emptyset) & \text{otherwise} \end{cases}$$

The sets of states, transitions, accounting registers, and deterministic states will be extended with some extra elements, taken from the specification, if the tester assumes that the last state of the observation is isomorphic to a state of the specification. The sets $\mathcal{S}'$, $\mathcal{T}'$, $\mathcal{A}'$, and $\mathcal{D}'$ are formally defined in [1]

The iterative application of the previously introduced (*fusion*) rule (see Section 2.3) will allow us to join different models created from different temporal observations into a single model.

At this point, the inclusion of those hypotheses that are not covered by observations will begin. During this new phase, we will usually need several models to represent all the `TFSM`s that are compatible with a set of observations and hypotheses. Some of the rules use the `modelElim` function. If we find out that a state of the model coincides with another one, we will eliminate one of the states and will allocate all of its constraints to the other one. The `modelElim` function modifies the components that define the model, in particular the accounting set. A similar function appeared in the original formulation of $\mathcal{HOTL}$. However, due to the inclusion of time issues, this function must be adapted to deal with the new considerations. A formal definition of this function can be found in [1]. It is only necessary to consider that those rules using `modelElim` have to consider the temporal version of this function. The rest of the rules belonging to $\mathcal{HOTL}$ do not vary in their formulation.

In $\mathcal{HOTL}$ we have some rules that may lead to inconsistent models. In some of these cases, an empty set of models is produced, that is, the inconsistent model is eliminated. Before granting conformance, we need to be sure that at least one model belonging to the set is consistent. $\mathcal{HOTL}$ already provides us with a rule that labels a model as *consistent*. Let us note that the inconsistences created by a rule can be detected by the subsequent applications of rules. Thus, a model is free of inconsistencies if for any other rule either it is not applicable to the model or the application does not modify the model. Due to space limitations we do not include the details of this rule (the formal definition can be found in [2]).

Similar to $\mathcal{HOTL}$, as explained at the end of Section 2, in order to check whether a model conforms to the specification we have to take into account that only the conformance of consistent models will be considered. In addition, given a consistent model, we will check its conformance with respect to the specification by considering the *worst* instance of the model, that is, if this instance conforms to the specification then any other instance extracted from the model does so. This worst instance is constructed as follows: For each state s and input i such that the behavior of s for i is not closed *and* either s is not deterministic or no transition with input i exists in the model, a new *malicious* transition is created. The new transition is labelled with a special output *error* that does

not belong to $\texttt{outputs}_{spec}$. This transition leads to a new state $\perp$ having no outgoing transitions. Since the specification cannot produce the output *error*, this worst instance will conform to the specification only if the unspecified parts of the model are not relevant for the correctness of the IUT it represents.

Definition 6. Let $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \texttt{Obs})$ be a model. We define the *worst temporal instance* of the model m with respect to the considered specification *spec*, denoted by $\texttt{worstTempCase}(m)$, as the TFSM

$$\left(\begin{array}{l} \mathcal{S} \cup \{\perp\}, \texttt{inputs}_{spec}, \texttt{outputs}_{spec} \cup \{error\}, \\ \left\{ s \xrightarrow{i/o}_{\hat{d}} s' \;\middle|\; \begin{array}{l} s \xrightarrow{i/o} s' \in \mathcal{T} \wedge \\ \exists\, outs, f, \delta, n : (s, i, outs, f, \delta, n) \in \mathcal{A} \wedge \\ o \in outs \;\wedge\; \pi_1(\delta(s \xrightarrow{i/o} s')) = \hat{d} \end{array} \right\} \\ \qquad\qquad\qquad\qquad \bigcup \\ \left\{ s \xrightarrow{i/error}_{[o,\infty]} \perp \;\middle|\; \begin{array}{l} s \in \mathcal{S} \;\wedge\; i \in \texttt{inputs}_{spec} \wedge \\ \nexists\, outs : (s, i, outs, f, \delta, \top) \in \mathcal{A} \wedge \\ (s \notin \mathcal{D} \vee \;\nexists\, s', o : s \xrightarrow{i/o} s') \end{array} \right\}, \mathcal{I} \end{array}\right)$$

□

Thus, the rule for indicating the correctness of a model is

$$(correct)\,\frac{\begin{array}{c} m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \texttt{Obs}) \;\wedge\; \texttt{consistent}(m) \;\wedge \\ H = \texttt{reduce}(\texttt{Obs}) \;\wedge\; \texttt{worstTempCase}(m)\; \texttt{conf}_{int}^{H}\; spec \end{array}}{\texttt{models}\,(\{\texttt{correct}(m)\})}$$

where

$$\texttt{reduce}(\texttt{Obs}) = \{(i_1/o_1/t_1, \ldots, i_n/o_n/t_n) | (a_1, i_1/o_1/t_1, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \texttt{Obs}\}$$

Now we can consider the conformance of a set of models. A set conforms to the specification if all the elements do so and the set contains at least one element. Note that an empty set of models denotes that all the models were inconsistent.

$$(allCorrect)\,\frac{\texttt{models}\,(\mathcal{M}) \;\wedge\; \mathcal{M} \neq \emptyset \;\wedge\; \mathcal{M} = \{\texttt{correct}(m_1), \ldots, \texttt{correct}(m_n)\}}{\texttt{allModelsCorrect}}$$

Now that we introduce a correctness criterion. In the next definition, in order to uniquely denote observations, fresh names are assigned to them. Besides, let us note that all hypothesis predicates follow the form $h \in \texttt{Hyp}$ for some h belonging to Hyp.

Definition 7. Let *spec* be a TFSM, Obs be a set of observations, and Hyp be a set of hypotheses. Let $A = \{ob = o \mid ob \text{ is a fresh name } \wedge\; o \in \texttt{Obs}\}$ and $B = \{h_1 \in \texttt{Hyp}, \ldots, h_n \in \texttt{Hyp}\}$, where $\texttt{Hyp} = \{h_1, \ldots, h_n\}$.

If the deduction rules allow to infer allModelsCorrect from the set of predicates $C = A \cup B$, then we say that C *logically conforms to spec* and we denote it by C logicConf *spec*. □

In order to prove the validity of our method, we have to relate the deductions obtained by using our logic with the notion of conformance introduced in Definition 5. The *semantics* of a predicate is described in terms of the set of TFSMs that fulfill the requirements given by the predicate; given a predicate p, we denote this set by $\nu(p)$. Despite the differences, the construction is similar to that in [2] for classical finite state machines. Let us consider that P is the conjunction of all the considered observation and hypothesis predicates. Intuitively, the set $\nu(P)$ denotes all the TFSMs that can produce these observations and fulfill these hypotheses, that is, all the TFSMs that, according to our knowledge, can *define* the IUT. So, if our logic deduces that these TFSMs conform to the specification then the IUT actually conforms to the specification.

Theorem 1. Let *spec* be a TFSM, Obs be a set of observations, and Hyp be a set of hypotheses. Let $A = \{ob = o \mid ob \text{ is a fresh name } \wedge\ o \in \texttt{Obs}\} \neq \emptyset$ and $B = \{h_1 \in \texttt{Hyp}, \ldots, h_n \in \texttt{Hyp}\}$, where $\texttt{Hyp} = \{h_1, \ldots, h_n\}$. Let $C = A \cup B$ be a set of predicates and $H = \texttt{reduce}(\texttt{Obs})$. Then, $C\ \texttt{logicConf}\ spec$ iff for all TFSM $M \in \nu(\bigwedge_{p \in C})$ we have $M\ \texttt{conf}_{int}^{H}\ spec$ and $\nu(\bigwedge_{p \in C}) \neq \emptyset$.

5 Conclusions and Future Work

In this paper we have provided an extension of $\mathcal{HOTL}$ to deal with systems presenting temporal information. What started as a simple exercise, where only a couple of rules were going to be modified, became a much more difficult task. As we have already commented, the inclusion of time complicates not only the original framework, with a more involved definition of the *accounting* and the functions that modify it, but adds some new complexity with the addition of new rules. These new rules have not been included in this paper due to lack of space, but they can be found in [1]. Regarding future work, the first task is to show the validity of the method, that is, to formally prove the soundness and completeness of $\mathcal{THOTL}$. The second task is to construct a stochastic version of $\mathcal{HOTL}$. Taking the current paper as basis this task should be easy.

References

1. Merayo, M.G., Núñez, M., Rodríguez, I.: THOTL: A timed extension of HOTL (2007), Available at http://kimba.mat.ucm.es/testing/papers/thotl.pdf
2. Rodríguez, I., Merayo, M.G., Núñez, M.: HOTL: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming (in press, 2007) http://dx.doi.org/10.1016/j.jlap.2007.03.002

On-the-Fly Model Checking of Fair Non-repudiation Protocols

Guoqiang Li and Mizuhito Ogawa

Japan Advanced Institute of Science and Technology
Asahidai, Nomi, Ishikawa, 923-1292 Japan
{guoqiang,mizuhito}@jaist.ac.jp

Abstract. A fair non-repudiation protocol should guarantee, (1) when a sender sends a message to a receiver, neither the sender nor the receiver can deny having participated in this communication; (2) no principals can obtain evidence while the other principals cannot do so. This paper extends the model in our previous work [12], and gives a sound and complete on-the-fly model checking method for fair non-repudiation protocols under the assumption of a bounded number of sessions. We also implement the method using Maude. Our experiments automatically detect flaws of several fair non-repudiation protocols.

1 Introduction

Fair non-repudiation protocols intend a reliable exchange of messages in the situation that each principal can be dishonest, who tries to take advantage of other principals by aborting the communication or sending fake messages. A fair non-repudiation protocol needs to ensure two properties, non-repudiation and fairness. Non-repudiation means that when a sender sends a message to a receiver, neither the sender nor the receiver can deny participation in this communication. Fairness means no principals can obtain evidence while the other principals cannot do so. Difficulties in verifying these security properties come from various factors of infinity,

– each principal can initiate or respond to an unbounded number of sessions;
– each principal may communicate with an unbounded number of principals;
– each intruder can produce, store, duplicate, hide, or replace an unbounded number of messages based on the messages sent in the network, following the Dolev-Yao model [7].
– each dishonest principal may disobey the prescription of the protocol, sending an unbounded number of messages it can generate.

This paper proposes a sound and complete on-the-fly model checking method for fair non-repudiation protocols under the restriction of a bounded number of sessions. This method is based on trace analysis. To the best of our knowledge, this is the first model checking method applied to the non-repudiation property.

To describe non-repudiation protocols, we choose a process calculus based on a variant of Spi calculus [1]. The calculus uses environment-based communication, instead of channel-based communication, with the following features.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 511–522, 2007.
© Springer-Verlag Berlin Heidelberg 2007

- The calculus excludes recursive operations, so that only finitely many sessions are represented.
- To represent an unbounded number of principals, a binder is used to represent intended destination of messages [12].
- Following the Dolev-Yao model, a deductive system, which can generate infinitely many messages, is exploited to describe abilities of intruders [3].
- Another deductive system is introduced to generate infinitely many messages that dishonest principals may produce and send [17].

A finite *parametric model* is proposed by abstracting/restricting the infinities. It is sound and complete under the restriction of a bounded number of sessions. The on-the-fly model checking on the parametric model is implemented by Maude, which successfully detects flaws of several fair non-repudiation protocols.

Due to the lack of space, we omit proofs of lemmas and theorems; these can be found in the extended version [13].

2 Concrete Model for Protocol Description

Assume four countable disjoint sets: $\mathcal{L}$ for labels, $\mathcal{N}$ for names, $\mathcal{B}$ for binder names and $\mathcal{V}$ for variables. Let $a, b, c, \ldots$ indicate labels, $m, n, A, B, \ldots$ indicate names, $\mathtt{m}, \mathtt{n}, \mathtt{k}, \ldots$ indicate binder names, and let $x, y, z, \ldots$ indicate variables.

Definition 1 (Messages). *Messages $M, N, L \ldots$ in a set $\mathcal{M}$ are defined iteratively as follows:*

$$\begin{aligned} pr &::= n \mid x \\ M, N, L &::= pr \mid \mathtt{m}[pr, \ldots, pr] \mid (M, N) \mid \{M\}_L \mid \mathcal{H}(M) \end{aligned}$$

A message is ground, if it does not contain any variables.

A binder, $\mathtt{m}[pr_1, \ldots, pr_n]$ is a message that can be regarded as a special name indexed by its parameters. One usage of binders is to denote encryption keys. For instance, $+\mathtt{k}[A]$ and $-\mathtt{k}[A]$ represent A's public key and private key, respectively.

Definition 2 (Processes). *Processes in a set $\mathcal{P}$ are defined as follows:*

$$\begin{aligned} P, Q, R ::= \mathbf{0} \mid \overline{a}M.P \mid a(x).P \mid [M = N]\, P \mid (\mathtt{new}\ x)P \mid (\nu n)P \mid \\ let\ (x, y) = M\ in\ P \mid case\ M\ of\ \{x\}_L\ in\ P \mid P + Q \mid P \| Q \end{aligned}$$

Variables x and y are bound in $a(x).P$, $(\mathtt{new}\ x)P$, let $(x, y) = M$ in P, and case M of $\{x\}_L$ in P. Sets of free variables and bound variables in P are denoted by $f_v(P)$ and $b_v(P)$, respectively. A process P is closed if $f_v(P) = \emptyset$. A name is free in a process if it is not restricted by a restriction operator ν. Sets of free names and local names of P are denoted by $f_n(P)$ and $l_n(P)$, respectively.

A new process, summation $P + Q$ that behaves like P or Q, intends a dishonest principal that has several choices, such as aborting communication, or running a recovery stage.

Messages that the environment can generate are started from the current finite knowledge, denoted by S ($\subseteq \mathcal{M}$), and deduced by an *environmental deductive system*. Here, we presuppose a countable set $\mathcal{E}$ ($\subseteq \mathcal{M}$), for those public names and ground binders, such as each principal's name, public keys, and intruders' names. The environmental deductive system is shown in Fig. 1.

$$\frac{}{S \vdash M}\ M \in \mathcal{E}\ \ Env \qquad \frac{}{S \vdash M}\ M \in S\ \ Ax$$

$$\frac{S \vdash M\ \ S \vdash N}{S \vdash (M,N)}\ Pair_intro \quad \frac{S \vdash (M,N)}{S \vdash M}\ Pair_elim1 \quad \frac{S \vdash (M,N)}{S \vdash N}\ Pair_elim2$$

$$\frac{S \vdash \{M\}_{\mathtt{k}[A,B]}\ \ S \vdash \mathtt{k}[A,B]}{S \vdash M}\ Senc_elim \qquad \frac{S \vdash M\ \ S \vdash \mathtt{k}[A,B]}{S \vdash \{M\}_{\mathtt{k}[A,B]}}\ Senc_intro$$

$$\frac{S \vdash \{M\}_{\pm\mathtt{k}[A]}\ \ S \vdash \mp\mathtt{k}[A]}{S \vdash M}\ Penc_elim \qquad \frac{S \vdash M\ \ S \vdash \pm\mathtt{k}[A]}{S \vdash \{M\}_{\pm\mathtt{k}[A]}}\ Penc_intro$$

Fig. 1. Environmental deductive system

A process P that describes a dishonest principal A can send out all messages generated through $\vdash$, and can also encrypt messages with A's private key and shared key. A *P-deductive system* is defined in Fig. 2.

$$\frac{S \vdash M}{S \vdash_P M} \quad \frac{S \vdash_P M}{S \vdash_P \{M\}_{\mathtt{k}[A,B]}} \quad \frac{S \vdash_P M}{S \vdash_P \{M\}_{-\mathtt{k}[A]}}$$

Fig. 2. A P-deductive system

An *action* is a term of form $\overline{a}M$ or $a(M)$. It is ground if its attached message is ground. The messages in a concrete trace s, represented by $msg(s)$, are those messages in output actions of the concrete trace s. We use $s \vdash M$ to abbreviate $msg(s) \vdash M$, and $s \vdash_P M$ to abbreviate $msg(s) \vdash_P M$.

Definition 3 (Concrete trace and configuration). *A concrete trace s is a ground action string that satisfies each decomposition $s = s'.a(M).s''$ implies $s' \vdash M$, and each $s = s'.\overline{a}M.s''$ implies $s' \vdash_P M$, where P is a closed process that contains the label a. ϵ represents an empty trace. A concrete configuration is a pair $\langle s, P\rangle$, in which s is a concrete trace and P is a closed process.*

The transition relation of concrete configurations is defined by the rules listed in Fig. 3. Two symmetric forms, $(RSUM)$ of $(LSUM)$, and $(RCOM)$ of $(LCOM)$ are omitted from the figure. Furthermore, a function `Opp` is defined for complemental key in decryption and encryption. Thus we have $\mathtt{Opp}(+\mathtt{k}[A]) = -\mathtt{k}[A]$, $\mathtt{Opp}(-\mathtt{k}[A]) = +\mathtt{k}[A]$ and $\mathtt{Opp}(\mathtt{k}[A,B]) = \mathtt{k}[A,B]$.

For convenience, we say a concrete configuration $\langle s, P\rangle$ *reaches* $\langle s', P'\rangle$, if $\langle s, P\rangle \longrightarrow^* \langle s', P'\rangle$. A concrete configuration is a *terminated configuration* if no

$$
\begin{array}{rl}
(INPUT) & \langle s, a(x).P\rangle \longrightarrow \langle s.a(M), P\{M/x\}\rangle \;\; s \vdash M \\
(OUTPUT) & \langle s, \overline{a}M.P\rangle \longrightarrow \langle s.\overline{a}M, P\rangle \\
(DEC) & \langle s, case\ \{M\}_L\ of\ \{x\}_{L'}\ in\ P\rangle \longrightarrow \langle s, P\{M/x\}\rangle \;\; L' = \mathtt{Opp}(L) \\
(PAIR) & \langle s, let\ (x,y) = (M,N)\ in\ P\rangle \longrightarrow \langle s, P\{M/x, N/y\}\rangle \\
(NEW) & \langle s, (\mathtt{new}\ x)P\rangle \longrightarrow \langle s, P\{M/x\}\rangle \quad s \vdash_P M \\
(RESTRICTION) & \langle s, (\nu n)P\rangle \longrightarrow \langle s, P\{m/n\}\rangle \quad m \notin f_n(P) \\
(MATCH) & \langle s, [M=M]P\rangle \longrightarrow \langle s, P\rangle \\
(LSUM) & \langle s, P+Q\rangle \longrightarrow \langle s, P\rangle \\
(LCOM) & \dfrac{\langle s, P\rangle \longrightarrow \langle s', P'\rangle}{\langle s, P\|Q\rangle \longrightarrow \langle s', P'\|Q\rangle}
\end{array}
$$

Fig. 3. Concrete transition rules

transition rules can be applied to it. A sequence of consecutive concrete configurations is named a *path*. A concrete configuration $\langle s, P\rangle$ *generates* a concrete s', if $\langle s, P\rangle$ reaches $\langle s', P'\rangle$ for some P'.

3 Representing Protocols and Security Properties

3.1 Representing Protocols

For simplicity of representation, we use several convenient abbreviations. Pair splitting is applied to input and decryption.

$$
a(x_1, x_2).P \triangleq a(x).let\ (x_1, x_2) = x\ in\ P
$$
$$
case\ M\ of\ \{x_1, x_2\}_L\ in\ P \triangleq case\ M\ of\ \{x\}_L\ in\ let\ (x_1, x_2) = x\ in\ P
$$

Similarly, we write $let\ (x_1, x_2, \ldots, x_n) = M\ in\ P$, $a(x_1, x_2, \ldots, x_n).P$, and $case\ M\ of\ \{x_1, x_2, \ldots, x_n\}_L\ in\ P$ for tuples of messages.

We will use a simplified variation of Zhou-Gollmann non-repudiation protocol to illustrate how our system works. The full ZG protocol is proposed in [19]. Note that besides a standard flow description, a fair non-repudiation protocol also contains a description on what are evidences for participated principals.

$$
A \longrightarrow B : \quad \{B, N_A, \{M\}_K\}_{-K_A} \tag{1}
$$
$$
B \longrightarrow A : \quad \{A, N_A, \{M\}_K\}_{-K_B} \tag{2}
$$
$$
A \longrightarrow S : \quad \{B, N_A, K\}_{-K_A} \tag{3}
$$
$$
S \longrightarrow A : \quad \{A, B, N_A, K\}_{-K_S} \tag{4}
$$
$$
S \longrightarrow B : \quad \{A, B, N_A, K\}_{-K_S} \tag{5}
$$

The evidence that A *sends the message* M *to* B (referred as M_1) is the pair of messages that B accepted in (1) and (5). In (1), A sends a signed message to B, and B can confirm that the intended receiver of (1) is B by decrypting it by the public key $+K_A$. In (5), B checks whether N_A in (5) coincides with that in (1). If they match, B can confirm that the TTP S has received K from A in (3).

