

Klaus Havelund
Rupak Majumdar
Jens Palsberg (Eds.)

LNCS 5156

Model Checking Software

15th International SPIN Workshop
Los Angeles, CA, USA, August 2008
Proceedings

 Springer

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

Alfred Kobsa

University of California, Irvine, CA, 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

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Klaus Havelund Rupak Majumdar
Jens Palsberg (Eds.)

Model Checking Software

15th International SPIN Workshop
Los Angeles, CA, USA, August 10-12, 2008
Proceedings



Springer

Volume Editors

Klaus Havelund
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, USA
E-mail: klaus.havelund@jpl.nasa.gov

Rupak Majumdar
Department of Computer Science
University of California
Los Angeles, CA, USA
E-mail: rupak@cs.ucla.edu

Jens Palsberg
Department of Computer Science
University of California
Los Angeles, CA, USA
E-mail: palsberg@ucla.edu

Library of Congress Control Number: 2008931715

CR Subject Classification (1998): F.3, D.2.4, D.3.1, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-85113-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85113-4 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 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12443697 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 15th International SPIN Workshop on Model Checking of Software (SPIN 2008), which took place at the University of California, Los Angeles, August 10–12, 2008. The SPIN workshops form a forum for researchers and practitioners interested in model checking techniques for the verification and validation of software systems. Model checking is the process of checking whether a given structure is a model of a given logical formula. The structure normally represents a set of tasks executing in parallel in an interleaved fashion, resulting in a non-deterministic set of executions. The main focus of the workshop series is software systems, including models and programs. Subjects of interest include theoretical and algorithmic foundations as well as tools for software model checking. The workshop in addition aims to foster interactions and exchanges of ideas with related areas in software engineering, such as static analysis, dynamic analysis, and testing.

There were 41 submissions, including 38 full papers and 3 tool papers. Each submission was reviewed by at least three Programme Committee members. The committee decided to accept 18 papers, including 17 regular papers and 1 tool paper. The programme also included five invited talks (in alphabetical order): Matthew Dwyer (University of Nebraska) “Residual Checking of Safety Properties”, Daniel Jackson (MIT) “Patterns of Software Modelling: From Classic To Funky”, Shaz Qadeer (Microsoft Research) “The Case for Context-Bounded Verification of Concurrent Programs”, Wolfram Schulte (Microsoft Research) “Using Dynamic Symbolic Execution to Improve Deductive Verification”, and Yannis Smaragdakis (University of Oregon) “Combining Static and Dynamic Reasoning for the Discovery of Program Properties”.

We would like to thank the authors of submitted papers, the invited speakers, the Programme Committee members, the additional reviewers, and the Steering Committee for their help in composing a strong programme. We thank Springer for having agreed to publish these proceedings as a volume of *Lecture Notes in Computer Science*. The EasyChair conference system was used for the entire management of paper submissions for the workshop. This includes paper submission, reviewing, and final generation of the proceedings.

June 2008

Klaus Havelund
Rupak Majumdar
Jens Palsberg

Conference Organization

General Chair

Jens Palsberg UC Los Angeles, USA

Program Chairs

Klaus Havelund NASA JPL/Caltech, USA
Rupak Majumdar UC Los Angeles, USA

Program Committee

Christel Baier Bonn, Germany
Dragan Bošnački Eindhoven, Netherlands
Lubos Brim Brno, Czech
Stefan Edelkamp Dortmund, Germany
Dawson Engler Stanford, USA
Kousha Etessami Edinburgh, UK
Susanne Graf Verimag, France
John Hatcliff Kansas State University, USA
Gerard Holzmann NASA JPL/Caltech, USA
Franjo Ivančić NEC, USA
Sarfraz Khurshid UT Austin, USA
Kim Guldstrand Larsen Aalborg, Denmark
Madan Musuvathi Microsoft, USA
Joel Ouaknine Oxford, UK
Corina Pasareanu NASA Ames, USA
Doron Peled Warwick, UK
Paul Pettersson Uppsala, Sweden
Koushik Sen Berkeley, USA
Natasha Sharygina Lugano, Switzerland
Eran Yahav IBM, USA

Steering Committee

Dragan Bošnački Eindhoven, Netherlands
Stefan Edelkamp Dortmund, Germany
Susanne Graf Verimag, France
Stefan Leue Konstanz, Germany
Antti Valmari Tampere, Finland
Pierre Wolper Liege, Belgium

Advisory Committee

Gerard Holzmann	NASA JPL/Caltech, USA
Amir Pnueli	Weizmann Institute, Israel
Moshe Vardi	Rice University, USA

External Reviewers

Jiri Barnat	Cristina Seceleanu
Roberto Bruttomesso	Ohad Shacham
Jan Carlson	Nishant Sinha
Jyotirmoy Deshmukh	Damian Sulewski
Michael Emmi	Ashish Tiwari
Jeffrey Fischer	Oksana Tkachuk
Malay Ganai	Stefano Tonetta
Pierre Ganty	Simon Tschirner
John Håkansson	Aliaksei Tsitovich
Pallavi Joshi	Martin Vechev
Peter Kissmann	Helmut Veith
Shuhao Li	Chao Wang
Edgar Pek	Ru-Gang Xu
Noam Rinetzky	Greta Yorsh
Andrey Rybalchenko	Damian Sulewski
Sriram Sankaranarayanan	

Table of Contents

Invited Contributions

Residual Checking of Safety Properties	1
<i>Matthew B. Dwyer and Rahul Purandare</i>	
The Case for Context-Bounded Verification of Concurrent Programs	3
<i>Shaz Qadeer</i>	
Combining Static and Dynamic Reasoning for the Discovery of Program Properties	7
<i>Yannis Smaragdakis</i>	
Using Dynamic Symbolic Execution to Improve Deductive Verification	9
<i>Dries Vanoverberghe, Nikolaj Bjørner, Jonathan de Halleux, Wolfram Schulte, and Nikolai Tillmann</i>	

Regular Papers

Automated Evaluation of Secure Route Discovery in MANET Protocols	26
<i>Todd R. Andel and Alec Yasinsac</i>	
Model Checking Abstract Components within Concrete Software Environments	42
<i>Tonglaga Bao and Mike Jones</i>	
Generating Compact MTBDD-Representations from Probmela Specifications	60
<i>Frank Ciesinski, Christel Baier, Marcus Größer, and David Parker</i>	
Dynamic Delayed Duplicate Detection for External Memory Model Checking	77
<i>Sami Evangelista</i>	
State Focusing: Lazy Abstraction for the Mu-Calculus	95
<i>Harald Fecher and Sharon Shoham</i>	
Efficient Modeling of Concurrent Systems in BMC	114
<i>Malay K. Ganai and Aarti Gupta</i>	
Tackling Large Verification Problems with the Swarm Tool	134
<i>Gerard J. Holzmann, Rajeev Joshi, and Alex Groce</i>	

Formal Verification of a Flash Memory Device Driver – An Experience Report	144
<i>Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim</i>	
Layered Duplicate Detection in External-Memory Model Checking	160
<i>Peter Lamborn and Eric A. Hansen</i>	
Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes	176
<i>Stefan Leue, Alin Ștefănescu, and Wei Wei</i>	
Improved On-the-Fly Equivalence Checking Using Boolean Equation Systems	196
<i>Radu Mateescu and Emilie Oudot</i>	
Resource-Aware Verification Using Randomized Exploration of Large State Spaces	214
<i>Nazha Abed, Stavros Tripakis, and Jean-Marc Vincent</i>	
Incremental Hashing for SPIN	232
<i>Viet Yen Nguyen and Theo C. Ruys</i>	
Verifying Compiler Based Refinement of Bluespec™ Specifications Using the SPIN Model Checker	250
<i>Gaurav Singh and Sandeep K. Shukla</i>	
Symbolic Context-Bounded Analysis of Multithreaded Java Programs	270
<i>Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon</i>	
Efficient Stateful Dynamic Partial Order Reduction	288
<i>Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby</i>	
Symbolic String Verification: An Automata-Based Approach	306
<i>Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra</i>	
Verifying Multi-threaded C Programs with SPIN	325
<i>Anna Zaks and Rajeev Joshi</i>	
Author Index	343

Residual Checking of Safety Properties

Matthew B. Dwyer and Rahul Purandare

Department of Computer Science and Engineering University of Nebraska-Lincoln
{dwyer, rpuranda}@cse.unl.edu

Abstract. Program analysis and verification techniques have made great strides, yet, as every researcher in the field will admit it is easy to find a program and property for which a given technique is not cost-effective. Investigating the conventional wisdom that programs are *mostly correct*, we have observed that even failed program analyses usually produce a wealth of information about the parts of the program that operate correctly. Leveraging this information can help focus subsequent analysis and verification activities to make them more cost-effective.

1 The Limits of Verification

There have been great advances in static program analysis and verification techniques over the past two decades. These techniques have scaled to increasingly large and complex programs, significantly improved the utility and precision of analysis results, and targeted more expressive correctness property specifications. Despite these advances, there remain very real practical limits to the application of those techniques. Given nearly any program analysis or verification technique, it is relatively easy to find a program and correctness property for which it is either intractable or uselessly imprecise.

There is a broadly held belief that, with sufficient investments, program analysis and verification techniques will continue to improve over the next decade [1], but these techniques will be applied to software of increasing size and complexity. Consequently, it is likely that a *practical* limit on the applicability of state-of-the-art program analysis and verification techniques will exist for some time.

2 Prove What You Can and Focus on the Remainder

When the limit of a program verification technique is encountered an inconclusive result will be produced. Partial results may have been computed, if scalability was the obstacle, or false error reports may have been produced, if precision was the obstacle. In either case, it is sensible to ask whether the information that was calculated is of any value in focusing further program analysis or testing. We believe that this information can be extremely valuable.

The conventional wisdom that underlies software fault tolerance [2] and mutation testing [3] approaches asserts that well-developed programs are *mostly correct*. If this is the case, then it may be possible to infer that non-trivial portions of a program are *correct* by analyzing the intermediate results of a failed

program analysis or verification run. Our recent work [4], which considers static tpestate analysis [5], and recent work by Lal et al. [6], which considers push-down model checking, demonstrate this to be the case. The techniques differ in their technical details, but both calculate sets of program locations that never participate in violations of the analyzed property; we refer to the remaining program locations as the analysis *residue*.

The residue can be used for multiple purposes. A subsequent verification or analysis could focus solely on those statements by, for example, customizing its abstractions based on those statements. A test generation method could target those statements. In our work [4], we exploit this information to eliminate instrumentation for statements that are not in the residue to significantly reduce the cost of run-time monitoring of tpestate properties.

Classic examples of staged analyses, such as those used for array dependence testing [7], have illustrated the advantages of focusing on the analysis residue for state properties. We believe that there is significant potential for staging multiple forms of program analyses and verification approaches that are focused on safety properties.

Acknowledgments

This work was supported in part by the Army Research Office through DURIP award W91NF-04-1-0104, and by the National Science Foundation through CSR award 0720654, CCF awards 0429149 and 0541263, and CRI award 0454203.

References

1. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50, 63–69 (2003)
2. Randell, B., Lee, P.A., Treleaven, P.C.: Reliability issues in computing system design. ACM Comput. Surv. 10, 123–165 (1978)
3. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11, 34–41 (1978)
4. Dwyer, M.B., Purandare, R.: Residual dynamic tpestate analysis: exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA, November 5–9, 2007, pp. 124–133 (2007)
5. Strom, R.E., Yemini, S.: Tpestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12, 157–171 (1986)
6. Lal, A., Kidd, N., Reps, T.W., Touili, T.: Abstract error projection. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 200–217. Springer, Heidelberg (2007)
7. Wolfe, M.: High performance compilers for parallel computing. Addison-Wesley, Reading (1996)

The Case for Context-Bounded Verification of Concurrent Programs

Shaz Qadeer

Microsoft Research

Concurrent programs are difficult to get right. Subtle interactions among communicating threads in the program can result in behaviors unexpected to the programmer. These behaviors typically result in bugs that occur late in the software development cycle or even after the software is released. Such bugs are difficult to reproduce and difficult to debug. As a result, they have a huge adverse impact on the productivity of programmers and the cost of software development. Therefore, tools that can help detect and debug concurrency errors will likely provide a significant boost to software productivity.

The problem of automatic and precise defect-detection for programs, whether concurrent or sequential, is undecidable in general. The importance of this problem has led researchers, over the past 50 years, to devise various techniques to circumvent this undecidability barrier. An important breakthrough in this direction is the idea of analyzing models of a program rather than the program itself. The fundamental insight is that although the set of behaviors of a concrete program might be insurmountably large, there is usually a simple abstract model of the program that contains enough information to prove that the program satisfies a particular partial specification. If the problem of analyzing the abstract model is decidable, then analysis of the model could help in pinpointing defects in the program. Of course, the model itself cannot be created automatically in general; algorithms for creating these models typically require input from the programmer.

Two models have been widely and successfully used for defect detection in sequential programs—finite-state machines and pushdown machines. For these models, the basic analysis problem, known as the *reachability* problem, is the following:

Given an error state e , is there an execution of the machine from the initial state to e ?

For both models, there are polynomial-time algorithms for solving the reachability problem; the complexity is linear for finite-state machines and cubic for pushdown machines. A big reason for the success of these models in software verification is the relatively low complexity of reachability analysis for them. Unfortunately, the same modeling tools have not worked so well for concurrent programs. The natural extensions of these models to handle concurrency are communicating finite-state machines and communicating pushdown machines

respectively. While reachability analysis is PSPACE-complete for communicating finite-state machines, it is undecidable for communicating pushdown machines. Consequently, the use of abstract models for verifying concurrent programs has seen limited success.

This position paper argues that the high complexity of reachability analysis for models of concurrent computation can be mitigated by performing context-bounded verification of these models. To understand the idea of context-bounded verification, we take a closer look at concurrent behaviors. We model a concurrent computation as an interleaving of actions performed by tasks that are concurrently executing and communicating using shared resources. A context switch occurs in such a computation whenever the execution of a task is temporarily interrupted by some other task. The *context-bounded reachability* problem, for both communicating finite-state machines and communicating pushdown machines, is the following:

Given an error state e and a bound $c \geq 0$, is there an execution of the machine from the initial state to e with no more than c context switches?

Note that context-bounded reachability is significantly different from depth-bounded reachability, where executions are restricted to some length $d \geq 0$. Bounding the number of context switches in an execution does not bound the length of the execution because a task may perform an unbounded number of steps prior to a context switch.

Bounding the number of context switches has a significant impact on the complexity of the reachability problem; context-bounded reachability is NP-complete both for communicating finite-state machines and communicating pushdown machines [53]. Since the problem is NP-complete, it is unlikely that there is a polynomial-time algorithm for it. However, there is reason to be optimistic. In the last decade, significant advances have been made in satisfiability solving. Researchers have built a number of powerful satisfiability solvers that have managed to solve practical and real-world instances of hardware and software verification problems. Since context-bounded reachability is in NP, it can be translated to satisfiability thereby allowing these powerful solvers to be leveraged. In addition, there exist algorithms for solving context-bounded reachability that are polynomial in the size of the model and exponential in c [52]. These algorithms could be used for scalable verification, at least for small values of c .

If an oracle for the context-bounded reachability problem returns the answer **No** for a particular bound c , then we are assured that the error state is unreachable via executions with up to c context switches. However, this answer says nothing about executions with more than c context switches. To decrease the likelihood of missed errors, we could start with a small value of c (zero, for example) and keep incrementing it as long as the oracle returns the answer **No**, until the validation resources allocated to the program are exhausted. There are two advantages of this pay-as-you-go approach to verification. First, successful context-bounded verification for a bound c is a useful and intuitive coverage

metric suitable for concurrent programs and orthogonal to sequential coverage metrics such as line or branch coverage. Second, we believe that the number of context switches in an execution is a good metric of the complexity of that execution. Hence, an erroneous execution returned by this iterative approach is one of the simplest witnesses to the error and could help the programmer localize the cause of the error quickly.

Obviously, the pay-as-you-go approach to verification of concurrent programs is just as easily facilitated by depth-bounding as by context-bounding. However, depth-bounding lacks the two aforementioned advantages of context-bounding because the depth of explored executions is not an intuitive metric for the complexity of concurrent executions.

The above discussion provides motivation for context-bounded verification from the point of view of computational complexity. However, our argument would rightly be considered weak if errors in concurrent programs manifested only in executions with a large number of context switches. Fortunately, there is significant empirical evidence indicating that is not the case. Over the past few years, we have implemented context-bounded reachability analysis in three software model checkers—KISS [6], ZING [1], and CHESS [4]. We have applied these tools to many real-world concurrent programs and discovered numerous errors exhibited by executions with a small number of context switches.

Context-bounding is a novel, interesting, and useful perspective on the problem of verifying concurrent systems. Recent work has provided both theoretical and practical evidence of the power of context-bounded verification. However, there are many important challenges ahead. First, to apply context-bounded verification to a concurrent program requires the identification of program tasks. In many programs, such as multithreaded programs manipulating shared data-structures, the task abstraction is obvious and corresponds to the syntactic abstraction of a thread. However, for message-passing or event-driven programs, the task abstraction is not easily discerned. Consequently, we need linguistic techniques to specify and analysis techniques to discover tasks in concurrent programs. Second, to make use of satisfiability solvers we need efficient encodings of the context-bounded reachability problem into the satisfiability problem. Finally, we need techniques to construct concurrent finite-state and pushdown models from software implementations; these techniques must interact well with the encoding techniques.

References

1. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. Technical Report MSR-TR-2004-10, Microsoft Research (2004)
2. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: CAV 2008: Computer Aided Verification (2008)
3. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems (2008)

4. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI 2007: Programming Language Design and Implementation, pp. 446–455 (2007)
5. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
6. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI 2004: Programming Language Design and Implementation, pp. 14–24. ACM Press, New York (2004)

Combining Static and Dynamic Reasoning for the Discovery of Program Properties

Yannis Smaragdakis

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403-1202, USA
yannis@cs.uoregon.edu

Abstract. Combinations of static and dynamic analyses can be profitably employed in tasks such as program understanding, bug detection, behavior discovery, etc. In the past several years, we have explored a particular scheme for improving the quality of bug reports in a sequence of tools: *JCrasher*, *Check 'n' Crash*, and *DSD-Crasher*. We have additionally explored the combination of dynamic and symbolic execution for the purpose of inferring program invariants in the *DySy* tool. In this talk, we discuss such approaches, while distinguishing the conceptual benefits and drawbacks of each approach from the abilities and shortcomings of the current representative tools.

The idea of combining static and dynamic program analysis has been recognized as highly promising and explored by researchers for over two decades. The distinction of “static” and “dynamic” analyses is itself fairly fluid, with researchers often using the terms to refer to techniques (e.g., symbolic execution vs. concrete execution) rather than to whether the analysis depends on dynamic input. We generally refer to an analysis as “static” if it emphasizes data-flow richness/generality over control-flow accuracy and “dynamic” if it emphasizes control-flow accuracy over data-flow richness/generality.

We have used combinations of static and dynamic analyses over several years for the purpose of discovering program properties. This work produced a sequence of progressively more complex tools for finding defects (bugs) in Java programs: *JCrasher* [1], *Check 'n' Crash* [2], and *DSD-Crasher* [3, 4]. Some of these tools have been widely used in the research community. A major feature of our approach is an emphasis on reducing false bug warnings, rather than on enabling more bug reports. This emphasis informs many of the design and implementation choices in these tools.

Specifically, we use a static analysis based on symbolic reasoning as a central piece of our approach. Program effects are represented as logic sentences, and a reasoning engine attempts to discover all violations of correctness properties by producing the full space of preconditions for each undesirable state. Left on its own, this analysis would produce false error reports of two different kinds. First, some reports would be unsound: they do not correspond to reproducible bugs but are due to the abstraction techniques of our symbolic reasoning. For

instance, if our symbolic reasoning engine cannot establish true properties of arithmetic expressions, it is likely to report errors where none exist. A second kind of false error report is due to reasoning about a program (or a program module) in isolation, without taking into account conditions established by its environment. In order to filter out these two kinds of false error reports, we employ two different dynamic analysis approaches. To address the first kind of false report, we attempt to produce concrete examples demonstrating each reported error. The examples are then executed and we dynamically observe if the predicted violation truly occurs, in which case it is reported to the user. To address the second kind of false report, a *dynamic invariant inference* step tries to “read the programmer’s mind” and establish preconditions of a program or a module based on existing regression test inputs. This directs the static analysis to only report problems consistent with the preconditions.

This chain of analyses is promising and our current tool instantiation is an early but far from perfect representative of the approach. Several conceptual research problems arise in this process. Some correspond to established research directions (e.g., counterexample generation, general constraint solving). Others have to do with addressing the impedance mismatch between analyses (e.g., inferred invariants often need to satisfy consistency properties, such as behavioral subtyping, or they result in contradictions when supplied as facts to a symbolic reasoning engine). In the talk, we discuss in detail one specific research direction, which also represents our latest work: The use of symbolic execution in combination with dynamic execution of a program, for the purpose of invariant inference, as represented by the DySy tool [5].

References

- [1] Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34(11), 1025–1050 (2004)
- [2] Csallner, C., Smaragdakis, Y.: Check ‘n’ Crash: Combining static checking and testing. In: *Proc. 27th International Conference on Software Engineering (ICSE)*, May 2005, pp. 422–431. ACM, New York (2005)
- [3] Csallner, C., Smaragdakis, Y.: DSD-Crasher: A hybrid analysis tool for bug finding. In: *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2006, pp. 245–254. ACM, New York (2006)
- [4] Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodologies (TOSEM)* 17(2) (April 2008)
- [5] Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference. In: *International Conference on Software Engineering (ICSE)* (May 2008)

Using Dynamic Symbolic Execution to Improve Deductive Verification

Dries Vanoverberghe, Nikolaj Bjørner, Jonathan de Halleux, Wolfram Schulte,
and Nikolai Tillmann

Microsoft Research

One Microsoft Way, Redmond WA 98052, USA

{t-drivan*, nbjorner, jhalleux, schulte, nikolait}@microsoft.com

Abstract. One of the most challenging problems in deductive program verification is to find inductive program invariants typically expressed using quantifiers. With strong-enough invariants, existing provers can often prove that a program satisfies its specification. However, provers by themselves do not find such invariants. We propose to automatically generate executable test cases from failed proof attempts using dynamic symbolic execution by exploring program code as well as contracts with quantifiers. A developer can analyze the test cases with a traditional debugger to determine the cause of the error; the developer may then correct the program or the contracts and repeat the process.

1 Introduction

Many modern specification and verification systems such as Spec# [3], JML [25] and so forth, use a design-by-contract approach [27], where the specification language is typically an extension of the underlying programming language. The verification of these contracts often uses verification condition generation (VCG). The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas are then fed to interactive or automatic theorem provers.

In practice, there are two limitations of this VCG and proving approach. First, most program verification tools do not by themselves produce sufficiently strong contracts. Such contracts need to be crafted or synthesized independently, usually by a human being. Second, if the program verification tool fails to prove the desired properties, then discovering the mismatch between the contracts and the code, or why the contracts were not strong enough, remains a difficult task. In practice, the program verification tools offer only little help.

Most of the above mentioned program verification tools employ an automated solver to prove program properties. These solvers must typically be able to handle a combination of domains, such as integers, bit-vectors, arrays, heaps, and data-types, which are often found in programming languages. In addition most interesting contracts involving functional correctness and the heap involve quantifiers, which solvers must reason about, as well. Solvers, that combine various theories are called SMT (Satisfiability Modulo Theories). Such solvers have recently gained a lot of attention, see for instance

* Permanent email address: Dries.Vanoverberghe@cs.kuleuven.be

Simplify [16], CVC3 [4], Fx7 [28], Verifun [19], Yices [18], and Z3 [15]. To support quantifiers, a commonly used technique for SMT solvers is to use pattern matching to determine relevant quantifier instantiations. Pattern matching happens inside of the solver on top of the generated verification condition. When a proof attempt fails, pinpointing the insufficient annotation within the context of the SMT solver is obscured by the indirection from the program itself. To give good feedback to the developer in such a case, the SMT solver should provide a human-readable counter-example, i.e. a model of the failed proof attempt. However, producing (informative) models from quantified formulas remains an open research challenge. Producing models for quantifier-free formulas, is on the other hand easy and relatively well understood.