Alternatively, the evidence that B *receives the message* M *from* A (referred as M_2) is the pair of the messages that A accepted in (2) and (4).

Fresh variables are used to denote the sub-messages that the principal can use to deceive another principal. These variables are bound by the `new` primitive. After receiving messages, each principal may abort the communication. Thus a summation " + " is used to represent nondeterministic choices of a principal.

$$\begin{aligned}
A \triangleq\, & (\nu N_A)(\mathtt{new}\, x_1, x_2)\, \overline{a1}\{x_1, N_A, x_2\}_{-\mathtt{k}[A]}).a2(x_3). \\
& \mathit{case}\ x_3\ \mathit{of}\ \{x_4, x_5, x_6\}_{+\mathtt{k}[x_1]}\ \mathit{in}\ [x_4 = A]\,[x_5 = N_A]\,[x_6 = x_2]\,(\mathbf{0}+ \\
& (\mathtt{new}\, x_7, x_8)\, \overline{a3}\{x_7, x_8\}_{-\mathtt{k}[A]}.a4(x_9).\ \mathit{case}\ x_9\ \mathit{of}\ \{x_{10}, x_{11}, x_{12}, x_{13}\}_{+\mathtt{k}[S]} \\
& \mathit{in}\ [x_{10} = A]\,[x_{11} = x_1]\,[x_{12} = N_A]\,[x_{13} = x_8].\mathbf{0}) \\
B \triangleq\, & b1(y_1).\mathit{case}\ y_1\ \mathit{of}\ \{y_2, y_3, y_4\}_{+\mathtt{k}[A]}\ \mathit{in}[y_2 = B]\,(\mathbf{0}+ \\
& (\mathtt{new}\, y_5)\, \overline{b2}\{A, y_5\}_{-\mathtt{k}[B]}.b3(y_6).\mathit{case}\ y_6\ \mathit{of}\ \{y_7, y_8, y_9, y_{10}\}_{+\mathtt{k}[S]}\ \mathit{in} \\
& [y_7 = A]\,[y_8 = B]\,[y_9 = y_3].\mathbf{0}) \\
S \triangleq\, & s1(z_1).\mathit{case}\ z_1\ \mathit{of}\ \{z_2\}_{+\mathtt{k}[z_3]}\ \mathit{in}\ \overline{s2}\{z_3, z_2\}_{-\mathtt{k}[S]}.\overline{s2}\{z_3, z_2\}_{-\mathtt{k}[S]}.\mathbf{0} \\
SYS^{ZG} \triangleq\, & A \| B \| S
\end{aligned}$$

3.2 Probing Transformation

Given a process P, the *context* $P[.]$ is obtained when all occurrences of $\mathbf{0}$ in P are replaced by *holes*, $[.]$. Let $\phi(P)$ be the set of holes in $P[.]$.

Definition 4 (Probing transformation). *Given a process P that represents a protocol, a probing transformation is generated from P, by applying the following two transformations, and returns a process (named a probing process).*

- *Declaration process insertion: Let $P[.]$ be the context of P. Given a set $\psi \subseteq \phi(P)$, and a message M, $P_{\psi,M}$ is a probing process generated from P, such that holes in ψ are inserted by the same process $\overline{\mathtt{c}}\, M.\mathbf{0}$ with a fresh label $\overline{\mathtt{c}}$ (named declaration process), and holes in $\phi(P) - \psi$ are inserted by $\mathbf{0}$ in $P[.]$.*
- *Guardian process composition: A probing process P_g is formed of P composed with a process $c(x).\mathbf{0}$ with a fresh label c (named guardian process), that is, $P \| c(x).\mathbf{0}$.*

Intuitively, declaration process insertion is used to show that a principal can provide a message M at the end of the session. Guardian process composition is used to check whether a message is observable in the environment.

3.3 Action Terms

Definition 5. *Let α range over the set of actions. Action terms are defined as follows:*

$$T ::= \alpha \mid \neg T \mid T \wedge T \mid T \vee T$$
$$\sigma ::= T \mid T \hookleftarrow T \mid T \hookrightarrow_F T$$

Action terms inductively defined by T are *state action terms*, and those defined by σ are *path action terms*. A state action term is also a path action term.

We define two relations: the relation $\models_t$ between a concrete trace and a state action term, and $\models$ between a concrete configuration and a path action term.

- $s \models_t \alpha$: there exists a ground substitution ρ from variables to ground messages such that $\alpha\rho$ occurs in s.
- $s \models_t \neg T$: $s \not\models_t T$.
- $s \models_t T_1 \wedge T_2$: $s \models_t T_1$ and $s \models_t T_2$.
- $s \models_t T_1 \vee T_2$: $s \models_t T_1$ or $s \models_t T_2$.
- $\langle s, P\rangle \models T$: for each concrete trace s' generated by $\langle s, P\rangle$, $s' \models_t T$ holds.
- $\langle s, P\rangle \models T_1 \hookleftarrow T_2$: for each concrete trace s' generated by $\langle s, P\rangle$, if there is a ground substitution ρ such that $s' \models_t T_2\rho$, then $s' \models_t T_1\rho$, and $T_1\rho$ occurs before $T_2\rho$ in s'.
- $\langle s, P\rangle \models T_1 \hookrightarrow_F T_2$: for each concrete configuration $\langle s', P'\rangle$ reached by $\langle s, P\rangle$, if there is a ground substitution ρ such that $s' \models_t T_1\rho$, then for every path starting from $\langle s', P'\rangle$, there exists a concrete trace s'' such that $s'' \models_t T_2\rho$.

3.4 Representing Security Properties

For the simplified ZG protocol, evidences M_1 and M_2 in Section 3.1 correspond to the two non-repudiation properties [21,9], respectively.

- *Non-repudiation of origin (NRO)* is intended to protect against the sender's false denial of having sent the messages.
- *Non-repudiation of receipt (NRR)* is intended to protect against the receiver's false denial of having received the message.

The evidence M_1 (resp. M_2) is the pair of messages in (1) and (5) (resp. (2) and (4)). In the protocol description, they are messages received at $b1$ and $b3$ (resp. $a2$ and $a4$) as y_1 and y_6 (resp. x_3 and x_9). Then the declaration process is $\overline{evid}_A(y_1, y_6).\mathbf{0}$ (resp. $\overline{evid}_B(x_3, x_9).\mathbf{0}$).

Each process may have several action paths, since it may contain the summation $+$. The probing transformation replaces $\mathbf{0}$ reachable by paths containing both $b1$ and $b3$ (resp. $a2$ and $a4$) with the declaration process.

$$\begin{aligned}
A_p \triangleq\ & (\nu N_A)(\mathtt{new}\, x_1, x_2)\, \overline{a1}\{x_1, N_A, x_2\}_{-\mathtt{k}[A]}).a2(x_3).\\
& case\ x_3\ of\ \{x_4, x_5, x_6\}_{+\mathtt{k}[x_1]}\ in\ [x_4 = A]\,[x_5 = N_A]\,[x_6 = x_2]\,(\mathbf{0}+\\
& (\mathtt{new}\, x_7, x_8)\, \overline{a3}\{x_7, x_8\}_{-\mathtt{k}[A]}.a4(x_9).\ case\ x_9\ of\ \{x_{10}, x_{11}, x_{12}, x_{13}\}_{+\mathtt{k}[S]}\\
& in\ [x_{10} = A]\,[x_{11} = x_1]\,[x_{12} = N_A]\,[x_{13} = x_8].\overline{\mathbf{evid}_\mathbf{B}}(\mathbf{x_3}, \mathbf{x_9}).\mathbf{0})\\
B_p \triangleq\ & b1(y_1).case\ y_1\ of\ \{y_2, y_3, y_4\}_{+\mathtt{k}[A]}\ in[y_2 = B]\,(\mathbf{0}+\\
& (\mathtt{new}\, y_5)\, \overline{b2}\{A, y_5\}_{-\mathtt{k}[B]}.b3(y_6).case\ y_6\ of\ \{y_7, y_8, y_9, y_{10}\}_{+\mathtt{k}[S]}\ in\\
& [y_7 = A]\,[y_8 = B]\,[y_9 = y_3].\overline{\mathbf{evid}_\mathbf{A}}(\mathbf{y_1}, \mathbf{y_6}).\mathbf{0})\\
SYS_p^{ZG} \triangleq\ & A_p \| B_p \| S
\end{aligned}$$

$$
\begin{array}{rl}
(PINPUT) & \langle \hat{s}, a(x).P \rangle \longrightarrow_p \langle \hat{s}.a(x), P \rangle \\
(POUTPUT) & \langle \hat{s}, \overline{a}M.P \rangle \longrightarrow_p \langle \hat{s}.\overline{a}M, P \rangle \\
(PDEC) & \langle \hat{s}, case\ \{M\}_L\ of\ \{x\}_{L'}\ in\ P \rangle \longrightarrow_p \langle \hat{s}\theta, P\theta \rangle \\
 & \qquad \theta = \mathtt{Mgu}(\{M\}_L, \{x\}_{\mathtt{Opp}(L')}) \\
(PPAIR) & \langle \hat{s}, let\ (x, y) = M\ in\ P \rangle \longrightarrow_p \langle \hat{s}\theta, P\theta \rangle \ \ \theta = \mathtt{Mgu}((x, y), M) \\
(PNEW) & \langle \hat{s}, (\mathtt{new}\ x)P \rangle \longrightarrow_p \langle \hat{s}, P\{y/x\} \rangle \quad y \notin f_v(P) \cup b_v(P) \\
(PRESTRICTION) & \langle \hat{s}, (\nu n)P \rangle \longrightarrow_p \langle \hat{s}, P\{m/n\} \rangle \ \ m \notin f_n(P) \\
(PMATCH) & \langle \hat{s}, [M = M']P \rangle \longrightarrow_p \langle \hat{s}\theta, P\theta \rangle \qquad \theta = \mathtt{Mgu}(M, M') \\
(PLSUM) & \langle \hat{s}, P + Q \rangle \longrightarrow_p \langle \hat{s}, P \rangle \\
(PLCOM) & \dfrac{\langle \hat{s}, P \rangle \longrightarrow_p \langle \hat{s}', P' \rangle}{\langle \hat{s}, P \| Q \rangle \longrightarrow_p \langle \hat{s}', P' \| Q' \rangle} \quad Q' = Q\theta \text{ if } \hat{s}' = \hat{s}\theta \text{ else } Q' = Q
\end{array}
$$

Fig. 4. Parametric transition rules

Characterization 1 (NRO in simplified ZG protocol). *Given the description with probing process of simplified ZG protocol, the NRO is satisfied, if*

$$
\langle \epsilon, SYS_p^{ZG} \rangle \models \overline{a1}\{B, x, y\}_{-\mathtt{k}[A]} \wedge \overline{a3}\{B, x, z\}_{-\mathtt{k}[A]} \hookleftarrow \overline{evid}_A(\{B, x, y\}_{-\mathtt{k}[A]}, \{A, B, x, z\}_{-\mathtt{k}[S]})
$$

Characterization 2 (NRR in simplified ZG protocol). *Given the description with probing process of simplified ZG protocol, the NRR is satisfied if*

$$
\langle \epsilon, SYS_p^{ZG} \rangle \models \overline{evid}_B(\{A, x, y\}_{-\mathtt{k}[B]}, \{A, B, x, z\}_{-\mathtt{k}[S]}) \hookrightarrow_F \overline{b2}\{A, x, y\}_{-\mathtt{k}[B]} \wedge \overline{s2}\{A, B, x, z\}_{-\mathtt{k}[S]}
$$

4 Parametric Simulation

All messages in concrete traces generated by transition rules in Fig. 3 are ground. In this section, parametric traces, in which irrelevant messages to a protocol execution are replaced with free variables, are presented.

4.1 Parametric Model

Definition 6 (Parametric trace and configuration). *A parametric trace $\hat{s}$ is a string of actions. A parametric configuration is a pair $\langle \hat{s}, P \rangle$, in which $\hat{s}$ is a parametric trace and P is a process.*

The transition relation of parametric configurations [3] is given by the rules listed in Fig. 4. Two symmetric forms $(PRSUM)$ of $(PLSUM)$, and $(PRCOM)$ of $(PLCOM)$ are omitted from the figure. A function $\mathtt{Mgu}(M_1, M_2)$ returns the *most general unifier* of M_1 and M_2.

Definition 7 (Concretization and abstraction). *Given a parametric trace $\hat{s}$, if there exists a substitution ϑ that assigns each parametric variable to a ground*

message, and which satisfies $s = \hat{s}\vartheta$*, where* s *is a concrete trace, we say that* s *is a concretization of* $\hat{s}$ *and* $\hat{s}$ *is an abstraction of* s*.* ϑ *is named a concretized substitution.*

According to the definition of parametric configurations, a concrete configuration $\langle \epsilon, P \rangle$ is also a parametric configuration. We name it an *initial configuration*. From an initial configuration, each concrete trace has an abstraction generated by parametric transition rules. On the other hand, if a parametric trace has a concretization, then the concretization is generated by concrete transition rules. Otherwise the parametric trace cannot be instantiated to any concrete trace.

Theorem 1. *(Soundness and completeness) Let* $\langle \epsilon, P \rangle$ *be an initial configuration, and let* s *be a concrete trace.* $\langle \epsilon, P \rangle$ *generates* s*, if and only if there exists* $\hat{s}$*, such that* $\langle \epsilon, P \rangle \longrightarrow_p^* \langle \hat{s}, P' \rangle$ *for some* P'*, and* s *is a concretization of* $\hat{s}$*.*

4.2 Satisfiable Normal Form

Theorem 1 shows that each concrete trace generated by an initial configuration has an abstraction. However, a parametric trace may or may not have concretizations.

Example 1. Consider a naive protocol, A sends a message $\{A, M\}_{K_{AB}}$ to B. There exists a parametric trace $b1(\{A, x\}_{\mathtt{k}[A,B]})$. Since $\mathtt{k}[A, B]$ was not leaked to the environment, before A or B sends an encrypted message protected by $\mathtt{k}[A, B]$, B cannot accept any message encrypted by $\mathtt{k}[A, B]$. Thus, the parametric trace $b1(\{A, x\}_{\mathtt{k}[A,B]})$ has no concretizations.

We name a message like $\{A, x\}_{\mathtt{k}[A,B]}$ a *rigid message*. A rigid message is the pattern of a requirement of an input action. The requirement can only be satisfied by the messages generated by a proper principal. If there are no appropriate messages satisfying the requirement, the parametric trace has no concretizations.

Definition 8 (Rigid message). *Given a parametric trace* $\hat{s}$*,* $\{N\}_L$ *in* M *is a rigid message if*

- M *is included in an input action such that* $\hat{s} = \hat{s}'.a(M).\hat{s}''$*, and*
  - *if* L *is a shared key or a private key, then* $\hat{s}' \not\vdash L$ *and* $\hat{s}' \not\vdash \{N\}_L$*;*
  - *if* L *is a public key, then there exists a rigid message, or at least one name or binder in* N*, which cannot be deduced by the* $\hat{s}'$*, and* $\hat{s}' \not\vdash \{N\}_L$*.*
- M *is included in an output action such that* $\hat{s} = \hat{s}'.\overline{a}\, M.\hat{s}''$*, and*
  - $\{N\}_L$ *satisfies the above three conditions, and*
  - L *is not known by the principal that contains the label* a*.*

A parametric trace with a rigid message needs to be further substituted by trying to unify the rigid message to the atomic messages in output actions of its prefix parametric trace. Such unification procedures will terminate because the number of atomic messages in the output actions of its prefix parametric trace is finite. We name these messages *elementary messages*, and use $el(\hat{s})$ to represent the set of elementary messages in $\hat{s}$.

Given a parametric trace $\hat{s}$ and a message N, we say N is $\hat{\rho}$*-unifiable* in $\hat{s}$, if there exists $N' \in el(\hat{s})$ such that $\hat{\rho} = \mathtt{Mgu}(N, N')$.

Definition 9 (Deductive relation). *Let $\hat{s}$ be a parametric trace such that $\hat{s} = \hat{s}_1.l(M).\hat{s}_2$, in which l is an input or an output label. If there exists a rigid message N in M such that $N \notin el(\hat{s}_1)$, and N is $\hat{\rho}$-unifiable in $\hat{s}_1$, then $\hat{s} \rightsquigarrow \hat{s}\hat{\rho}$.*

For two parametric traces $\hat{s}$ and $\hat{s}'$, if $\hat{s} \rightsquigarrow^* \hat{s}'$ and there is no $\hat{s}''$ that satisfies $\hat{s}' \rightsquigarrow \hat{s}''$, we name $\hat{s}'$ the *normal form* of $\hat{s}$. The set of normal forms of $\hat{s}$ is denoted by $\mathtt{nf}_{\rightsquigarrow}(\hat{s})$. "A parametric trace has concretizations" is equivalent to there exists a parametric trace in its $\mathtt{nf}_{\rightsquigarrow}(\hat{s})$ that has concretizations.

Lemma 1. *Let $\hat{s}$ be a parametric trace, and let $\hat{s}'$ be a normal form in $\mathtt{nf}_{\rightsquigarrow}(\hat{s})$. $\hat{s}'$ has a concretization, if and only if, for each decomposition $\hat{s}' = \hat{s}'_1.l(M).\hat{s}'_2$ in which l is either an input label or an output label, each rigid message N in M satisfies $N \in el(\hat{s}'_1)$.*

A satisfiable normal form is a normal form of $\hat{s}$ that satisfies the requirements in Lemma 1. $\mathtt{snf}_{\rightsquigarrow}(\hat{s})$ denotes the set of *satisfiable normal forms* of $\hat{s}$.

Theorem 2. *A parametric trace $\hat{s}$ has a concretization iff $\mathtt{snf}_{\rightsquigarrow}(\hat{s}) \neq \emptyset$.*

4.3 Simulation on a Parametric Model

Definition 10. *Let T be a state action term, and let $\hat{s}$ be a parametric trace that has concretizations. We say $\hat{s} \models_t T$, if for each concretization s of $\hat{s}$, $s \models_t T$.*

Definition 11. *Let σ be a path action term, and let $\langle \hat{s}, P\rangle$ be a parametric configuration, where $\hat{s}$ has concretizations. We say $\langle \hat{s}, P\rangle \models \sigma$, if for each concretization s of $\hat{s}$, where $s = \hat{s}\vartheta$, $\langle \hat{s}\vartheta, P\vartheta\rangle \models \sigma$.*

An action α is *$\hat{\rho}$-unifiable* in a parametric trace $\hat{s}$ if the parametric message in α can be unified to the message attached to the same label as α in $\hat{s}$, and $\hat{\rho}$ is the result of the unification.