Symbolic execution [24] is a well-known technique to generate test cases. In particular, test cases that expose errors help a developer in debugging problems. Symbolic execution analyzes individual execution traces, where each trace is characterized by a path condition, which describes an equivalence class of test inputs. A constraint solver is used to decide the feasibility of path conditions, and to obtain concrete test inputs as representatives of individual execution paths. Note that constraints can also be solved by SMT solvers with model generation capabilities. Recently symbolic execution has been extended to deal with contracts, even contracts involving quantifiers over (sufficiently small) finite domains [30].

In this paper, we propose an extension of symbolic execution for programs involving contracts with quantifiers over very large, and potentially unbounded domains. This is of benefit for debugging failed proof attempts. If a deductive proof fails due to insufficient quantified assertions, we use symbolic execution to generate concrete test cases that exhibit mismatches between the program and its contracts with quantifiers. Quantifiers are furthermore instantiated using symbolic values encountered during a set of exhibited runs. In this setting, quantifier instantiation is limited to values supplied to or produced by a symbolic execution. With a sufficient set of instances, we can derive test cases that directly witness limitations of the auxiliary assertions. The SMT solver no longer needs to handle these quantifiers.

In particular, we handle branch conditions with quantifiers as follows: When a branch condition of the program involves an unbounded universal quantifier, we first recover the quantified formula $\phi(x)$ (which must be embedded by the compiler into the code), we introduce a Boolean variable t that represents whether the quantifier holds, and we introduce a branch in the program over t . Conceptually, program execution forks at this branch. If $t = \text{false}$, we introduce another additional test input c that represents the bound variable, and explore the quantified formula until we find a c such that $\neg\phi(c)$, If $t = \text{true}$, then we proceed with the program. Quantifier instantiations are identified lazily using pattern matching over the symbolic trace.

An extended form of pattern matching, called E-matching [12], is used for instantiating quantifiers in SMT solvers. E-matching admits matching modulo a set of equalities. We lift E-matching to combine run-time information with symbolic execution. To this end, we use *dynamic* symbolic execution, which executes the program for particular test inputs in order to obtain derived run-time values. Run-time values are used to determine which symbolic terms *may* be equal. Thus, pattern matching is performed on the

Algorithm 2.1. Dynamic symbolic execution

Set $J := false$ loop Choose program input i such that $\neg J(i)$ Output i Execute $P(i)$; record path condition C Set $J := J \vee C$ end loop	<i>intuitively, J is the set of already... ...analyzed program inputs stop if no such i can be found in particular, $C(i)$ holds</i>
---	--

symbolic trace, where concrete run-time values are used to supply a set of alternative views of values appearing in a trace.

We implemented a prototype of our approach as an extension to the dynamic symbolic execution platform Pex [34,32] for .NET, which we develop at Microsoft Research. Pex contains a complete symbolic interpreter for safe programs that run in the .NET virtual machine. Pex uses Z3 [15,14] as a constraint solver, using Z3’s ability to compute models for satisfiable constraint systems. Pex has been used within Microsoft to test core .NET components developed at Microsoft. Pex is integrated with Microsoft Visual Studio.

The rest of the paper is structured as follows: Section 2 gives an overview of dynamic symbolic execution. Section 3 walks through a simple example of how our approach generates test cases for a program with contracts involving quantifiers. Section 4 describes in detail how we extend dynamic symbolic execution to handle quantifiers. Section 5 discusses related work. Section 6 concludes.

2 Dynamic Symbolic Execution

2.1 Introduction

Dynamic symbolic execution [22,5] is a variation of conventional static symbolic execution [24]. Dynamic symbolic execution consists in executing the program, starting with arbitrary inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. Then a constraint solver is used to compute variations of the previous inputs in order to steer future program executions along different execution paths. In this way, all execution paths will be exercised eventually.

Algorithm 2.1 shows the general dynamic symbolic execution algorithm.

The advantage of dynamic symbolic execution over static symbolic execution is that the abstraction of execution paths can leverage observations from concrete executions, and not all operations must be expressed and reasoned about symbolically. Using concrete observations for some values instead of fully symbolic representations leads to an under-approximation of the set of feasible execution paths, which is appropriate for testing. Such cases can be detected, e.g. when a function is called that is defined outside of the scope of the analysis. Our tool reports them to the user.

2.2 Symbolic State Representation

In concrete execution, the program's state is given by a mapping from program variables to concrete values, such as 32 or 64-bit integers and heap-allocated (object) pointers. In symbolic execution, the state is a mapping from program variables to terms built over symbolic input values, together with a predicate over symbolic input values, the so-called path condition.

The terms represent symbolically which computations were performed over the symbolic inputs. For example, if x, y, z are the symbolic inputs for the program variables x, y, z , then the statement

```
u = x * (y + z);
```

causes the program variable u to be mapped to the term $x * (y + z)$. If the concrete inputs for x, y, z are 2, 3, and 4, respectively, then the concrete value of u will be 14.

The path condition is the conjunction of all the guards of all conditional branches that were performed to reach a particular point in an execution trace. For example, when the function

```
void foo(int x) {
  if (x>0) {
    int y = x*x;
    if (y==0) {
      // target
    }
  }
}
```

reaches the *target*, then the path condition consists of two conjuncts, $(x > 0)$ and $(x * x == 0)$. Note that the symbolic state is always expressed in terms of the symbolic input values, that is why the value of the local variable y was expressed with the symbolic input x .

2.3 Test Inputs and Non-deterministic Programs

Dynamic symbolic execution determines a set of concrete test inputs for a program. In practice, this means that we determine parameter values for a designed top-level function. In addition to the immediate parameter values, our dynamic symbolic execution platform Pex [34] allows the code-under-test to call the generic function `Choose` in order to obtain an additional test input. In C# syntax, the function has the following signature:

```
T Choose<T> ();
```

Each invocation of this function along an execution trace provides the program with a distinct additional symbolic test input. In the following, we will also refer to the functions `ChooseTruth` and `ChooseBoundVariable`. They work just as `Choose`, and their main purpose is to easily distinguish between different choices.

2.4 Making Basic Contracts Executable

Most design-by-contract languages support function pre-conditions and post-conditions, class invariants and loop invariants. In the following, we describe how most contracts

Program 2.1. Implementation of Assume and Assert in C# syntax

```

void Assume(bool b) {
    if (!b) throw new AssumptionException();
}
void Assert(bool b) {
    if (!b) throw new AssertionException();
}

```

can be turned into executable code using `Assert` and `Assume`. This code can then be explored by symbolic execution.

As shown in Program 2.1, the `Assume` and `Assert` functions contain a conditional branch over their Boolean parameter, and they throw an exception when the argument is false. An `AssumptionException` is treated by the symbolic execution engine as a filter: test inputs that cause this exception to be thrown are not shown to the user. An `AssertionException` indicates an error, since a mismatch between the program under test and its contracts has been found. Test inputs that cause this exception are shown to the user.

We reduce a class invariant to both a pre- and post-condition of each affected function (see e.g. [27]), and a loop invariant to a call to an `Assert` function with the positive condition at the loop entry and with the negative condition at the loop exit. Given a designated top-level function to explore, we turn its pre-conditions into calls to the `Assume` function, and all pre-conditions of called functions into calls of the `Assert` function which are placed at the beginning of the functions. We turn all post-conditions into calls to the `Assert` function which are placed at the end of the function. We discuss the treatment of universal quantifiers in depth in Section 4.

3 Example

Program 3.1 shows an implementation of a swap function. In this example we use C# syntax extended with pre-conditions and post-conditions (`requires` and `ensures`), and `old` expressions.¹ It has two pre-conditions: the array `a` must not be null, and the indices must be within the bounds of the array. The three post-conditions express that the elements at index `lo` and `hi` in the array `a` are swapped, and that all remaining elements of the array are identical to the old elements in the array. (Note that the last `if` statement introduces an error into program.)

Program 3.2 shows the translation of the `swap` function, including its pre and post-condition as described in 2.4. The `old` expression is realized by creating a copy of the referenced values in the initial state. The `Forall<int>(i => p(i))` expression refers to a generic function `Forall` that takes a predicate expression `p(i)` as an argument. Intuitively, it represents $\forall i.p(i)$. shows the translated program.

¹ In fact, the syntax we use is Spec# [3], except for our `Forall` function that does not involve bounds. In contrast, the universal quantifier in Spec# must state a finite enumeration of possible values for the bound variable.

Program 3.1. Swap Example

```

public void Swap(byte[] a, int lo, int hi)
    requires a != null;
    requires 0 <= lo && lo <= hi && hi < a.Length;

    ensures a[hi] == old(a[lo]);
    ensures a[lo] == old(a[hi]);
    ensures Forall<int>(i =>
        !(i >= 0 && i < a.Length && i !=lo && i !=hi)
        || a[i] == old(a[i]));
{
    byte tmp = a[hi];
    a[hi] = a[lo];
    a[lo] = tmp;
    if (lo != 0 && hi != 0)
        a[0] = 42;
}

```

When symbolic execution of this program reaches the `Assert(Forall(...))` statement, our treatment of the quantifier (that we describe in detail in Section 4) will consider the case in which the quantifier does not hold. To this end, our technique explores the body of the quantifier using symbolic execution with the intention to find test inputs that make the asserted quantifier true and false.

Case 1: Let’s assume the quantifier does not hold. The following code snippet represents a test case that was generated during the search.

```

IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(false)
    .ChooseBoundVariable(1610612732);
byte[] bs0 = new byte[2];
TestSwap2(bs0, 1, 1);

```

In this code, `oracle` is initialized by a call to `OnComprehension`, which indicates which quantifier is about to be initialized (here, 0 indicates that it is the first quantifier in the execution trace). Calls to `ChooseTruth` and `ChooseBoundVariable`, set the truth value of the quantifier and the value of the bound variable (in this case, the truth value `false` indicates that the quantifier should not hold, and the bound variable `i` gets the value 1610612732). The argument for the parameter `a` is an array of size 2, so the index `i`, chosen earlier, is outside of the range of the array. With these assignments the body of the quantifier evaluates to `true`. Since we are looking for a case where the quantifier does not hold, these test inputs get pruned and the search for a counterexample continues.

When the exploration finds the case in which `i` is zero, it discovers that the body of the quantifier evaluates to `false`. In this case, the `Assert` statement fails and a failing test case has been found:

Program 3.2. Translated Swap Example

```

public void Swap(byte[] a, int lo, int hi) {
    Assume(a != null);
    Assume(0 <= lo && lo <= hi && hi < a.Length);
    byte[] old_a = a.Clone();

    byte tmp = a[hi];
    a[hi] = a[lo];
    a[lo] = tmp;
    if (lo != 0 && hi != 0)
        a[0] = 42;

    Assert(a[hi] == old_a[lo]);
    Assert(a[lo] == old_a[hi]);
    Assert(Forall<int>(i =>
        !(i >= 0 && i < a.Length && i != lo && i != hi)
        || a[i] == old_a[i]));
}

```

```

IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(false)
    .ChooseBoundVariable(0);
byte[] bs0 = new byte[3];
TestSwap2(bs0, 1, 2);

```

Case 2: Let's assume the quantifier holds. In this case, the quantifier is instantiated lazily using pattern matching when relevant constraints become available. The execution continues normally at first. Whenever the program tries to observe a value that should not exist according to the asserted quantifier, the path will be pruned. For example, suppose that we add the code in Program 3.3 at the end of Program 3.2.

In this case, the code checks if the value of `a` at an arbitrary index `j` satisfying `j >= 0 && j < a.Length && lo != j && hi != j` is still equal to its old value.

At this point our pattern matching engine will detect a match with the quantifier, and thus the engine instantiates the quantifier. However, the instantiated quantifier together with the current path condition is unsatisfiable. We detect that the path is infeasible, stop its execution, and prune the path. The `Assert(false)` statement will never be executed.

The following code snippet shows a test case for which the quantifier holds:

```

IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(true);
byte[] bs0 = new byte[3];
TestSwap2(bs0, 1, 2);

```


Program 3.3. Extra code

```

int j = Choose<int>();
if (j >= 0 && j < a.Length &&
    lo != j && hi != j && a[j] != old(a[j]))
    Assert(false);

```

[-] QuantifierTest [details](#) | [coverage](#)

Name	Duration	Tests
Swap(Byte[], Int32, Int32) log parameter values details coverage	00:00:04.40	total, failures, exceptions, inconclusive 6, 1, 1, 0

```

Test(18): SwapByteInt32Int32_20080531_122313_003
IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(true);
oracle.ChooseAt(0, 2);
byte[] bs0 = new byte[3];
Swap(bs0, 0, 2);

```

```

Failing Test(23): SwapByteInt32Int32_20080531_122314_005, unexpected PexAssertionViolationException
See Pex documentation on Assertion Violation for more information why this test fails.

```

```

IPexOracleRecorder oracle = PexOracle.NewTest();
oracle.OnComprehension(0)
    .ChooseTruth(false)
    .ChooseBoundVariable(0);
byte[] bs0 = new byte[3];
Swap(bs0, 1, 2);

```

Exception not caught by test:

PexAssertionViolationException:

[-] QuantifierTest.Swap(Byte[], Int32, Int32), QuantifierTest.cs (601)

Fig. 1. Example Report of Pex

When exploring the `if` statement of Program 3.3, the exploration observes the term `a[j]`. Since this term matches with a subterm of the quantifier body, the body of the quantifier gets instantiated and added as an extra constraint (`!(j >= 0 && j < a.Length && j != lo && j != hi) || a[j] == old(a[j])`). After entering the body of the `if` test, we know that the left disjunct of this constraint is `false`, the right disjunct is `true`. This contradicts the `if` test that `a[j] == old(a[j])` and therefore this path is infeasible.

Figure 1 shows a partial report that is the result of running Pex on the example (including the extra code). The failing test is the same as the one we discussed earlier.

4 Quantifiers

4.1 Introduction

Universal and existential quantifiers are a noteworthy example of an extension of a programming language to support design-by-contract specifications. To make quantifiers executable, existing approaches typically require the bound variables to be in a range or from a set. A significant drawback is that executing these quantifiers for all elements within a range is often impractical. For instance, a contract that involves bound variables that ranges over all records in a database or over all 64-bit integers will require significant resources and be impractical to check repeatedly at run-time. Our approach does not require such bounds.

In the following, we assume that the quantifier body is a pure expression, i.e. that it does not have side-effects. Also, we do not consider nested quantifiers, and leave it for future work to explore several alternative ways of handling nested quantifiers (such as, relying on the SMT solver for these, using prenex forms, or executing the body in a separate run so that we can apply the techniques recursively).

4.2 Compiling a Quantifier to a Non-deterministic Program

We describe how quantifiers within contracts can be transformed into executable, non-deterministic code. We focus on the treatment of universal quantifiers. Since $\exists x.\varphi(x)$ is equivalent to $\neg\forall x.\neg\varphi(x)$, we can use the same approach for existential quantifiers.

Program 4.1 shows the executable version of a universal quantifier. It is implemented as a library function `forall<T>` where `T` is the type of the bound variable. It takes the body of the quantifiers as input as a predicate `p` (`Predicate<T>` is the type of a function which takes an input of type `T` and returns a Boolean value).

Program 4.1. Executable Version of Universal Quantifier

```
bool forall<T>(Predicate<T> p) {
    bool q_holds = ChooseTruth<bool>();
    if (q_holds) {                // Quantifier holds
        Assumeforall<T>(p);
    } else {                      // Quantifier does not hold.
        T val = ChooseBoundVariable<T>();
        Assume(!p(val));
    }
    return q_holds;
}
```

The implementation first obtains a value by calling `ChooseTruth` and stores it in the variable `q_holds`. The value represents whether the quantifier holds. The implementation branches over this value. This is a non-deterministic choice.

Case 1: When q_holds is true. In this case, the rest of the computation should assume that the quantifier indeed is true for all values. This is represented by a call to the function `AssumeForall<T>(p)`, which we will explain in detail in Section 4.3. In a nutshell, the quantifier is added to a list of active quantifiers, and the symbolic execution engine will look out for *relevant values* that appear in the program.

For all such relevant values v_1, \dots, v_n , the instantiated quantifier $p(v_i)$ represents a condition that the test inputs must fulfill. All execution paths in which the quantifier does not hold for these values will be cut off.

In other words, the statement

```
AssumeForall<T>(p);
```

conceptually just represents a set of statements

```
Assume(p(v1));
Assume(p(v2));
...
Assume(p(vn));
```

for all relevant values v_1, \dots, v_n .

However, since the relevant values might only be discovered as the program execution continues, these additional `Assume(p(...))` clauses are realized by our symbolic execution engine.

Case 2: When q_holds is false. In this case, the implementation will attempt to obtain a value `val` for which the predicate `p` is false. This is realized by performing another non-deterministic choice to obtain a value `val`, checking whether the predicate holds on that value, and pruning all execution paths where `p(val)` did not hold by calling `Assume(!p(val))`. The underlying dynamic symbolic execution engine will attempt to obtain values for `val` such that the execution proceeds beyond the call to `Assume`. When no such value is found, all execution paths starting from the assumption that q_holds is false will be effectively cut off.

Illustration. Figure 2 shows an execution tree for evaluating a quantifier whose branches represent the choices introduced by the `ChooseTruth` and `ChooseBoundVariable` functions. The first node with the outgoing branches `false` and `true` represents the program branch over q_holds . If q_holds is true, the predicate $p(i)$ is added to the list of quantifiers; If q_holds is false, the body of the quantifier `p` is explored in order to find a value for which the predicate does not hold.

4.3 Pattern Based Quantifier Instantiation

SMT solvers based on integrations of modern sat-solving techniques [20], theory lemmas and theory combination [13] have proven highly scalable, efficient and suitable for integrating theory reasoning. However, numerous applications from program analysis and verification require furthermore to handle proof-obligations involving quantifiers. As we notice here, quantifiers are often used for capturing frame conditions over loops, summarizing auxiliary invariants over heaps. Quantifiers can also be used

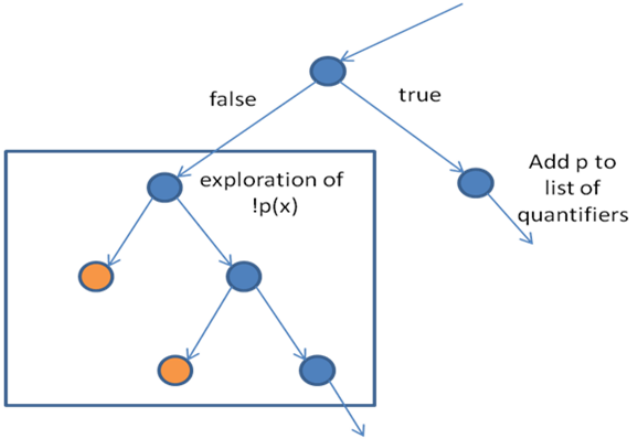


Fig. 2. Example Exploration Tree

for supplying axioms of theories that are not already equipped with solvers. A well known approach for incorporating quantifier reasoning with ground decision procedures uses an E-matching algorithm that works with the set of ground equalities asserted during search to instantiate bound variables. The ground equalities are stored in a data-structure called an *E-graph*. E-matching is used in several theorem provers: Simplify [16], CVC3 [4], Fx7 [28], Verifun [19], Yices [18], Zap [2], and Z3 [12].

We will now describe the quantifier instantiation process and E-matching problem in more detail.

Quantifiers and SAT-Solvers. Suppose φ is a quantifier free formula we wish to show unsatisfiable (conversely $\neg\varphi$ is valid), then φ can be converted into an equi-satisfiable set of clauses [35] of the form $(l_1 \vee l_2 \vee l_3) \wedge (l_4 \vee \dots) \wedge \dots$, where each literal l_i is an atom or a negation of an atom, each atom is either an equality $t \simeq s$, a predicate symbol P , or some other relation applied to ground arguments. Then SAT solving techniques are used for searching through truth assignments to the atoms [29]. An equality $t \simeq s$ assigned to *true* cause as a side-effect a partition of ground terms in the E-graph for φ to be collapsed. When an equality $t \simeq s$ is assigned to *false*, the theory solvers check that s and t do not appear in the same partition; otherwise, the assignment is contradictory. If φ contains quantifiers, then quantified sub-formulas are treated as atomic predicates. Thus, if φ contains a sub-formula of the form $\forall x.\psi$, then this sub-formula is first replaced by a predicate $p_{\forall x.\psi}$. The SAT solver core, and the E-graph structure can then work in tandem to find a satisfying assignment to φ [$\forall x.\psi \leftarrow p_{\forall x.\psi}$]. Suppose first the SAT solver core chooses to set $p_{\forall x.\psi}$ to *false*; it means that under the current assignment of truth values to sub-formulas of φ , it must be the case that $\neg\forall x.\psi$, or in other words, there is some (Skolem) constant sk , such that $\neg\psi[x \leftarrow sk]$. Thus we may add the additional fact

$$\neg p_{\forall x.\psi} \rightarrow \neg\psi[x \leftarrow sk]$$

to φ , propagate the truth assignment for $p_{\forall x.\phi}$, and have the resulting formula $\neg\psi[x \leftarrow sk]$ participate in subsequent search. Conversely, if the SAT solver core chooses to set $p_{\forall x.\psi}$ to *true*, then for the assignment to be consistent with φ , it must be the case that $\forall x.\psi$. In this case, ϕ holds for every instantiation of x . We will later describe how suitable instantiations for x are determined, but suppose for a moment that an instantiation t_1 is identified. We can then add the following fact to φ

$$p_{\forall x.\psi} \rightarrow \psi[x \leftarrow t_1]$$

while maintaining satisfiability, and use the current truth-assignment to propagate the instantiation.

The E-matching Problem. As mentioned above, a widely used algorithmic component for finding suitable instantiations consists of an E-matching algorithm. The E-matching problem is more precisely defined as: *Given* a set of ground equations E , where E is a set of the form $\{t_1 \simeq s_1, t_2 \simeq s_2, \dots\}$, a ground term t and a term p possibly containing variables. *Provide* the set of substitutions θ , modulo E , over the variables in p , such that $E \models t \simeq \theta(p)$. Two substitutions are equivalent if their right hand sides are pairwise congruent modulo E .

When solving the E-matching problem, it is common to build a *congruence closure* based on the equalities in E . The congruence closure partitions terms in E and ground sub-terms from p and t ; it is the least partition (\equiv_C) closed under equivalence (reflexivity, symmetry, and transitivity), and functionality: if $t_1 \equiv_C s_1, t_2 \equiv_C s_2$, and $f(t_1, t_2)$, and $f(s_1, s_2)$ are sub-terms, then $f(t_1, t_2) \equiv_C f(s_1, s_2)$. Efficient implementations of congruence closure typically use union-find data-structures and use-lists [17]. A congruence closure is the finest partition induced by the equalities E .

The E-matching problem is NP-complete, but in the context of SMT problems the harder practical problem is to handle a massive number of patterns and a dynamically changing set of patterns and equalities E . Efficient data-structures and algorithms for these situations are described in [12].

Patterns. So what do we E-match against? For most practical purposes, the answer is a set of sub-terms in the quantified formula (from the above example, ψ) that contain the bound variables (from the above example, the variable x).

Example 1. Consider one of the axioms used for characterizing arrays [26]:

$$\forall a, i, j, v . i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) \simeq \text{read}(a, j) .$$

When should it be instantiated? One clear case is when the sub-term $\text{read}(\text{write}(a, i, v), j)$ matches a term in set of current ground terms. A less obvious case is when both $\text{write}(a, i, v)$ and $\text{read}(a, j)$ occur as terms. These terms combined contain all bound variables, so they can be used for instantiating the quantifier. The latter condition refers to two occurrences of a . These two occurrences can match any pair of terms in the current context as long as they belong to the same equality partition.

In Simplify [16], such patterns are annotated together with the quantifier.

$$\forall a, i, j, v . (\text{PATS } \text{read}(\text{write}(a, i, v), j) \text{ (MPAT } \text{write}(a, i, v) \text{ read}(a, j))) : \\ i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) \simeq \text{read}(a, j) .$$

We here perform a partial lifting of the concept of patterns to the context of C# programs.

Example 2. A contract that assumes the array `a` to be 0 on every index `i` can be formulated as:

```
Assume(Forall<int>(i =>
    !(i >= 0 && i < a.Length)
    || Pattern<int>(a[i]) == 0));
```

In this contract we have used the generic function:

```
T Pattern<T>(T value);
```

The pattern specifies that the quantifier on `i` be instantiated whenever a sub-term of the form `a[i]` is created during search. Multiple occurrences of `Pattern` are treated as alternatives. Our pattern extension in `Pex` does not currently provide a counter-part to multi-patterns (conceptually it is a relatively easy extension, that has yet to be exercised: add a numeric argument to `Pattern`, all occurrences using the same numeral argument belong to the same multi-pattern).

Note that this does not directly limit the set of values for instantiating the quantifier. There are still 2^{32} or 2^{64} possible values of the type `int` to instantiate the quantifier. Operationally, the `Pattern` function implements the identity function.

4.4 Run-Time-Guided Pattern Matching

As we outlined, modern SMT solvers use E-matching for instantiating patterns. E-matching uses congruence relations between ground terms. During search for unsatisfiability or satisfiability, congruence relations encode equalities that hold under all possible interpretations of the current state of the search. We will here deviate from this use of congruence relations for finding pattern matches. The basic observation is that, instead of searching through a set of different congruence relations, we use the model produced by a concrete execution for identifying a (coarse) partition of terms appearing in the corresponding symbolic trace. Two terms in a symbolic trace are treated as potentially equal if their run-time values are equal. E-matching can now be replaced by a pattern matching function that uses run-time values. We call this version the *M-matching* problem, where \mathcal{M} refers to a model. The \mathcal{M} -matching problem is more precisely, given a model \mathcal{M} , that provides an interpretation for a set of terms \mathcal{T} , and a ground term $t \in \mathcal{T}$ and pattern p ; provide the set of substitutions θ mapping variables in p to terms in \mathcal{T} , such that $\mathcal{M} \models t \simeq \theta(p)$. Notice that \mathcal{M} -matching is an approximation of E-matching, since $E \models t \simeq \theta(p)$ and $\mathcal{M} \models E$ implies that $\mathcal{M} \models t \simeq \theta(p)$. On the other hand, \mathcal{M} -matching allows going beyond congruences of uninterpreted function symbols: the model provided by a concrete run provides interpretations to arbitrary functions. \mathcal{M} -matching may be implemented in a way similar to E-matching, using code-trees [12], but using a model \mathcal{M} instead of relying on a congruence closure.

The main steps used by the \mathcal{M} -matching algorithm are summarized below.

1. Let \mathcal{T} be the set of all terms appearing in a symbolic state.
2. Let p be a pattern we wish to match with terms in \mathcal{T} .

3. Recursively, match function symbols used in p with all possible matching symbols from \mathcal{T} . For example, if p is of the form $f(p_1, p_2)$, then select every occurrence of $f(t_1, t_2)$ in \mathcal{T} and create the sub-matching problems p_1, t_1 and p_2, t_2 .
4. If, in the recursive matching step, p_i is ground, then check if the matching terms p_i^M equals t_i^M . If p_i is a bound variable x , then bind the value t_i^M to x if x has not been bound before. If x was bound before to t_j^M , then check $t_i^M = t_j^M$.

A match succeeds if the concrete run-time values coincide. The symbolic representations may be different, but the use of run-time values ensures that every match that may be valid at a give execution point is found by using the run-time values. Thus, this process may be used for supplying a superset of useful values for parameters to quantified contracts. Our method restricts quantifier instantiation to observed values. Symbolic values that are not observed are don't cares from the point of view of the program under test, so we admit test inputs that violate contracts on unobserved values.

Example 3. Suppose we seek to match a pattern of the form:

$$w * ((w + u) + V)$$

where w is an identifier used in a program and V is the bound variable of a quantifier. Consider the program fragment:

```

y = u + 4;
w = 4;
if (x == y - u) {
    u = x * (y + z);
    . . .
}

```

We match the pattern $w * ((w + u) + V)$ against the term $x * ((u + 4) + z)$ which got built by expanding the assignment to y by $u + 4$. Matching proceeds by following the structure of the pattern:

$$\begin{aligned}
& Match(w * ((w + u) + V), x * ((u + 4) + z)) = \\
& \quad Match(w, x) \text{ and} \\
& \quad Match(((w + u) + V), ((u + 4) + z))
\end{aligned}$$

Since the pattern occurs only under the if-condition ($x == y - u$) it must be the case that the run-time value of both x and w is 4. So by using the run-time values, the first match reduces to

$$Match(4, 4)$$

which holds. The second call to *Match* reduces to:

$$\begin{aligned}
& Match((w + u), (u + 4)) \\
& Match(V, z)
\end{aligned}$$

Where the first matching obligation can be solved by looking at the run-time values of w and u :

$Match(8, 8)$

The second matching obligation binds the variable V to z .

Models induced by run-time values will produce a possibly coarser partition than a corresponding congruence closure, so more terms may be identified as matches than really exist. We can compensate for the approximation by adding a side-condition to the instantiated quantifier. Namely, if

$$\forall x . pat(x) : \phi(x)$$

is a quantifier with bound variable x and pattern $pat(x)$, and the symbolic term t is identified as a run-time match of $pat(x)$, with the instantiation $x \leftarrow s$, then we can create the instantiated formula:

$$pat(s) \simeq t \rightarrow \phi(s)$$

5 Related Work

Automated testing has been used in the past to guide the refinement of invariants when proof attempts fail [11], however their work was not applied to design-by-contract specifications.

The use of specifications as test oracle to decide the result for a particular test case is a well-known technique. This idea was first explored by Peters and Parnas [31]. Many approaches have followed this technique to run-time check design-by-contract specifications for JML [6], Eiffel [27] and Spec# [3]. Design-by-contract specifications have also been used for test generation using more or less random approaches to the test input generation problem, e.g. for JML [8,7] and Eiffel [9,10]. Notably, their approaches do not handle unbounded quantifiers. Unlike existing approaches, we provide a way to evaluate unbounded quantifiers (or quantifiers over an impractically large domain).

The idea of symbolic execution was pioneered by [24]. Dynamic symbolic execution was first suggested in DART [22]. Their tool analyzes C programs. Several related approaches followed [33,5,23]. They differ between each other in the extent of how much concrete information is lifted in the analysis and how much is treated symbolically, i.e. the extent of the under-approximation that they perform. We describe how to extend dynamic symbolic execution with a symbolic treatment of quantifiers.

Contracts can be used to make dynamic symbolic execution more modular and thus scalable [30]. Several attempts have been made to even infer such contracts dynamically [21,1].

Our approach relies on using the SMT solver Z3 to generate inputs that drive a program into its different reachable configurations. An overview of related work on E-matching is detailed in [12].

6 Conclusion

This paper described an approach for using dynamic testing for debugging deductive verification of contracts with quantifiers. We extended symbolic execution to handle

unbounded quantifiers. We translated quantifiers to non-deterministic programs and introduced \mathcal{M} -matching as a technique for finding quantifier instances among symbolic values exercised in a run. Future work includes assessing scalability and the coverage exercised by the quantifier instances.

Acknowledgments

We would like to thank Ernie Cohen, Herman Venter, and Songtao Xia for the discussions and their support.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Proc. of TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
2. Ball, T., Lahiri, S.K., Musuvathi, M.: Zap: Automated theorem proving for software analysis. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 2–22. Springer, Heidelberg (2005)
3. 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, pp. 49–69. Springer, Heidelberg (2005)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 322–335. ACM Press, New York (2006)
6. Cheon, Y.: A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, The author's Ph.D. dissertation. (April 2003), <http://archives.cs.iastate.edu>
7. Cheon, Y.: Automated random testing to detect specification-code inconsistencies. Technical report, Department of Computer Science The University of Texas at El Paso, 500 West University Avenue, El Paso, Texas, USA (2007)
8. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Proc. 16th European Conference Object-Oriented Programming, pp. 231–255 (June 2002)
9. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 84–94. ACM, New York (2007)
10. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo: adaptive random testing for object-oriented software. In: ICSE 2008: Proceedings of the 30th international conference on Software engineering, pp. 71–80. ACM, New York (2008)
11. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 48–65. Springer, Heidelberg (2008)
12. de Moura, L., Bjørner, N.: Efficient E-matching for SMT Solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
13. de Moura, L., Bjørner, N.: Model-based Theory Combination. Electron. Notes Theor. Comput. Sci. 198(2), 37–49 (2008)
14. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver (2007), <http://research.microsoft.com/projects/Z3>

15. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
16. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *J. ACM* 52(3), 365–473 (2005)
17. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. *J. ACM* 27(4), 758–771 (1980)
18. 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)
19. Flanagan, C., Joshi, R., Saxe, J.B.: An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Laboratories, Palo Alto (2004)
20. 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)
21. Godefroid, P.: Compositional dynamic test generation. In: Proc. of POPL 2007, pp. 47–54. ACM Press, New York (2007)
22. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. *SIGPLAN Notices* 40(6), 213–223 (2005)
23. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of NDSS 2008 (Network and Distributed Systems Security), pp. 151–166 (2008)
24. King, J.C.: Symbolic execution and program testing. *Commun. ACM* 19(7), 385–394 (1976)
25. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University (June 1998)
26. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
27. Meyer, B.: Eiffel: The Language. Prentice Hall, New York (1992)
28. Moskal, M., Lopuszanski, J.: Fast quantifier reasoning with lazy proof explication (2006), <http://nemerle.org/malekith/smt/smt-tr-1.pdf>
29. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 38th Design Automation Conference (DAC 2001) (2001)
30. Mouy, P., Marre, B., Williams, N., Gall, P.L.: Generation of all-paths unit test with function calls. In: Proceedings of ICST 2008 (International Conference on Software Testing, Verification and Validation), pp. 32–41 (2008)
31. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. *IEEE Trans. Softw. Eng.* 24(3), 161–173 (1998)
32. Pex development team. Pex (2007), <http://research.microsoft.com/Pex>
33. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. of ESEC/FSE 2005, pp. 263–272. ACM Press, New York (2005)
34. Tillmann, N., de Halleux, J.: Pex – white box test generation for .NET. In: Proc. of Tests and Proofs (TAP 2008), Prato, Italy, April 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
35. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, pp. 466–483. Springer, Heidelberg (1983)

Automated Evaluation of Secure Route Discovery in MANET Protocols

Todd R. Andel^{1,*} and Alec Yasinsac^{2,**}

¹ Air Force Institute Technology, Wright-Patterson AFB, OH, USA
todd.andel@afit.edu

² University of South Alabama, Mobile, AL, USA
ayasinsac@usouthal.edu

Abstract. Evaluation techniques to analyze security properties in ad hoc routing protocols generally rely on manual, non-exhaustive approaches. Non-exhaustive analysis techniques may conclude a protocol is secure, while in reality the protocol may contain an unapparent or subtle flaw. Using formalized exhaustive evaluation techniques to analyze security properties increases protocol confidence. In this paper, we offer an automated evaluation process to analyze security properties in the route discovery phase for on-demand source routing protocols. Using our automated security evaluation process, we are able to produce and analyze all topologies for a given network size. The individual network topologies are fed into the SPIN model checker to exhaustively evaluate protocol abstractions against an attacker attempting to corrupt the route discovery process.

1 Introduction

Mobile ad hoc networks (MANETs) consist of portable wireless nodes that do not use fixed infrastructure. Unlike their wired counterparts that depend on fixed routers for network connectivity and message forwarding, MANETs rely on each node to provide routing functionality. Ad hoc routing protocols allow wireless nodes to communicate with nodes outside their local transmission distance. MANET routing protocols [1,2,3,4] commonly utilize two-phased routing approaches in which a route is first discovered during a route discovery phase and data is subsequently forwarded over the discovered route. In order for the protocol to meet its desired goal to deliver messages, both phases must be secured to protect the protocol against malicious activity.

There are numerous approaches proposed to analyze security properties in MANET routing protocols. These techniques include visual inspection, network simulation, analytical proofs, simulatability models, and formal methods [5]. Unfortunately, most MANET literature generally follows unstructured approaches and non-exhaustive methods to evaluate route security.

* The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

** This material is based upon work supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grants numbered W91NF-04-1-0415.

In this paper, we utilize the formal methods approach through automated model checking to evaluate the ad hoc route discovery process against route corruption. Given the network topology, our analysis models exhaustively check all routing combinations to evaluate if an attacker can corrupt the route discovery phase by returning validated routes that are not consistent within the network topology. Our modeled protocol abstractions employ the SPIN model checker [6] to provide exhaustive analysis against the specified protocol security property. Although SPIN has been used to evaluate MANET routing protocol operation (e.g., loop freedom) [7, 8, 9], to the best of our knowledge we are the first to use SPIN to evaluate security properties in ad hoc routing protocols, as presented by our initial work in [10]. This paper extends our initial work on using SPIN to evaluate MANET route security by adding additional protocol models, a more advanced attacker model, and provides an automated process to examine all possible network topologies for a given network size.

In the remainder of this paper, we discuss requirements for secure routing and the current analysis techniques being used to evaluate secure ad hoc routing protocols. Our primary contribution is the secure routing model abstraction, attacker abstractions, and the exhaustive network topology generation and analysis process. We exercise our automated security analysis process to discover an undocumented attack on the Ariadne [11] protocol and an undocumented attack on the endairA protocol [12].

2 Evaluating Secure Routing

Secure ad hoc routing protocols must be analyzed to ensure their security goals are met and to identify under what adversarial environments they may fail. We first define secure routing protocol requirements, discuss the current techniques used to evaluate MANET security properties, and provide an overview of our exhaustive analysis approach.

2.1 Requirements for Secure Routing Protocols

The term *security protocol* customarily refers to an *authentication protocol*, in which the goal is to securely share information between two nodes. Security analysis for authentication protocols evaluates if it is possible for a third party to obtain protected information, regardless of the communication path [13]. Conversely, security evaluations for MANET secure routing protocols must consider the complete communication path, or route. In particular, we must consider route accuracy and protocol reliability.

A routing protocol provides *route accuracy* if it produces routes that exist within the current network topology. Route accuracy is an integrity issue, ensuring that an attacker has not corrupted the path identified during route discovery. Since the routes obtained during route discovery can fail due to both malicious actions and non-malicious failures (e.g., mobility, hardware failures, etc.), the routing protocols must also provide *reliability*. Once a route fails, reliability mechanisms initiate a new route discovery process, use a previously found path if multi-path protocols [14, 15] are being utilized, or may attempt to detect and remove malicious nodes via probing protocols [16].

2.2 Current Evaluation Techniques

There are many techniques used to evaluate security properties in MANET routing protocols. These techniques include visual inspection, network simulation, analytical methods, simulatability models, and formal methods. Visual inspection, analytical proofs, and simulatability models are not automated. Visual inspection and network simulation do not provide exhaustive attacker analysis. Work in [5] provides detailed analysis on the capabilities and limitations of these analysis methods for use in MANET environments.

Our research follows the formal methods [17] and automated model checking [18] paradigms to evaluate route validity for routes returned in on-demand source routing protocols. Automated formal methods have shown success in evaluating authentication protocols [19,20,21]. There has also been some initial work [22,23] using automated formal methods to evaluate MANET route security for ad hoc on-demand distance vector protocols such as AODV and Secure AODV (SAODV), where the attacker's goal is to corrupt a node's next-hop entries stored in the node's routing table. However, the work was isolated to evaluating a few network topologies.

2.3 An Exhaustive Evaluation Approach

Figure 1 illustrates our complete analysis procedure. In the following sections we describe SPIN model development over the Ariadne and endairA protocols, attacker development, and our automated process that generates and analyzes all network topologies for a given network size N .

We transform the routing protocol, attacker actions, and security property into a model abstraction for analysis using the SPIN model checker. SPIN [6] is a general purpose model checker developed to verify the operation of distributed systems. Once a protocol abstraction and desired goals are specified in the Promela modeling language, SPIN generates a finite state automata (FSA) and performs exhaustive state analysis to determine if the desired goal holds or is violated. If the system goal is violated, SPIN generates an event sequence leading to the failure.

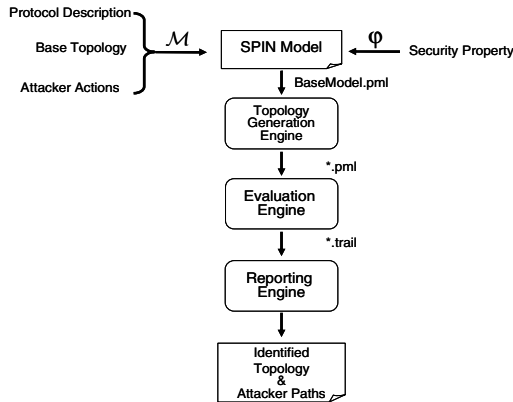


Fig. 1. Automated Security Analysis Process

A primary factor in choosing SPIN for our research is its success in verifying security properties in authentication protocols [24] and its previous use to evaluate loop freedom in MANET routing protocols in non-malicious environments [7, 8, 9]. Our research combines these areas to analyze security properties in MANET routing protocols.

Additional factors in choosing SPIN include Promela's ability to model message passing and the ability to model independent distributed processes. These factors allow us to model the routing process in a MANET environment. Finally, Promela's ability to code non-deterministic choices allows us to model the wireless message deliverability environment. Since wireless message deliverability is not guaranteed, an attacker may be able to inject false routing information in place of dropped messages. For example, we use the following *if* construct to model wireless message reception:

```

if
  :: message received ->
    process message according to protocol
  :: message received ->
    drop message and take no actions
fi .

```

Since both entry statements are true, then either statement and its corresponding actions may be non-deterministically chosen during SPIN simulations. During SPIN exhaustive analysis, all possible non-deterministic choices are examined independently. Using SPIN in this fashion allows us to model all possible message interactions to determine if an attack may occur.

3 Model Checking Secure Routing

We focus our model abstraction on the MANET route discovery phase, analyzing if an attacker can corrupt the path during route discovery. Our model development and analysis targets source routing protocols, in which routes are explicitly embedded into protocol messages. We break the model into the three primary processes: the wireless medium, non-malicious node, and attacker node.

For each primary process, we isolate the model development to focus on malicious failures. Non-malicious routing failures occur due to mobility or failed nodes, which affects a protocol's message deliverability and overall protocol performance. Malicious failures may also occur, which to the routing protocol cannot be distinguished from non-malicious failures.

In addition to the inconclusive results encountered by including non-malicious failures, modeling non-malicious failures may not be feasible in a finite space model checker. For example, Wibling et al. [8] evaluate protocol loop freedom and show that it is not feasible to model mobility in model checking paradigms due to rapid state-space explosion. While model checkers may encounter the same state-space limitations when isolating malicious failures, modeling only the required failures of interest reduces the model complexity and increases the chances that the model checker can evaluate the given security property.

Based on these observations, we focus on modeling malicious failures against secure routing protocols. Eliminating non-malicious failures allows us to effectively isolate discovered route failures due to malicious activity. Any security flaws which

may arise due to mobility causing an undelivered message is taken into account by exhaustively searching for all possible paths in the given topology under analysis, since transmissions in general cannot be guaranteed in wireless environments.

3.1 Modeling the Wireless Medium

Modeling message transmission in a wireless environment is a significant challenge for simulating and model checking MANET protocols. When a wireless node transmits a message in a physical MANET implementation, all nodes within its transmission footprint receive the message virtually simultaneously. Each wireless recipient then determines if it must process or drop the received packet.

SPIN does not provide direct broadcast communication support; therefore, we model the required communication components to ensure all neighbor nodes receive the message. Since time is implicitly modeled by evaluating all computational possibilities, we model one hop broadcast messages via multiple unicast messages. In [25], Ruys discusses how to implement simple broadcast communication via a common bus structure, a matrix of channels for each node combination, or as a broadcast server process.

We model MANET communication between nodes using a *wireless medium server*, similar to Ruys' [25] broadcast server process. By modeling the wireless medium as its own process rather than modeling node-to-node channels or using a variable length bus, we limit the state-space and modularize our approach to eliminate modeling dependencies based on the network size being evaluated. We model a node's transmission range by connectivity rather than by distance, by ensuring only reachable neighbors receive transmitted messages. We store the network topology for the configuration we are evaluating in a two-dimensional array. Figure 2 illustrates the associated array for a four node network topology. The model's wireless medium process uses the array to determine the transmitting node's local neighbors and transmits messages accordingly.

The array consists of N rows by N columns, where N is the total number of non-malicious nodes plus the total number of malicious nodes. Rows indicate the transmitting node and columns indicate the node's local neighbors. Each array element holds a Boolean value, with true (or 1) indicating a connection exists between node pairs. The shaded areas indicate array symmetry in a bidirectional environment.

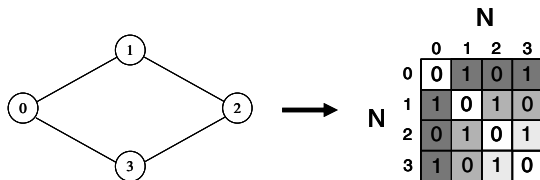


Fig. 2. Representing Network Topology

Using the independent wireless medium process with the global connectivity array allows us to model any MANET routing protocol. However, we tune the wireless medium to model on-demand source routing protocols in order to maintain a reduced state-space. Our wireless medium server approach delivers broadcast route request

(*rreq*) messages to each adjacent node. Each recipient node processes the message and transmits the message to the next hop using the wireless medium server. The unicast route reply (*rrep*) can use an identical process, with the exception that each node receiving the message must determine if it is included in the path before retransmitting the message to the next upstream host. In order to limit state-space, the wireless medium server only transmits to the intended recipient identified in the unicast *rrep* message and to all attacker nodes within the current transmission footprint. This approach reduces state-space yet still captures the protocol's required elements for security analysis.

3.2 Modeling Source Routing Protocols

MANET two-phased routing protocols can be classified as *proactive* or *reactive*. Proactive protocols attempt to continuously maintain fresh routes between all source-destination pairs. Reactive protocols establish a route between a source and destination only when required. Reactive protocols, also known as on-demand protocols, generally use either distance vector mechanisms or source routing. In on-demand distance vector protocols, such as AODV, tables are used in each node to direct the next hop to a given destination. On-demand source routing protocols, such as DSR, explicitly embed the complete route into each message.

Our research goal is to automate the security analysis process for evaluating routing attacks against the route discovery phase in on-demand source routing protocols. In our initial work [10], we modeled the DSR protocol and the SRP extension to DSR. In this work, we provide models for the Ariadne and endairA security extensions based on the DSR protocol. Modeling these protocols proved to be more difficult, as the added security requirements posed additional challenges to ensure the cryptographic primitives were maintained. We make several abstraction choices to simplify the resulting model and limit the resulting state-space. Model checking over the abstraction exhaustively examines all routing possibilities for a static network. Our model abstraction does not reflect message timing issues, since message deliverability is not guaranteed in a wireless environment. Our resultant models allow us to search for possible route violations rather than probable outcomes.