Lemma 2. *Given a parametric trace $\hat{s}$,*

1. *$\hat{s} \models_t \alpha$ if and only if, α is $\hat{\rho}$-unifiable in $\hat{s}$, and for each satisfiable normal form in $\mathtt{snf}_{\rightsquigarrow}(\hat{s}\hat{\rho})$ satisfying $\hat{s}\hat{\rho}\hat{\rho}'$, $\alpha\hat{\rho}\hat{\rho}'$ occurs in $\hat{s}\hat{\rho}\hat{\rho}'$.*
2. *$\hat{s} \models_t \neg\alpha$ if and only if $\mathtt{snf}_{\rightsquigarrow}(\hat{s}\hat{\rho}) = \emptyset$ when α is $\hat{\rho}$-unifiable in $\hat{s}$.*
3. *For any state action term T, $\hat{s} \models_t T$ is decidable.*

Theorem 3. *Given an initial configuration $\langle \epsilon, P\rangle$,*

1. *Given a state action term T, $\langle \epsilon, P\rangle \models T$, if and only if for each parametric trace $\hat{s}$ generated by $\langle \epsilon, P\rangle$, $\hat{s} \models_t T$.*
2. *Given two state action terms T_1 and T_2, $\langle \epsilon, P\rangle \models T_2 \hookleftarrow T_1$, if and only if for each parametric trace $\hat{s}$ generated by $\langle \epsilon, P\rangle$, if T_1 is $\hat{\rho}$-unifiable in $\hat{s}$, then for each normal form in $\mathtt{snf}_{\rightsquigarrow}(\hat{s}\hat{\rho})$ satisfying $\hat{s}\hat{\rho}\hat{\rho}'$, $T_2\hat{\rho}\hat{\rho}'$ occurs before $T_1\hat{\rho}\hat{\rho}'$.*
3. *Given two state action terms T_1 and T_2, $\langle \epsilon, P\rangle \models T_1 \hookrightarrow_F T_2$, if and only if for each parametric configuration $\langle \hat{s}', P'\rangle$ reached by $\langle \epsilon, P\rangle$, if T_1 is $\hat{\rho}$-unifiable in $\hat{s}'$, then for each terminated parametric configuration $\langle \hat{s}''\hat{\rho}, P''\hat{\rho}\rangle$ reached by $\langle \hat{s}'\hat{\rho}, P'\hat{\rho}\rangle$, either $\hat{s}''\hat{\rho}$ cannot deduce any satisfiable normal forms, or $T_2\hat{\rho}\hat{\rho}'$ occurs in each satisfiable normal form $\hat{s}''\hat{\rho}\hat{\rho}'$ in $\mathtt{snf}_{\rightsquigarrow}(\hat{s}'\hat{\rho})$.*

Actually, Theorem 3 implicitly shows the algorithm to check whether a system satisfies a path action term.

5 Experimental Results

We implement the on-the-fly model checking method using Maude [6], since Maude can describe model generation rules by equational rewriting, instead of describing a model directly. Thus each property can be checked at the same time when a model is generated. It is named an on-the-fly model checking method.

Due to the space limitation, we have only explained the non-repudiation property. Fairness for fair non-repudiation protocols [17,9] that is classified into FAIRO, FAIRR, and FAIRM, is presented in the extended version [13].

In experiments with one session bound, the attacks for NRO, FAIRO and FAIRM of simplified ZG protocol were detected automatically. For comparison, we also implemented the analysis for the full ZG protocol, which guarantees those three properties [1]. We also tested some protocols proposed by the ISO [8].

The results are summarized in Fig. 5, in which the column "protocol spec." is the number of lines for a protocol specific part. In addition to these lines, each Maude file also contains about 400 lines for the common description.

protocols	property	protocol spec.	states	times(ms)	flaws
Simplified ZG protocol	NRO	50	513	3,954	detected
	NRR	50	780	3,905	secure
	FAIRO	55	770	2,961	detected
	FAIRR	55	846	3,903	secure
	FAIRM	50	4,109	45,545	detected
Full ZG protocol	NRO	50	633	7,399	secure
	FAIRO	55	788	3,394	secure
	FAIRM	60	788	3,490	secure
ISO/IEC13888-2 M2	NRO	50	1,350	7,710	detected
	FAIRO	65	1,977	6,827	detected
	FAIRR	65	2,131	7,506	secure
ISO/IEC13888-3 M-h	FAIRO	60	295	918	detected
	FAIRR	60	305	1,040	secure

Fig. 5. Experimental results

The experiments were carried out using Maude 2.2, and were performed on a Pentium 1.4 GHz, 1.5 GB memory PC, under Windows XP.

6 Related Work

Gavin Lowe first used trace analysis on process calculus CSP, and implemented a model-checker FDR to discover numerous attacks [14,15]. In his work, the intruder was represented as a recursive process. He restricted the state space to be finite by imposing upper-bounds upon messages generated by intruders, and also upon the number of principals in the network.

[1] For formal definitions of the properties of full ZG protocol, refer to [17].

Many of our ideas are inspired by Michele Boreale's symbolic approach [3]. In his research, he restricted the number of principals and intruders, and represented that each principal explicitly communicates with an intruder. Our model finitely describes an unlimited number of principals and intruders in the network.

David Basin et al. proposed an on-the-fly model checking method (OFMC) [4]. In their work, an intruder's messages are instantiated only when necessary, known as *lazy intruder*, which is similar to the use of a rigid message in our model. Unlike our method, an intruder's role is explicitly assigned. This is efficient, but the process needs to be performed several times to ensure that no intruders can attack a protocol in any roles.

Schneider proposed a trace analysis to prove non-repudiation and fairness properties of the ZG protocol based on CSP [17]. He used a deductive system to describe a dishonest principal and *failures* of a process to define these properties. We borrow the idea of the dishonest principal description from his research.

Jianying Zhou et al. proposed several non-repudiation protocols, and proved their correctness by SVO logic in their papers and book [19,20,21]. We use their definition for non-repudiation in this paper.

G. Bella and L. Paulson extended their previous Isabelle/HOL theorem proving approach for authentication property [2,16] to the ZG protocol, and proved the correctness of its non-repudiation and fairness properties [5]. The approach need not restrict the number of states to be finite, yet cannot be fully automated.

There were several studies based on game-theoretic model checking method on the fairness property. S. Kremer firstly analyzed several protocols, and also summarized and compared many formal definitions of fairness in his thesis [11]. Recently, D. Kähler et al. proposed a more powerful AMC-model checking method for verifying the fairness property [10].

V. Shmatikov et al. analyzed fairness of two contract signing protocols based on a finite-state model checker Murφ [18]. His model was limited to a bounded number of sessions and principals, and bounded number of messages that an intruder generates. We have released the bounds for principals and messages, using a parametric abstraction on an unlimited number of messages.

7 Conclusion

This paper proposed a sound and complete on-the-fly model checking method for fair non-repudiation protocols under the restriction of a bounded number of sessions. It extended our previous work [12] to handle all infinity factors of fair non-repudiation protocols. We implemented the method using Maude. It successfully detected the flaws of several examples automatically.

Our future work will be: First, to extend the method with pushdown model checking for recursive processes, so that a protocol with infinitely many sessions can be analyzed. Second, to check properties of other kinds of fair exchange protocols, such as digital contract signing protocols, and certified e-mail protocols.

Acknowledgements. The authors thank Dr. Yoshinobu Kawabe for discussions. This research is supported by the 21st Century COE "Verifiable and

Evolvable e-Society" of JAIST, funded by Japanese Ministry of Education, Culture, Sports, Science and Technology.

References

1. Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The Spi Calculus. In: Proceedings of the CCS 1997, pp. 36–47. ACM Press, New York (1997)
2. Bella, G.: Inductive Verification of Cryptographic Protocols. PhD thesis, University of Cambridge (2000)
3. Boreale, M.: Symbolic Trace Analysis of Cryptographic Protocols. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 667–681. Springer, Heidelberg (2001)
4. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A Symbolic Model Checker for Security Protocols. International J. of Information Security 4(3), 181–208 (2005)
5. Bella, G., Paulson, L.C.: A Proof of Non-repudiation. In: Christianson, B., Crispo, B., Malcolm, J.A., Roe, M. (eds.) SPW 2001. LNCS, vol. 2467, pp. 119–125. Springer-Verlag, Heidelberg (2002)
6. Clavel, M., Durán, F., Eker, S., Lincolnand, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (2005), http://maude.cs.uiuc.edu/maude2-manual/
7. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols. IEEE Trans. on Information Theory 29, 198–208 (1983)
8. ISO/IEC13888: Information Technology - Security Techniques - Non-repuduation - Part 1–Part 3 (1997)
9. Kremer, S., Markowitch, O., Zhou, J.: An Intensive Survey of Fair Non-repudiation Protocols. Computer Communications 25, 1606–1621 (2002)
10. Käehler, D., Küesters, R., Truderung, T.: Infinite State AMC-Model Checking for Cryptographic Protocols. In: Proceedings of the LICS 2007 (2007)
11. Kremer, S.: Formal Analysis of Optimistic Fair Exchange Protocols. PhD thesis, Universite Libre de Bruxelles (2003)
12. Li, G., Ogawa, M.: On-the-fly Model Checking of Security Protocols and Its Implementation by Maude. IPSJ Trans. on Programming 48 (SIG 10) 50–75 (2007)
13. Li, G., Ogawa, M.: On-the-fly Model Checking of Fair Non-repudiation Protocols (extended version) (2007) http://www.jaist.ac.jp/~guoqiang/Fullnon.pdf
14. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-key Using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
15. Lowe, G.: Some New Attacks upon Security Protocols. In: Proceedings of the CSFW 1996, pp. 162–169. IEEE Computer Society Press, Los Alamitos (1996)
16. Paulson, L.C.: The Inductive Approach to Verifying Cryptographic Protocols. J. of Computer Security 6, 85–128 (1998)
17. Schneider, S.: Formal Analysis of a Non-repudiation Protocol. In: Proceedings of the CSFW 1998, pp. 54–65. IEEE Computer Society Press, Los Alamitos (1998)
18. Shmatikov, V., Mitchell, J.C.: Finite-state Analysis of Two Contract Signing Protocols. Theoretical Computer Science 283, 419–450 (2002)
19. Zhou, J., Gollmann, D.: A Fair Non-repudiation Protocol. In: Proceedings of the S&P 1996, pp. 55–61. IEEE Computer Society Press, Los Alamitos (1996)
20. Zhou, J., Gollmann, D.: Towards Verification of Non-repudiation Protocols. In: Proceedings of 1998 International Refinement Workshop and Formal Methods Pacific, pp. 370–380. Springer, Heidelberg (1998)
21. Zhou, J.: Non-repudiation in Electronic Commerce. Artech House (2001)

Model Checking Bounded Prioritized Time Petri Nets

Bernard Berthomieu, Florent Peres, and François Vernadat

LAAS-CNRS, Université de Toulouse, Toulouse, France
fax: +33 (0)5.61.33.64.11; Tel.: +33 (0)5.61.33.63.63
{Bernard.Berthomieu,Florent.Peres,Francois.Vernadat}@laas.fr

Abstract. In a companion paper [BPV06], we investigated the expressiveness of Time Petri Nets extended with Priorities and showed that it is very close to that Timed Automata, in terms of weak timed bisimilarity. As a continuation of this work we investigate here the applicability of the available state space abstractions for Bounded Time Petri Nets to Bounded Prioritized Time Petri Nets. We show in particular that a slight extension of the "strong state classes" construction of [BV03] provides a convenient state space abstraction for these nets, preserving markings, states, and LTL formulas. Interestingly, and conversely to Timed Automata, the construction proposed does not require to compute polyhedra differences.

1 Introduction

Since their introduction in [Mer74], Time Petri nets (TPN for short) have been widely used for the specification and verification of systems in which time plays an essential role like communication protocols, hardware, or realtime systems.

TPNs extend Petri nets with temporal intervals associated with transitions, specifying firing delay ranges for the transitions. Assuming transition t became last enabled at time θ, and the end-points of its time interval are α and β, then t cannot fire earlier than time $\theta + \alpha$ and must fire no later than $\theta + \beta$, unless disabled by firing some other transition. Firing a transition takes no time.

Finite state space abstractions for bounded TPN's, preserving various classes of properties, can be computed in terms of so-called state classes [BM83] [BD91] [YR98] [BV03] [Had06]. State classes abstract sets of states by a marking and a polyhedron capturing temporal information. The polyhedra can be represented by difference systems, built and compared in polynomial time.

Though priorities are pervasive in some families of realtime systems, they are not supported by the Time Petri Net models, and cannot be generally encoded within. In a companion paper [BPV06] we proposed an extension of TPNs with Priorities: in a Prioritized TPN ($PrTPN$ for short) a transition is not allowed to fire if some transition with higher priority is firable at the same instant. We then proved that priorities strictly extend the expressiveness of Time Petri nets, and in particular that Bounded $PrTPN$s can be considered equivalent to Timed Automata, in terms of weak timed bisimularity.

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 523–532, 2007.
© Springer-Verlag Berlin Heidelberg 2007

As a continuation of this work, we investigate in this paper state space abstractions for Bounded Prioritized Time Petri Nets. We first explain why the classical "state classes" construction of [BM83] is unable to cope with priorities. Then, we show that a minor extension of the "strong state classes" construction of [BV03] (also called "state zones" by some authors) is a suitable state space abstraction, preserving the markings, states, and the formulas of linear time temporal logics of the state space of $PrTPN$s. A refinement of this construction also preserves the CTL properties of the state space. Interestingly, and conversely to the constructions proposed for model checking Prioritized Timed Automata [LHHC05], [DHLP06], the constructions required for $PrTPN$s preserve convexity of state classes; they do not require to compute expensive polyhedra differences.

The paper is organized as follows. Section 2 recalls the definition, semantics and main properties of Prioritized Time Petri Nets. Section 3 discusses their state space abstractions; not all available abstractions for TPNs can be extended to cope with priorities. The modeling power of $PrTPN$s and the capabilities of the proposed abstractions is illustrated in Section 4 by a simple scheduling example. Finally, related work and some side-issues are discussed in Section 5.

2 Time Petri Nets with Priorities

2.1 Definition

Let $\mathbf{I}^+$ be the set of non empty real intervals with non negative rational end-points. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left end-point, and $\uparrow i$ its right end-point, or ∞ if i is unbounded. For any $\theta \in \mathbb{R}^+$, let $i \dot{-} \theta = \{x - \theta | x \in i \wedge x \geq \theta\}$.

Definition 1. *[BPV06] A* Prioritized Time Petri Net *(PrTPN for short) is a tuple* $\langle \mathbf{P}, \mathbf{T}, \mathbf{Pre}, \mathbf{Post}, \succ, m_0, \mathbf{I}_s \rangle$ *in which:*

- $\langle \mathbf{P}, \mathbf{T}, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ *is a Petri net.* $\mathbf{P}$ *is the set of places,* $\mathbf{T}$ *is the set of transitions,* m_0 *is the initial marking, and* $\mathbf{Pre}$, $\mathbf{Post} : \mathbf{T} \rightarrow \mathbf{P} \rightarrow \mathbb{N}^+$ *are the precondition and postcondition functions, respectively.*
- $\mathbf{I_s} : \mathbf{T} \rightarrow \mathbf{I}^+$ *is the static interval function.*
- $\succ$ *is the priority relation, assumed irreflexive, asymmetric and transitive.*

TPNs extend PNs with the static Interval function $\mathbf{I}_s$, $PrTPN$s extend TPNs with the priority relation $\succ$ on transitions. Priorities will be represented by directed arcs between transitions, the source transition having higher priority.

For $f, g : \mathbf{P} \rightarrow \mathbf{N}^+$, $f \geq g$ means $(\forall p \in P)(f(p) \geq g(p))$ and $f\{+|-\}g$ maps $f(p)\{+|-\}g(p)$ with every p. A marking is a function $m : \mathbf{P} \rightarrow \mathbf{N}^+$. A transition $t \in \mathbf{T}$ is *enabled* at m iff $m \geq \mathbf{Pre}(t)$.

Definition 2. *A* state *of a TPN is a pair $s = (m, I)$ in which m is a marking and I is a function called the* interval *function. Function $I : \mathbf{T} \rightarrow \mathbf{I}^+$ associates a temporal interval with every transition enabled at m.*

The temporal components in states can be conveniently seen as firing domains, rather than interval functions: The *firing domain* of state (m, I) is the set of real vectors $\{\underline{\phi} | (\forall k)(\underline{\phi}_k \in I(k))\}$.

2.2 Semantics

Definition 3. *The semantics of a PrTPN* $\langle \mathbf{P}, \mathbf{T}, \mathbf{Pre}, \mathbf{Post}, \succ, m_0, \mathbf{I}_s \rangle$ *is the timed transition system* $\langle S, s_0, \rightsquigarrow \rangle$ *where:*

- S *is the set of states* (m, I) *of the PrTPN*
- $s_0 = (m_0, I_0)$ *is the initial state, where* m_0 *is the initial marking and* I_0 *is the static interval function* $\mathbf{I}_s$ *restricted to the transitions enabled at* m_0.
- $\rightsquigarrow \subseteq S \times \mathbf{T} \cup \mathbb{R}^+ \times S$ *is the state transition relation, defined as follows* ($(s, a, s') \in \rightsquigarrow$ *is written* $s \overset{a}{\rightsquigarrow} s'$):
  - *Discrete transitions: we have* $(m, I) \overset{t}{\rightsquigarrow} (m', I')$ *iff* $t \in \mathbf{T}$ *and:*
    1) $m \geq \mathbf{Pre}(t)$
    2) $0 \in I(t)$
    3) $(\forall t' \in \mathbf{T})(m \geq \mathbf{Pre}(t) \wedge (t' \succ t) \Rightarrow 0 \notin I(t'))$
    4) $(\forall k \in \mathbf{T})(m' \geq \mathbf{Pre}(k) \Rightarrow$
    $I'(k) = \textit{if } k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k) \textit{ then } I(k) \textit{ else } \mathbf{I}_s(k))$
  - *Continuous transitions: we have* $(m, I) \overset{\theta}{\rightsquigarrow} (m, I')$ *iff* $\theta \in \mathbb{R}^+$ *and:*
    5) $(\forall k \in \mathbf{T})(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
    6) $(\forall k \in \mathbf{T})(m \geq \mathbf{Pre}(k) \Rightarrow I'(k) = I(k) \dot{-} \theta)$

Transition t may fire from (m, I) if t is enabled at m, firable instantly, and no transition with higher priority satisfies these conditions. In the target state, the transitions that remained enabled while t fired (t excluded) retain their intervals, the others are associated with their static intervals. A continuous transition by θ is possible iff θ is not larger than the right endpoint of any enabled transition.

This definition differs from that used for Time Petri nets [BPV06] by the addition of condition (3), that prevents a transition to fire when some other transition also firable instantly has higher priority. Note that priorities do not modify the time-elapsing rules: all enabled transitions k are considered in condition (5), whether or not t has priority over k.

The *State Graph* of a $PrTPN$ is the timed transition system $SG = (S, s_0, \rightsquigarrow)$. From the properties of continuous transitions, any sequence of transitions of SG ending with a discrete transition is equivalent to a sequence alternating delay and discrete transitions, called a *firing schedule* or a *time transition sequence*.

Following [BM83], an alternative definition of the state space is often used for TPNs, capturing only the states reachable by some firing schedule. It amounts to interpret time-elapsing as nondeterminism. The *Discrete State Graph* of a $PrTPN$ is the triple $DSG = (S, s_0, \rightarrow)$, with $\rightarrow \subseteq S \times \mathbf{T} \times S$ and:

$$s \xrightarrow{t} s' \Leftrightarrow (\exists \theta)(\exists s'')(s \overset{\theta}{\rightsquigarrow} s'' \wedge s'' \overset{t}{\rightsquigarrow} s').$$

Any state of SG which is not in DSG is reachable from some state of DSG by a continuous transition. Both the SG and DSG are dense graphs: states may have uncountable numbers of successors.

A property of the state graphs of $PrTPN$s is worth to be mentioned: As in TPNs, but conversely to Timed Automata, at least one of the enabled transitions at a state is guaranteed to be firable after some delay. Time-elapsing may only increase the number of firable transitions and temporal deadlocks cannot happen.

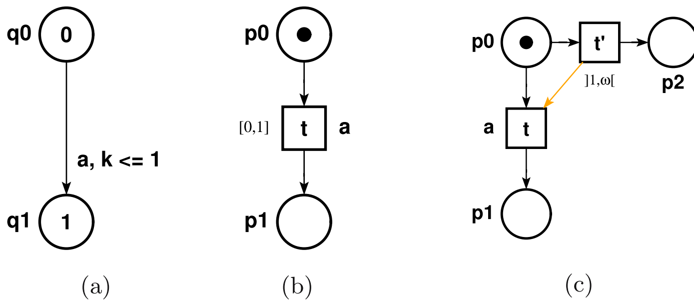
2.3 Properties

Boundedness: A PN is *bounded* if the marking of each place is bounded, boundedness implies finiteness of the set of reachable markings. Boundedness is undecidable for TPNs, and thus for $PrTPN$s, but there are a number of decidable and convenient sufficient conditions for this property [BM83], [BV07].

Composability: Adding to their definition a labeling of transitions by actions, a product operator can be defined for Prioritized Time Petri Nets. Under certain restrictions, this product is compositional in the sense that the semantics of a product of $PrTPN$s is equal to the product of their semantics [BPV06]. Compositional verification in presence of priorities has been investigated in length in a more general framework in [GS03] and other works by these authors.

Expressiveness: It was shown in [BCH+05] that bounded TPNs are equivalent to *Timed automata* (TAs for short) in terms of language acceptance, but that TAs are strictly more expressive in terms of weak timed bisimilarity. Adding priorities to TPNs preserves their expressiveness in terms of language acceptance, but strictly increases their expressiveness in terms of weak timed bisimilarity: it is shown in [BPV06] that any TA with invariants of the form $\wedge_i(k_i \leq c_i)$ is weak time bisimilar to some bounded $PrTPN$, and conversely.

To illustrate this result, let us consider the TA and nets in the figure below. As shown in [BCH+05], no TPN is weakly timed bisimilar with TA (a). In particular, the TPN (b) is not: when at location q_0 in the TA, time can elapse of an arbitrary amount, while time cannot progress beyond 1 in TPN (b). Consider now the $PrTPN$ (c), in which $t \prec t'$. Transition t' is silent and is firable at any time greater than 1, transition t bears label a and the default interval $[0, \infty[$. t may fire at any time not larger than 1, but not later, since t' is then firable and it has priority over t. Indeed, $PrTPN$ (c) is weakly timed bisimilar with TA (a).



(a) (b) (c)

3 State Space Abstractions for *PrTPN*s

The state graphs of $PrTPN$s are generally infinite, even when bounded. To model check $PrTPN$s, one needs finite abstractions of their state graphs. If G is

some state space, A some abstraction of it, and f a logical formula of the family one wish to preserve, then we must have $G \models f$ iff $A \models f$. Traditionally, we will focus on abstractions of the DSG rather than the SG.

For Time Petri nets, state space abstractions are available that preserve markings (including deadlocks) [BV07], markings and all traces [BM83], [BD91], states [BH06], states and traces [BV03], [Had06], and states, traces and branching properties [YR98], [BV03], [BH06]. We investigate in this section extensions of these abstractions to Prioritized Time Petri Nets, when applicable.

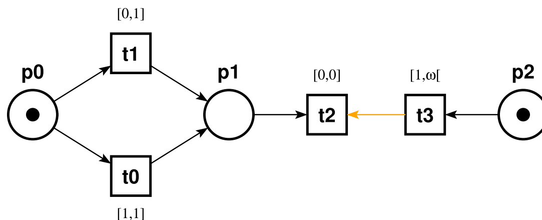
3.1 Abstractions Preserving Markings and Traces

Definition 4. *For each firable sequence $\sigma \in \mathbf{T}^*$, let C_σ be the set of states inductively defined by: $C_\epsilon = \{s_0\}$ and $C_{\sigma.t} = \{s' | (\exists s \in C_\sigma)(s \xrightarrow{t} s')\}$*

C_σ the set of states reached in the discrete state graph by firing schedules of support sequence σ. Each state of the DSG is in some C_σ. For each such set C_σ, let $\mathcal{M}(C_\sigma)$ be the marking of any state in C_σ (all states in C_σ bear the same marking), and $\mathcal{F}(C_\sigma)$ be the union of the firing domains of all states in C_σ. Finally, let $C_\sigma \cong C_{\sigma'}$ iff $\mathcal{M}(C_\sigma) = \mathcal{M}(C_{\sigma'})$ and $\mathcal{F}(C_\sigma) = \mathcal{F}(C_{\sigma'})$.

The classical state class construction of [BM83], [BD91], termed SCG in the sequel, stems from the observation that, if $C_\sigma \cong C_{\sigma'}$, then any firing schedule firable from some state in C_σ is firable from state in $C_{\sigma'}$, and conversely. The state classes of [BM83] denote the above sets C_σ, for all firable sequences σ, considered modulo equivalence $\cong$. The set of classes is equipped with a transition relation: $C_\sigma \xrightarrow{t} X \Leftrightarrow C_{\sigma.t} \cong X$, it is finite iff the net is bounded.

The SCG is a convenient abstraction for LTL model checking, it preserves the markings and traces of the net. A weaker abstraction, only preserving markings, is obtained from the SCG by merging the classes related by inclusion of their firing domains. Unfortunately, both these abstractions are too coarse to preserve the effects of priorities. In fact, the founding observation that sets of states equivalent by $\cong$ have same futures in terms of firing schedules simply does not hold in presence of priorities, as illustrated by the following example.



Firing t_0 or t_1 in the above net leads to the same SCG class. Now, because t_3 has higher priority than t_2 and remains enabled while t_0 or t_1 fires, t_2 can never fire after t_0, and may only fire after t_1 if t_1 fired earlier than time 1.

3.2 Abstractions Preserving States and Traces

The previous SCG construction considers the sets C_σ in Definition 4 modulo equivalence $\cong$. In contrast, the *strong state classes* construction ($SSCG$ for short) proposed in [BV03], also called *state zones* by some authors, exactly coincides with those sets C_σ. For building the $SSCG$, one first needs a canonical representation for the sets C_σ, clock domains serve this purpose.

Clock domains: With each reachable state, one may associate a *clock function* γ. Function γ associates with each transition enabled at the state the time elapsed since it was last enabled. Clock functions may also be seen as vectors $\underline{\gamma}$ indexed over the transitions enabled.

In the $SSCG$ construction, a class is represented by a marking and a clock system, but classes still denote sets of states as in Definition 2 (defined from interval functions). The set of states denoted by a marking m and a clock system $Q = \{G\underline{\gamma} \leq \underline{g}\}$ is the set $\{(m, \Phi(\underline{\gamma}))|\underline{\gamma} \in \langle Q \rangle\}$, where $\langle Q \rangle$ is the solution set of Q and firing domain $\Phi(\underline{\gamma})$ is the solution set in $\underline{\phi}$ of:

$$\underline{0} \leq \underline{\phi}\ ,\ \underline{e} \leq \underline{\phi} + \underline{\gamma} \leq \underline{l} \ \text{ where } \ \underline{e}_k = \downarrow \mathbf{I}_s(k) \text{ and } \underline{l}_k = \uparrow \mathbf{I}_s(k)$$

Each clock vector denotes a state, but, unless the static intervals of all transitions are bounded, different clock vectors may denote the same state, and clock systems with different solution sets (possibly an infinity) may denote the same set of states. For this reason we introduce equivalence $\equiv$:

Definition 5. *Given $c = (m, Q = \{G\underline{\gamma} \leq \underline{g}\})$ and $c' = (m', Q' = \{G'\underline{\gamma}' \leq \underline{g}'\})$, let $c \equiv c'$ iff $m = m'$ and clock systems Q and Q' denote the same sets of states.*

Equivalence $\equiv$ is clearly decidable, efficient methods for checking it are discussed e.g. in [BV03] and [Had06], relying upon some relaxations of clock systems. When the static intervals of all transitions are bounded, $\equiv$ reduces to equality of the solution sets of the clock systems.

Construction of the $SSCG$: Strong state classes are represented by a marking m and a system $Q = \{G\underline{\gamma} \leq \underline{g}\}$ describing a clock domain for the enabled transitions. Clock variable $\underline{\gamma}_i$ is associated with the ith transition enabled at m.

The first step in the computation of a successor class, when building the $SSCG$, is to determine which transitions are firable from the current class. In absence of priorities, transition t is firable from some state in class (m, Q) iff there is some delay $\theta \in \mathbb{R}^+$ such that, augmented by θ, the clocks of all enabled transitions are not larger than the right endpoint of their respective static intervals, and within that interval for transition t:

(a1) $\theta \geq 0$
(a2) $\theta + \underline{\gamma}_t \in \mathbf{I}_s(t)$
(a3) $(\forall i \neq t)(m \geq \mathbf{Pre}(i) \Rightarrow \theta + \underline{\gamma}_i \leq \uparrow \mathbf{I}_s(i))$

In presence of priorities, a fourth condition must be added, asserting that no enabled transition with higher priority than t is firable at θ:

(a4) $(\forall i)(m \geq \mathbf{Pre}(i) \wedge i \succ t \Rightarrow \theta + \underline{\gamma}_i \notin \mathbf{I}_s(i))$

$\theta + \underline{\gamma}_i \notin \mathbf{I}_s(i)$ holds iff $\theta + \underline{\gamma}_i > \uparrow\mathbf{I}_s(t)$ or $\theta + \underline{\gamma}_i < \downarrow\mathbf{I}_s(t)$, but the former case may not happen since it contradicts condition (a3). The $SSCG$ for a $PrTPN$ is built as for a TPN, just augmenting the enabling conditions as explained:

Algorithm 1. Computing the Strong State Class Graph with Priorities

---

- $R_\epsilon = (m_0, \{0 \leq \underline{\gamma}_t \leq 0 \mid m_0 \geq \mathbf{Pre}(t)\})$
- If σ is firable and $R_\sigma = (m, Q)$ then $\sigma.t$ is firable iff
  - (i) $m \geq \mathbf{Pre}(t)$
  - (ii) Q augmented with
    $\theta \geq 0\ ,\ \ \theta \geq \downarrow\mathbf{I}_s(t) - \underline{\gamma}_t$
    $\{\theta \leq \uparrow\mathbf{I}_s(i) - \underline{\gamma}_i \mid m \geq \mathbf{Pre}(i)\}$
    $\{\theta < \downarrow\mathbf{I}_s(j) - \underline{\gamma}_j \mid m \geq \mathbf{Pre}(j) \wedge j \succ t\}$
    is consistent
- If $\sigma.t$ is firable then $R_{\sigma.t} = (m', Q')$ is computed from $R_\sigma = (m, Q)$:
  m' = m - **Pre**(t) + **Post**(t)
  Q' obtained by :
  - (a) A new variable is introduced, θ, constrained by (ii) above;
  - (b) $\forall k \in \mathbf{T} : m' \geq \mathbf{Pre}(k)$, introduce a new variable $\underline{\gamma'}_k$, such that:
    $\underline{\gamma'}_k = \underline{\gamma}_k + \theta$ if $k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k)$
    $0 \leq \underline{\gamma'}_k \leq 0$ otherwise
  - (c) The variables $\underline{\gamma}$ and θ are eliminated.

The temporary variable θ stands for the possible delays after which t can fire. There is an arc labeled t between R_σ and c iff $c \equiv R_{\sigma.t}$. As for Time Petri Nets, the $SSCG$ of a $PrTPN$ is finite iff the net is bounded. The clock systems are difference systems, for which canonical forms can be computed here in time complexity $\mathcal{O}(n^2)$, following the observation of [Rok93] that they can be computed incrementally. The $SSCG$ preserves all traces of the net and also permits to decide state reachability [BV07].

Compared to the methods proposed for model checking Prioritized Timed Automata [LHHC05] [DHLP06], Algorithm 1 does not require the expensive ($\mathcal{O}(n^4)$) DBM subtractions mandatory there. As was explained, this follows from the fact that time-elapsing in $PrTPN$s is bounded by the smallest of the deadlines of the enabled transitions, whatever the priority relation.

3.3 Abstractions Preserving Branching Properties

The branching properties are those one can express in branching time temporal logics like CTL, or modal logics like HML or the μ-calculus. Neither the SCG (for TPNs) nor the $SSCG$ (for TPNs or $PrTPN$s) preserve these properties.

An abstraction preserving branching properties of the DSG was first proposed in [YR98], called the *Atomic state class graph* ($ASCG$ for short), for the subclass of TPNs in which all transitions have bounded static intervals. An alternative construction was proposed in [BV03], in which the $ASCG$ is obtained from the $SSCG$ of the net by a partition refinement process. This construction remains applicable to $PrTPN$s, it can be summarized as follows.

Bisimilarity is known to preserve branching properties. Let $\rightarrow$ be a binary relation over a finite set U, and for any $S \subseteq U$, let $S^{-1} = \{x | (\exists y \in S)(x \rightarrow y)\}$. A partition P of U is *stable* if, for any pair (A, B) of blocks of P, either $A \subseteq B^{-1}$ or $A \cap B^{-1} = \emptyset$. Computing a bisimulation, starting from an initial partition P of states, is computing a stable refinement of P [ACH+92]. In our case, a suitable initial partition of the state space is the set of classes of the $SSCG$ (it is a cover rather than a partition, but the method remains applicable). Computing the $ASCG$ is computing a stable refinement of the $SSCG$, the technical details can be found in [BV03], [BV07].

4 An Example: Rate Monotonic Scheduling

This example illustrates the use of priorities for expressing scheduling policies on systems of tasks. The system is made of three realtime tasks, to be scheduled by a rate monotonic policy, the corresponding $PrTPN$ is shown in Figure 1.

Task i has the following transitions:

- $\mathbf{P}_i$ Periodically generates a token in place newP_i, representing a new period event from which a task will be released,
- $\mathbf{R}_i$ Marks the task release, i.e. the instant at which the task enter its idle state and waits to be started. Updates its state from done_i to $\mathrm{notdone}_i$,
- $\mathbf{S}_i$ Starts the task, changes its state from idle_i to exec_i,
- $\mathbf{E}_i$ Completes the task: frees the resource and restores its state to done_i,
- $\mathbf{DL}_i$ Signals a deadline miss, occuring when the task is still executing and a new period event is generated.

A scheduler in a realtime system is often coded by a *policy*, which tells, when the unpoliticized system is nondeterministic, how to resolve this nondeterminism. The rate monotonic policy gives priorities to tasks according to their period value, the task with smaller period having the highest priority. The policy is expressed here using priorities. In this example, the upper task has the smallest period (3), followed by the middle task (5), and the lower task (11).

Thus, the RM policy requires here that every transition of T1 that can lead to a state enabling S_i in zero time has to possess a higher priority than all the corresponding transitions in T2, and similarly for T2 and T3. That is $P_1, R_1, S_1 \succ P_2, R_2, S_2 \succ P_3, R_3, S_3$ (to preserve readability, priorities are omitted in Figure 1). This priority relation represents exactly the RM policy.

Algorithm 1 has been implemented and integrated in an experimental version of the $Tina$ toolbox [BRV04]. For this example, the unprioritized $SSCG$, computed by $Tina$ for the net with priorities removed, is unbounded. Taking

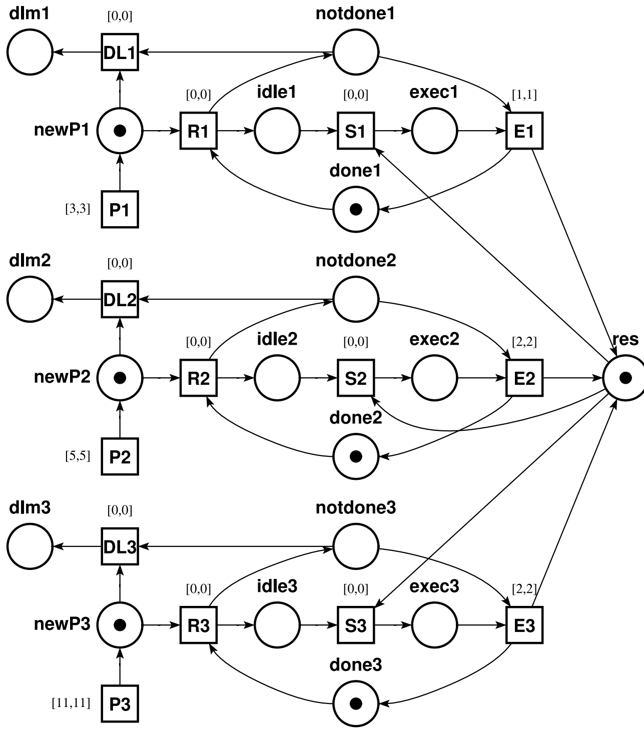


Fig. 1. A task system $(P_1, R_1, S_1 \succ P_2, R_2, S_2 \succ P_3, R_3, S_3)$

priorities into account, the tool builds a finite $SSCG$ with 557 state classes and 671 transitions. All markings are safe (the places hold at most one token) and it can be checked on the graph produced that no deadline miss can occur (transitions DLi are dead).

5 Conclusion

The results presented in this paper increase the range of systems one can represent and analyze using Time Petri nets and similar models. Beside their modeling convenience, priorities do extend the expressiveness of Time Petri nets. Further, it has been shown that Prioritized Petri nets can be analyzed with methods similar to the well known state class based methods for Time Petri nets. These methods are easy to implement and have a tractable complexity.

As mentioned in the text, analyzing Prioritized Time Petri nets does not require to use polyhedra differences, adding priorities to TPNs does not augment the complexity of computation of classes.

The version of Prioritized TPNs introduced in this paper makes use of the simplest possible notion of priority: static priorities. Technically, nothing

prevents to replace it by more flexible notions of priorities like dynamic priorities depending on markings.

References

[ACH+92] Alur, R., Courcoubetis, C., Halbwachs, N., Dill, D.L., Wong-Toi, H.: Minimization of timed transition systems. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 340–354. Springer, Heidelberg (1992)

[BCH+05] Bérard, B., Cassez, F., Haddad, S., Roux, O.H., Lime, D.: Comparison of the Expressiveness of Timed Automata and Time Petri Nets. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 211–225. Springer, Heidelberg (2005)

[BD91] Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Tr. on Soft. Eng. 17(3), 259–273 (1991)

[BH06] Boucheneb, H., Hadjidj, R.: Using inclusion abstraction to construct atomic state class graphs for time petri nets. International Journal of Embedded Systems 2(1/2) (June 2006)

[BM83] Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. IFIP Congress Series 9, 41–46 (1983)

[BPV06] Berthomieu, B., Peres, F., Vernadat, F.: Bridging the gap between timed automata and bounded time petri nets. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 82–97. Springer, Heidelberg (2006)

[BRV04] Berthomieu, B., Ribet, P.-O., Vernadat, F.: The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research 42(14), 2741–2756 (2004)

[BV03] Berthomieu, B., Vernadat, F.: State class constructions for branching analysis of time Petri nets. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 442–457. Springer, Heidelberg (2003)

[BV07] Berthomieu, B., Vernadat, F.: State Space Abstractions for Time Petri Nets. In: Lee, I., Leung, J.Y.T., Son, S. (eds.) Handbook of Real-Time and Embedded Systems, CRC Press, Boca Raton (2007)

[DHLP06] David, A., Håkansson, J., Larsen, K.G., Pettersson, P.: Model checking timed automata with priorities using DBM subtraction. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 128–142. Springer, Heidelberg (2006)

[GS03] Goessler, G., Sifakis, J.: Priority systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 314–329. Springer, Heidelberg (2004)

[Had06] Hadjidj, R.: Analyse et validation formelle des systèmes temps réel. PhD Thesis, Ecole Polytechnique de Montréal, Univ. de Montréal (February 2006)

[LHHC05] Lin, S.-W., Hsiung, P.-A., Huang, C.-H., Chen, Y.-R.: Model checking prioritized timed automata. In: Peled, D.A., Tsay, Y.K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 370–384. Springer, Heidelberg (2005)

[Mer74] Merlin, P.M.: A Study of the Recoverability of Computing Systems. PhD Thesis, Irvine (1974)

[Rok93] Rokicki, T.G.: Representing and Modeling Circuits. PhD Thesis, Stanford Univ. Stanford, CA (1993)

[YR98] Yoneda, T., Ryuba, H.: CTL model checking of Time Petri nets using geometric regions. IEEE Trans. on Inf. and Systems E99-D(3), 1–10 (1998)

Using Patterns and Composite Propositions to Automate the Generation of LTL Specifications

Salamah Salamah, Ann Q. Gates, Vladik Kreinovich, and Steve Roach

Dept. of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA

Abstract. Property classifications and patterns, i.e., high-level abstractions that describe common behavior, have been used to assist practitioners in generating formal specifications that can be used in formal verification techniques. The Specification Pattern System (SPS) provides descriptions of a collection of patterns. Each pattern is associated with a scope that defines the extent of program execution over which a property pattern is considered. Based on a selected pattern, SPS provides a specification for each type of scope in multiple formal languages including Linear Temporal Logic (LTL). The (Prospec) tool extends SPS by introducing the notion of Composite Propositions (CP), which are classifications for defining sequential and concurrent behavior to represent pattern and scope parameters.