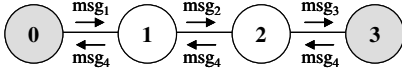
Modeling Ariadne. Ariadne [11] is an extension to the DSR protocol that attempts to secure the forward *rreq* from being corrupted by computing a one-way per-hop hash value at each intermediate node.

The Ariadne message formats follow as:

- $\langle rreq, initiator, target, id, hash_value, accum_path, sig_list \rangle$
- $\langle rrep, target, initiator, accum_path, sig_list, target_sig \rangle$.

We illustrate the Ariadne protocol using the network topology and messages shown in Figure 3, with initiator node 0, target node 3, H is a cryptographically secure one-way hash function, and SK_i is node i 's signing key.

The hash value (h_x) is included to guard against an attacker removing a node from the embedded path. The initial hash seed is a secret known only to the initiator-target pair. Each intermediate node adds its id to the route path (*accum_path*), calculates and inserts a new per-hop hash value (*hash_value*), and appends its signature before



$msg_1 = (rreq, 0, 3, id, h_0, (), ())$
$h_0 = H[0, initiator\text{-}target\ secret]$
$msg_2 = (rreq, 0, 3, id, h_1, (1), (sig_1))$
$h_1 = H[1, h_0]$
$sig_1 = SK_1\{rreq, 0, 3, id, h_1, (1)\}$
$msg_3 = (rreq, 0, 3, id, h_2, (1, 2), (sig_1, sig_2))$
$h_2 = H[2, h_1]$
$sig_2 = SK_2\{rreq, 0, 3, id, h_2, (1, 2), (sig_1)\}$
$msg_4 = (rrep, 3, 0, (1, 2), (sig_1, sig_2), (sig_3))$
$sig_3 = SK_3\{rrep, 0, 3, (1, 2), (sig_1, sig_2)\}$

Fig. 3. Ariadne Protocol Messages

retransmitting the *rreq*. In addition to the hash value, each intermediate node computes and appends a signature over the complete packet to make sure the path contains only trusted insiders. Ariadne allows the signature to be a message authentication code (MAC) computed with a pairwise secret key, a MAC computed with the delayed key via the TESLA broadcast key distribution scheme, or computed as a digital signature. For the purposes of model development, we use digital signatures and refer to node x 's signature as sig_x .

Once the target receives the *rreq*, it validates the one-way hash value by iteratively performing the hash computation against all nodes in the accumulated path. If the *rreq* validates, the target signs the path and returns the path and signature in a *rrep*. During the *rrep*, the intermediate nodes along the unicast path relay the reply to the next upstream node indicated in the embedded path. The initiator checks the signatures and accepts the route if all checks validate.

We express Ariadne using Promela to facilitate SPIN analysis. The first step is to define the message format. The identifier (*id*) is dropped from the message since we are analyzing one route discovery round. We replace the *id* with a Boolean value in each node to ensure only one *rreq* is processed for the route discovery round. To maintain common message formatting and subsequent channel modeling, the *rreq* and *rrep* messages are modeled following the same format. The signatures of the intermediate and target nodes are combined producing the following interim message abstraction:

$$\langle msg_type, initiator, target, accum_path, hash_chain, sig_list \rangle.$$

Ariadne does not use the *hash_chain* in the *rrep* and it is set to zero after the target processes the *rreq*. The two fields of interest are the *hash_chain* and the *sig_list*, which model the cryptographic one-way hash value and the list of cryptographic signatures respectively.

The *hash_chain* represents a cryptographically secure one-way computation. At each stage an intermediate node rehashes the value after appending its node id to the previous hash value. This process produces the one-way hash chain with i representing the current node and $i-1$ representing the previous node along the forward *rreq*:

$$h_i = H[n_i, h_{n(i-1)}] = H[n_i, H[n_{i-1}, \dots H[n_0, h_0] \dots]].$$

During protocol operation the computed hash value is transmitted. For modeling purposes we remove the hash operation, listing the hash chain as: $[n_i, n_{i-1}, \dots, n_1, n_0]$. Modeled nodes may not corrupt or remove an earlier appended value to the hash chain, since we are modeling a one-way hash value based on the chain's node identifiers in the appended array. We capture the hash computation by restricting a node's

actions against the modeled hash chain to either replaying the entire chain or computing a hash value by appending to any previous hash chain captured from the wireless environment.

We represent the hash chain in Promela with an append only accumulated array. To capture the properties of the one-way hash computation, the model does not allow an intermediate node to change the *hash_chain*. The modeled hash chain is different than the accumulated path (*accum_path*), as the model does not restrict changes to the accumulated path. Once the target receives the *rreq*, it checks the accumulated path against the hash chain to ensure they match. This process captures the effect of the target computing a one-way hash value using the path contained in the *rreq* message and comparing it against the hash value delivered in the *rreq*.

The Ariadne accumulated intermediate node signatures (*sig_list*) protect the protocol from including malicious outsiders in the routing path. The initiator validates the signatures against the received accumulated path. We model the *sig_list* in the same fashion as the hash value, appending a node id to the *sig_list* to indicate a node has signed the given message. The signatures cannot be reordered, but can be dropped in the reverse order to match any attacker dropping nodes from the end of the accumulated path. Assuming the signature process is cryptographically secure and limiting our future attack process to dropping nodes from the end of the path, we do not explicitly model the intermediate signatures in order to reduce the state-space. We do model the target signature over the path to secure the embedded path returned in the route reply. The target models the signature by copying the accumulated path from the forward *rreq* into the signature in the *rrep*. The signature cannot be corrupted during the *rrep*, allowing the initiator to check the returned accumulated path against the target signature to ensure routing attacks performed against the return *rrep* are detected.

The Ariadne model therefore uses the following message format, where *accum_pos* and *hash_pos* track the current position for the non-malicious node to append itself to the accumulated path and abstracted hash chain to the current forward *rreq* message. We use a single array called *crypt_val* to model both the hash chain and target signature. During the forward *rreq*, *crypt_val* holds the hash chain. During the return *rrep*, *crypt_val* holds the target signature.

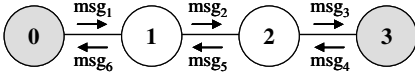
<msg_type, initiator, target, accum_path, accum_pos, crypt_val, hash_pos>.

Modeling endairA. The endairA protocol [12] attempts to secure DSR by only securing the *rrep* using signatures, as illustrated in Figure 4. Node 0 is the initiator, node 3 is the target, and SK_i is node i 's signing key.

The endairA message formats follow as:

- *<rreq, initiator, target, id, accum_path>*
- *<rrep, initiator, target, accum_path, sig_list>* .

Instead of protecting the forward *rreq* process, the target computes a signature over the accumulated path received in the *rreq* and adds the signature to the *rrep*. During the *rrep*, the intermediate nodes sign the message and forwards to the next hop. Once the *rrep* reaches the initiator, the initiator checks the target signature and verifies that each node in the return path has signed the message in reverse order. While the target may sign corrupted paths received by *rreq*, the protocol authors contend these paths should not make it back to the initiator with the correct appended signatures.



```

msg1 = (rreq, 0, 3, id, ( ) )
msg2 = (rreq, 0, 3, id, (1) )
msg3 = (rreq, 0, 3, id, (1, 2) )
msg4 = (rrep, 0, 3, (1, 2), (sig3) )
           sig3 = SK3{rrep, 0, 3, id, (1, 2), ()}
msg5 = (rrep, 0, 3, (1, 2), (sig3, sig2) )
           sig2 = SK2{rrep, 0, 3, (1, 2), (sig3)}
msg6 = (rrep, 0, 3, (1, 2), (sig3, sig2, sig1) )
           sig1 = SK1{rrep, 0, 3, (1, 2), (sig3, sig2)}
  
```

Fig. 4. endairA Protocol Messages

The approach to modeling endairA is similar to the Ariadne abstraction. As in Ariadne, we drop the *id* from the message and replace it with a Boolean value to ensure only one *rreq* is processed for the route discovery round. To maintain common message formatting and subsequent channel modeling, the *rreq* and *rrep* messages are modeled following the same format. Our message abstraction follows as:

$\langle \text{msg_type}, \text{initiator}, \text{target}, \text{accum_path}, \text{accum_pos}, \text{target_sig}, \text{sig_list}, \text{sig_pos} \rangle$.

The target signature (*target_sig*) holds the signature over the accumulated path that the target calculates and embeds into the *rrep*. The value is modeled by copying the *accum_path* array into the *target_sig* array. The attacker cannot change the *target_sig*, which protects the path from being corrupted during the *rrep*. The signature list (*sig_list*) keeps track of each node that has signed the *rrep* during its message traversal. We model this activity in an array, where each node adds its id to the *sig_list* during the *rrep*. Once the initiator receives the *rrep*, it checks the target signature to ensure the accumulated path matches the signature and verifies that all nodes in the accumulated path have signed the message by checking the signature list.

3.3 Modeling the Attacker

There are known attacks against MANET routing protocols that have no known detection or prevention mechanism. Attacks such as the invisible node attack (INA) [26] are a continued threat since no current mechanisms can positively identify the node that transmitted a given message. Additionally, malicious insiders can refuse to participate in routing protocols at will, even though they are trusted to follow the protocol rules.

This research focuses on an attacker that actively corrupts the embedded route during the MANET route discovery phase, resulting in routes that do not match the current network topology. The attacker model incorporates static analysis techniques to produce a finite attacker model that an automated model checker can execute. If we allowed an attacker to arbitrarily change messages, the state-space for all possible messages would quickly exceed computational capabilities. The routing messages and structure are dependent on the interactions between the intermediate nodes and the network topology, which complicates the analysis structure as compared to authentication protocol analysis.

We leverage the fact that a simplified three node (source, destination, attacker) structure can model end-to-end authentication protocol security requirements [13]. Rather than using model checking to generate all possible message structures, we

evaluate all possible path sequences through the wireless network structure. To develop the attacker model we follow the approach in [24] to limit the attacker's actions based on the information provided by the protocol messages that could possibly enable an attack. This process requires static analysis over the possible messages the attacker can capture and extracts only the information required by the attacker to perform a successful attack. The information obtained by the static analysis allows the attacker to model a finite set of attempted attack sequences.

Ariadne Attack Development. We use the messages in Fig. 3 for static analysis to develop the Ariadne attacker. Since the target node signs the accumulated path received in the *rreq*, the path cannot be corrupted during the *rrep*. Thus, the target signature limits the attacker to the forward *rreq*. Examining the *rreq* message structure indicates that the unprotected accumulated path can be changed as long as the corresponding hash chain and signatures match the path changes. The forward intermediate signatures do not allow the attacker to reorder the path or add an outsider to the path. Since we are not explicitly modeling the forward signatures, we limit the malicious insider to only appending or dropping a node from the end of the accumulated path. We also limit the malicious outsider to only dropping a node from the end of the accumulated path, since the outsider cannot generate a valid signature without a trusted key.

The final message element to consider is the one-way hash value. Due to its cryptographic properties, the hash value cannot be directly corrupted as was the previous case in [10]. In [10], the attacker simply dropped a node from the accumulated path in the *rreq* and did not attempt to corrupt or compute the cryptographic message authentication code in SRP. The Ariadne attacker in the current case must actively compute a valid cryptographic value. Any attacker that attempts to drop a node from the accumulated path must compute a hash value that corresponds to the new path. The only way for an attacker to strip a node from the end of the accumulated path and compute a correct corresponding hash value is to generate the hash value based on capturing a previous hash value from an upstream neighbor during the current route discovery round. Hash values from previous route discovery rounds would not match due to the message id or sequence number. For instance, if an attacker wishes to remove node 2 from the protocol example in Figure 3, it needs to know the value h_1 . If h_1 is not captured from msg_2 it can be calculated from h_0 (captured in msg_1), since $h_1 = H[1, h_0]$. The resulting *malicious insider* attacker process proceeds as follows. Upon receiving a *rreq* from an intermediate node, the attacker stores the associated node's hash value and checks to see if it knows a hash value from a previous upstream neighbor. If the attacker has this knowledge, it drops the last node and adds its id to the accumulated path. The attacker also replaces the hash value with its local hash calculation. To compute a valid hash, the modeled attacker appends nodes to a previous hash-chain until it matches the current accumulated path. During the *rrep*, the attacker attempts to relay the *rrep* to any upstream neighbor. The *malicious outsider* follows the same process except that it does not add its id to the path after dropping the last node from the accumulated path.

These attacks are possible as long the attacker can capture a previous enabling hash value and the *rrep* is deliverable to one of the attacker's upstream neighbors.

endairA Attack Development. We use the messages in Figure 4 for static analysis to develop the endairA attacker. As in Ariadne, the target signature on the accumulated

path ensures the path cannot be corrupted during the *rrep*. Since the signatures added by the intermediate nodes during the *rrep* must match the reverse order as the path signed by the target in the *rrep*, any attack must be limited to malicious insiders. If a malicious node strips a node during the forward *rreq*, it requires a direct link to an upstream node to ensure the next node can produce the appropriate signature during the *rrep*. However, this link would constitute a valid path. The only avenue that allows the malicious insider the ability to drop nodes without this direct link is the ability to generate signatures for more than one node. If we assume the cryptographic process is secure in polynomial time, the only way the attacker can produce multiple signatures is to have multiple keys. In a colluding environment, we assume that all colluding attackers have previously shared their keying material.

The resulting attacker model in a colluding environment proceeds as follows. If the forward *rreq* passes through two attackers, the second attacker strips the nodes between the two attackers before sending the *rreq* to the next hop. Once the target signs the corrupted accumulated path, it responds with the *rrep*. During the *rrep* the second attacker signs for both attackers in the correct order to ensure the signatures match the nodes in the accumulated path. The requirement to enable this attack is that the second attacker must be able to relay information to the upstream node prior to the first attacker.

A unique aspect to modeling the endairA attacker from our work in [10] and the Ariadne attacker is that we demonstrate how Promela models can generate attacks based on colluding attackers. In the model, an attacker node is able to discern if a colluding attacker has added its id to the route discovery path and it can react accordingly to initiate the attack.


4 Automated Topology Generation and Analysis

The protocol and attacker models allow us to use SPIN to examine if a message sequence exists that allows an attack for the given network topology being analyzed. In [10] we chose an enabling attacker topology based on static analysis; however, a complete automated analysis capability should not rely on manually choosing an enabling topology. One of the biggest impediments to manual analysis methods is the intuition to choose a network configuration in which the attack exists.

To solve this problem, we use our SPIN models to evaluate all network topologies for N number of nodes. The evaluation process consists of specifying N and evaluating the model against each possible topology. The process relies on duplicating the Promela file for each topology, adapting the network connectivity array to reflect each configuration.

The network and corresponding connectivity array in Figure 2 represent a symmetric (i.e., bi-directional) network. The shaded areas indicate which array portions are symmetric. If modeling an asymmetric (i.e., directional) network, the symmetric areas within the connectivity array do not exist. In the asymmetric case the number of topologies is determined by:

$$2^{(N-1)} \times 2^{(N-1)} \times \dots \times 2^{(N-1)} = 2^{N(N-1)} .$$



When modeling symmetric networks, the number of topologies is based on the possibilities for the shaded areas to one side of the zero diagonal. Since the number of elements that determine the symmetric topology are half of the asymmetric case, we can calculate the number of symmetric topologies by: $2^{N(N-1)/2}$.

In this research, we analyzed the modeled protocols using a 5-node symmetric network. With $N = 5$, a total of 1,024 unique topologies are possible. Following the complete analysis process shown in Figure 1, the remaining steps consist of the Topology Generation Engine, the Evaluation Engine, and the Reporting Engine. These steps utilize Perl routines to generate, evaluate, and report on the attacker capability for all possible network topologies for a given network size N .

The Topology Generation Engine generates a new Promela file for each possible network topology for the given network size N . The 5-node symmetric topology array can be illustrated according to Figure 5.

	0	1	2	3	4
0	0	val-9	val-8	val-7	val-6
1	val-9	0	val-5	val-4	val-3
2	val-8	val-5	0	val-2	val-1
3	val-7	val-4	val-2	0	val-0
4	val-6	val-3	val-1	val-0	0

Fig. 5. 5-Node Symmetric Topology Array Values

By listing the network connectivity array values in this manner, we can represent the upper (or lower) half of the array with a binary string value as illustrated in Figure 6. We iterate the binary string through all possible values to generate all possible topologies, starting with the base topology that stores the initial network as fully disconnected (i.e., all links set to 0). Since $N = 5$, the binary string is enumerated between 0 and 1023 to represent all possible topologies. For each network topology we read the base Promela file and adjust the network connectivity array as appropriate for the current binary string representation, saving the output as *filename_case-#.pml*.

val-9	val-8	val-7	val-6	val-5	val-4	val-3	val-2	val-1	val-0
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

Fig. 6. Binary String Representation

The next step in the analysis process is the Evaluation Engine, which compiles and executes each Promela file for SPIN exhaustive analysis. Any successful attacks are stored in individual SPIN trail files that capture the attack event sequence. The final step occurs in the Reporting Engine, which reads each error file and associated Promela model file to report the discovered attacks. The information returned is the initiator, target, and attacker nodes, along with the corrupted path and the current network topology.

5 Evaluation Criteria and Results

In addition to modeling the protocol and attacker, we define the desired security property (φ) as:

$\varphi =$ all returned routes must exist in the current network topology.

To evaluate the property in SPIN we add an analysis check for the path once it is received and accepted by the node that initiated the route discovery process. The node evaluates the returned path to ensure each link exists in the model's connectivity array. If any link check fails, an assertion violation is raised and SPIN halts execution, creating a trail file that lists the event sequence leading to the failure.

Our automated process ensures all topologies for a given network size N are generated and subsequently evaluate using SPIN. The following analysis assumes that $N=5$.

5.1 Attacking Ariadne

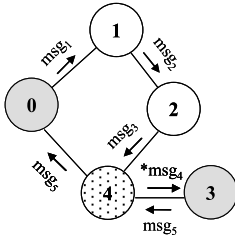
Running the analysis procedure against Ariadne and a malicious insider produces the following attack output as generated by the Report Engine:

```
Processing case: ari_nda-rreq_5node_case-611.pml
Initiator Node: 0   Target Node: 3   Attacker Node: 4   Corrupted Path: 0 - 1 - 4 - 3
The topology is:
net_con[0].to[0] = 0 net_con[0].to[1] = 1 net_con[0].to[2] = 0 net_con[0].to[3] = 0 net_con[0].to[4] = 1
net_con[1].to[0] = 1 net_con[1].to[1] = 0 net_con[1].to[2] = 1 net_con[1].to[3] = 0 net_con[1].to[4] = 0
net_con[2].to[0] = 0 net_con[2].to[1] = 1 net_con[2].to[2] = 0 net_con[2].to[3] = 0 net_con[2].to[4] = 1
net_con[3].to[0] = 0 net_con[3].to[1] = 0 net_con[3].to[2] = 0 net_con[3].to[3] = 0 net_con[3].to[4] = 1
net_con[4].to[0] = 1 net_con[4].to[1] = 0 net_con[4].to[2] = 1 net_con[4].to[3] = 1 net_con[4].to[4] = 0
```

Further investigation into the associated SPIN trail file shows the message sequence leading to the route corruption attack. Figure 7 illustrates the message sequence for the identified topology that returns the corrupted route. Recall we are not explicitly modeling the signatures; however, the attacker actions from the trial file result in the complete attack. The attacker node removed node 2 from msg_3 , added itself to the current accumulated path, computed the matching hash value, and transmitted the corrupted path in msg_4 . The resulting corrupt path is returned in msg_5 as $0-1-4-3$. This attack violates both the security property φ and an original Ariadne security claim. In [11], the Ariadne authors claim that a single malicious insider cannot remove a node during route discovery. This attack was also previously indicated in [12] using the simulatability method and subsequently reported using visual inspection techniques. This research demonstrates the ability to automatically discover the attack through model checking techniques.

Analysis over Ariadne against a malicious outsider reports a similar attack, with the exception that the attacker node 4 does not add itself to the accumulated path in msg_4 . The resulting corrupt path is $0-1-3$. This attack violates both the security property φ and an original Ariadne security claim. In [11], the Ariadne authors claim an attacker with no comprised keys and any number of attacker nodes, can only perform a wormhole attack or force the protocol to choose an attacker desired path by rushing,

which occurs when the attacker node responds to the route discovery process faster than the protocol expects. We have shown how a malicious outsider can actively change the embedded routing path. This attack is the first to our knowledge that shows a malicious outsider can actively corrupt the Ariadne route discovery process. In order to drop a node from the accumulated path during the *rreq*, the attacker requires only the ability to generate the appropriate hash value, based on capturing a hash from an upstream node in the path. The attacker also must be able to relay the return *rrep* to any upstream neighbor in the unicast *rrep*.



```

msg1 = (rreq, 0, 3, id, h0, (), ())
      h0 = H[0, initiator-target secret]
msg2 = (rreq, 0, 3, id, h1, (1), (sig1))
      h1 = H[1, h0]
      sig1 = SK1{rreq, 0, 3, id, h1, (1)}
msg3 = (rreq, 0, 3, id, h2, (1, 2), (sig1, sig2))
      h2 = H[2, h1]
      sig2 = SK2{rreq, 0, 3, id, h2, (1, 2), (sig1)}
*msg4 = (rreq, 0, 3, id, h4, (1, 4), (sig1, sig4))
      h4 = H[4, h1] = H[4, H[1, h0]]
      sig4 = SK4{rreq, 0, 3, id, h4, (1, 4), (sig1)}
msg5 = (rreq, 0, 3, (1, 4), (sig1, sig4), (sig2))
      sig3 = SK3{rreq, 0, 3, id, h4, (1, 4), (sig1, sig4)}

```

Fig. 7. Ariadne Attack Sequence

5.2 Attacking endairA

Running the analysis procedure against endairA and two colluding malicious insiders produces the following attack output as generated by the Report Engine:

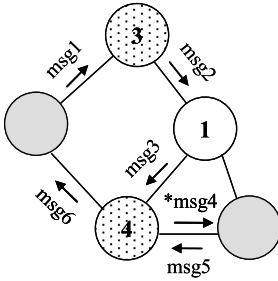
```

Processing case: end_nda-2_5node1_case-250.pml
Initiator Node: 0   Target Node: 2   Attacker Node: 4   Corrupted Path: 0 - 3 - 4 - 2
The topology is:
net_con[0].to[0] = 0 net_con[0].to[1] = 0 net_con[0].to[2] = 0 net_con[0].to[3] = 1 net_con[0].to[4] = 1
net_con[1].to[0] = 0 net_con[1].to[1] = 0 net_con[1].to[2] = 1 net_con[1].to[3] = 1 net_con[1].to[4] = 1
net_con[2].to[0] = 0 net_con[2].to[1] = 1 net_con[2].to[2] = 0 net_con[2].to[3] = 0 net_con[2].to[4] = 1
net_con[3].to[0] = 1 net_con[3].to[1] = 1 net_con[3].to[2] = 0 net_con[3].to[3] = 0 net_con[3].to[4] = 0
net_con[4].to[0] = 1 net_con[4].to[1] = 1 net_con[4].to[2] = 1 net_con[4].to[3] = 0 net_con[4].to[4] = 0

```

Figure 8 illustrates the attacker message sequence revealed in the associated SPIN trail file.