In this work, we provide definitions of patterns and scopes when defined using CP classes. In addition, we provide general (template) LTL formulas that can be used to generate LTL specifications for all combinations of pattern, scope, and CP classes.

1 Introduction

Although the use of formal verification techniques such as model checking [4] and runtime monitoring [8] improve the dependability of programs, they are not widely adapted in standard software development practices. One reason for the hesitance in using formal verification is the high level of mathematical sophistication required for reading and writing the formal specifications required for the use of these techniques [3].

Different approaches and tools such as the Specification Pattern System (SPS) [2] and the Property Specification Tool (Prospec) [5] have been designed to provide assistance to practitioners in generating formal specifications. Such tools and approaches support the generation of formal specifications in multiple formalizations. The notions of patterns, scopes, and composite propositions (CP) have been identified as ways to assist users in defining formal properties. Patterns capture the expertise of developers by describing solutions to recurrent problems. Scopes on the other hand, allow the user to define the portion of execution where a pattern is to hold.

The aforementioned tools take the user's specifications and provide formal specifications that matches the selected pattern and scope in multiple formalizations. SPS for example provides specifications in Linear Temporal Logic (LTL)

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 533–542, 2007.
© Springer-Verlag Berlin Heidelberg 2007

and computational Tree Logic (CTL) among others. On the other hand, Prospec provides specifications in Future Interval Logic (FIL) and Meta-Event Definition Language (MEDL). These tools however, do not support the generation of specifications that use CP in LTL. The importance of LTL stems from its expressive power and the fact that it is widely used in multiple formal verification tools. This work provides a set of template LTL formulas that can be used to specify a wide range of properties in LTL.

The paper is organized as follows: Section 2 provides the background related information including description of LTL and the work that has been done to support the generation of formal specifications. Section 3 highlights the problems of generating formal specifications in LTL. Sections 4 and 5 provide the general formal definitions of patterns and scopes that use CP. Section 6 motivates the need for three new LTL operators to simplify the specifications of complex LTL formulas. Last, the general LTL template formulas for the different scopes are described followed by summary and future work.

2 Background

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a prominent formal specification language that is highly expressive and widely used in formal verification tools such as the model checkers SPIN [4] and NUSMV [1]. LTL is also used in the runtime verification of Java programs [8].

Formulas in LTL are constructed from elementary propositions and the usual Boolean operators for *not*, *and*, *or*, *imply* (*neg*, $\wedge$, $\vee$, $\rightarrow$, respectively). In addition, LTL allows for the use of the temporal operators *next* (X), *eventually* ($\diamond$), *always* ($\Box$), *until*, (U), *weak until* (W), and *release* (R). In this work, we only use the first four of these operators. These formulas assume discrete time, i.e., states $s = 0, 1, 2, \ldots$ The meaning of the temporal operators is straightforward. The formula XP holds at state s if P holds at the next state $s + 1$. $P\,U\,Q$ is true at state s, if there is a state $s' \geq s$ at which Q is true and, if s' is such a state, then P is true at all states s_i for which $s \leq s_i < s'$. The formula $\diamond P$ is true at state s if P is true at some state $s' \geq s$. Finally, the formula $\Box P$ holds at state s if P is true at all moments of time $s' \geq s$. Detailed description of LTL is provided by Manna et al. [6].

2.2 Specification Pattern System (SPS)

Writing formal specification, particularly those involving time, is difficult. The Specification Pattern System [2] provides patterns and scopes to assist the practitioner in formally specifying software properties. These patterns and scopes were defined after analyzing a wide range of properties from multiple industrial domains (i.e., security protocols, application software, and hardware systems). *Patterns* capture the expertise of developers by describing solutions to recurrent problems. Each pattern describes the structure of specific behavior and

Table 1. Description of CP Classes in LTL

CP Class	**LTL Description** (P^{LTL})
$AtLeastOne_C$	$p_1 \vee \ldots \vee p_n$
$AtLeastOne_E$	$(\neg p_1 \wedge \ldots \wedge \neg p_n) \wedge ((\neg p_1 \wedge \ldots \wedge \neg p_n)\, U\, (p_1 \vee \ldots \vee p_n))$
$Parallel_C$	$p_1 \wedge \ldots \wedge p_n$
$Parallel_E$	$(\neg p_1 \wedge \ldots \wedge \neg p_n) \wedge ((\neg p_1 \wedge \ldots \wedge \neg p_n)\, U\, (p_1 \wedge \ldots \wedge p_n))$
$Consecutive_C$	$(p_1 \wedge X(p_2 \wedge (\ldots (\wedge X p_n)) \ldots))$
$Consecutive_E$	$(\neg p_1 \wedge \ldots \wedge \neg p_n) \wedge ((\neg p_1 \wedge \ldots \wedge \neg p_n)\, U\, (p_1 \wedge \neg p_2 \wedge \ldots \wedge \neg p_n \wedge X(p_2 \wedge \neg p_3 \wedge \ldots \wedge \neg p_n \wedge X(\ldots \wedge X(p_{n-1} \wedge \neg p_n \wedge X p_n)) \ldots))$
$Eventual_C$	$(p_1 \wedge X(\neg p_2\, U\, (p_2 \wedge X(\ldots \wedge X(\neg p_{n-1}\, U\, (p_{n-1} \wedge X(\neg p_n\, U\, p_n)))) \ldots))))$
$Eventual_E$	$(\neg p_1 \wedge \ldots \wedge \neg p_n) \wedge ((\neg p_1 \wedge \ldots \wedge \neg p_n)\, U\, (p_1 \wedge \neg p_2 \wedge \ldots \wedge \neg p_n \wedge ((\neg p_2 \wedge \ldots \wedge \neg p_n)\, U\, (p_2 \wedge \neg p_3 \wedge \ldots \wedge \neg p_n \wedge (\ldots \wedge (p_{n-1} \wedge \neg p_n \wedge (\neg p_n\, U\, p_n)) \ldots)))))$

defines the pattern's relationship with other patterns. Patterns are associated with scopes that define the portion of program execution over which the property holds.

The main patterns defined by SPS are: *Universality*, *Absence*, *Existence*, *Precedence*, and *Response*. In SPS, each pattern is associated with a *scope* that defines the extent of program execution over which a property pattern is considered. There are five types of scopes defined in SPS: *Global*, *Before R*, *After L*, *Between L And R*, and *After L Until R*. A detailed description of these patterns and scopes can be found in Dewyer [2].

2.3 Composite Propositions (CP)

The idea of CP was introduced by Mondragon et al. [5] to allow for patterns and scopes to be defined using multiple propositions. In practical applications, we often need to describe properties where one or more of the pattern or scope parameters are made of multiple propositions, i.e., composite propositions (CP). For example, the property that every time data is sent at state s_i the data is read at state $s_1 \geq s_i$, the data is processed at state s_2, and data is stored at state s_3, can be described using the $Existence(P)$ pattern within the *Between L and R* scope. In this example L stands for "data is sent", R stands for 'date is stored' and P is composed of p_1 and p_2 (data is read and data is processed, respectively).

To describe such patterns, Mondragon et al. [5] extended SPS by introducing a classification for defining sequential and concurrent behavior to describe pattern and scope parameters. Specifically, the work formally described several types of CP classes and provided formal descriptions of these CP classes in LTL. Mondragon et al defined eight CP classes and described their semantics using LTL. The eight CP classes defined by that work are $AtLeastOne_C$, $AtLeastOne_E$, $Parallel_C$, $Parallel_E$, $Consecutive_C$, $Consecutive_E$, $Eventual_C$, and $Eventual_E$. The subscripts C and E describe whether the propositions in the CP class are asserted as Conditions or Events

respectively. A proposition defined as a condition holds in one or more consecutive states. A proposition defined as event means that there is an instant at which the proposition changes value in two consecutive states.

This work modified the LTL description of the CP classes $AtLeastOne_E$, $Eventual_C$, and $Eventual_E$. The work changed the the semantics of the $AtLeastOne_E$ class to one that is more consistent with the other CP classes of type E. The LTL description of the other two CP classes were modified to a semantically equivalent LTL formulas. Table 1. provides the semantics of the CP classes used in this paper in LTL.

3 Problem with Direct Substitution

Although SPS provides LTL formulas for basic patterns and scopes (ones that use single, "atomic", propositions to define L, R, P, and Q) and Mondragon et al. provided LTL semantics for the CP classes as described in Table 1., in most cases it is not adequate to simply substitute the LTL description of the CP class into the basic LTL formula for the pattern and scope combination. Consider the following property: "The delete button is enabled in the main window only if the user is logged in as administrator and the main window is invoked by selecting it from the Admin menu.". This property can be described using the $Existence(Eventual_C(p_1, p_2))\ Before(r)$ where p_1 is "the user logged in as an admin", p_2 is "the main window is invoked", and r is "the delete button is enabled". As mentioned above, the LTL formula for the $Existence(P)\,Before(R)$ is "$(\Box\neg R) \vee (\neg R\, U\, (P \wedge \neg R))$", and the LTL formula for the CP class $Eventual_C$, as described in Table 1, is $(p_1 \wedge X(\neg p_2\, U\, p_2))$. By replacing P by $(p_1 \wedge X(\neg p_2\, U\, p_2))$ in the formula for the pattern and scope, we get the formula: "$(\Box\neg R) \vee (\neg R\, U\, ((p_1 \wedge X(\neg p_2\, U\, p_2)) \wedge \neg R))$" This formula however, asserts that either R never holds or R holds after the formula $(p_1 \wedge X(\neg p_2\, U\, p_2))$ becomes true. In other words, the formula asserts that it is an acceptable behavior if R ("the delete button is enabled") holds after p_1 ("the user logged in as an admin") holds and before p_2 ("the main window is invoked") holds, which should not be an acceptable behavior.

As seen by the above example, the temporal nature of LTL and its operators means that direct substitution could lead to the description of behaviors that do not match the actual intent of the specifier. For this reason, it is necessary to provide abstract LTL formulas that can be used as templates for the generation of LTL specifications for all patterns, scopes, and CP classes combinations, which is the goal of this paper.

4 Patterns Defined with Composite Propositions

As we mentioned in Section 2.2, Dwyer et al. defined the notions of patterns and scopes to assist in the definition of formal specifications. Patterns provide common solutions to recurring problems, and scopes define the extent of program execution where the pattern is evaluated. In this work we are concerned with

the following patterns: the absence of P, the existence of P, Q precedes P, Q strictly precedes P, and Q responds to P.

Note that the strict precedence pattern was defined by Mondragon et al. [5], and it represents a modification of the precedence pattern as defined by Dwyer et al. The following subsections describe these patterns when defined using single and composite propositions.

The absence of P means that the (single or composite) property P never holds, i.e., for every state s, P does not hold at s. In the case of CP classes, this simply means that P^{LTL} (as defined in Table 1 for each CP class) is never true. The LTL formula corresponding to the absence of P is:

$$\Box\neg P^{LTL}$$

The existence of P means that the (single or composite) property P holds at some state s in the computation. In the case of CP classes, this simply means that P^{LTL} is true at some state of the computation. The LTL formula corresponding to the existence of P is:

$$\Diamond P^{LTL}$$

For single proposition, the meaning of "precedes", "strictly precedes", and "responds" is straightforward. To extend the meanings of these patterns to ones defined using CP, we need to explain what "after" and "before" mean for the case of CP. While single propositions are evaluated in a single state, CP, in general, deal with a sequence of states or a time interval (this time interval may be degenerate, i.e., it may consist of a single state). Specifically, for every CP $P = T(p_1, \ldots, p_n)$, there is a beginning state b_P – the first state in which one of the propositions p_i becomes true, and an ending state e_P – the first state in which the condition T is fulfilled. For example, for $Consecutive_C$, the ending state is the state $s+(n-1)$ when the last statement p_n holds; for $AtLeastOne_C$, the ending state is the same as the beginning state – it is the first state when one of the propositions p_i holds for the first time.

For each state s and for each CP $P = T(p_1 \ldots, p_n)$ that holds at this state s, we will define the beginning state $b_P(s)$ and the ending state $e_P(s)$. The following is a description of b_P and e_P for the CP classes of types condition and event defined in Table 1 (to simplify notations, wherever it does not cause confusion, we will skip the state s and simply write b_P and e_P):

- For the CP class $P = AtLeastOne_C(p_1, \ldots, p_n)$ that holds at state s, we take $b_P(s) = e_P(s) = s$.
- For the CP class $P = AtLeastOne_E(p_1, \ldots, p_n)$ that holds at state s, we take, as $e_P(s)$, the first state $s' > s$ at which one of the propositions p_i becomes true and we take $b_P(s) = (e_P(s) - 1)$.
- For the CP class $P = Parallel_C(p_1, \ldots, p_n)$ that holds at state s, we take $b_P(s) = e_P(s) = s$.
- For the CP class $P = Parallel_E(p_1, \ldots, p_n)$ that holds at state s, we take, as $e_P(s)$, the first state $s' > s$ at which all the propositions p_i become true and we take $b_P(s) = (e_P(s) - 1)$.

- For the CP class $P = Consecutive_C(p_1, \ldots, p_n)$ that holds at state s, we take $b_P(s) = s$ and $e_P(s) = s + (n-1)$.
- For the CP class $P = Consecutive_E(p_1, \ldots, p_n)$ that holds at state s, we take, as $b_P(s)$, the last state $s' > s$ at which all the propositions were false and in the next state the proposition p_1 becomes true, and we take $e_P(s) = s' + (n)$.
- For the CP class $P = Eventual_C(p_1, \ldots, p_n)$ that holds at state s, we take $b_P(s) = s$, and as $e_P(s)$, we take the first state $s_n > s$ in which the last proposition p_n is true and the previous propositions $p_2, \ldots, p_{n-1}$ were true at the corresponding states $s_2, \ldots, s_{n-1}$ for which $s < s_2 < \ldots < s_{n-1} < s_n$.
- For the CP class $P = Eventual_E(p_1, \ldots, p_n)$ that holds at state s, we take as $b_P(s)$, the last state state s_1 at which all the propositions were false and in the next state the first proposition p_1 becomes true, and as $e_P(s)$, the first state s_n in which the last proposition p_n becomes true.

Using the notions of beginning and ending states, we can give a precise definitions of the Precedence, Strict Precedence, and Response patterns with Global scope:

Definition 1. *Let P and Q be CP classes. We say that Q* precedes *P if once P holds at some state s, then Q also holds at some state s' for which $e_Q(s') \leq b_P(s)$. This simply indicates that Q precedes P iff the ending state of Q is the same as the beginning state of P or it is a state that happens before the beginning state of P.*

Definition 2. *Let P and Q be CP classes. We say that Q* strictly precedes *P if once P holds at some state s, then Q also holds at some state s' for which $e_Q(s') < b_P(s)$. This simply indicates that Q strictly precedes P iff the ending state of Q is a state that happens before the beginning state of P.*

Definition 3. *Let P and Q be CP classes. We say that Q* responds to *P if once P holds at some state s, then Q also holds at some state s' for which $b_Q(s') \geq e_P(s)$. This simply indicates that Q responds to P iff the beginning state of Q is the same as the ending state of P or it is a state that follows the ending state of P.*

5 Non-global Scopes Defined with Composite Propositions

So far we have discussed patterns within the "Global" scope. In this Section, we provide a formal definition of the other scopes described in Section 2.2.

We start by providing formal definitions of scopes that use CP as their parameters[1].

[1] These definitions use the notion of beginning state and ending state as defined in Section 4.

- For the “Before R” scope, there is exactly one scope – the interval $[0, b_R(s_f))$, where s_f is the first state when R becomes true. Note that the scope contains the state where the computation starts, but it does not contain the state associated with $b_R(s_f)$.
- For the scope “After L”, there is exactly one scope – the interval $[e_L(s_f), \infty)$, where s_f is the first state in which L becomes true. This scope, includes the state associated with $e_L(s_f)$.
- For the scope “Between L and R”, a scope is an interval $[e_L(s_L), b_R(s_R))$, where s_L is the state in which L holds and s_R is the first state $> e_L(s_L)$ when R becomes true. The interval contains the state associated with $e_L(s_L)$ but not the state associated with $b_R(s_R)$.
- For the scope “After L Until R”, in addition to scopes corresponding to “Between L and R”, we also allow a scope $[e_L(s_L), \infty)$, where s_L is the state in which L holds and for which R does not hold at state $s > e_L(s_L)$.

Using the above definitions of scopes made up of CP, we can now define what it means for a CP class to hold within a scope.

Definition 4. *Let P be a CP class, and let S be a scope. We say that P s-holds (meaning, P holds in the scope S) in S if P^{LTL} holds at state $s_p \in S$ and $e_P(s_P) \in S$ (i.e. ending state $e_P(s_p)$ belongs to the same scope S).*

Table 2 provides a formal description of what it means for a pattern to hold within a scope.

6 Need for New Operations

To describe LTL formulas for the patterns and scopes with CP, we need to define new “and” operations. These operations will be used to simplify the specification of the LTL formulas in Section 7.

In non-temporal logic, the formula $A \wedge B$ simply means that both A and B are true. In particular, if we consider a non-temporal formula A as a particular case of LTL formulas, then A means simply that the statement A holds at the given state, and the formula $A \wedge B$ means that both A and B hold at this same state.

In general a LTL formula A holds at state s if some “subformulas” of A hold in s and other subformulas hold in other states. For example, the formula $p_1 \wedge X p_2$ means that p_1 holds at the state s while p_2 holds at the state $s+1$; the formula $p_1 \wedge X \diamond p_2$ means that p_1 holds at state s and p_2 holds at some future state $s_2 > s$, etc. The statement $A \wedge B$ means that different subformulas of A hold at the corresponding different states but B only holds at the original state s. For patterns involving CP, we define an “and” operation that ensures that B holds at all states in which different subformulas of A hold. For example, for this new “and” operation, $(p_1 \wedge X p_2)$ and B would mean that B holds both at the state s and at the state $s+1$ (i.e. the correct formula is $(p_1 \wedge B \wedge X(p_2 \wedge B)))$. Similarly, $(p_1 \wedge X \diamond p_2)$ and B should mean that B holds both at state s and at state $s_2 > s$

Table 2. Description of Patterns Within Scopes

Pattern	Description
Existence	We say that there is an *existence of* P *within a scope* S if P s-holds at some state within this scope.
Absence	We say that there is an *absence of* P *within a scope* S if P never s-holds at any state within this scope.
Precedence	We say that Q *precedes* P *within the scope* s if once P s-holds at some state s, then Q also s-holds at some state s' for which $e_Q(s') \leq b_P(s)$.
Strict Precedence	We say that Q *strictly precedes* P *within the scope* s if once P s-holds at some state s, then Q also s-holds at some state s' for which $e_Q(s') < b_P(s)$.
Response	We say that Q *responds to* P *within the scope* s if once P s-holds at some state s, then Q also s-holds at some state s' for which $b_Q(s') \geq e_P(s)$.

when p_2 holds. In other words, we want to state that at the original state s, we must have $p_1 \wedge B$, and that at some future state $s_2 > s$, we must have $p_2 \wedge B$. This can be described as $(p_1 \wedge B) \wedge X \diamond (p_2 \wedge B)$.

To distinguish this new "and" operation from the original LTL operation $\wedge$, we will use a different "and" symbol & to describe this new operation. However, this symbol by itself is not sufficient since people use & in LTL as well; so, to emphasize that our "and" operation means "and" applied at several different moments of time, we will use a combination $\&_r$ of several & symbols.

In addition to the $A \&_r, B$ operator, we will also need two more operations:

- The new operation $A \&_l B$ will indicate that B holds at the *last* of A-relevant moments of time.
- The new operation $A \&_{-l} B$ will indicate that B holds at the all A-relevant moments of time except for the last one.

For the lack of space, this paper does not include the detailed description of these new LTL operators. The formal descriptions of these LTL operators along with examples of their use is provided by Salamah [7].

7 General LTL Formulas for Patterns and Scopes with CP

Using the above mentioned new LTL operators, this work defined template LTL formulas that can be used to define LTL specifications for all pattern/scope/CP combinations. The work defined three groups of templates; templates to generates formulas for the *Global* scope, templates to generate formulas for the $BeforeR$ scope, and templates to generate formulas for the remaining scopes. The templates for these remaining scopes use the templates for the *Global* and $BeforeR$ scopes. For the lack of space, we show an example template LTL formula from each of these three groups. The remaining templates are available in Salamah [7].