To the best of our knowledge, we provide the first active route corruption attack against endairA performed by two malicious insiders. During the *rreq*, node 4 removes the intermediate node between itself and the colluding attacker node 3. During the *rrep*, node 4 signs for both itself and node 3 since the colluding nodes share their cryptographic keys. The initiator believes *msg₆* is from node 3, since the signatures are correct and the initiator cannot physically identify that node 4 actually sent the message. The resulting corrupt path is 0-3-4-2. This attack violates both the specified security property ϕ and an original endairA security claim. In [12], the authors claim that endairA cannot be attacked by colluding adversaries unless the adversaries are local neighbors to one another.



```

msg1 = (rreq, 0, 2, id, ( ))
msg2 = (rreq, 0, 2, id, (3))
msg3 = (rreq, 0, 2, id, (3, 1))
*msg4 = (rreq, 0, 2, (3, 4))
Node 4 drops node 1
msg5 = (rrep, 0, 2, (3, 4), (sig2))
sig2 = SK2{rrep, 0, 2, (3, 4)}
msg6 = (rrep, 0, 2, (3, 4), (sig2, sig4, sig3))
sig4 = SK4{rrep, 0, 2, (3, 4), (sig2)}
sig3 = SK3{rrep, 0, 2, (3, 4), (sig2, sig4)}

```

Fig. 8. endairA Attack Sequence

6 Conclusion

In this paper, we provide an automated model checking technique to evaluate route corruption attacks against the route discovery phase for on-demand source routing protocols. The existing security analysis techniques used to evaluate security properties in MANET routing protocols are not automated or do not provide exhaustive attacker analysis.

Our automated analysis process uses the SPIN model checker to examine all message event sequences for a given topology. We additionally provide exhaustive topology generation to ensure all possible network topologies for a given network size N are evaluated. Each topology is subsequently analyzed with SPIN, identifying any configuration and corresponding event sequence producing attacks along with the attack sequence.

Through the use of our automated evaluation process we identified previously undocumented attacks against the Ariadne and endairA protocol and have shown the feasibility of using model checking to automate attack analysis in MANET routing protocols.

Our future work includes introducing more detail into the protocol models. As we weigh the low level protocol details against the required state-space, our goal is to develop a protocol entirely in the model checking paradigm to meet its goals and provide subsequent compilation to produce executable routing code for network devices.

References

- [1] Royer, E.M., Toh, C.-K.: A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications* 6, 46–55 (1999)
- [2] Hu, Y.C., Perrig, A.: A survey of secure wireless ad hoc routing. *IEEE Security & Privacy* 2, 28–39 (2004)
- [3] Argyroudis, P.G., O’Mahony, D.: Secure routing for mobile ad hoc networks. *IEEE Communications Surveys & Tutorials* 7, 2–21 (2005)
- [4] Djenouri, D., Khelladi, L., Badache, A.N.: A survey of security issues in mobile ad hoc and sensor networks. *IEEE Communications Surveys & Tutorials* 7, 2–28 (2005)
- [5] Andel, T.R., Yasinsac, A.: Surveying Security Analysis Techniques in MANET Routing Protocols. *IEEE Communications Surveys & Tutorials* 9, 70–84 (2007)

- [6] Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 279–295 (1997)
- [7] De Renesse, F., Aghvami, A.H.: Formal verification of ad-hoc routing protocols using SPIN model checker. In: 12th IEEE Mediterranean Electrotechnical Conference, vol. 3, pp. 1177–1182 (2004)
- [8] Wibling, O., Parrow, J., Pears, A.: Automatized Verification of Ad Hoc Routing Protocols. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 343–358. Springer, Heidelberg (2004)
- [9] Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. *Journal of the ACM* 49, 538–576 (2002)
- [10] Anđel, T.R., Yasinsac, A.: Automated Security Analysis of Ad Hoc Routing Protocols Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA 2007), Wrocław, Poland, pp. 9–26 (2007)
- [11] Hu, Y.-C., Perrig, A., Johnson, D.B.: Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks. *Wireless Networks* 11, 21–38 (2005)
- [12] Ács, G., Buttyán, L., Vajda, I.: Provably secure on-demand source routing in mobile ad hoc networks. *IEEE Transactions on Mobile Computing* 5, 1533–1546 (2006)
- [13] Ryan, P., Schneider, S.: *Modelling and Analysis of Security Protocols*. Addison-Wesley, Harlow (2001)
- [14] Burmester, M., Van Le, T.: Secure multipath communication in mobile ad hoc networks. In: International Conference on Information Technology: Coding and Computing (ITCC 2004), vol. 2, pp. 405–409 (2004)
- [15] Kotzanikolaou, P., Mavropodi, R., Douligeris, C.: Secure Multipath Routing for Mobile Ad Hoc Networks. In: Second Annual Conference on Wireless On-demand Network Systems and Services (WONS 2005), pp. 89–96 (2005)
- [16] Awerbuch, B., Holmer, D., Nita-Rotaru, C., Rubens, H.: An on-demand secure routing protocol resilient to byzantine failures. In: 3rd ACM Workshop on Wireless Security, pp. 21–30. ACM Press, Atlanta (2002)
- [17] Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. *ACM Computing Surveys* 28, 626–643 (1996)
- [18] Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (1999)
- [19] Meadows, C.: The NRL Protocol Analyzer: An Overview. *The Journal of Logic Programming* 26, 113–131 (1996)
- [20] Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
- [21] Song, D., Berezin, S., Perrig, A.: Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security* 9, 47–74 (2001)
- [22] Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. *Theoretical Computer Science* 367, 203–227 (2006)
- [23] Yang, S., Baras, J.S.: Modeling vulnerabilities of ad hoc routing protocols. In: 1st ACM Workshop on Security of Ad hoc and Sensor Networks, pp. 12–20 (2003)
- [24] Maggi, P., Sisto, R.: Using SPIN to Verify Security Properties of Cryptographic Protocols. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 187–204. Springer, Heidelberg (2002)
- [25] Ruys, T.C.: Towards effective model checking. Department of Computer Science. University of Twente, Deventer, The Netherlands (2001)
- [26] Anđel, T.R., Yasinsac, A.: The Invisible Node Attack Revisited, pp. 686–691. *IEEE SoutheastCon*, Richmond (2007)

Model Checking Abstract Components within Concrete Software Environments

Tonglaga Bao and Mike Jones

Computer Science Department, Brigham Young University,
Provo, UT
{tonga, jones}@cs.byu.edu

Abstract. In order to model check a software component which is not a standalone program, we need a model of the software which completes the program. This is typically done by abstracting the surrounding software and the environment in which the entire system will be executed. However, abstracting the surrounding software artifact is difficult when the surrounding software is a large, complex artifact. In this paper, we take a new approach to the problem by abstracting the software component under test and leaving the surrounding software concrete. We compare three abstraction schemes, bitstate hashing and two schemes based on predicate abstraction, which can be used to abstract the components. We show how to generate the mixed abstract-concrete model automatically from a C program and verify the model using the SPIN model checker. We give verification results for three C programs each consisting of hundreds or thousands of lines of code, pointers, data structures and calls to library functions. Compared to the predicate abstraction schemes, bitstate hashing was uniformly more efficient in both error discovery and exhaustive state enumeration. The component abstraction results in faster error discovery than normal code execution when pruning during state enumeration avoids repeated execution of instructions on the same data.

1 Introduction

One way to manage the complexity of large software engineering projects is to factor the problem into cooperating components. Each component must then be written to implement that component's functionality in the context of other components. The modular verification problem is the problem of showing that each component behaves correctly in the execution environment created by the other components.

We focus on modular formal verification when no formal model of the surrounding software exists. Such a formal model would most likely be missing because it is too expensive to generate. This can happen, for example, when the implementations of some components deviate from their specifications but the nature of those deviations has not been precisely characterized. Or, there may simply be no formal model of the entire system. In these situations, the key

verification question is: does the component under test satisfy a set of properties in the context provided by the other components even though there is no formal model of the other components?

Modular verification without a formal specification of the surrounding software is important to engineers who must implement and verify components for existing software. This problem can occur when existing software is upgraded or when software development organizations decide to use formal verification after having developed a significant amount of software. In these and similar cases, a technique is needed to verify formal properties of new components in the context of existing software.

Formal approaches to modular software verification have been proposed for quite some time. However, in every case, the component is left concrete while the environment is abstracted. This poses two problems. First, the component itself may be too complex to admit formal analysis without abstraction. Second, abstraction of the software environment requires a formal model of the software environment. This means that the surrounding software must be converted to a formal model during or before abstraction. For large software environment, this is prohibitively expensive.

Fortunately, a precise model does exist for every executable software artifact. This model, though difficult to describe analytically, is simply the behavior of the software on the computational platform on which it was intended to be executed. The idea of defining semantics through execution is not new and lies at the foundation of advances in explicit state-space representation model checking [6,10,13].

Similarly, discovering the precise definition of software meaning through execution lies at the core of our approach. The main difficulty is creating an efficient interface between the abstract component and unabstracted surrounding software. The interface must be defined so that execution can be quickly passed to concrete software across the abstraction boundary. For example, over-approximation schemes are unsuitable because a single abstract state may represent thousands of concrete states—many of which are infeasible. Each of these concrete states would need to be passed and executed by the surrounding software.

In this paper, we present a new approach to component verification in which the component is abstracted and the environment is left concrete. We evaluate three abstraction techniques which are compatible with this approach to modular verification.

We have implemented our idea in the SPIN model checker [6]. Given a program written in C, we translate it to a model with PROMELA proctypes and C functions using an extension of the CIL compiler [11]. Abstract components are modeled by PROMELA proctypes and the surrounding software is left as C code with little modification.

Previously, Holzman and Joshi extended PROMELA to have better communication with programs written in C using the `c_code` and `c_track` mechanism. The C code enclosed inside a `c_code` block is executed directly like a normal C

program. The `c_track` block encloses variables from the C program which will be tracked. Tracked variables are included in the PROMELA state vector. `c_track` declarations can be marked as “matched” or “unmatched” to indicate that the variables will be stored into the hash table or not.

Inside a component, we translate branching instruction to PROMELA in order to expose the program control flow to the model checker. We use `c_code` and `c_track` mechanism. The environment is enclosed in `c_code` block and the component is a mixed model with `c_code` instructions interspersed with PROMELA instructions. In the component, variables are strategically tracked and matched in order to support data abstraction. When execution reaches the component boundary, the concrete state in the state exploration queue or stack can be passed directly to the software environment.

We tried three potential abstraction scheme with the component. The first one is described in [12]. It is an under-approximated abstraction scheme. Refinement is achieved by checking the preciseness of the abstraction through weakest pre-conditions. The second one is described in [9], in which refinement is obtained by checking the value range of the variables. Third one is abstraction through supertrace.

Our approach can verify software components that interact with complex surrounding software. For example, the surrounding software might contain mathematics for which first order logic theorem provers, as used in predicate abstraction, can provide no useful information. This can happen even for relatively simple operations like multiplying two variables or for more complex operations like exponentiation or trigonometry. The environment might also contain pointers and references which are too difficult to track in a formal semantic model. Our model runs without problem in these cases since we simply execute the environment instead of reasoning about it.

It would seem natural, at this point, to just execute the component instead of reasoning about it as well. After all, executing the environment sidesteps range of thorny semantic issues.

However, the surrounding software just provides the environment in which to verify the component so simply it, with no attempt at analysis or verification, is sufficient. But the component is the target of the verification effort, so tools, such as abstraction, to improve the utility verification effort are warranted.

The main contribution of this paper is a model of components which supports data abstraction for C programs. Our model supports model checking inside unmodified, complex software environments and allows nested function calls between component and environment. Our state exploration algorithm is a simple modification of standard state enumeration algorithms. The value of this work is that it enlarges the class of C programs to which standard state enumeration algorithms can be applied.

Experimental results show this algorithm is indeed able to deal with complex surrounding software with complex control structures and suggest that our algorithm can be used as a complementary method of testing to discover error faster

by covering the state space faster. It is a good way to deal with programs that are too complex to be reasoned about by traditional model checking techniques.

Our test result shows abstracting the component using supertrace outperforms the other two abstraction schemes in terms of verification speed and error discovery speed. It also outperforms concrete execution in terms of error discovery speed when the state space includes many copies of the same large region of states. In this case, concrete state exploration still needs to execute the already visited states since it does not have a memory about what state is already visited. Abstraction, on the other hand, can jump out of the partial state space it already visited and start to explore new state space faster.

In the next section we survey closely related work in abstraction for explicit model checking and component-based verification. Section 3 contains an explicit model checking algorithm for exploring the state space of abstract components in concrete environments. Section 4 shows how we implement this approach in SPIN. Section 5 provides experimental results. We close with conclusions and ideas for future work in Section 6.

2 Related Work

In this section, we first discuss the prior work on which this work is built, then present related work in abstraction and environment generation.

One way to view this paper is an extension of Holzmann and Joshi’s work on model-driven software verification. In [7], Holzmann and Joshi describe a method for mixing C code with PROMELA models such that data abstractions can be performed on C variables. We build on their work by creating an abstraction model for components based on data abstractions which under-approximate the state space.

Our approach to modeling the software environment is fundamentally different than that used by Holzmann and Joshi. While Holzmann and Joshi use PROMELA instructions as the test harness for C code, we use code from the surrounding software as the test harness. Our approach is appropriate when a PROMELA model of the surrounding software is too expensive to either build or execute. Our approach also admits the use of PROMELA as a test harness for parts of the system—such as for input from users.

FeaVer [8] also verifies C program or part of a C program. It uses a tool called Modex to extract a PROMELA model from the C source code. When verifying some specific functions of a program, Modex requires the user to provide a test harness in which to test these functions. Our approach has similar goals and is built on PROMELA commands, such as `c_code` and `c_decl`, which were used in Modex. However, our approach simplifies the process of defining a PROMELA model and test harness from a C program and our approach is designed to verify data operations of C programs rather than just the concurrent behavior of a threaded program. Our approach supports concurrency but the implementation does not. Implementation of support for concurrency is a topic for future work.

In order to simplify the translation between branching instructions to PROMELA in components, we used the CIL [11] compiler to compile C programs into a C intermediate language, and then translate the resulting C intermediate language into PROMELA. CIL can compile all valid C programs into a few core syntactic constructs. This simplifies translation while allowing us to handle a large subset of C.

2.1 Environment Generation

When verifying only part of a program, the problem of simulating the program environment can be split into two parts: generating the test harness and simulating the surrounding software. The test harness provides the inputs to the program. Most existing work in component verification combines these two problems together to provide an abstract environment to the program under test which simulates both the test harness and the surrounding software.

Bandera [4] divides a given java program into two parts: the unit under test and the environment. The component under test is verified concretely while the environment is abstracted to provide the unit the necessary behaviors. The abstract environment is obtained through a specification written by the user or through the source code analysis. The drawbacks of this approach are: first, abstracting the environment is a time consuming and error prone process. Moreover, it is hard to avoid the semantic gap between the concrete environment and the abstract representation of it. Second, if the component under test is complex, then model checking it concretely might cause state space explosion.

Slam [2] is a software model checker developed by Ball et al. to verify Windows device drivers. A device driver is a program which communicates with the operating system kernel on behalf of a peripheral device such as a mouse or printer. The surrounding software for a device driver is the entire kernel, so device driver verification requires a model of the kernel in order to close the execution environment. Since the Windows kernel is a large, complex program with no formal specification, manually generating a formal model of the kernel is expensive and error-prone.

Instead, Ball et al. [1] generate kernel models via merging different abstractions of the kernel procedure. Slam selects a set of device drivers that utilize a specific kernel procedure. These drivers then used as a training set by linking each driver with this specific kernel procedure and executing it in Slam. Slam automatically generates Boolean abstractions for the kernel procedure with each device driver. The Boolean abstractions for the procedure can be extracted and merged to create a library of Boolean programs which are used to verify future device drivers which utilize that kernel procedure. In this work, we take a different approach. Instead of abstracting the kernel, we abstract the device driver and leave the kernel software concrete. In our approach, a device driver would be verified by abstracting the device driver then verifying the abstract model of it in the context of the actual Windows kernel. An external environment that generates IO requests to drive verification would also be needed.

2.2 Abstraction

We abstract the component under test during verification. Abstraction is a widely studied topic in software model checking. Abstraction methods can be split based on whether they over approximate and under approximate the reachable states of a program. SLAM, which was mentioned previously, uses predicate abstraction [2] to abstract the device drivers and corresponding procedures in the kernel. SLAM uses over approximation abstraction techniques.

In this work, we need an abstraction scheme which stores abstract states in the hash table but uses concrete states in the queue of states to be expanded because this simplifies passing states between component and environment.

We have investigated three abstraction schemes which have this property: under approximation using predicates which requires a theorem prover for refinement, under approximation using predicates which do not require a theorem prover for refinement and bitstate hashing. Results for component verification using each of these abstractions are given later.

Pasareanu et al. proposed an under approximation abstraction approach which uses predicates and a theorem prover to manage refinement [12]. They explore the concrete state space, push concrete states into the stack and store their corresponding abstract states into the hash table. Abstract states are generated by evaluating a set of predicates on variable values. The state vector includes one bit per predicate and each bit is set based on the predicate's truth value. Refinement is done by examining each transition relation in terms of the existing predicates. If the existing predicates do not imply the weakest pre-condition of the next state in terms of the current transition, then the abstraction is not precise and the weakest pre-condition is added into the existing predicate set. Otherwise, abstraction is precise and no refinement is necessary. Since the state exploration is driven only by reachable concrete states in the stack, the abstraction never introduces new behaviors to the system. The abstraction misses behaviors when two concrete states satisfy the same set of predicates but lead to different program states.

Kudra and Mercer hypothesized that under approximation with predicates could be made faster by eliminating the theorem prover in refinement checking [9]. Pasareanu posed the same hypothesis, but eliminated the theorem prover by always assuming that the abstraction is imprecise. Kudra and Mercer eliminated the theorem prover by picking predicates which can be evaluated without a first order logic theorem prover and tracking extra data required to test the validity of those predicates. For each abstract state, they store the minimum and maximum value of each variables. When the minimum and maximum value of a variable is different, then they know this abstract state corresponds to at least two concrete states and needs to be refined. Eventually, this method will explore all the concrete states. However, before exploring all the concrete states, it covers more of the state space in less time in order to find errors faster. For some programs, eliminating the theorem prover results in faster error discovery because states could be generated more quickly.

Bitstate hashing stores concrete states in the queue of states to be expanded, but represents visited states using a single bit (or small set of bits) in the hash table [5]. In bitstate hashing, the location for a state in the hash table is determined by applying a hash function directly to the entire state vector. The resulting value is used as an index into the hash table and the bit at that index is set to true to indicate that the state has been visited. This abstraction misses behaviors when two concrete states hash to the same value but result in different program behaviors.

For the purposes of this work, under approximating predicate abstraction and bit state hashing are the same process with the difference that predicates are used to hash states in predicate abstraction while a hash function is used to hash states in bit state hashing. In this sense, predicate abstraction is a semantically based abstraction in which well-chosen predicates differentiate concrete states based on their meaning while bit state hashing is purely a structural abstraction in which the meanings of data values are ignored by the hashing function.

3 Algorithm

In this section first we describe the state exploration algorithm, then we discuss how abstraction is done for components.

3.1 State Exploration

Figure 1 shows our state enumeration algorithm for verifying abstract components in the context of concrete software. We have omitted property checking in order to simplify the presentation. Safety property checking can be added.

Given a program *prog*, we call the procedure **init** in line 1. \mathcal{P} stores the initial set of predicates in line 2. The initial set of predicates is all of the guards in the entire program. \mathcal{P}_{new} stores the set of new predicates used to refine the abstraction after each iteration and is initialized to the empty set in line 3. We check the starting instruction of the program in line 6. If the starting instruction is in the environment, then we execute instructions in the environment until control returns to the component. An instruction is in the environment if the state generated by that instruction is in the environment. The environment is specified as a range of program counter values, so an instruction is in the environment if it generates a state with a program counter value which lies outside that range. The environment execution functions returns the first state which lies in the component.

Now that we have a start state which lies in the component, we push it on the stack at line 8 and begin verification by calling the **component** function in line 9. When using predicate abstraction with refinement, we repeatedly verify the entire program until no refinement is necessary, as shown in line 10.

The **component** function is a variation of explicit state exploration in which abstract states are stored in the hash table, concrete states are stored in the stack and transitions which leave the component are serially executed without

storing states. We obtain the transition out of the current state in line 19. If that transition exits the component, then we execute instructions in the environment until an instruction returns control to the component at line 20. The next state is generated by applying the current transition to the current state, line 21, or in the **environment** function at line 28. State exploration then continues by pushing the next state into the stack in line 22.

In the **environment** function, when the next instruction is in the environment, we will simply execute it at line 28. When the next instruction returns control to the component, we return the first state which lies in the component at line 31.

```

1  proc init(prog)
2     $\Phi := \text{Guards}(\text{prog})$ 
3     $\Phi_{new} := \emptyset$ 
4    do
5       $\Phi := \Phi \cup \Phi_{new}$ 
6      if start_instr  $\in$  environment then
7        start_state = environment(start_instr, start_state)
8        push(start_state)
9        component()
10     while  $\Phi_{new} \not\subseteq \Phi$ 
11  end
12
13 proc component()
14   while size(stack)  $\neq$  0
15     cur_state = top(stack)
16      $\alpha = \text{abstract}(\text{cur\_state})$ 
17     if ( $\alpha \notin$  hash table)
18       insert  $\alpha$  into hash table
19       cur_inst = transition(cur_state)
20       if (cur_inst  $\notin$  comp) next_state = environment (cur_inst, cur_state)
21       else next_state = cur_inst(cur_state)
22       push (next_state)
23     else pop(stack)
24  end
25
26 proc environment(inst, state)
27   do
28     next_state = inst(state)
29     inst = transition (next_state)
30   while (inst  $\in$  environment)
31   return (inst(next_state))
32 end

```

Fig. 1. State enumeration algorithm that combines under-approximation with concrete execution

3.2 Abstraction

The **abstract** function takes a concrete state s and returns an abstract state. abs denotes an abstract state represented by a bit vector. The algorithm requires an abstraction which stores concrete states in the stack and stores abstract states in the hash table. If abstract states are stored in the stack, then passing control between the component and environment at lines 7 and 20 of Figure 1 would be more difficult because we would need to create a concrete state which represents the abstract current state.

<pre> 1 proc abstract(s) 2 foreach $\phi_i \in \Phi$ do 3 if $\phi_i(s)$ then $abs_i := 1$ 4 else $abs_i := 0$ 5 if (refinement check is not valid) 6 addNewPreds(Φ_{new}) 7 return abs 8 end </pre>	<pre> 1 proc abstract(s) 2 $abs = \text{hash}(s)$ 3 return abs 4 end </pre>
(a), (b)	(c)

Fig. 2. Three abstraction schemes which store concrete states in the stack and abstract states in the hash table and are compatible with our approach to component verification. (a), (b) Predicate abstraction with or without a theorem prover as in [12] and [9], the difference being that the precision check at line 6 is done with or without a theorem prover. (c) Bit state hashing [5].

Figure 2 shows the abstractions which we have investigated as part of our component model in this paper. The first pair of abstractions, (a) and (b), are shown together because they differ only in the manner in which refinement is checked at line 6. In both cases, refinement is checked by determining if the abstraction of the previous state implies the abstraction of the next state with substitutions made using assignments in the instruction between the states. A detailed description can be found in [12]. In Pasareanu’s case, the validity of the implication is checked using an automatic theorem prover. In Kudra’s case, the validity of the implication can be checked by determining if the variable’s value falls within a certain range.

Figure 2(c) shows bitstate hashing interpreted as an abstraction function. This is included to clarify the relationship between bitstate hashing and predicate abstraction as used in our work. Bitstate hashing is an abstraction in which a hash function is used to compute the abstract state. The abstract state is not stored directly in the hash table, but is used as an address at which to set a bit indicating that a state with that hash code has been visited. Refinement is not possible using bitstate hashing so the predicate set Φ_{new} is not updated and the while loop in line 10 of Figure 1 is simply ignored. However, bistate hashing can be made more precise by re-running the algorithm with a different hash function.

Each of the three abstractions in Figure 2 under approximate the state space. Every abstract state corresponds to at least one concrete state since the **abstract** function is only applied to already existing concrete states. States can be missed when two concrete states have the same abstract representation and only one of them is expanded. Like other under-approximation techniques, every error found using our algorithm is a feasible error, but finding no errors does not guarantee that the component is error free.

For predicate abstraction, both with and without a theorem proving support, our under-approximation scheme can not be refined to include all behaviors of the system since we ignore system behaviors in the environment and these parts can not be included in the refinement check. More specifically, substituting the right side of an assignment for the left side of the assignment when that variable appears in the abstraction predicates can not be done safely for sequences of transitions that pass through the environment. Multiple syntactic substitutions for the transitions in the environment can mask program behavior and cause the precision check to succeed when behaviors have been ignored.

Interestingly, this loss of information in the precision check is adjustable. When the component grows to include the whole system, it is no longer required to chain together transitions in the syntactic substitution for the precision check and the refinement process works as described in [12]. A detailed discussion of the properties of refinement for predicate abstraction in the context of our component modeling method can be found in [3].

4 Implementation

We have implemented the algorithm in SPIN model checker using CIL for pre-processing of C code. For this implementation, we have assumed that a function is the basic unit of a component or the environment. In other words, each function in a C program either belongs entirely to the component or belongs entirely in the environment. We group interesting functions together to be verified as a single component, and group everything else into the environment.

Each function in the component is translated into a proctype in SPIN to be verified. The functions in environment remain unchanged as C functions. The details of how a function is translated into a SPIN proctype and how the SPIN proctypes interacts with the functions in the environment are discussed below.

SPIN supports embedded C code by providing five different primitives identified by the following keywords: `c_code`, `c_track`, `c_decl`, `c_state`, and `c_expr`. Everything enclosed inside `c_code` block is compiled directly by GCC then executed and interpreted as one atomic PROMELA state. `c_track` specifies the C variables we want to track as part of the state vector. `c_track` can be used with the `Matched` or `UnMatched` keywords. `Matched` variables are stored both in the state stack and hash table, while `UnMatched` variables are only stored in the state stack. For example

```

1  main() {
2    int i;
3    for (i = 0; i < 10; i++)
4      if (i == 4)
5        break;
6      else i = i * 2;
7  }

```

Fig. 3. A simple C program

```

1  c_decl{ int i; char abs[predNum];}
2  c_track "&i" "sizeof(int)" "UnMatched"
3  c_track "&abs" "sizeof(abs)" "Matched"
4  proctype main() {
5    do
6      :: c_expr{i < 10} → c_code{i++; abstraction();}
7      if
8        :: c_expr{i == 4} → break;
9        :: else → c_code{i = i * 2; abstraction();};
10     fi;
11   od;
12 }

13 c_code {
14   void abstraction(){
15     for ( i = 0; i < predNum; i ++ )
16       if (preds[i] == true)abs[i] = 1;
17       else abs[i] = 0;
18   }
19 };

```

Fig. 4. The promela code generated from the C code shown in figure 3

```
c_track "&i" "sizeof(int)" "UnMatched"
```

indicates C variable i will be tracked but not matched, and

```
c_track "&i" "sizeof(int)" "Matched"
```

indicates i is both tracked and matched. A kind of forgetful data abstraction can be obtained by tracking a variable but not matching it [7].

If a C function is contained within the component, then we translate it into an equivalent PROMELA proctype by enclosing non-branching statements in `c_code` blocks and translated branching statements into PROMELA. We do this in three steps.

First, we use CIL compiler to translate C into the C intermediate language (CIL) [11]. The CIL compiler compiles a valid C program into a C program which has a reduced number of syntactic constructs. By translating from C

to a syntactic subset of C using CIL, we obtain a C program with simpler syntax, which makes translation from C to PROMELA much easier. We have implemented an extension of the CIL compiler to translate the CIL language into PROMELA.

Next, we enclose each non-branching statement of the component in a `c_code` block, so that every statement of the component is treated as a single PROMELA transition.

Finally, although each statement of the component is enclosed by a `c_code` block, control statements are translated entirely into PROMELA. This allows SPIN to expose the branching structure of the component during verification.

As an example, consider the C code in Figure 3 which is translated into the PROMELA code shown in Figure 4 assuming the use of a predicate abstraction. Abstraction through bit state hashing is simple as it does not require additional arrays for predicates or their Boolean values. In Figure 4, `predNum` gives the number of predicates, `abs[predNum]` is a vector that contains abstract states, and `preds[predNum]` contains the given predicates. The function `abstraction()` on line 14 computes the abstraction by evaluating the predicates then storing their evaluation in the `abs[i]` vector of bits.

Predicate abstraction in the component is achieved by tracking and matching a bit vector. We declare an array of bits, called `abs[]`, which are also marked as `Matched`. After every assignment statement or function call in the component, we insert a call to a C function named `abstraction()`. `abstraction()` checks the current set of predicates and sets the corresponding bit values in `abs[]`. We then store only the values of `abs[]` and ignore all other variables.

Abstraction through bit state hashing is achieved by using SPIN's built-in implementation of hashing.

The software environment is modeled by wrapping it in a single `c_code` block. This means that segments of the environment are executed as needed by SPIN based on the behavior of the instructions in the component.

The next issue in the implementation is managing function calls within and between components and the environment. When translating C into a mixed C-and-PROMELA model, the most difficult problem is enforcing execution order in the presence of function calls. Since SPIN is designed to run concurrent code, we need to do some work to force it to avoid inappropriate interleavings in otherwise sequential programs. On the other hand, modeling concurrent components is more difficult because functions in the environment must support multiple active invocations. For purely sequential components and environments, there are four cases to consider depending on the location of the caller and the callee.

Inside a component, when a proctype calls a proctype, channels are used to enforce the order of the execution. An example is given in Figure 5. In Figure 5, `proctype main` calls `proctype proc` in line 7, and waits for `proc` to return at line 8. `proctype proc` signals the end of execution by pushing 1 into the channel at line 26. This signals the main process that it may resume execution.

When the code in a proctype calls a function in the environment, we pass an integer pointer `rtnFunFlag` with the function call. The callee indicates its return

```

1  proctype main() {
2    c_code{fun(rtnFunFlag, -1)};
3    do
4      :: c_expr{*rtnFunFlag == 1} → break;
5      :: else → skip;
6    od;
7    run proc();
8    c ? 1;
9  }
10 c_code {
11   fun(int* rtnFunFlag, int callerLabel){
12     if (callerLabel) goto label;
13     addproc(1);
14     goto end;
15     label:
16     *rtnFunFlag = 1;
17     end: ;
18  }
19 proctype proc(chan c, int callerID, int callerLabel) {
20   c_code{
21     if (callerID){
22       funarray[callerID](callerLabel);
23       goto end;
24     }
25   }
26   c ! 1;
27   c_code { end: ; };
28 }

```

Fig. 5. Functions in C translated into PROMELA

by setting `rtnFunflag` to 1 at the end of the function, at which time the caller continues execution. This is illustrated by figure 5 in lines 2 to 6 and line 16.

The most difficult case is when a function in the environment calls a proctype in the component. Since the `c_code` block is designed to be executed without interruption, if there is a call to a proctype in the middle, we must break out of the `c_code` block and run the proctype using just straight C. In the code generated by the SPIN, we find that calling a proctype is translated into an `addproc` function. In line 13 of figure 5, we add an `addproc` function to invoke the corresponding proctype `proc`. We pass the return program counter, `pc` value and function ID to `proc` so that it knows where to jump back to after execution. Then the caller function will jump out of the `c_code` block as shown in line 14. Then the `proc` begins execution and jumps back to the right place depending on the arguments.

When two functions in the environment call each other, it will be handled in unmodified C code with little additional effort. One important thing to note is that when a series of environment functions call each other, may be one of them may in turn call a function in the component, which means we need to stop

execution in the environment immediately and return to the component. In this case it is important to keep a stack of function names and labels so that each environment function knows where to jump back after the component function returns back to the environment.

5 Results

The implementation of the algorithm allows us to take C code and model check parts of it. The C code can include complex data structures with pointers, and calls to library functions.

We choose three models to illustrate the result. They are matrix multiplication, sorting algorithms, and a program that simulates the operating system’s dynamic storage allocator. The Matrix Multiplication and Sorting algorithm implementations are downloaded from the Internet. The dynamic storage allocator is taken from an assignment in an undergraduate operating system class.

Matrix Multiplication is a program that takes two matrices from the user and returns the product of those two matrices. This is an interesting problem for our component model because the code contains much data and many predicates, which makes it a good candidate for the predicate abstraction scheme. We supply 1000 pairs of matrices to the program. Each matrix has a user-defined number of column and rows. We insert an assert function to check that the dimension of the column of the first matrix equals to the row dimension of the second matrix. The result of the verification is shown in table [1](#).

Table 1. Matrix Multiplication, All Times in Seconds, Memory in MB, INFI indicates the result is not known because either time or space limitation is reached

matrix	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	120	270	5.3M	744	0	21	0.21	14
PA+TP	120	270	INFI	INFI	INFI	INFI	INFI	INFI
PA+NTP	120	270	INFI	INFI	INFI	INFI	INFI	INFI

In table [1](#), the first column gives the different abstraction schemes we test. PA+TP indicates predicate abstraction with theorem prover, and PA+NTP indicates predicate abstraction without theorem prover. In the first row, cLine is the number of lines of code in the component, eLine is the number of lines of code in the environment. There are also several library function calls which we do not include in the line number count. Matrix Multiplication uses library calls like “printf” and “assert”. States is the total number of states generated from the component, mem is amount of memory (in Mbytes) used to store the state space of the component, predicates is the total number of predicates used in the verification. Time is the total time (in seconds) needed to complete the verification without seeded errors, and eTime stands for total time to find seeded error. Match shows the number of states that are matched inside the hash table. We group several functions that do the main computation together to compose the

component and leave the rest of the software as the environment. This model consists of total of 5 million states. Bit state hashing performs best in matrix multiplication. Both of the other two algorithms fail to explore the total state space or find errors in the given time and space limit.

Table 2 also contains results for the matrix multiplication model, but this time we decrease the size of the component and increase the size of the environment by 80 lines. All three algorithms run to completion for this model. Observe that bitstate hashing generates the least number of states while TA+NTP generates the most. That is because bitstate hashing only needs one iteration of the entire program, but the other two do a refinement on their abstractions and continue exploring the whole state space until the state space covers all concrete states. PA+TP is the slowest in both error discovery and generating the whole state space. That is because it has to call the theorem prover to decide which, if any new predicates are needed for the refinement.

Table 2. Matrix Multiplication with smaller component, All Times in Seconds, Memory in MB

matrix	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	40	350	3718	2	0	0.01	0.001	0
PA+TP	40	350	12095	54	102	170	41	198
PA+NTP	40	350	192516	111	101	5.1	0.08	100

Table 3 shows the results for verifying a C model called sorting. It consists of several different sort algorithms. They are selection sort, insertion sort, bubble sort, and quick sort. We pick selection sort as a component. The property we check is asserting a value is less than the value after it in a list after returning back from the sorting functions. As the above models, bitstate hashing again outperforms the other two abstraction schemes.

Table 3. Sorting Model, All Times in Seconds, Memory in MB

sorting	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	44	110	8226	86	0	0.73	0.2	0
PA+TP	44	110	INFI	INFI	INFI	INFI	41	INFI
PA+NTP	40	110	491830	405	449	51	3.5	8604

In all the above models, bitstate hashing is by far more efficient than the other abstraction techniques. In fact, in the previous models, concrete exploration will be even more effective than bitstate hashing. However, there are several advantages to explore and store abstract states instead of simply executing them concretely. One advantage is through abstraction, the state space is covered faster because previously visited regions of the state space can be avoided through duplicate state detection using the bittable.

In table 4, we have a larger model that simulates part of an operating system which allocates, reallocates, and frees blocks of memory. This code has a bigger

Table 4. Malloc Model, All Times in Seconds, Memory in MB

malloc	cLine	eLine	matched time	
bitstate	100	2000	10	1.2
concrete	0	2100	0	169

and more complex environment compared with the other two models. In this model, the component is 100 lines of codes and the environment is 2000 lines of codes. Both the component and environment uses library calls like “malloc”, “realloc” etc. These libraries plus the environment make it difficult to model the code formally. By concretely executing them, we don’t need a formal model of them.

We add a loop to make part of the code executes repeatedly, and we put an assert function outside of the loop. The purpose of doing that is to see if bitstate hashing can find an error faster than concrete exploration by recognizing already visited states and going to another part of the state space. The experimental result shows that it takes concrete exploration 169 seconds to discover the error, but bitstate hashing find it in only 1.2 seconds. The reason for that is bitstate hashing is able to track the states. When it sees an already explored state, it will backtrack and explore the other part of the state space. Table 5 shows a similar result. In this test, we increase the size of the component. Bitstate algorithm again finds error faster than concrete exploration.

Table 5. Malloc Model, All Times in Seconds, Memory in MB

malloc	cLine	eLine	states	matched time	
bitstate	400	1700	142352	8	5.3
concrete	0	2100	0	0	169

6 Conclusion and Future Work

In this paper, we have presented a technique for component-based verification that supports abstraction of the component under test rather than the environment in which the component is embedded. The abstraction can be applied when the source code for, but not a model of, the surrounding software is available.

The main purpose of this approach to abstraction is to save space and time by verifying only the part of the program under test rather than reasoning about the entire program. This approach assumes that errors which occur outside of the component under test are irrelevant and can be ignored. The focus is on detecting errors which are located inside the component, but which may have been caused by behaviors outside the component. Similarly, errors detected in the context of a specific software environment say nothing about errors in the context of even a slightly different software environment. The salient assumption here is that errors within a specific environment are of more interest than errors that exist in a family of environments.

Experimental result shows that we can verify a C program with the SPIN model checker automatically with little change to the original software. This software also can run in complex environments and call any library function. We abstract the component under test. The experiments suggest that bitstate hashing is the most efficient abstraction for this approach to component verification. Abstraction based on predicates did not reduce the abstract state space enough to justify the additional time to interpret states using predicates. The experiments also demonstrate that errors can be found in abstracted components more quickly than errors can be found by simply executing the component. This happens when the model checker prunes the search during state enumeration. Abstraction of components finds errors more quickly than executing the component as is when the state space includes many copies of the same large region of states. In this case, concrete state exploration still needs to execute the already visited states since it does not have a memory about what state is already visited. Abstraction, on the other hand, can jump out of the partial state space it already visited and start to explore new state space faster. Of the three abstraction methods we used, bitstate hashing found errors in the least time, mostly because it does not need a refinement.

We have not yet investigated methods for extracting components from software. Instead, we have simply assumed that the component is given by a set of *pc* values. One avenue for future work is developing methods for extracting useful components from software based on a set of verification properties. Future work also includes investigating other abstraction schemes and extending the implementation to handle concurrent software.

References

1. Ball, T., Levin, V., Xie, F.: Automatic creation of environment models via training. In: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Barcelona, Spain, pp. 93–107 (2004)
2. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), New York, pp. 1–3 (2002)
3. Bao, T.: Refinement for predicate abstraction in the context of abstract component model. Brigham Young University (2007)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, pp. 439–448 (2000)
5. Holzmann, G.J.: An analysis of bitstate hashing. In: Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, pp. 301–314 (1995)
6. Holzmann, G.J.: The model checker SPIN. *Software Engineering* 23(5), 279–295 (1997)
7. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: 11th International SPIN Workshop on Model Checking of Software (SPIN), Barcelona, Spain, pp. 76–91 (2004)
8. Holzmann, G.J., Smith, M.H.: Feaver 1.0 user guide.

9. Dritan Kudra and Eric G. Mercer. Finding termination and time improvement in predicate abstraction with under-approximation and abstract matching. MS thesis, Brigham Young University (2007)
10. Mercer, E., Jones, M.: Model checking machine code with the gnu debugger. In: 12th International SPIN Workshop on Model Checking of Software (SPIN), San Francisco, CA, pp. 251–265 (2005)
11. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. *Computational Complexity*, 213–228 (2002)
12. Pasareanu, C.S., Pelanek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 52–66. Springer, Heidelberg (2005)
13. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: 15th IEEE International Conference on Automated Software Engineering (ASE), Washington, DC, p. 3 (2000)

Generating Compact MTBDD-Representations from **Probmela** Specifications

Frank Ciesinski¹, Christel Baier¹, Marcus Größer¹,
and David Parker²

¹ Technical University Dresden, Institute for Theoretical Computer Science, Germany

² Oxford University Computing Laboratory, Oxford, UK

Abstract. The purpose of the paper is to provide an automatic transformation of parallel programs of an imperative probabilistic guarded command language (called **Probmela**) into probabilistic reactive module specifications. The latter serve as basis for the input language of the symbolic MTBDD-based probabilistic model checker **PRISM**, while **Probmela** is the modeling language of the model checker **LiQuor** which relies on an enumerative approach and supports partial order reduction and other reduction techniques. By providing the link between the model checkers **PRISM** and **LiQuor**, our translation supports comparative studies of different verification paradigms and can serve to use the (more comfortable) guarded command language for a MTBDD-based quantitative analysis. The challenges were (1) to ensure that the translation preserves the Markov decision process semantics, (2) the efficiency of the translation and (3) the compactness of the symbolic BDD-representation of the generated **PRISM**-language specifications.

1 Introduction

Model checking plays a crucial role in analyzing quantitative behaviour of a wide range of system types such as randomised distributed algorithms and randomised communication protocols. One of the key ingredients of a model checking tool for a randomized system is an appropriate modeling language which should be expressive and easy to learn and must be equipped with a formal semantics that assigns MDPs to the programs of the modeling language. For efficiency reasons, it is also important that the MDP-semantics has a formalization by means of rules that support the automated generation of a compact internal representation of the MDP from a given program. The modeling languages of most model checkers for MDPs use probabilistic variants of modeling languages of successful nonprobabilistic model checkers. The MDP-fragment of the model checker **PRISM** [9] uses reactive module-like specifications [1] extended by the feature that statements can have a probabilistic effect. Probabilistic reactive modules rely on a declaration of pre- and postconditions of variables. Thus, their nature is rather close to symbolic representations which makes them well suited for the generation of a multiterminal binary decision diagram (MTBDD) [6,20] for the system. On the other hand, probabilistic reactive modules do not

support complex data structures (e.g., arrays and channels) and require the encoding of conditional or repetitive commands by means of pre- and postconditions. Modeling languages, like *Probmela* [2] which is a probabilistic dialect of the prominent (nonprobabilistic) modeling language *Promela* [10], that combine features of imperative programming language (such as complex datatypes, conditional commands, loops) with message passing over channels and communication over shared variables are much more comfortable. Many protocols and systems can be formally described within such a high-level modeling language in a rather elegant and intuitive way. The MDP-semantics of a given *Probmela*-program can be obtained as a DFS-based enumeration of the reachable states (similar to the on-the-fly generation of the transition system for a given *Promela*-program as it is realized in *SPIN* [10]). However, the generation of a compact symbolic MTBDD-representation is nontrivial. Although several reduction techniques that rely on an analysis of the underlying graph of the MDP or the control graphs of *Probmela*-programs can be applied to make the quantitative analysis competitive with the symbolic approach concerning the time required for the quantitative analysis [5], the enumerative approach often fails to handle very large systems with many parallel processes which can still be verified with the symbolic approach by *PRISM*.

The purpose of this paper is to combine the advantages of both approaches by providing an automatic translation from *Probmela*-programs into *PRISM* language and to derive a compact MTBDD representation from the generated *PRISM* code. The implementation of this translation (called *Prismela*) yields the platform to use the (more comfortable) modeling language *Probmela* for the MTBDD-based quantitative analysis of *PRISM* and supports comparative studies of different verification paradigms: the symbolic approach realized in *PRISM* [9] and the enumerative approach of *LiQuor* [4]. The main challenges are

- (1) to ensure that the translation preserves the Markov decision process semantics, (without introducing extra steps and intermediate pseudo-states that serve to simulate a single step of the original guarded command specification),
- (2) the efficiency of the translation and
- (3) the compactness of the symbolic BDD-representation of the generated *PRISM*-module specifications.

Our work is conceptually related to [3], where a translation schema is presented that allows for the transformation of a core fragment of *Promela* to the input format of the (nonprobabilistic) symbolic model checker *SMV* [15]. Beside the probabilistic features (probabilistic choice, lossy channels, random assignments), we treat some more language concepts than [3] such as message passing via handshaking through synchronous channels. Furthermore, we describe the translation of atomic regions in more detail and describe our implemented automated heuristics to calculate a good variable ordering for a given model. Another symbolic approach for *Promela* specifications has been presented in [22] using a nonstandard decision diagram, called *DDD*.

After a brief summary of the main concepts of *Probmela* and the PRISM input language (Section 2), we present the translation (Section 3), discuss heuristics that address item (3) and attempt finding good variable orderings for the MTBDD-representation and determining the variable ranges (Section 4). In Section 5, we explain the main features of our implementation on the top of the model checkers LiQuor and PRISM and report on experimental results.

2 Preliminaries

We give here brief intuitive explanations on the syntax and semantics of the (core fragment of the) modeling language *Probmela* and PRISM’s language, and suppose that the reader is familiar with the main concepts of *Promela* [10] and reactive modules [1].

The modeling language Probmela [2] is a probabilistic dialect of SPIN’s input language *Promela* [10]. In the core language, programs are composed by a finite number of processes that might communicate over shared (global) variables or channels. Programs consist of a declaration (types, initial values) of the global variables and channels, and the code for the processes. The processes can access the global variables and channels, but they also can have local variables and channels. We skip these details here and suppose for simplicity that the names of all (local or global) variables and channels are pairwise distinct. The channels can be synchronous or fifo-channels of finite capacity. The fifo channels can be declared to be either perfect or lossy with some failure probability $\lambda \in]0, 1[$. The meaning of λ is that the send-operation might fail with probability λ . The operational behavior of the processes is specified in a guarded command language as in *Promela* with (deterministic) assignments $x = \text{expr}$, communication actions $c?x$ (receiving a value for variable x along channel c) and $c!\text{expr}$ (sending the current value of an expression along channel c), the statement `skip`, conditional and repetitive statements over guarded commands (`if ... fi` and `do ... od`), and atomic regions. The probabilistic features of *Probmela* are lossy fifo-channels (see above), a probabilistic choice operator $\text{pif}[\pi_1] \Rightarrow \text{cmd}_1 \dots [\pi_k] \Rightarrow \text{cmd}_k \text{fip}$ (where π_1, \dots, π_k are probabilities, i.e., real numbers between 0 and 1 such that $\pi_1 + \dots + \pi_k \leq 1$ and $\text{cmd}_1, \dots, \text{cmd}_k$ are *Probmela* commands) and random assignments $x = \text{random}(V)$. The intuitive meaning of the `pif...fip` command is that with probability π_i , command cmd_i is executed next. The value $1 - (\pi_1 + \dots + \pi_k)$ is the deadlock probability where no further computation of the process is possible. In a random assignment $x = \text{random}(V)$, x is a variable and V a finite set of possible values for x . The meaning is that x is assigned to some value in V according to the uniform distribution over V . In addition, *Probmela* permits jumps by means of `goto`-statements.

Probmela also supports the creation, stopping, restarting and destruction of processes. Since the PRISM language assumes a fixed number of variables and modules, such dynamic features are not included in the translation and are therefore irrelevant for the purposes of this paper.