An example of a template LTL formula within the *Global* scope, is the template LTL formula for *Q Responds* to *P*:

- $\Box(P^{LTL} \rightarrow (P^{LTL} \,\&_l\, \diamond Q^{LTL}))$

An example of a template LTL formula within the $Before\ R$ scope, is the template LTL formula for $Q\, Precedes\ P_C\ Before\, R_C$:

- $(\diamond R^{LTL}) \rightarrow ((\neg(P^{LTL}\&_r\neg R^{LTL}))\, U\, ((Q^{LTL}\&_{-l}\neg P^{LTL}) \vee R^{LTL}))$

Finally, template formulas for the three remaining scopes can be constructed based on the templates for the *Global* and $BeforeR$ scopes. The formulas for the $AfterL$ scope can be built using the formulas for the *Global* scope as follows:

- $\neg((\neg L^{LTL})\, U\, (L^{LTL}\&_l\neg\mathcal{P}_G^{LTL}))$

This means that for any pattern, the formula for this pattern within the $ASfterL$ scope can be generated using the above formula and simply substituting the term $\mathcal{P}_G^{LTL}$ by the formula for that pattern within the *Global* scope.

In these examples and the remaining templates, the subscripts C and E attached to each CP indicates whether the CP class is of type condition or event, respectively. In the case where no subscript is provided, then this indicates that the type of the CP class is irrelevant and that the formula works for both types of CP classes.

8 Summary and Future Work

The work in this paper provided formal descriptions of the different composite propositions (CP) classes defined by Mondragon et al. [5]. In addition, we formally described the patterns and scopes defined by Dweyer et al. [2] when using CP classes. The main contribution of the paper is defining general LTL formulas that can be used to generate LTL specifications of properties defined by patterns, scopes, and CP classes. The general LTL formulas for the Global scope have been verified using formal proofs [7]. On the other hand, formulas for the remaining scopes were verified using testing and formal reviews [7].

The next step in this work consists of providing formal proofs for formulas of the remaining scopes. In addition, we aim at enhancing the Prospec tool by including the generation of LTL formulas that use the translations provided by this paper.

Acknowledgments. This work is funded in part by the National Science Foundation (NSF) through grant NSF EAR-0225670, by Texas Department of Transportation grant No. 0-5453, and by the Japan Advanced Institute of Science and Technology (JAIST) International Joint Research Grant 2006-08. The authors are thankful to the anonymous referees for important suggestions.

References

[1] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: a new Symbolic Model Verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, Springer, Heidelberg (1999)
[2] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specification for Finite State Verification. In: Proceedings of the 21st international conference on Software engineering, Los Angeles, CA, pp. 411–420 (1999)
[3] Hall, A.: Seven Myths of Formal Methods. IEEE Software 11(8) (1990)
[4] Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2004)
[5] Mondragon, O., Gates, A.Q.: Supporting Elicitation and Specification of Software Properties through Patterns and Composite Propositions. Intl. Journal Software Engineering and Knowledge Engineering 14(1) (2004)
[6] Manna, Z., Pnueli, A.: Completing the Temporal Picture. Theoretical Computer Science 83(1), 97–130 (1991)
[7] Salamah, I.S.: Defining LTL formulas for complex pattern-based software properties, University of Texas at El Paso, Department of Computer Science, PhD Dissertation, July (2007)
[8] Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Fifth Workshop on Runtime Verification (July 2005)

Pruning State Spaces with Extended Beam Search

Mohammad Torabi Dashti and Anton J. Wijs

CWI, Amsterdam
{dashti,wijs}@cwi.nl

Abstract. This paper focuses on using beam search, a heuristic search algorithm, for pruning state spaces while generating. The original beam search is adapted to the state space generation setting and two new search variants are devised. The resulting framework encompasses some known algorithms, such as A^*. We also report on two case studies based on an implementation of beam search in μCRL.

1 Introduction

State space explosion is still a major problem in the area of model checking. Over the years a number of techniques have emerged to prune, while generating, parts of the state space that are not relevant given the task at hand. Some of these techniques, such as partial order reduction algorithms (e.g. see [8]), guarantee that no essential information is lost after pruning. Alternatively, this paper focuses mainly on heuristic pruning methods which heavily reduce the generation time and memory consumption but generate only an approximate (partial) state space. The idea is that a user-supplied heuristic function guides the generation such that ideally only relevant parts of the state space are actually explored. This is, in fact, at odds with the core idea of model checking when studying qualitative properties of systems, i.e. to exhaustively search the complete state space to find any corner case bug. However, heuristic pruning techniques can very well target performance analysis problems as approximate answers are usually sufficient.

In this paper, we investigate how *beam search* can be integrated into the state space generation setting. Beam search (BS) is a heuristic method for combinatorial optimisation problems, which has extensively been studied in artificial intelligence and operations research, e.g. see [9,12]. BS is similar to breadth-first search as it progresses level by level through a highly structured search tree containing all possible solutions to a problem, but it does not explore all the encountered nodes. At each level, all the nodes are evaluated using a heuristic cost (or priority) function but only a fixed number of them is selected for further examination. This aggressive pruning heavily decreases the generation time, but may in general miss essential parts of the search tree, since wrong decisions may be made while pruning. Therefore, BS has so far been mainly used in searching trees with a high density of goal nodes. Scheduling problems, for instance, have been perfect targets for using BS as their goal is to optimally schedule a certain number of jobs and resources, while near-optimal schedules, which densely populate the tree, are in practice good enough.

The idea of using BS in state space generation is an attempt towards integrating functional analysis, to which state spaces are usually subjected, and quantitative analysis. Since model checkers, such as SPIN, UPPAAL and μCRL, which generate these

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 543–552, 2007.
© Springer-Verlag Berlin Heidelberg 2007

state spaces, usually have highly expressive input languages, BS for state spaces can be applied in a more general framework compared to its traditional use. Applying BS to search state spaces tightly relates to directed model checking (DMC) [3] and guided model checking (for timed automata) [1], where heuristics are used to guide the search for finding counter-examples to a functional property (usually in LTL) with a minimal exploration of the state space. Using A^* [3] and genetic algorithms [5] to guide the search are among notable works in this field. In contrast to DMC, we generate a partial state space in which an arbitrary property can be checked afterwards (the result would not be exact, hence being useful when near-optimal solutions suffice). However, there are strong similarities as well: A^* can be seen as an instantiation of BS (see § 3.4).

Contributions. We motivate and thoroughly discuss adapting the BS techniques to deal with arbitrary structures of state spaces. Next, we extend the classic BS in two directions. First, we propose *flexible* BS, which, broadly speaking, does not stick to a fixed number of states to be selected at each search level. This partially mitigates the problem of determining this exact fixed number in advance. Second, we introduce the notion of *synchronised* BS, which aims at separating the heuristic pruning phase from the underlying exploration strategy. Combinations of these variants create a spectrum of search algorithms that, as will be described, encompasses some known search techniques, such as A^*. We have implemented these variants of BS in the μCRL state space generation toolset [2]. Experimental results for two scheduling case studies are reported.

Road map. BS is described in § 2. § 3 deals with the adaptation of existing BS variants to the state space generation setting. There we also propose our extensions to BS. Memory management and choosing heuristics are also discussed there. § 4 reports on two case studies, § 5 presents our related work and § 6 concludes the paper.

2 Beam Search

Beam search [9,12] is a heuristic search algorithm for combinatorial optimisation problems, which has extensively been applied to scheduling problems, for example in systems designed for job shop environments [12].

BS is similar to breadth-first search as it progresses level by level. At each level of the search *tree* (in § 3 we extend BS to handle cycles), it uses a heuristic evaluation function to estimate the promise of encountered nodes[1], while the *goal* is to find a path from the root of the tree to a leaf node that possesses the minimal evaluation value among all the leafs. In each level, only the β most promising nodes are selected for further examination and the other nodes are permanently discarded. The *beam width* parameter β is fixed to a value before searching starts. Because of this aggressive pruning the search time is a linear function of β and is thus heavily decreased. However, since wrong decisions may be made while pruning, BS is neither complete, i.e. is not guaranteed to find a solution when there is one, nor optimal, i.e. does not guarantee finding an optimal solution. To limit the possibility of wrong decisions, β can be increased at the cost of increasing the required computational effort and memory.

[1] In this section we use nodes and edges, as opposed to states and transitions, to distinguish between the traditional setting of BS and our adaptations.

Two types of evaluation functions have traditionally been used for BS [12]: *priority* evaluation functions and *total-cost* evaluation functions, which lead to the *priority* and *detailed* BS variants, respectively. In priority BS at each node the evaluation function calculates a priority for each successor node and selects based on those priorities. At the root of the search tree, up to β most promising successors (i.e. those with the highest priorities) are selected, while in each subsequent level only one successor with the highest priority is selected per examined node. In detailed BS at each node the evaluation function calculates an estimate of the total-cost of the best path that can be found continuing from the current node. At each level up to β most promising nodes (i.e. those with the lowest total-cost values) are selected regardless of who their parent nodes are. When $\beta \to \infty$, detailed and priority BS behave as exhaustive breadth-first search.

3 Adapting Beam Search for State Space Generation

Motivation. In its traditional setting, BS is typically applied on highly structured search trees, which contain all possible orderings of a given number of jobs, e.g. see [7,14]. Such a search tree starts with n jobs to be scheduled, which means that the root of the tree has n outgoing transitions. Each node has exactly $n-k$ outgoing transitions, where k is the level in the tree where the node appears. State spaces, however, supposedly contain information on all possible behaviours of a system. Therefore, they may contain cycles, confluence of traces, and have more complex structures than the well-structured search trees usually subjected to BS. This necessitates modifying the BS techniques to deal with arbitrary structures of state spaces. Moreover, the BS algorithms search for a particular node (or schedule) in the search space, while in (and after) generating state spaces one might desire to study a property beyond simple reachability. We therefore extend BS to a state space *generation* (SSG) setting, as opposed to its traditional setting that focuses only on *searching*. The notion of a particular "goal" (cf. § 2) is thus removed from the adapted BS (see § 3.5 for possible optimisations when restricting BS to verify reachability properties). This along with the necessary machinery for handling cycles raise memory management issues in BS, that we discuss in § 3.5.

Below, DBS and PBS correspond, respectively, to the detailed and priority beam searches extended to deal with arbitrary state spaces (§ 3.1 and § 3.2). The F and S prefixes refer to the flexible and synchronised variants (§ 3.3 and § 3.4). we start with introducing labelled transition systems.

Labelled transition system. (LTS) is a tuple (Σ, s_0, Act, Tr), where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Act is a finite set of action labels and $Tr \subseteq \Sigma \times Act \times \Sigma$. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in Tr$. In this paper we consider LTSs with finite Σ.

3.1 Priority Beam Search for State Space Generation

Below we first present the PBS algorithm and then describe and motivate the changes that we have made to the traditional priority BS.

PBS is shown in figure 1. The sets *Current*, *Next* and *Expanded* denote, respectively, the set of states of the current level, the next level and the set of states that have been

expanded. The user-supplied function $priority : Act \to \mathbb{Z}$ provides the priority of actions, as opposed to states.[2] We motivate this deviation by noting that *jobs* in the BS terminology correspond more naturally with *actions* in LTSs.

The set *Buffer* temporarily keeps seemingly promising transitions. The function $prio_{\min} : \mathscr{P}(Tr) \to \mathbb{Z}$ returns the lowest priority of the actions of a given set of transitions, with $prio_{\min}(\emptyset) = -\infty$. The function $getprio_{\min} : \mathscr{P}(Tr) \to Tr$, given a set of transitions, returns one of the transitions having an action with the lowest priority. Expanding the set $Current \setminus Expanded$ in the **while** loop ensures that no state is revisited in case cycles or confluent traces exist in the search space. The algorithm terminates when it has explored all the states in its beam.

```
Current := {s_0}
Expanded := ∅; Buffer := ∅
level := 0; limit := α
while Current \ Expanded ≠ ∅ do
  Next := ∅
  for all s ∈ Current \ Expanded do
    for all s -a-> s' ∈ en(s) do
      if priority(a) > prio_min(Buffer) then
        if |Buffer| = limit then
          Buffer := Buffer \
                    {getprio_min(Buffer)}
        Buffer := Buffer ∪ {s -a-> s'}
    Next := Next ∪ nxt(s, Buffer)
    Buffer := ∅
  Expanded := Expanded ∪ Current
  Current := Next
  level := level + 1
  if level = l then limit := 1
```

Fig. 1. Priority BS

In priority BS, originally, up to β children of the root are selected. The resulting beam of width β is then maintained by expanding only one child per node in subsequent levels. In state spaces, however, the root has typically much less outgoing transitions than the average branching factor of the state space. Fixing the beam width at such an early stage is therefore not reasonable.

To mitigate this problem, instead of β, the algorithm of figure 1 is provided with the pair (α, l), where $\alpha, l \in \mathbb{N}$ and $\alpha^l = \beta$. The idea is that the algorithm uses the *priority* function to prune non-promising states from the very first level, but in two phases: before reaching nearly β states in a single level, it considers the most promising α transitions for further expansion, but after that, it expands only one child per state.

3.2 Detailed Beam Search for State Space Generation

In detailed BS a total-cost evaluation function $f : \Sigma \to \mathbb{N}$ is used to guide the search. This function is decomposed into $f(s) = g(s) + h(s)$. The $g(s)$ function represents the cost taken to reach s from the initial state s_0, which is defined as $g(s) = g(s') + cost(a)$ if $s' \xrightarrow{a} s$. The user-supplied function $cost : Act \to \mathbb{N}$ assigns weights to actions that can, e.g., denote the time needed to perform different jobs in a scheduling problem. These weights are fixed before search starts. Since the range of *cost* is non-negative numbers, we have $s \to^* s' \implies g(s') \geq g(s)$. The user-supplied function $h(s)$ estimates the cost it would take to efficiently reach a goal state (or complete the schedule) continuing from s.

[2] In general, *priority* can also depend on states: $priority : \Sigma \to Act \to \mathbb{Z}$. In this paper, we only consider fixed priorities, which resembles *dispatch* scheduling in AI terminology [12].

Thus, for a goal state s, $h(s) = 0$. The f function is called *monotonic* if $s \rightarrow^* s'$ implies $f(s) \leq f(s')$.

The original idea of detailed BS does not need to change much to fit into the SSG setting except for when handling cycles. When exploring a cyclic LTS, to guarantee the termination of the algorithm, it is necessary to store the set of explored states to avoid exploring a state more than once (cf. the *Expanded* set in figure 1). However, if a state is reached via a path with a lower cost, the state has to be re-examined. This is because the total-cost of each state depends on the cost to reach that state from the root, cf. § 3.4.

3.3 Flexible Beam Search

A major issue that still remains unaddressed in the BS adaptations of § 3.1 and 3.2 is the *tie-breaking* problem: How should equally competent candidates, e.g. having the same f values, be pruned? These selections are beyond the influence of the evaluation function and can undesirably make the algorithm non-deterministic. Hence, we propose two variants of BS that we call *flexible detailed* and *flexible priority* beam searches, in which the beam width can change during state space generation.

In flexible detailed BS, at each level, up to β most promising states are selected plus any other state which is as competent as the worst member of these β states. This achieves closure on the worst (i.e. highest) total-cost value being selected. Similarly, in flexible priority BS, at each state, all the transitions with the same priority as the most promising transition of that particular state are selected. Note that in FPBS, in contrast to FDBS, if the beam width is stretched, it cannot be readjusted to the intended β.

3.4 Synchronised Beam Search

As is described in § 2, the classic BS algorithms were tailored for the breadth-first exploration strategy. Below, we explain a way to do BS on top of best-first [10] exploration algorithms. Broadly speaking, we separate the exploration strategy from the pruning phase, so that the exploration is guided with a (possibly different[3]) heuristic function. This is particularly useful when checking reachability properties on-the-fly.

Below, we inductively describe $\mathcal{G}$-*synchronised* xBS, where $\mathcal{G} : \Sigma \rightarrow \mathbb{N}$ is the function that guides the exploration and $x \in \{\mathrm{D}, \mathrm{P}, \mathrm{FD}, \mathrm{FP}\}$ (denoting the BS variants described previously). Let $\hat{S}_i$ denote the set of states to be explored at round i.[4] We partition this set into equivalence classes $c_0, \cdots, c_n$, where $n \in \mathbb{N}$, such that $\hat{S}_i = c_0 \cup \cdots \cup c_n$ and $\forall s \in \hat{S}_i.\ s \in c_j \iff \mathcal{G}(s) = j$. The pruning algorithm xBS is subsequently applied only on c_k where $c_k \neq \emptyset \wedge \forall j < k.\ c_j = \emptyset$. According to the pruning algorithm (which can possibly employ an evaluation function different from $\mathcal{G}$), some of the successors of c_k are selected, constituting the set $\hat{S}$. The next round starts with $\hat{S}_{i+1} = \hat{S} \cup \hat{S}_i \setminus c_k$. Since synchronised beam search separates the exploration algorithm from the pruning algorithm, it can be perfectly combined with the other variants of BS introduced earlier.

[3] Using different functions for guiding exploration and pruning in principle allows dealing with multi-priced optimisation problems, cf. [1].

[4] "Round" i corresponds to a logical (i.e. not necessarily horizontal) level in the state space, which is processed in the ith iteration of SSG algorithm.

Using any constant function as $\mathcal{G}$ in SDBS would clearly result in BS with breadth-first exploration strategy.

Figure 2 shows $\mathcal{G}$-SDBS in detail. The sets *Current*, *Next* and *Expanded* contain pairs of states and corresponding g values, i.e. $\langle s, s.g\rangle$. The function $getf_{\max} : \mathcal{P}(\Sigma) \to \Sigma$, given a set of states, returns one of the states that has the highest f value. Here $unify(X)$ and $update(X,Y)$ are defined as follows: $unify(X) = \{\langle s,g\rangle \in X \mid \forall \langle s,g'\rangle \in X.\ g \leq g'\}$ and $update(X,Y) = \{\langle s,g\rangle \in X \mid \neg\exists \langle s,g'\rangle \in Y.\ g' \leq g\}$. In this algorithm, a state will be revisited only if it is reached via a path with a lower cost than the g cost assigned to it (see also § 3.2).

```
s0.g := 0; Current := {⟨s0, s0.g⟩}
Expanded := ∅
while Current ≠ ∅ do
  Next := ∅; i := −1; found := F
  while found = F do
    i := i + 1
    c_i := {⟨s, s.g⟩ ∈ Current | 𝒢(s) = i}
    if c_i ≠ ∅ then
      Current := Current \ c_i
      found := T
  while |c_i| > β do
    c_i := c_i \ {getf_max(c_i)}
  for all s ∈ c_i do
    for all s −a→ s' ∈ en(s) do
      s'.g := s.g + cost(a)
      Next := Next ∪ {⟨s', s'.g⟩}
  Expanded := unify(Expanded ∪ c_i)
  Current := update(unify(Next ∪
                 Current), Expanded)
```

Fig. 2. Synchronised detailed BS

To mention a practically interesting candidate for $\mathcal{G}$, we temporarily deviate from our general setting. Consider the problem of finding a path of minimal cost that leads to a particular state in the search space. Recall that the total-cost function in DBS can be decomposed into $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the trace leading from the root to s. If $\mathcal{G}(s) = g(s)$ in $\mathcal{G}$-synchronised DBS, once the goal state is found, searching can safely terminate. This is because at a goal state s, $f(s) = g(s)$ and since the algorithm always follows paths with minimal g (remember that g is monotonic), state s is reached before another state s' iff $g(s) \leq g(s')$. We observe that in g-SDBS no state is re-explored, because states with minimal g are taken first and thus a state can be reached again only via paths with higher costs (cf. § 3.2). Both g-SDBS and g-SPBS have been used in our experiments of § 4, where minimal-time traces to a particular state are searched for.

As another variant of synchronised search, we note that given a *monotonic* total-cost function $f(s) = g(s) + h(s)$ (cf. § 3.2), f-SFDBS with arbitrary $\beta > 0$, corresponds to the well-known A^* search algorithm (e.g. see [10]).[5] Due to space constraints, we refer to [13] for a proof of this relation.