PRISM's *input language* [18]. For the purposes of the paper, only the fragment of PRISM that has an MDP-semantics is relevant. In this fragment, a PRISM program consists of several *modules* $\mathcal{P} = Q_1 \parallel \dots \parallel Q_n$ that run in parallel. Each module consists of a variable definition and a finite set of statements. The statements are equipped with a precondition (guard) on the current variable evaluation. The effect of the statements on the variables can be probabilistic. A PRISM statement $s \in \text{Stmt}$ has the form

$$[\sigma] \text{guard} \rightarrow \pi_1 : \text{upd}_1 + \dots + \pi_k : \text{upd}_k$$

where *guard* is a Boolean condition on the variables and π_1, \dots, π_k are probabilities, i.e., real numbers between 0 and 1 that sum up to 1. (If $k = 1$ then $\pi_1 = 1$ and we simply write $[\sigma] \text{guard} \rightarrow \text{upd}_1$ rather than $[\sigma] \text{guard} \rightarrow 1 : \text{upd}_1$.) The terms upd_i are “updates” that specify how the new values of the variables are obtained from the current values. Formally, the updates are conjunctions of formulas of the type $x' = \text{expr}$ where x is a program variable and its primed version x' refers to the value of x in the next state and expr is an expression built by constants and (unprimed) variables. If an update does not contain a conjunct $x' = \dots$ then the meaning is that the value of variable x remains unchanged. (In this way, the updates upd_i specify unique next values.) The symbol σ is either ε or a synchronization label. Statements of different modules with the special symbol ε (simply written $[\]$ rather than $[\varepsilon]$) are executed in an interleaved way, i.e., without any synchronization. The meaning of statements with a synchronization label σ is that all modules have to synchronize over a statements labeled by σ . No other channel-based communication concept is supported by the PRISM language, i.e., there is no asynchronous message passing over fifo-channels and no (explicit) operator modeling handshaking between two modules. Furthermore, PRISM does not support data types like arrays.

Markov decision processes (MDP). Both *Probmela* programs and PRISM programs have an operational semantics in terms of a Markov decision process (MDP) [19]. In this context, the MDP for a program consists of a finite state space S and a transition relation $\rightarrow \subseteq S \times \text{Act} \times \text{Distr}(S)$ where Act is a set of actions and $\text{Distr}(S)$ denotes the set of (sub) distributions over S (i.e., functions $\mu : S \rightarrow [0, 1]$ such that $\sum_{s \in S} \mu(s) \leq 1$). Furthermore, there is a distinguished state that is declared to be initial.

The states in the MDP for a *Probmela* program consist of local control states for all processes, valuations for the local and global variables and a component that specifies the current contents of the fifo-channels. The transition relation \rightarrow is formally presented by means of SOS-rules [2]. In our implementation, we slightly departed from [2] and used a MDP-semantics that relies on a representation of each process by a control graph, which can then be unfolded into an MDP and put in parallel with the MDPs for the other processes. (Parallel composition is understood as ordinary interleaving and synchronization in the handshaking principle for message passing over synchronous channels.)

In the sequel, let Var be the set of all global variables of the given program and LocVar_i the set of local variables of process Q_i . For simplicity, we suppose

that $\text{Var} \cap \text{LocVar}_i = \emptyset$. We write Var_i for $\text{Var} \cup \text{LocVar}_i$, the set of variables that can appear in the statements of process Q_i . If V is a set of variables then $\text{Eval}(V)$ denotes the set of all (type-consistent) valuations for the variables in V . In the control graph for process Q_i , the nodes are called locations of Q_i . They play the role of a program counter and are obtained by assigning identifiers to each command in the **Probmela**-code for Q_i . The edges have the form $\ell \xrightarrow{g:\alpha} \nu$ where ℓ is a location, g is a guard (Boolean condition on the variables in Var_i) and α an action which can be viewed as a function $\alpha : \text{Eval}(\text{Var}_i) \rightarrow \text{Distr}(\text{Eval}(\text{Var}_i))$ and ν a distribution over the locations of Q_i . If ν assigns probability 1 to some location ℓ' (and probability 0 to all other locations) then we simply write $\ell \xrightarrow{g:\alpha} \ell'$. Furthermore, the trivial guard $g = \text{true}$ is omitted and we simply write $\ell \xrightarrow{\alpha} \nu$ rather than $\ell \xrightarrow{\text{true}:\alpha} \nu$. For instance, the location ℓ assigned to the command $\text{pif}[\pi_1] \Rightarrow \text{cmd}_1 \dots [\pi_k] \Rightarrow \text{cmd}_k \text{fip}$ has just one outgoing edge $\ell \xrightarrow{id} \nu$ where id is the identity¹. Let ℓ_j be the location representing the command cmd_j then distribution ν is given by $\nu(\ell_j) = \pi_j$ and $\nu(\ell') = 0$ for all other locations ℓ' . If location ℓ stands for a nondeterministic choice $\text{if}[g_1] \Rightarrow \text{cmd}_1 \dots [g_k] \Rightarrow \text{cmd}_k \text{fi}$, then there are k outgoing edges $\ell \xrightarrow{g_j:id} \ell_j$. Similarly, for a loop $\text{do}[g_1] \Rightarrow \text{cmd}_1 \dots [g_k] \Rightarrow \text{cmd}_k \text{od}$, there are k outgoing edges $\ell \xrightarrow{g_j:id} \ell_j$ for $1 \leq j \leq k$ where ℓ_j is the location representing cmd_j ².

If ℓ represents an assignment $x = \text{expr}$, then ℓ has a single outgoing edge $\ell \xrightarrow{\text{true}:\alpha} \ell'$ with the trivial guard true and the action α that modifies x according to expr and keeps all other variables unchanged. Again, location ℓ' stands for the command after the command represented by ℓ in the **Probmela** code for Q_i . The effect of an atomic regions is modeled in the control graph by a single edge that represents the cumulative effect of all activities inside the atomic region.

For a given **PRISM** program, the MDP is obtained as follows. The states are the evaluations of the program variables. Given a state s , then for each statement $\text{stmt} = [\text{guard} \rightarrow \pi_1 : \text{upd}_1 + \dots + \pi_k : \text{upd}_k$ (in some module) where $s \models \text{guard}$ there is a transition $s \xrightarrow{\text{stmt}} \nu$ with $\nu(s') = \sum_j \pi_j$ where j ranges over all indices in $\{1, \dots, k\}$ such that upd_j evaluates to true when the unprimed variables are interpreted according to s , while the values of the primed variables are given by s' . Furthermore, s and s' must agree on all variables x where x' does not appear in upd_j .

3 From **Probmela** to **PRISM**

We now suppose that we are given a **Probmela** program \mathcal{P} (of the core language without dynamic features). The goal is to generate automatically a **PRISM** program $\tilde{\mathcal{P}}$ that has the same MDP-semantics and a compact MTBDD-representation.

¹ i.e., $id(\eta)(\eta) = 1$ and $id(\eta)(\eta') = 0$ if $\eta \neq \eta'$

² As in **Promela** loops are terminated by a special command **break**. Its control semantics is given by a control edge from the location of the break-command to the next location after the loop.

The general workflow of the translation (see Fig. II) starts with a *Probmela* program \mathcal{P} consisting of n processes Q_1, \dots, Q_n and derives a PRISM program $\tilde{\mathcal{P}}$ with n modules $\tilde{Q}_1, \dots, \tilde{Q}_n$. The global variables of \mathcal{P} are also global in $\tilde{\mathcal{P}}$. Furthermore, $\tilde{\mathcal{P}}$ contains additional global variables that serve to mimic the arrays and channels in \mathcal{P} and other features of *Probmela* that have no direct translation. The last two steps attempt to minimize the MTBDD-representation and rely on heuristics to determine a good variable ordering and algorithms that fix appropriate bit-sizes (ranges) of variables (Section 4).

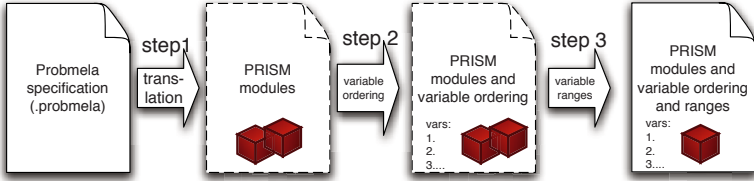


Fig. 1. Translation scheme from *Probmela* to PRISM

In the first step, each *Probmela* process is translated into an equivalent PRISM module. It relies on the control graph semantics of *Probmela* and translates each control edge into one or more PRISM statements. Given a *Probmela* process Q_i , the corresponding PRISM module \tilde{Q}_i has the same local variables $LocVar_i$ and an additional integer variable pc_i that serves as a program counter for Q_i . Intuitively, the possible values of pc_i encode the locations of the control graph of Q_i . The rough idea is to replace each edge $\ell \xrightarrow{g:\alpha} \nu$ in the control graph of Q_i with the PRISM statement

$$[[\tilde{g} \wedge pc_i = \ell] \rightarrow \nu(\ell_1) : (\tilde{\alpha}) \wedge (pc_i = \ell_1) + \dots + \nu(\ell_k) : (\tilde{\alpha}) \wedge (pc_i = \ell_k)]$$

where \tilde{g} and $\tilde{\alpha}$ are the translations of g and α to PRISM code with primed and unprimed variables and ℓ_1, \dots, ℓ_k are the locations that have positive probability under distribution ν .

This basic translation schema is directly applicable for deterministic or randomized assignments. For instance, if α stands for the assignment $x = y + z$ then $\tilde{\alpha}$ is the update $(x' = y + z)$. Since PRISM and *Probmela* support the same operations on basic types, the translation of Boolean conditions and actions representing simple assignments is straightforward, even when they involve more complex operations like multiplication, division, etc. Thus, a deterministic assignment given by an edge $\ell \xrightarrow{\alpha} \ell'$ in the control graph where α is given by the command $x = \text{expr}$ is translated into the PRISM statement $[[pc_i = \ell \rightarrow (x' = \text{expr}) \wedge (pc'_i = \ell')]$. For a randomized assignment, given by an edge $\ell \xrightarrow{\alpha} \ell'$ where α is given by the command $x = \text{random}(0, 1)$, the translation schema yields the PRISM statement

$$[[pc_i = \ell \rightarrow \frac{1}{2} : (x' = 0) \wedge (pc'_i = \ell') + \frac{1}{2} : (x' = 1) \wedge (pc'_i = \ell')].$$

Arrays are very useful to model, e.g. memory slots or network packages. *Probmela* uses a C-like syntax to define arrays (e.g., `int[3]a` defines an integer array with 3 cells). Access to the array cells is possible either using a variable, a constant or an arithmetical expression as an index, e.g. `a[i + j]`. Arrays are also supported in classical reactive modules [11], but they are difficult to implement if a reactive module specification has to be transferred into symbolic representation and are therefore not supported in *PRISM*. For an action α which contains an array access of the form `a[expr]` the corresponding condition $\tilde{\alpha}$ is obtained by introducing fresh *PRISM* variables a_j for all array cells `a[j]` and replacing `a[j]` by a_j or a'_j (depending on whether `a[j]` appears on the left or right hand side of an assignment). A more complex case occurs if the array index to which is referenced is an array access itself. For instance, if action α in the control edge $\ell \xrightarrow{\alpha} \ell'$ is the assignment `a[j[k]] = 7` then the corresponding *PRISM* code consists of several statements that represent the possible combinations of values for k and $j[k]$:

$$\begin{aligned}
& [](\text{pc}_i = \ell) \wedge (k = 0) \wedge (j_0 = 0) \rightarrow (a'_0 = 7) \wedge (\text{pc}'_i = \ell') \\
& [](\text{pc}_i = \ell) \wedge (k = 0) \wedge (j_0 = 1) \rightarrow (a'_1 = 7) \wedge (\text{pc}'_i = \ell') \\
& [](\text{pc}_i = \ell) \wedge (k = 0) \wedge (j_0 = 2) \rightarrow (a'_2 = 7) \wedge (\text{pc}'_i = \ell') \\
& \vdots \\
& [](\text{pc}_i = \ell) \wedge (k = 1) \wedge (j_1 = 0) \rightarrow (a'_0 = 7) \wedge (\text{pc}'_i = \ell') \\
& [](\text{pc}_i = \ell) \wedge (k = 1) \wedge (j_1 = 1) \rightarrow (a'_1 = 7) \wedge (\text{pc}'_i = \ell') \\
& [](\text{pc}_i = \ell) \wedge (k = 1) \wedge (j_1 = 2) \rightarrow (a'_2 = 7) \wedge (\text{pc}'_i = \ell') \\
& \vdots
\end{aligned}$$

The case where multiple arrays are connected via arithmetical operations, e.g. `a[j[k]] + l[m] = 42`, can be treated in a similar way.

Remark 1. The treatment of probabilistic choices (`pif...fip`), nondeterministic choices (`if...fi`), loops (`do...od`) and jumps is inherent in our translation schema which operates on the control graph semantics of *Probmela* (and where the meaning of probabilistic, nondeterministic choices and loops is already encoded). However, it is worth noting that our translation yields a rather natural and intuitive encoding in *PRISM* for these language concepts. The translation of a probabilistic choice `pif` $[\pi_1] \Rightarrow \text{cmd}_1 \dots [\pi_k] \Rightarrow \text{cmd}_k \text{fip}$, specified by an edge $\ell \xrightarrow{id} \nu$ (where $\nu(\ell_j) = \pi_j$ and ℓ_j is the location for `cmdj`) yields the *PRISM* statement

$$[](\text{pc}_i = \ell) \rightarrow \pi_1 : (\text{pc}'_i = \ell_1) + \dots + \pi_k : (\text{pc}'_i = \ell_k).$$

For a nondeterministic choice `if` $:: g_1 \Rightarrow \text{cmd}_1 \dots :: g_k \Rightarrow \text{cmd}_k \text{fi}$ or loop `do` $:: g_1 \Rightarrow \text{cmd}_1 \dots :: g_k \Rightarrow \text{cmd}_k \text{od}$ specified by k control edges $\ell \xrightarrow{g_j:id} \ell_j$ we get k *PRISM* statements $[](\tilde{g}_j) \wedge (\text{pc}_i = \ell) \rightarrow (\text{pc}'_j = \ell_j)$ for $1 \leq j \leq k$. Similarly, the basic translation schema can directly be applied to treat *Probmela*'s `goto`-command, formalized by control edges of the form $\ell \xrightarrow{g} \ell'$ for a conditional jump. The corresponding *PRISM* statement has the form $[](\tilde{g}) \wedge (\text{pc}_i = \ell) \rightarrow (\text{pc}'_i = \ell')$. \square

Perfect asynchronous channels can be regarded as arrays with restricted access according to the fifo principles. In the internal representation of our model checker LiQuor, asynchronous channels are realized as arrays with an additional variable c_{fill} that keeps track of the number data items currently stored inside the channel. That is, $c_{\text{fill}} = k$ iff channel c contains k messages. A send operation $c!v$ is enabled iff c_{fill} is strictly smaller than the capacity of c (which is defined in the channel declaration), while a receive operation $c?x$ requires $c_{\text{fill}} > 0$. When executing a send or receive operation, variable c_{fill} is incremented or decremented, respectively. The translation of send and receive operations into PRISM statements can therefore be realized in a similar way as array access. E.g., the control edge $\ell \xrightarrow{g:c!v} \ell'$ where c is a perfect channel of capacity m and v a constant value is translated into the PRISM statements:

$$\begin{aligned} & [](pc_i = \ell) \wedge (\tilde{g} \wedge c_{\text{fill}} = j - 1) \rightarrow \\ & (pc'_i = \ell') \wedge (c'_{\text{fill}} = j) \wedge (c'_j = c_{j-1}) \wedge \dots (c'_2 = c_1) \wedge (c'_1 = v), \end{aligned}$$

while control edges $\ell \xrightarrow{g:c?x} \ell'$ representing a receive operation are realized in PRISM by the statements

$$[](pc_i = \ell) \wedge (\tilde{g}) \wedge (c_{\text{fill}} = j) \rightarrow (pc'_i = \ell') \wedge (c'_{\text{fill}} = j - 1) \wedge (x' = c_j)$$

where $1 \leq j \leq m$. The shift operation that is inherent in the PRISM code for the send operation serves to avoid that the same channel configuration is presented by several states.

Lossy asynchronous channels. In *Probmela*, asynchronous channels can be declared to be lossy, i.e. the enqueueing process loses the message with some predefined probability. For such lossy channels, we modify the translation for perfect asynchronous channels by dealing with a probabilistic choice for the PRISM statement modeling the send operation. Suppose that the send operation of process Q_i is modeled by the control edge $\ell \xrightarrow{c!v} \ell'$ and that the failure probability of c is 0.3. The corresponding PRISM statements are:

$$\begin{aligned} & [](pc_i = \ell) \wedge (c_{\text{fill}} = 2) \rightarrow 0.7 : (c_{\text{fill}} = 3) \wedge (c'_2 = c_1) \wedge (c'_1 = c_0) \wedge (c'_0 = v) \wedge (pc' = \ell') + \\ & \quad \quad \quad 0.3 : (pc' = \ell) \\ & [](pc_i = \ell) \wedge (c_{\text{fill}} = 1) \rightarrow 0.7 : (c_{\text{fill}} = 2) \wedge (c'_1 = c_0) \wedge (c'_0 = v) \wedge (pc' = \ell') + \\ & \quad \quad \quad 0.3 : (pc' = \ell) \\ & [](pc_i = \ell) \wedge (c_{\text{fill}} = 0) \rightarrow 0.7 : (c_{\text{fill}} = 1) \wedge (c'_0 = v) \wedge (pc' = \ell') + 0.3 : (pc' = \ell) \end{aligned}$$

Synchronous channels are syntactically defined in *Probmela* (as well in *Promela*) as channels with capacity 0. They require pairwise message passing by handshaking between processes. If c is a synchronous channel then the send operation $c!v$ can only be performed if there is another process ready to immediately execute a receive operation $c?x$, where x is an arbitrary program variable. The PRISM language also supports synchronization, but without message passing and not

in a pairwise manner. Instead, synchronization in PRISM language modules is over the synchronization labels and requires the participation of all modules. To translate *Probmela*'s communication actions for a synchronous channel c into PRISM code, we generate appropriate synchronization labels all potential handshakings along channel c . That is, for each pair of control edges $e_1 = \ell_i \xrightarrow{c?x} \ell'_i$ in the control graph of process Q_i and $e_2 = \ell_j \xrightarrow{c!expr} \ell'_j$ in the control graph of another process Q_j with matching handshaking actions we introduce a fresh synchronization label $\sigma(e_1, e_2)$ and use the following PRISM statements:

$$\begin{array}{ll} [\sigma(e_1, e_2)] (pc_i = \ell_i) \rightarrow (pc'_i = \ell'_i) \wedge (x' = \text{expr}) & \text{(in module } \tilde{Q}_i) \\ [\sigma(e_1, e_2)] (pc_j = \ell_j) \rightarrow (pc'_j = \ell'_j) & \text{(in module } \tilde{Q}_j) \\ [\sigma(e_1, e_2)] (pc_k = \ell_k) \rightarrow (pc'_k = \ell_k) & \text{(in module } \tilde{Q}_k, k \notin \{i, j\}) \end{array}$$

Note that the use of synchronization labels $c_{i,j}$ that just indicate the channel and synchronization partners would not be sufficient since a process might request a synchronous communication actions at several locations.

Atomic regions collapse several commands to one single step. They can be used to effectively shrink the state space if it is known that certain calculations need not (or must not) to be carried out interleaved. To the user this language element appears as a builtin mutual exclusion protocol that can be used to execute certain calculations exclusively without actually implementing a mutual exclusion mechanism as part of the specification.

In the simple case, the atomic region consists of a sequential composition of independent assignments, e.g., $a = 1; b = 2; c = 3$, which corresponds to the PRISM statement $[]\text{true} \rightarrow (a' = 1) \wedge (b' = 2) \wedge (c' = 3)$. However, more complicated types of atomic regions that are allowed in *Probmela* that require a more involved translation into PRISM. First, atomic regions can write a single variable more than once. Consider for instance the atomic region `atomic{i + +; i + +}` which would (according to the simple translation scheme) lead to an update $\dots - > (i' = i + 1) \wedge (i' = i + 1)$. Such statements, however, are not allowed in PRISM. Instead, such commands must either be subsumed to one expression (i.e. $i' = i + 2$) or encoded in two separate transitions. Second, atomic regions may contain (nested) probabilistic or nondeterministic choices that can hardly be accumulated into a single step. To provide the PRISM code for complex atomic regions, an additional global variable *proc* is added to the PRISM program $\tilde{\mathcal{P}}$.

We set *proc* to an initial value of -1 and extend the guards of PRISM statements in each module \tilde{Q}_i by the condition $proc = -1 \vee proc = i$. This ensures that for $proc = -1$ all modules can potentially perform steps, while for $proc = i$ the transitions of all module \tilde{Q}_j with $j \neq i$ are disabled. Furthermore, we extend the PRISM code for module \tilde{Q}_i to ensure that when an atomic region of process Q_i is entered then the current value of *proc* is set to i and that *proc* is reset to -1 when Q_i leaves the atomic region.

Soundness of the translation. The reachable fragments of the MDPs for a given *Probmela* program \mathcal{P} without atomic regions and the generated PRISM

program $\tilde{\mathcal{P}}$ are *isomorphic*. The isomorphism is obtained by identifying the state $s = \langle \ell_1, \dots, \ell_n, \eta \rangle$ in the MDP for \mathcal{P} with the state $s' = \langle pc_1 = \ell_1, \dots, pc_n = \ell_n, \tilde{\eta} \rangle$ in the MDP for $\tilde{\mathcal{P}}$. Here, ℓ_i is a location in the control graph for process Q_i and η a variable and channel valuation. $\tilde{\eta}$ stands for the unique valuation of the variables in $\tilde{\mathcal{P}}$ that is consistent with η (i.e., agrees on all variables of \mathcal{P} and maps, e.g., the index-variables a_j for an array a in \mathcal{P} to the value of the j -th array cell $a[j]$ under η). To show that each outgoing transition has a matching transition from s' , and vice versa, we can make use of the fact that the outgoing transition from both s and s' arise by the control edges from the locations ℓ_i and that the PRISM statements are defined exactly in the way such that the enabledness and the effect of the control edges is preserved. This strong soundness result still holds if \mathcal{P} contains simple atomic regions. In case that \mathcal{P} contains complex atomic regions then we can establish a *divergence-sensitive branching bisimulation* [21, 7] between the (reachable fragments of the) MDPs for \mathcal{P} and $\tilde{\mathcal{P}}$ which identifies all (intermediate) states where the location of some process is inside an atomic regions. Thus, \mathcal{P} and $\tilde{\mathcal{P}}$ are still equivalent for all stutter-insensitive properties, e.g., specified by nextfree LTL or PCTL formulae.

4 Optimizations of the MTBDD Representation

The translation presented in the previous section combined with the PRISM tool yields an automatic way to generate a symbolic representation of the MDP for a ProbMela program as a multiterminal binary decision diagram (MTBDD) [6, 20]. In this section, we discuss techniques to obtain a compact MTBDD representation. First, we present a heuristic to find a good variable ordering for the MTBDD of a given PRISM program. Second, we address the problem of finding appropriate and small ranges for the variables in a PRISM program.

Determining good variable orderings automatically. Throughout this section, we assume some familiarity with (MT)BDDs. (Details can be found, e.g., in [16, 23].) It is well-known that the size of an (MT)BDD for a discrete function can crucially depend on the underlying variable ordering and that the problem of finding the optimal variable ordering is NP-complete. There are several heuristic approaches to find fairly good variable orderings. Some of them improve the variable ordering of a given (MT)BDD, while others attempt to derive a good initial variable ordering from the syntactic description of the function to be represented [8, 17]. We follow here the second approach and aim to determine a reasonable variable ordering from the PRISM code.