3.5 Discussions

Memory management is a challenging issue in SSG. Although BS reduces memory usage due to cutting away parts of the state space, still explored states need to be accessed to guarantee the termination of SSG in case of cyclic LTSs. This can be partially

[5] The monotonicity assumption on f is necessary for optimality of A^* [10].

counter-measured by taking into account specific characteristics of the problem at hand and the properties that are to be checked:

1. When aiming at a reachability property (such as reachability of a goal state, checking invariants and hunting deadlock states), once a state satisfying the desired property is reached the search can terminate and the witness trace can be reported. This however cannot be extended to arbitrary properties.
2. If there are no cycles in the state space, there is in principle no need to check whether a state has already been visited (in order to guarantee termination). Therefore, only the states from the current level need to be kept and the rest can be removed from memory[6], i.e. flushed to high latency media such as disks.

Heuristic functions and selecting the beam width. Effectiveness of BS hinges on selecting good heuristic functions. Heuristic functions heavily depend on the problem being solved. As our focus here is on exploration strategies that utilise heuristics, we do not discuss techniques to design the heuristic functions themselves. Developing heuristics constitutes a whole separate body of research and, here, we refer to a few of them. Among others, [3,6,12] complement the work we present in this paper, as they explain how to design heuristic functions when, e.g., analysing Java programs or provide approximate distance to deadlocks, etc.

Selecting the beam width β is another challenge in using BS. The beam width intuitively calibrates the time and memory usage of the algorithm on one hand and the accuracy of the results on the other hand. Therefore, in practice the time and memory limits of a particular experiment determine β. To reduce the sensitivity of the results to the exact value of β, flexible BS variants can be used. This, however, comes at the price of losing a tight grip on the memory consumption (see also § 3.3). For a general discussion on selecting β and its relation to the quality of answer see [12].

4 Experimental Results

In this section we report our experimental results.[7]

Cannibals and missionaries (C&M) problem is a classic river crossing puzzle and produces state spaces with interesting structures: they contain cycles, deadlocks and confluent traces. Assume that C missionaries and C cannibals stand on the left bank of a river that they wish to cross. The goal is to find a schedule for ferrying all the cannibals and all the missionaries across using a boat that can take up to B people at a time. The condition is that the cannibals never outnumber the missionaries, on a shore or in the boat. Moving the boat costs 1 time unit per passenger, and we wish to find a minimal cost path towards the goal. We use a μCRL implementation of BS and a SPIN implementation of the depth-first branch-and-bound algorithm to solve this problem.

[6] In this case, some states may be revisited due to confluent traces, hence undesirably increasing the search time.

[7] The experiments have been performed on a single machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2. See `http://www.cwi.nl/~wijs/TIPSy` for a complete report along with specs.

Table 1. Experimental results C&M. Times are in min:sec. o.o.t.: out of time (set to 12 hours); o.o.m.: out of memory (set to 900 MB).

Problem	Result	μCRL MCS		μCRL g-SFDBS				SPIN DFS		SPIN DFS BnB Prop.	
(C,B)	T	# States	Time	T	β	# States	Time	# States	Time	# States	Time
(3,2)	18	147	00:03.80	18	3	142	00:03.73	28,535	00:00.32	26,062	00:00.29
(20,4)	104	2,537	00:05.32	106	10	2,191	00:05.38	445,801	00:02.66	408,053	00:02.34
(50,20)	116	90,355	00:20.15	120	15	17,361	00:11.45	12,647,000	02:05.25	12,060,300	01:49.59
(100,10)	292	49,141	00:19.65	296	10	16,274	00:14.46	14,709,600	02:49.32	13,849,300	02:23.34
(100,30)	222	366,608	01:05.79	228	15	61,380	00:32.06	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,30)	680	1,008,436	04:10.72	684	15	205,556	02:30.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,50)	1,076	4,365,536	21:40.52	1,080	20	685,293	10:33.28	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,100)	1,036	17,248,979	77:16.36	1,040	20	1,170,242	16:47.10	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,250)	o.o.t.	o.o.t.	o.o.t.	2,032	20	5,317,561	240:22.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.

The results are shown in table 1. Since μCRL and SPIN count states in different ways, the numbers of states of the experiments using different tools are not comparable.

In μCRL, we first applied g-SFDBS with constant h (i.e. no estimation) with any $\beta > 0$, denoted minimal cost search MCS in table 1. MCS is an exhaustive search method, where the states are ordered based on the cost needed to reach them from the initial state. This search is used to find the minimum number of time units needed to solve the problem (shown in the *Result* column). As a comparison, we have also performed experiments with SPIN. In those cases, we followed the algorithm of [11], a prominent technique to use heuristics within SPIN. The idea is that the LTL formula that is checked is modified during verification to reflect the best solution found so far. This can effectively implement a branch-and-bound mechanism in SPIN, denoted DFS BnB Prop in table 1. This algorithm avoids exhaustive search, yet it is complete.

Besides that we used g-SFDBS with $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s) \rangle \times (2 \times C))$ as the heuristic part of DBS, where $C(s)$ and $M(s)$ are the numbers of cannibals and missionaries on the left bank in state s, respectively, and $\langle C(s) \neq M(s) \rangle$ is a Boolean expression returning 1 if $C(s) \neq M(s)$, and 0 otherwise. In table 1, the T column under g-SFDBS shows the minimum number of time units needed to solve the problem approximated by this search. The results show an example of what can be achieved when near-optimal solutions are acceptable. Our g-SFDBS algorithm should ideally be compared with other heuristic state space generation tools, such as HSF-SPIN [4]. We however leave this as future work.

Clinical Chemical Analyser (CCA) is a case study taken from industry [15]: it is used to analyse batches of test receipts on patient samples (blood, plasma, etc) that are uniquely described by a triple which indicates the number of samples of each fluid (see table 2). We have extensively described the CCA case in [16].

Table 2 reports the results of applying MCS, g-SDBS, g-SPBS and g-SFPBS to solve the problem of scheduling the CCA. The result column provides the total-cost (i.e. required time units) of the solution found. We remark that all these searches are tuned to find the optimal answer (for those cases where it was known to us). In case of g-SFPBS, the value of (α, l) is fixed to $(1, 1)$. The benefit of flexible variants of BS is

Table 2. Experimental results CCA. o.o.t.: out of time (set to 30 hours).

Case	Result	MCS		g-SDBS			g-SPBS			g-SFPBS	
		β	Time	β	#States	Time	(α,l)	#States	Time	#States	Time
(3,1,1)	36	3,375	00:10.35	25	1,461	00:03.43	1,1	48	00:03.03	821	00:03.70
(1,3,1)	39	13,194	00:30.48	41	2,234	00:03.93	1,1	179	00:03.08	1,133	00:04.06
(6,2,2)	51	341,704,322	1524:56.00	81	7,408	00:07.76	2,9	479	00:03.06	45,402	02:33.65
(1,2,7)	73	o.o.t.	o.o.t.	75,000	6,708,705	84:38.41	1,1	90	00:02.99	122,449	04:02.94
(7,4,4)	75	o.o.t.	o.o.t.	35,000	3,801,607	41:01.80	3,25	155,379	08:14.66	20,666,509	872:55.71

thus clear here: A stable beam width is mostly sufficient. However, as a draw-back we observe that FPBS exhibits early state space explosion, compared to PBS.[8]

We observe that β is not directly related to the number of fluids in a test case. We believe this can be due to the ordering of states while searching, since a stable β suffices when using the flexible SFPBS. We conclude this discussion with noting that CCA provides a case study which can better be tackled using priority BS, compared to detailed BS variants.

5 Related Work

BS is extended to a complete search in [18], by using a new data structure, called a beam stack. Thereby, it is possible to achieve a range of searches, from depth-first search ($\beta = 1$) to breadth-first search ($\beta \to \infty$). Considering our extensions for arbitrary state spaces, it would be interesting to try to combine these two approaches.

Notable works on scheduling using formal method tools are [1] and [11]. In [1], Behrmann et al. have extended timed automata with linearly priced transitions and locations, resulting in UPPAAL CORA tool. They deal with reachability analysis using the standard branch-and-bound algorithm. A number of basic exploration techniques can be used for branching, and bounding is done based on heuristics. In [11], the depth-first branch-and-bound technique is used for scheduling in SPIN. See also § 4.

In [17], we report on a distributed implementation of the BS variants proposed in this paper, where a number of machines together perform these search algorithms.

6 Conclusions

In this paper, we extended and made available an existing search technique to be used for quantitative analysis within a setting used for system verification.

Our experiments showed the usefulness and flexibility of these extensions. We observed that BS can be tuned to encompass some other (heuristic) search algorithms, thus providing a flexible state space generation framework.

Future work. Comparing our implementation with other heuristic state space generation tools, such as HSF-SPIN, is certainly a next step for this work. Also, BS can in

[8] In FPBS once the beam width is stretched, it cannot be readjusted to its initial value, see § 3.3.

principle deal with infinite state spaces given that the heuristic function does not cut away all finite paths. This application of BS has yet to be investigated.

Acknowledgements. We are grateful to Jaco van de Pol and Michael Weber for their insightful comments on the paper, and to Bert Lisser for implementing parts of BS variants.

References

1. Behrmann, G., Larsen, K., Rasmussen, J.: Optimal scheduling using priced timed automata. SIGMETRICS Perform. Eval. Rev. 32(4), 34–40 (2005)
2. Blom, S., Fokkink, W., Groote, J., van Langevelde, I., Lisser, B., van de Pol, J.: μCRL: A toolset for analysing algebraic specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
3. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2), 247–267 (2004)
4. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
5. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 266–280. Springer, Heidelberg (2002)
6. Groce, A., Visser, W.: Heuristics for model checking Java programs. STTT 6(4), 260–276 (2004)
7. Oechsner, S., Rose, O.: Scheduling cluster tools using filtered beam search and recipe comparison. In: Proc. 2005 Winter Simulation Conference, pp. 2203–2210. IEEE Computer Society Press, Los Alamitos (2005)
8. Peled, D.: Ten years of partial order reduction. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998)
9. Pinedo, M.: Scheduling: Theory, algorithms, and systems. Prentice-Hall, Englewood Cliffs (1995)
10. Russell, S., Norvig, P.: Artificial intelligence: A modern approach. Prentice-Hall, Englewood Cliffs (1995)
11. Ruys, T.: Optimal scheduling using Branch-and-Bound with SPIN 4.0. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 1–17. Springer, Heidelberg (2003)
12. Si Ow, P., Morton, E.: Filtered beam search in scheduling. Intl. J. Production Res. 26, 35–62 (1988)
13. Dashti, M.T., Wijs, A.J.: Pruning state spaces with extended beam search. Technical Report SEN-R0610, CWI, 2006. `ftp.cwi.nl/CWIreports/SEN/SEN-R0610.pdf`.
14. Valente, J., Alves, R.: Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. Comput. Ind. Eng. 48(2), 363–375 (2005)
15. Weber, S.: Design of Real-Time supervisory control systems. PhD thesis, TU/e (2003)
16. Wijs, A., van de Pol, J., Bortnik, E.: Solving scheduling problems by untimed model checking. In: Proc. FMICS 2005, pp. 54–61. ACM Press, New York (2005)
17. Wijs, A.J., Lisser, B.: Distributed extended beam search for quantitative model checking. In: MoChArt 2006. LNCS (LNAI), vol. 4428, pp. 165–182 (2007)
18. Zhou, R., Hansen, E.: Beam-stack search: Integrating backtracking with beam search. In: Proc. ICAPS 2005, pp. 90–98. AAAI (2005)

Using Counterexample Analysis to Minimize the Number of Predicates for Predicate Abstraction

Thanyapat Sakunkonchak, Satoshi Komatsu, and Masahiro Fujita

VLSI Design and Education Center, The University of Tokyo
2–11–16 Yayoi, Bunkyo-ku, Tokyo, 113–0032, Japan
{thong,komatsu}@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract. Due to the success of the model checking technique in the hardware domain, over the last few years, model checking methods have been applied to the software domain which poses its own challenges, as software tends to be less structured than hardware. Predicate abstraction is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type. The cost for computation of abstraction and model checking depends on the number of state variables in the abstract model. In this paper, we propose a simple, yet efficient method to minimize the number of predicates for predicate abstraction. Given a spurious counterexample, at least one predicate is assigned at each program location during the refinement process. The computational cost depends proportionally to the number of assigned predicates. In this paper, instead, we search the counterexample to find the conflict predicates that caused this counterexample to be spurious. Then, we assign the necessary predicates to the abstract model. We compare the performance of our technique with the interpolation-based predicate abstraction tool like BLAST. The proposed method presents significantly better experimental results on some examples with large set of predicates.

1 Introduction

Model checking is the formal verification technique most-commonly used in the verification of RTL or gate-level hardware designs. Due to the success of the model checking technique in the hardware domain [1,2], over the last few years, model checking methods have been applied to the software domain, and we have seen the birth of software model checkers for programming languages such as C/C++ and Java.

There are two major approaches to software model checking. The first approach emphasizes *state space exploration*, where the state space of a system model is defined as the product of the state spaces of its concurrent finite-state components. The state space of a software application can be systematically explored by driving the "product" of its concurrent processes via a run-time scheduler through all states and transitions in its state space. This approach is developed in the tool Verisoft [3]. The second approach is based on *static analysis and abstraction* of software. It consists of automatically extracting a model

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 553–563, 2007.
© Springer-Verlag Berlin Heidelberg 2007

out of a software application by statically analyzing its code and abstracting away details, and then applying symbolic model checking to verify this abstract model.

In the context of the second approach, *predicate abstraction* [4,5,6] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. The generated abstract model is conservative in the sense that for every execution in the original model there is a corresponding execution in the abstract model. Each predicate in the original model is corresponding to a Boolean variable in the abstract model. Abstraction is coarse in the beginning. The spurious counterexample from model checker gives an information to refine the abstract model to be more precise.

We address three problems on refinement of predicate abstraction.

- **Select new predicates:** The choice of selection of new predicates is important. At every refinement loop, spurious behaviors can be removed by adding new predicates to make the relationships between existing predicates more precise. New predicates from the wrong selection can cause the abstract model to be intractable during the refinement.
- **Scope of predicates:** When predicates are used globally, coping with relationship of all predicates through the entire model is impossible when the size of the abstract model is large. Hence, the idea of localizing predicates should be considered.
- **Number of predicates:** Although localization of predicates is utilized, the computation of predicate abstraction and model checking are exponential with the number of predicates. The cost of computation in abstraction process can be reduced drastically when the number of predicates is smaller.

There are many software model checking tools that apply predicate abstraction, e.g. SLAM [5], BLAST [6] or F-Soft [7]. The problems addressed above can be handled by these tools.

- SLAM uses the techniques called *Cartesian approximation* and *maximum cube length approximation* to reduce the number of predicates in each theorem prover query. The selection of new predicates is conducted by a separate refinement algorithm.
- In BLAST toolkit, besides *lazy abstraction* technique which does *on-the-fly* abstraction and refinement of abstract model, the new refinement scheme based on *Craig interpolation* [8] was proposed. With this method, localization for predicates and predicate selection problems can be handled.
- Recent work [7] describes localization and register sharing to reduce the number of predicates at each local part of the abstract model. *Weakest preconditions* are used to find new predicates. This approach has been implemented to the F-Soft toolkit.

This work is inspired by a localization and register sharing [7] which applied to reduce the number of predicates in the abstraction refinement process. In this

paper, we propose a different perspective to minimize number of predicates by analyzing counterexample. For any spurious counterexample, there is at least one variable that makes it infeasible. In contrast to BLAST and F-Soft where the computation of new predicates can be performed by using interpolation and weakest pre-conditions techniques, respectively, we traverse through the counterexample and use propagation-and-substitution of variables to find infeasible variables then assign new predicates. The proposed method produces equal or smaller number of predicates comparing to other approaches. The cost of traversing counterexample to find new predicates is small comparing to model checking and abstraction computation.

This paper is organized as follows. Some related researches on software model checking describes in next section. An illustrative example used to describe the idea of the proposed method is shown in Section 3. Formal definitions of predicate abstraction and refinement are presented in Section 4, while in Section 5 describes our approach of minimizing the number of predicates by analyzing the counterexample. Some experimental results comparing our approach with the public tool BLAST are presented in Section 6 and we conclude with Section 7.

2 Related Work

Software model checking poses its own challenges, as software tends to be less structured than hardware. In addition, concurrent software contains processes that execute asynchronously, and interleaving among these processes can cause a serious state-space explosion problem. Several techniques have been proposed to reduce the state-space explosion problem, such as partial-order reduction and abstraction. In the software verification domain, predicate abstraction [4,5,9] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. Counterexample-Guided Abstraction Refinement (CEGAR) [10] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property. Toolkits of various platforms like C or the system-level design languages like SystemC or SpecC have been implemented. We briefly introduce some of them.

The SLAM project [5] conducted by Ball and Rajamani has developed a model checking tool based on the interprocedural dataflow analysis algorithm presented in [11] to decide the reachability status of a statement in a Boolean program. The generation of an abstract Boolean program is expensive because it requires many calls to a theorem prover.

Clarke et al. [12] use SAT-based predicate abstraction. During the abstraction phase, instead of using theorem provers, a SAT solver is used to generate the abstract transition relation. Many theorem prover calls can potentially be replaced

by a single SAT instance. Then, the abstract Boolean programs are verified with SMV. In contrast to SLAM, this work is able to handle bit-operations as well. This idea is also extended to use with SpecC language [13]. The synchronization constructs *notify/wait* can be modeled, but it does not explain how to handle the timing constraints that are introduced by using *waitfor*.

Sakunkonchak et al. [14] propose a SpecC/synchronization verification tool (S-VeT) based on the predicate abstraction of ANSI-C programs of SLAM project [5]. Concurrency and synchronization can be handled by mathematically model SpecC programs by equalities/inequalities formulae. These formulae are solved by using the Integer Linear Programming (ILP) solver.

3 Motivating Example

Similar to existing methods, abstraction and refinement processes are automated based on the state-of-the-art model checking and counterexample-guided abstraction refinement (CEGAR). More specifically, starting with a coarse level of abstraction, the given property (e.g. reachability of a statement that caused an error in the model) is used to verify the abstract model. A counterexample is given when this property is not satisfied. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process continues until there is no error found or there is no solution for the given property.

Let us consider the program as shown in Figure 1(a). We would like to check whether a statement labeled "ERROR" is reachable. Predicate abstraction of the original program is applied as shown in Figure 1(b) where *Abstract*0 abstracts *Original*. At this moment, we do not consider any particular predicate. Statements are not considered (represented as "...") and branching conditions are considered as non-deterministic choices (represented as "*"). Note that the behaviors and control constructs at every abstraction and refinement step are preserved.

When model check the abstract model *Abstract*0 in Figure 1(b), ERROR label is reachable as shown by an abstract counterexample which goes through all program locations starting from 1 to 8 (for simplicity, we omit the declaration of variables in line 0). Concretization of this counterexample with respect to *Original* is applied. *Concretize* as shown in the left of Figure 1(c) is obtained by corresponding every location from an abstract counterexample to the same location in *Original*. Decision procedure is used to simulate this concrete path. The concrete path *Concretize* is obviously *infeasible* (explain below). To find new predicates to refine *Abstract*0, the technique described in [8,7] can be used to remove the infeasible trace by tracking exactly one predicate at each program location from 1 to 8. The method in [7] can further remove more predicates by traversing the counterexample with weakest pre-conditions.

The proposed method is simpler and more efficient. Simulation of the concrete path and finding new predicates can be done by storing information while traversing the counterexample. *Concretize* is simulated starting from location 1.

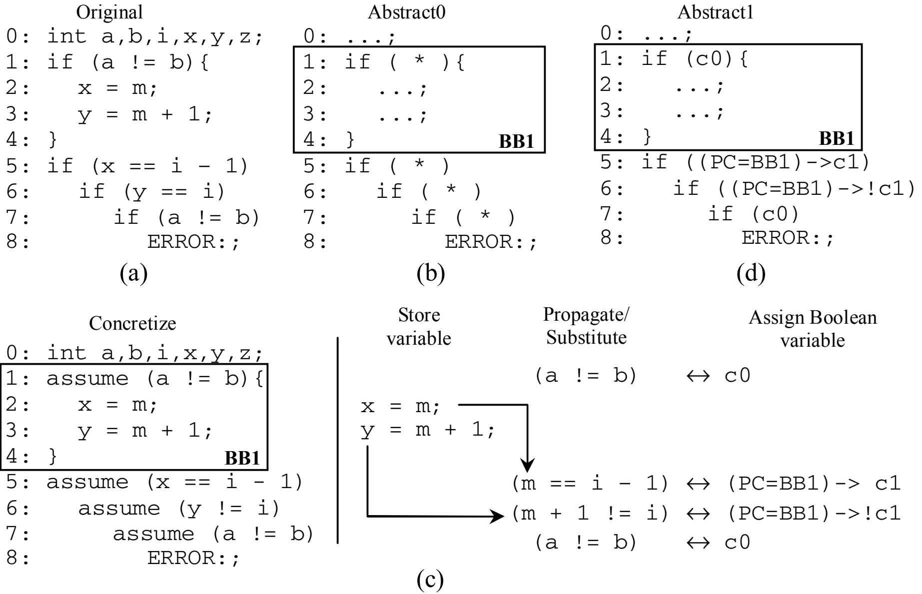


Fig. 1. (a) Original program. (b) First abstraction. The counterexample is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$. (c) Concretized the counterexample and use the propagate-and-substitute of variables x and y. Assign Boolean variables to corresponding predicates that caused infeasibility in counterexample. Note that only two predicates are used. (d) Abstract1 is the refined version of Abstract0 in (b). After model check, label ERROR is not reachable.

At each location during traversal, the program locations and assignment of variables are stored. For example, at location 2 and 3, the assignments of variables x and y are stored and these locations can be referred as they are in the same basic block $BB1$. As traversing to location 5 and 6, variables x and y are used. The symbolic values of x and y ($x \Rightarrow m$ and $y \Rightarrow m+1$) in location 2 and 3 are propagating-and-substituting to location 5 ($x == i - 1$) and 6 ($y\ != i$). We obtain ($m == i - 1$) and ($m + 1\ != i$) in location 5 and 6, respectively. This makes this counterexample infeasible. The process is repeated until error statement is found. Up to this point, the process of simulating the concrete path is the same with other methods except that the program locations and the assignments of variables are stored in memory. The concrete path is traversed and predicates that caused infeasibility are found. Boolean variables $c0, c1, !c1, c0$ can be assigned at location 1, 5, 6, and 7, respectively. However, at location 5 and 6, variables x and y depend on the execution of $BB1$. The conditions of dependent path must be added. Finally, the predicates assigned at location 1, 5, 6, 7 are $c0, (PC = BB1) \rightarrow c1, (PC = BB1) \rightarrow\ !c1, c0$, respectively.

Note that only two Boolean variables ($c0$ and $c1$) are used in this example. Although there is a program location ($PC = BB1$) to cooperate with $c1$, this is not an additional variable in the abstract model when performing model checking.

This is because all the program locations are already included in the abstract model. Therefore, only $c0$ and $c1$ are added for model checking and abstraction computation. According to this example, equal or larger number of predicates is produced by methods [8,7] and so as the cost of computation.

Finally, the abstract model $Abstract1$ in Figure 1(d) is constructed according to these new predicates. As the result of model checking, the error statement at location 8 is unreachable. Hence, this program is safe.

4 Predicate Abstraction and Refinement

4.1 Predicate Abstraction

Predicate abstraction is a technique to construct a conservative abstraction which maps an infinite state-space concrete program to an abstract program. Formally the concrete transition system is described by a set of *initial states* represented by $I(\bar{s})$ and a *transition relation* represented by $R(\bar{s}, \bar{s}')$ where S is a set of all states in concrete program and $\{\bar{s}, \bar{s}'\} \in S$ are a set of current and next states, respectively. The variables in the concrete program are represented by Boolean variables which correspond to one predicate on the variables in the concrete program. The abstraction is determined by a set of predicates $Pred = \{\phi_1, \ldots, \phi_k\}$ over the given program. If applying all predicates to a concrete state, a k-width vector of Boolean values, $\bar{b}$, is obtained. In other words, $\bar{b} = abs(\bar{s})$ maps the concrete states $\bar{s} \in S$ into the abstract states $\bar{b}$ where $abs(\cdot)$ is an *abstraction function*.

The term *conservative abstraction* means a transition from the abstract states $\bar{b}$ to $\bar{b}'$ in the abstract model exist if and only if there is a transition from $\bar{s}$ to $\bar{s}'$ in there concrete model where $\bar{b} = abs(\bar{s})$ and $\bar{b}' = abs(\bar{s}')$. The abstract initial states $\hat{I}(\bar{b})$ is

$$\hat{I}(\bar{b}) := \exists \bar{s} \in S : (abs(\bar{s}) = \bar{b}) \wedge I(\bar{s})$$

where the abstract transition relation $\hat{R}(\bar{b}, \bar{b}')$ can be shown as

$$\hat{R} := \{(\bar{b}, \bar{b}') | \exists \bar{s}, \bar{s}' \in S : R(\bar{s}, \bar{s}') \wedge (abs(\bar{s}) = \bar{b}) \wedge (abs(\bar{s}') = \bar{b}')\}$$

With this conservative abstraction, if the property $\hat{Prop}$ holds on an abstract state $\bar{b}$ in the abstract model, then the property must hold on all states $\bar{s}$ where $abs(\bar{s}) = \bar{b}$.

$$\hat{Prop}(\bar{b}) := \forall \bar{s} \in S : (abs(\bar{s}) = \bar{b}) \Rightarrow Prop(\bar{s})$$

Therefore, when model checking the abstract model, if property $\hat{Prop}$ holds on all reachable states, then $Prop$ also holds on all reachable states in concrete model. Otherwise, an abstract counterexample is obtained. In order to check if this abstract counterexample corresponds to a concrete counterexample, we need to simulate this abstract trace by concretizing it with the concrete model. If the simulation result tells that the abstract trace is really not feasible in the concrete model, this abstract counterexample is then called the *spurious counterexample*. To remove the spurious behaviors from the abstract model, the refinement of abstraction is needed.

4.2 Abstraction Refinement

If the abstract counterexample cannot be simulated, it is because the abstraction is too coarse. Spurious transitions make the abstract model not corresponding to the concrete model. In order to eliminate the spurious transitions from the abstract model, we find variables that caused infeasibility in the concrete trace.

A sequence of length $L + 1$ in the abstract model is a sequence of abstract states, $\bar{b}_0, \ldots, \bar{b}_L$ such that $\hat{I}(\bar{b}_0)$ holds and for each i from 0 to $L-1$, $\hat{R}(\bar{b}_i, \bar{b}_{i+1})$ holds. An *abstract counterexample* is a sequence $\bar{b}_0, \ldots, \bar{b}_L$ for which $\hat{Prop}(\bar{b}_L)$ holds. This abstract trace is *concrete* or so called *real counterexample* if there exists a concrete trace corresponding to it. Otherwise, if there are no concrete traces corresponding to this abstract trace, then it is called a *spurious counterexample.*

The abstraction function was defined to map sets of concrete states to sets of abstract states. The *concretization function*, *conc*$(\cdot)$, does the reverse. There is a concrete counterexample trace $\bar{s}_0, \ldots, \bar{s}_L$, where $\bar{s}_i$ corresponds to a valuation of all the k predicates $\phi_1, \ldots, \phi_k$, corresponding to the abstract counterexample trace $\bar{b}_0, \ldots, \bar{b}_L$ if these conditions are satisfied:

- For each $i \in 0, \ldots, L$, $conc(\bar{b}) = \bar{s}$ holds. This means that each concrete state $\bar{s}_i$ corresponds to the abstract state $\bar{b}_i$ in the abstract trace.
- $I(\bar{s}_0) \wedge R(\bar{s}_i, \bar{s}_{i+1}) \wedge \neg Prop(\bar{s}_L)$ holds, for each $i \in 0, \ldots, L-1$. With this, the concrete counterexample of length $L + 1$, starting from initial state, exists.

Thus, with the following formula,

$$\bigwedge_{i=0}^{L-1} (\bigwedge_{j=1}^{k} \phi_j = \bar{s}_i) \wedge \bigwedge_{i=0}^{L-1} R(\bar{s}_i, \bar{s}_{i+1})$$

if it is satisfiable then the abstract counterexample is concrete. Otherwise, it is a spurious counterexample.

5 Spurious Counterexample Analysis for Predicate Refinement

Let $|P|$ be the size of a given program (number of statements) and $|Pred|$ be the number of predicates in the abstraction refinement process. Computation cost of this model is $|P| \cdot 2^{|Pred|}$. It is obvious that the smaller the number of predicates in the abstraction refinement process, the exponential reduction in the cost of abstraction computation and model checking. Also, the size of the program can be considered.

This section describes the proposed method to reduce the size of the abstract model used in abstraction refinement process. The program size $|P_{BB}|$ where $|P_{BB}| \leq |P|$ is considered (instead of number of statements, number of basic blocks are considered). The number of predicates used in the proposed method

is $|Pred_{Proposed}|$ where $|Pred_{Proposed}| \leq |Pred|$. Thus, the computation cost of the proposed method is

$$|P_{BB}| \cdot 2^{|Pred_{Proposed}|} \leq |P| \cdot 2^{|Pred|}$$

Given a spurious counterexample, the proposed abstraction refinement method by analyzing the spurious trace is shown in Algorithm 1. In [7], localization and register sharing methods are used to reduce the number of predicates used in refinement process. Their refinement algorithm performs a backward weakest precondition propagation for each branching condition (`assume` statement) in the infeasible trace.

The abstraction refinement algorithm described in Algorithm 1 performs an analysis by traversing the spurious counterexample from the initial state. First, the spurious trace is traversed and information on the program locations and the assignments of variables are stored. Then, it is traversed again to find the conflict variables which can be represented by a conflict predicate. If this predicate is depending on execution of some program locations, then those program locations are associated to the predicate in the abstract model.

In [7], the number of predicates used depends proportionally on the number of *branching conditions*. In contrast, the number of predicates used in our approach is depending on the number of *conflict predicates*. Back to the example shown in Section 3, the method of [7] needs *four* predicates *plus* extra global constraints to make the relationship of those predicates more precise, while the proposed method needs *only two* predicates with some path dependences. And these conditions of path dependences do not make the abstract model larger because they are already represented the abstract states in the abstract model.

6 Experimental Results

The proposed method for abstraction refinement by counterexample analysis is implemented in SpecC/Synchronization Verification Tool (S-VeT) [14]. S-VeT tool accepts SpecC descriptions as input. In order to compare with BLAST, we generate SpecC wrappers for the C descriptions that used to verify with BLAST. These wrappers just make the program to be able to process by S-VeT tool. They do not introduce additional behaviors to the program.

The experiments are performed on a Pentium4 2.8GHz machine with 2GB memory running Linux. Several experiments are conducted to compare our technique against a public C-based verification tool, BLAST.

As reported in [7], BLAST fails to find the new set of predicates during refinement when applying with the default predicate discovery scheme. Options `craig2` and `predH7` were reported to give best performance. For comparison in this paper, we use the same options. Table 1 shows the comparison of various benchmarks running BLAST against our approach. Size of the program can be defined by the number of branching conditions used. The “TP” column gives the to total number of predicates in the program. “Pred” columns give the maximum number of predicates active at any program location in refinement process. Runtime related columns, “Abs”, “MC”, “Ref” and “Time” denote the execution

Algorithm 1. Abstraction refinement by analyzing counterexample to find conflict predicates and program location dependents

```
Declare
 1: A spurious counterexample s̄_0, ..., s̄_L
 2: Pred is a set of predicates (for simplicity, the subscript _Proposed is omitted)
 3: φ is a predicate that found to make counterexample infeasible
 4: Program locations are defined in term of basic block instead of individual state-
    ments (to make abstract model smaller)
 5: The stored information Store = (PC, Var, Symb) where PC is a set of program
    locations, Var is a set of variables to be assigned, and Symb is a set of symbolic
    values corresponding to the variable in that location
Begin
    /* Initially, store program locations and variable assignments */
 6: for i = 0 to L do
 7:     if s̄_i is an assignment then
 8:         PC_i := program location
 9:         Var_i := the assigned variable (left hand side of the assignment)
10:         Symb_i := symbolic value (right hand side of the assignment)
11:     end if
12: end for
    /* Traverse counterexample to find conflict variables and assign new predicates */

13: for i = 0 to L do /* Outer loop */
14:     if s̄_i is a branching condition then
15:         for j = 0 to i do /* Inner loop #1 */
16:             if variables in s̄_j contains the variable Var_j then
17:                 Propagate-and-substitute all existence of Var_j in s̄_j
18:             end if
19:         end for
20:         Simulate this path s̄_0, ..., s̄_j
21:         if predicate φ presents a conflict in the path is found then
22:             for j = i downto 0 do /* Inner loop #2 */
23:                 Track back to find if variables in φ are depending on any pro-
                    gram location PC_j
24:                 if variables in φ depends on execution of PC_j then
25:                     Associate PC_j to φ when constructing abstract model[1]
26:                 end if
27:             end for
28:             Pred := Pred ∪ φ
29:         end if
30:     end if
31: end for
End
```

[1] Abstract model $M = (PC, \phi)$ and we can check M using NuSMV.

Table 1. Experimental results comparing our approach implemented in S-VeT againt BLAST verification tool

Benchmark	TP	BLAST		S-VeT					Bug
		Pred	Time	Pred	Abs	MC	Ref	Time	
TCAS0	54	13	20.97	12	5	9	10	24	No
TCAS1	111	37	381.20	20	24	58	39	**122**	No
TCAS2	55	13	22.38	13	3	10	10	23	No
TCAS3	63	14	25.84	13	5	12	9	26	Yes
TCAS4	83	23	41.56	13	7	18	13	**39**	Yes
TCAS5	73	20	39.78	15	6	15	7	**28**	No
TCAS6	73	19	32.82	10	3	12	10	**25**	Yes
TCAS7	74	18	33.18	16	4	16	15	36	Yes
TCAS8	61	14	31.99	11	7	11	9	**28**	No
TCAS9	83	20	55.78	14	8	22	17	**38**	Yes
PredAbs1	29	26	1.38	1	0.45	0.97	0.99	2.45	No
PredAbs2	57	52	21.88	1	0.43	1.54	1.46	**3.44**	No
PredAbs3	88	78	140.16	1	0.43	3.46	2.28	**6.19**	No

time in seconds for abstraction, model checking, refinement process, and total runtime, respectively. The pre-process time is omitted.

The benchmarks Traffic Alert and Collision Avoidance System (TCAS) are tested with ten different properties. The benchmarks PredAbs are the fabricated examples used to validate the predicate abstraction refinement process. In all PredAbs benchmarks, only one predicate is sufficient to show that the counterexample is spurious. While our approach can find this predicate directly from analyzing the spurious trace, BLAST needs to interpret a large set of predicates before it knows that this trace is spurious. Although, BLAST is implemented with lazy abstraction technique and the complex refinement scheme based on Craig interpolation method, our approach can outperform by 8 out of the total of 13 benchmarks.

7 Conclusion

Predicate abstraction is a common and efficient technique in software verification domain. However, when the size of the program (number of branching conditions) is large, predicate abstraction suffers from the computation cost that increases exponentially as the number of predicates increases. Choice of predicate selection, scope and number of predicates are major artifacts for performance improvement. In this paper, we described a technique for improving the performance of the abstract refinement loop by analyzing the spurious counterexample. In order to get the smaller set of predicates, we analyze the spurious counterexample to find the conflict predicates. If any predicate is depended on any program location, that program location was associated with that predicate in the abstract model. Our approach is implemented with S-VeT toolkit. Experimental results present the

comparison of our method against BLAST toolkit. The results show that fewer number of predicates can be found with our approach.

Acknowledgement

We would like to thank Himanshu Jain for the Traffic Alert and Collision Avoidance System (TCAS) benchmarks.

References

1. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
2. Clarke, E.M., Grumberg, O., Dill, D.E.: Model checking and abstraction. In: ACM Transactions on Programming Languages and System TOPLAS, vol. 16 (1994)
3. Godefroid, P.: Model checking for programming languages using verisoft. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages (1997)
4. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
5. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
6. Henzinger, T.A., Jhala, R., Mujumdar, R., Sutre, G.: Lazy abstraction. In: ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (2002)
7. Jain, H., Ivancic, F., Gupta, A., Ganai, M.K.: Localization and register sharing for predicate abstraction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
8. Henzinger, T.A., Jhala, R., Mujumdar, R., McMillan, K.L.: Abstractions from proofs. In (POPL 2004). Proc. of the 31st Annual Symposium on Principles of Programming Languages (2004)
9. Henzinger, T.A., Jhala, R., Mujumdar, R., Sutre, G.: Lazy abstraction. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, Springer, Heidelberg (2003)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
11. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In (POPL 1995). Principles of Programming Languages (1995)
12. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. In: Proceeding of the Model Checking for Dependable Software-Intensive Systems Workshop (2003)
13. Clarke, E.M., Jain, H., Kroening, D.: Verification of SpecC using predicate abstraction. In: Second ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE 2004) (2004)
14. Sakunkonchak, T., Komatsu, S., Fujita, M.: Synchronization verification in system-level design with ILP solvers. In: Third ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2005) (2005)

Author Index

Abadi, Martín 1
Alizadeh, Bijan 129

Bauer, Jörg 35
Berthomieu, Bernard 523
Bošnački, Dragan 300
Boudali, Hichem 441
Bozzano, Marco 162
Bruttomesso, Roberto 237

Cassez, Franck 192
Cimatti, Alessandro 162, 237
Crouzen, Pepijn 441

Damm, Werner 425
David, Alexandre 192
Dax, Christian 223
Disch, Stefan 425
Donaldson, Alastair F. 300

Eisinger, Jochen 223
Elkind, Edith 378

Finkbeiner, Bernd 268, 474
Fujita, Masahiro 129, 553

Gates, Ann Q. 533
Gawlitza, Thomas 177
Geeraerts, Gilles 98
Genest, Blaise 378
Grumberg, Orna 410, 457

Habermehl, Peter 145
Haddad, Serge 362
Han, Tingting 331
Hasegawa, Atsushi 15
Hermanns, Holger 207
Hu, Alan J. 237
Huang, Geng-Dian 51
Hungar, Hardi 425

Iosif, Radu 145

Jacobs, Swen 425
Johnson, Jordan 1
Jones, Kevin 114

Katoen, Joost-Pieter 331
Kim, Moonzoo 489
Klaedtke, Felix 223
Komatsu, Satoshi 553
Kreinovich, Vladik 533
Kupferman, Orna 316

Larsen, Kim G. 192
Leuschel, Michael 300
Li, Guoqiang 511
Lime, Didier 192
Little, Scott 66, 114
Lustig, Yoad 316

Massart, Thierry 300
McMillan, K.L. 17
Merayo, Mercedes G. 501
Misra, Janardan 284
Mooij, Arjan 347
Myers, Chris 66, 114

Nain, Sumit 19
Núñez, Manuel 394, 501

Ogawa, Mizuhito 511
Oshman, Rotem 410

Pace, Gordon 82
Pang, Jun 425
Peled, Doron 378
Peres, Florent 523
Pigorsch, Florian 425
Pinchinat, Sophie 253
Prisacariu, Cristian 82

Rakamarić, Zvonimir 237
Raskin, Jean-François 98, 192
Recalde, Laura 362
Roach, Steve 533
Rodríguez, Ismael 394, 501
Rogalewicz, Adam 145
Romijn, Judi 347
Roy, Suman 284

Saha, Indranil 284
Sakunkonchak, Thanyapat 553

Salamah, Salamah 533
Schewe, Sven 268, 474
Schneider, Gerardo 82
Scholl, Christoph 425
Schuster, Assaf 457
Seidl, Helmut 177
Silva, Manuel 362
Spoletini, Paola 378
Stoelinga, Mariëlle 441

Tapparo, Francesco 162
Toben, Tobe 35
Torabi Dashti, Mohammad 543

Van Begin, Laurent 98
Vardi, Moshe Y. 19
Vernadat, François 523
Vojnar, Tomáš 145

Waldmann, Uwe 425
Walter, David 66, 114
Wang, Bow-Yaw 51
Wesselink, Wieger 347
Westphal, Bernd 35
Whitehead, Nathan 1
Wijs, Anton J. 543
Wirtz, Boris 425

Yadgar, Avi 457

Zhang, Lijun 207