Given a PRISM program we abstract away from the precise meaning of Boolean or arithmetic operations and analyze the dependencies of variables. For this, we treat the PRISM statements as statements that access variables by means of uninterpreted guards and operations. This leads to an *abstract syntax tree* (AST) presenting the syntactic structure of the given PRISM program $\tilde{\mathcal{P}}$. For this, we regard the PRISM statements as terms over the signature that contains constant symbols and the primed and unprimed versions of the program

variables as atoms and uses symbols like $+$, $*$, $=$, $<$, \rightarrow as function symbols. (The probabilities attached to updates are irrelevant and can simply be ignored). The node set in the AST for $\tilde{\mathcal{P}}$ consists of all statements in $\tilde{\mathcal{P}}$ and their subterms the primed and unprimed versions of the variables of $\tilde{\mathcal{P}}$ and nodes for all function symbols that appear in the statements of $\tilde{\mathcal{P}}$ (like comparison operators, arithmetic operators, the arrows between the guard and sum of updates in statements). Furthermore, the AST contains a special root node δ that serves to link all statements. The edge relation in the AST is given by the “subterm relation”. That is, the leaves stand for the primed or unprimed variables or constants.³ The children of each inner node v represent the maximal proper subterms of the term represented by node v . The children of the root node are the nodes representing the statements.

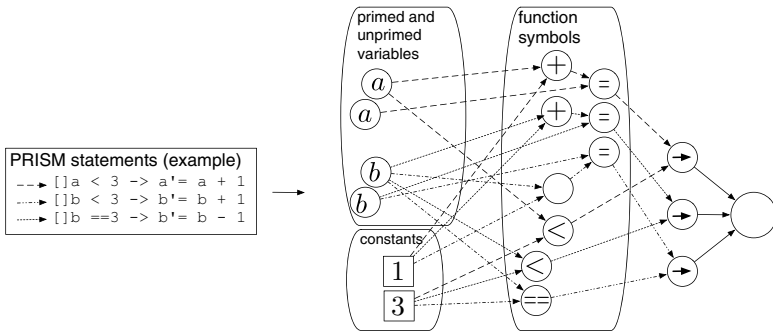


Fig. 2. Example AST

We now apply simple graph algorithms to the AST of $\tilde{\mathcal{P}}$ to derive a reasonable variable ordering for the MTBDD for $\tilde{\mathcal{P}}$. For this, we adapt heuristics that have been suggested for gate-level circuit representations of switching functions. We considered the fanin-heuristic [14] and the weight-heuristic [11] and adapted them for our purposes. The rough idea behind these heuristics is to determine a variable ordering such that (1) variables that affect the program at most should appear at the top levels, and (2) variables that are near to one another in the dataflow should be grouped together.

The *fanin-heuristic* is based on the assumption that input variables that are connected to the output variables via longer paths are more meaningful to the function and should be ordered first. For this a breadth-first-search is performed (starting from the leaves in the AST, i.e., the variables and constant symbols) which labels all nodes of the graph with the maximum distance to an input node, i.e., we compute the values $d(v)$ for all nodes in the AST where $d(v) = 0$ for the leaves and

$$d(w) = 1 + \max\{d(v) : v \text{ is a child of } w\}$$

³ At the bottom level, leaves representing the same variable or constant are collapsed. So, in fact, the AST is a directed acyclic graph, and possibly not a proper tree.

for all inner nodes w . The second step of the heuristic performs a depth-first-search starting at the root node with the additional property that the depth-first-search order in each node w that is visited is according to a descending ordering of the values $d(v)$. The visiting order of the variables then yields a promising variable ordering for the MTBDD for \tilde{P} .

The *weight-heuristic* relies on an iterative approach that assigns weights to all nodes of the AST and in each iteration the variable with the highest weight is the next in the variable ordering. This variable as well as any node that cannot reach any other variable is then removed from the AST and the next iteration yields the next variable in the ordering. (We suppose here that initially the leaves representing constants are removed from the AST.) In each iteration the weights are obtained as follows. We start with the root node and assign weight 1 to it and then propagate the weight to the leaves by means of the formula:

$$\text{weight}(v) = \frac{\text{weight}(\text{father}(v))}{|\text{number of children of father}(v)|}$$

Determining variable ranges. Besides the variable ordering, the bitsizes (and hence the value ranges) of the variables in a specification have great influence on the size of the MTBDD. This is unfortunately even the case if it turns out during model construction that in the reachable part of the model a particular variable does not fully exploit its defined range. Thus, it is highly desirable that the variable ranges in the PRISM model are as “tight” as possible. Often the user does this by applying her/his knowledge about the model and choosing just a reasonable range for each variable. Our tool also provides the possibility to determine reasonable variable ranges automatically. The idea of the algorithm for some program variable x is to perform a binary search in the interval $[1, k]$, where k is an upper bound for the bit size of x until an element i has been found such that $|\text{MDP}(\tilde{P}, x, i)| = |\text{MDP}(\tilde{P}, x, k)|$. Here, $|\text{MDP}(\tilde{P}, x, i)|$ denotes the number of states in the MDP for \tilde{P} when the bitsize of x is i . For efficiency purposes we implemented a modified version of this algorithm that starts with bitsize 1 and then increase it to the next 8 bit-border. If the model size changes we decrease by 4 bit to see if the lower size suffices as well, and so on.

5 Implementation and Results

The translation described in Section 3 and the heuristics of the previous section have been implemented on the top of our model checker LiQuor 4 and linked to the PRISM model checker. We called the resulting tool Prismela. It runs under the operating system *Microsoft Windows*. Using a graphical user interface (see Fig. 3) the user is able to load a *Probmela* model, control the translation process regarding variable orderings, variable ranges and start PRISM to build the model. It is also possible to combine user knowledge and automated procedures, for instance when the user already knows the value domain of particular

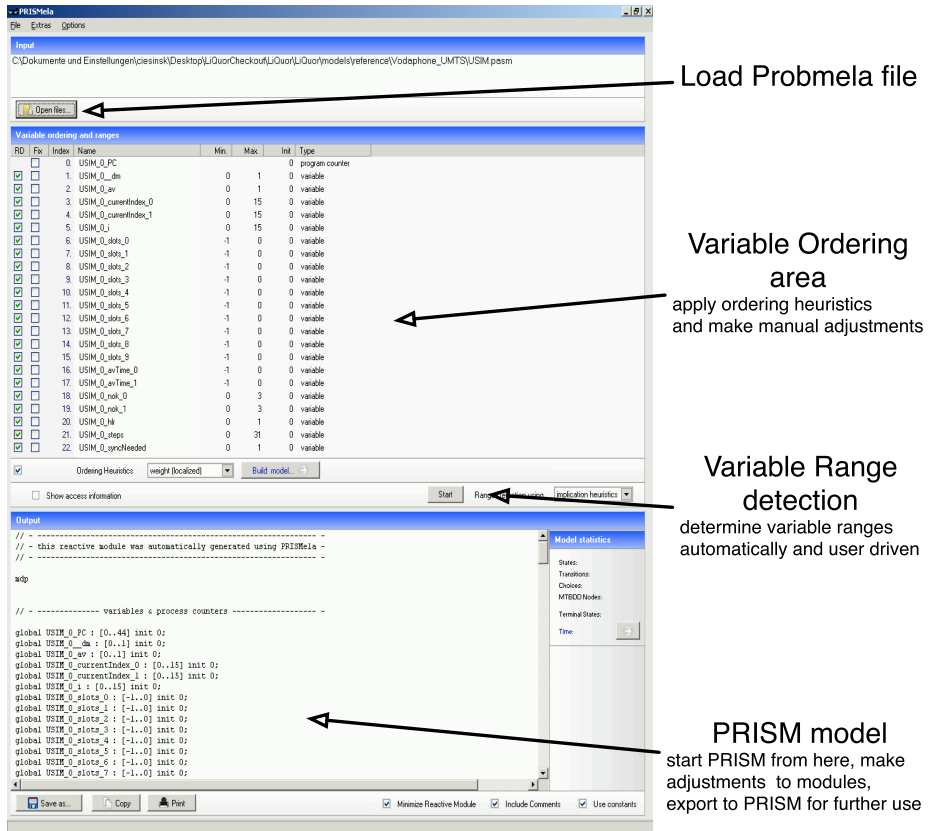


Fig. 3. Graphical user interface and functionality

variables or wants to fix the position of certain variables in the variable ordering. Then these variables can be excluded from the heuristics and variable range finding process. Furthermore the user has the option for manual changes on generated PRISM code that can then be exported for furthergoing use in PRISM. The relevant parts of LiQuor (The Probmela-compiler, PASM assembler, the virtual machine that generates the PRISM language model from the assembler code) were linked to Prismela so that the application runs independently from LiQuor.

Our model checker LiQuor [4] uses an intermediate representation of the Probmela program rather than the textual representation of the Probmela program itself. This intermediate representation is the result of a compiling process done by a compiler that was designed to translate Probmela into an assembler like formalism, called probabilistic assembler language (PASM), which is executed on a stack-based virtual processor during the model checking procedure. The virtual machine is connected to a storage module that can save and restore encountered the states of the MDP. The PASM-based approach has

model	MDP states	MDP trans.	time(LiQuor)	time(PRISM)	MTBDD-nodes
UMTS(10/3/20)	17.952	18.539	< 1s	17s	132.895
UMTS(30/5/60)	177.416	186.063	2s	1166s	$1.4 \cdot 10^6$
Din.Phil. (3)	635	2.220	< 1s	< 1s	2.011
Din.Phil. (6)	411255	$2.8 \cdot 10^6$	92s	< 1s	9645
Din.Phil. (10)	$2.2 \cdot 10^9$	$26 \cdot 10^9$	–	3s	41.953
Leader El.(3)	1.562	4.413	< 1s	< 1s	3410
Leader El.(6)	$4.2 \cdot 10^6$	$23 \cdot 10^6$	664s	6s	69.515
Leader El.(10)	$1.9 \cdot 10^{11}$	$1.7 \cdot 10^{12}$	–	–	926.585

Fig. 4. Some results with case studies

several advantages. One of them is that the correctness of the *Probmela* compiler can be easily be established by checking that the generated PASM code is consistent with the control graph semantics of *Probmela*. The crucial point for the purposes of this paper is that the generation of the PRISM language model can start from the PASM code rather than the *Probmela* specification. The generation of the PRISM modules is obtained by realizing the above translation steps for control edges on the level of PASM micro-commands. As a side effect, our translation is not affected by future extensions of *Probmela* (as long as they yield PASM code with a semantics based on control graphs as above) and is applicable to any other formalism with a PASM-translator.

Figure 4 shows some experimental results of the translation scheme. Among the case studies is one industrial motivated model (UMTS) that involves examining certain rare errors that occur when UMTS phones register to the network provider. This model involves complex storage behaviour in internal buffers of an UMTS end user device and uses almost every language element of *Probmela* discussed in this paper. Values given in parantheses are parameters for sizes and other characteristics and are not explained in detail. Larger numbers here indicate larger buffer tables and a larger number of potential entries in these tables, thus resulting in a larger model. Furthermore the table contains results from a randomized variant of the Dining Philosophers [13] (number of processes in parenthesis) and results from a randomized version of the Leader Election protocol [12] (number of processes in parenthesis). The results show that there exist models where one tool experiences great difficulties where the other may succeed rather quick, and vice versa. As expected for smaller models the explicit approach of LiQuor outperforms PRISM’s symbolic approach while the state explosion problem is more severe for the explicit approach of LiQuor.

Figure 5 illustrates the efficiency of our translation algorithm. The fanin-heuristic (as well as randomly chosen orderings) leads to very large MTBDDs. The amount of time to build the MTBDDs was always significantly lower when the weight heuristic was applied to calculate a variable ordering.

7 Dining Philosophers,

$3.3 \cdot 10^6$ states, $26 \cdot 10^6$ transitions.

15 PRISM variables (42 bits), 144 PRISM actions.

heuristic	MTBDD nodes	time
weight	9766	0.6s
fanin	51766	6s
random ordering (mean value)	61617	7s

10 Dining Philosophers,

$1.9 \cdot 10^9$ states, $22 \cdot 10^9$ transitions.

21 PRISM variables (51 bits), 202 PRISM actions.

heuristic	MTBDD nodes	time
weight	19684	2,2s
fanin	891604	359s
random ordering (mean value)	356627	171

Leader Election, 7 instances,

$62 \cdot 10^6$ states, $398 \cdot 10^6$ transitions.

28 PRISM variables (56 bits), 112 PRISM actions.

heuristic	MTBDD nodes	time
weight	$1,4 \cdot 10^5$	35s
fanin	$2,4 \cdot 10^6$	818s
random ordering (mean value)	$2,4 \cdot 10^6$	438s

Leader Election, 10 instances,

$194 \cdot 10^9$ states, $17 \cdot 10^{11}$ transitions.

28 PRISM variables (80 bits), 142 PRISM actions.

heuristic	MTBDD nodes	time
weight	886510	1081s
fanin	—	—
random ordering (mean value)	—	—

UMTS, 10/3/30,

35202 states, 36413 transitions.

23 PRISM variables (57 bits), 136 PRISM actions.

heuristic	MTBDD nodes	time
weight	218662	38s
fanin	233484	64s
random ordering (mean value)	252813	90s

Fig. 5. Influence of variable ordering heuristics on model generation with PRISM

6 Conclusion and Future Work

We presented an approach for the automatic translation of *Probmela* into the PRISM language. We thus can obtain an MTBDD representation for the *Probmela* program using PRISM. The translation process presented here is independent of the input language *Probmela* as it works on control graphs. It is therefore flexible for extensions of the input language and is, in principle, applicable to other modeling languages with a control graph semantics.

We also presented heuristics that serve to optimize the generation of the MTBDD from PRISM programs. These heuristics operate only on PRISM level and can therefore be applied to any PRISM program.

Future work on the presented topics include exhaustive comparisons between the symbolic and explicit model checking approach for probabilistic systems. Further improvements of the translation include language elements that were not covered yet. *Probmela* (as well as *Promela*) allows for dynamic creation of processes that would also be a desirable feature for *Prismela*.

Another target of future work will be the impact of static partial order reduction of *Probmela* programs on the use with symbolic model checkers.

References

1. Alur, R., Henzinger, T.A.: Reactive modules. *Formal Methods in System Design: An International Journal* 15(1), 7–48 (1999)
2. Baier, C., Ciesinski, F., Größer, M.: *Probmela*: a modeling language for communicating probabilistic systems. In: *Proceeding MEMOCODE (2004)*
3. Baldamus, M., Schröder-Babo, J.: p2b: a translation utility for linking *promela* and symbolic model checking (tool paper). In: Dwyer, M.B. (ed.) *SPIN 2001*. LNCS, vol. 2057, pp. 183–191. Springer, Heidelberg (2001)
4. Ciesinski, F., Baier, C.: LiQuor: a tool for qualitative and quantitative linear time analysis of reactive systems. In: *Proc. QEST*, pp. 131–132. IEEE CS Press, Los Alamitos (2007)
5. Ciesinski, F., Baier, C., Groesser, M., Klein, J.: Reduction techniques for model checking markov decision processes (submitted for publication, 2008)
6. Clarke, E., Fujita, M., McGeer, P., Yang, J., Zhao, X.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In: *International Workshop on Logic Synthesis, Tahoe City (1993)*
7. Größer, M., Norman, G., Baier, C., Ciesinski, F., Kwiatkowska, M., Parker, D.: On reduction criteria for probabilistic reward models. In: Arun-Kumar, S., Garg, N. (eds.) *FSTTCS 2006*. LNCS, vol. 4337, pp. 309–320. Springer, Heidelberg (2006)
8. Hermanns, H., Kwiatkowska, M., Norman, G., Parker, D., Siegle, M.: On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems* 56(1-2), 23–67 (2003)
9. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) *TACAS 2006*. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
10. Holzmann, G.J.: *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading (2003)
11. Minato, S.i., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In: *DAC 1990: Proceedings of the 27th ACM/IEEE conference on Design automation*, pp. 52–57. ACM Press, New York (1990)
12. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. *Information and Computation* 88(1) (1990)

13. Lehmann, D., Rabin, M.O.: On the advantage of free choice: A symmetric and fully distributed solution to the Dining Philosophers problem (extended abstract). In: Proc. Eighth Ann. ACM Symp. on Principles of Programming Languages, pp. 133–138 (1981); A classic paper in the area of randomized distributed algorithms. They show there is no deterministic, deadlock-free, truly distributed and symmetric solution to the Dining Philosophers problem, and describe a simple probabilistic alternative.
14. Malik, S., Wang, A.R., Brayton, R.K.: Logic verification using binary decision diagrams in a logic synthesis environment. In: ICCAD 1988: Digest of technical papers, pp. 6–9. IEEE Press, Los Alamitos (1988)
15. McMillan, K.L.: The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)
16. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design: OBDD-Foundations and Applications. Springer, Heidelberg (1998)
17. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. PhD thesis, University of Birmingham (2002)
18. PRISM web site, <http://www.prismmodelchecker.org>
19. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York (1994)
20. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic Decision Diagrams and Their Applications. In: IEEE /ACM International Conference on CAD, Santa Clara, California, November 1993, pp. 188–191. ACM/IEEE, IEEE Computer Society Press (1993)
21. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing* 2(2), 250–273 (1995)
22. Beaudenon, V., Encrenaz, E., Taktak, S.: Data decision diagrams for promela systems analysis. In: Software Tools and Technology Transfert (accepted for publication, 2008)
23. Wegener, I.: Branching Programs and Binary Decision Diagrams: Theory and Applications. In: Monographs on Discrete Mathematics and Applications. Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia (2000)

Dynamic Delayed Duplicate Detection for External Memory Model Checking

Sami Evangelista*

DAIMI, University of Aarhus, Denmark
evangel@daimi.au.dk

Abstract. Duplicate detection is an expensive operation of disk-based model checkers. It consists of comparing some potentially new states, the *candidate* states, to previous *visited* states. We propose a new approach to this technique called *dynamic delayed duplicate detection*. This one exploits some typical properties of states spaces, and adapts itself to the structure of the state space to dynamically decide when duplicate detection must be conducted. We implemented this method in a new algorithm and found out that it greatly cuts down the cost of duplicate detection. On some classes of models, it performs significantly better than some previously published algorithms.

Model checking is a method to prove that finite state systems match their specification. Given a model of the system and a property, e.g., a temporal logic formula, it explores all the possible configurations, i.e., the state space, of the system to check the validity of the property. Despite its simplicity, its practical application is limited due to the well-known state explosion problem: the state space can be far too large to be explored in reasonable time or to fit within the available main memory. Consequently, the design of methods able to cope with this problem has gained a lot of interest in the verification community.

A first family of techniques reduce the part of the state space that needs to be explored in such a way that all properties of interest are preserved. An example of such a technique is partial order reduction that limits redundant interleavings by exploiting the independence of some actions.

A more pragmatic approach does not aim at reducing the size of the problem but rather at making a more subtle use of the available resources (or augment them) to extend the range of problems that can be analyzed. Many options are available. We can, for example, compress states to virtually decrease the problem size, distribute the search to benefit from the aggregate computational power and memory of a cluster of machines, or make use of external memory.

In this work we look at this last option. Using disk storage instead of main memory is indeed very tempting since it considerably increases the amount of available memory thereby making it possible to solve problems that could not be solved even with the help of sophisticated techniques such as partial order

* Supported by the Danish Research Council for Technology and Production.

reduction. As a counterpart, disk accesses are much slower. In addition, the data structure used to store states is typically randomly accessed, involving an important runtime penalty when kept on disk. Hence, the use of disk storage requires a dedicated algorithm to be effective.

Most external memory algorithms are somehow based on the key idea of *delayed duplicate detection*. When a state is generated, the algorithm does not immediately check if the state has already been visited, i.e., if the state is a duplicate, since it would require a new disk access, and potentially the load of a disk block. Instead, the state is put into a *candidate* set that contains all the potentially new states and the comparison to the *visited* set stored on disk is delayed until it can be efficiently conducted. Hence, a large number of individual disk look-ups is replaced by a single file scan.

This paper reviews some algorithms proposed by the model checking and artificial intelligence community and introduces a new duplicate detection scheme based on breadth-first search. We refer to this scheme as *dynamic delayed duplicate detection*. Its principle is to exploit some typical properties of state spaces to decrease the cost of duplicate detection and to dynamically collect some data on the structure of the state space that will be used to decide when duplicate detection should be delayed or conducted.

Organization of the paper. We recall in Section 1 some basic elements on graphs and review existing works on disk based model checking. Section 2 introduces a simple variation of hash based delayed duplicate detection [10] that will be the basis of our dynamic algorithm. Some structural properties of state spaces that can be exploited are presented in Section 3. Section 4 contains the main contribution and introduces dynamic delayed duplicate detection that is experimentally evaluated in Section 5. Finally, Section 6 concludes this paper.

1 Background

Definitions and notations. We briefly give the ingredients that are relevant for understanding this paper. Figure 1 will help us illustrate these notions.

A **state space** is a directed graph (S, T, s_0) where S is a finite set of states, $T \subseteq S \times S$ is a set of transitions, and $s_0 \in S$ is the initial state. A state $s' \in S$ is a **successor** (resp. **predecessor**) of state $s \in S$, if $(s, s') \in T$ (resp. $(s', s) \in T$). We denote by $succ(s)$ (resp. $pred(s)$) the set of successors of s (resp. predecessors). The **distance** of a state s , denoted by $d(s)$ is the length of the shortest path from s_0 to s . The states at **level** k , noted $\mathcal{L}(k)$, is the set of all states which distance is k . It is defined recursively as $\mathcal{L}(0) = \{s_0\}$, $\mathcal{L}(k + 1) = \{s' \in S \mid \exists s \in \mathcal{L}(k) \cap pred(s')\} \setminus \cup_{i=0}^k \mathcal{L}(i)$. The **height** of a state space is its number of levels and its **width** is the size of its largest level, i.e., $\max_k |\mathcal{L}(k)|$. In our example, the height and width are respectively 4 and 3.

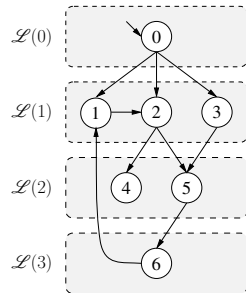


Fig. 1. A state space

The **average degree** is the ratio $\frac{|T|}{|S|}$. $t = (s, s')$ is a **forward transition** if $d(s') = d(s) + 1$. Otherwise it is a **backward transition** and we define its **length** as $d(s') - d(s)$. All transitions of our example are forward transitions except (1,2) and (6,1). Their lengths are respectively of 0, since 1 and 2 are on the same level, and 2.

We denote by $\mathcal{R}(k)$ the successors of states of level k , i.e., $s \in \mathcal{R}(k) \Leftrightarrow \exists s' \in \mathcal{L}(k) \cap \text{pred}(s)$. More generally, $\mathcal{R}^n(k)$ consists of all the states reachable from level k by a path of length n . Formally, $s \in \mathcal{R}^n(k) \Leftrightarrow \exists s' \in \mathcal{L}(k), (s_1, s_2), \dots, (s_n, s_{n+1}) \in T \mid s_1 = s' \wedge s_{n+1} = s$. The successors of states of level 1 in figure 1 is the set $\mathcal{R}(1) = \mathcal{L}(2) \cup \{2\} = \{2, 4, 5\}$ and $\mathcal{R}^2(1) = \{4, 5, 6\}$. By definition, it holds for any d that $\mathcal{L}(d+1) \subseteq \mathcal{R}(d)$ since $\mathcal{R}(d)$ contains level $d+1$ plus all the states reachable from level d by a backward transition.

A breadth-first search (BFS) explores a state space by maintaining a set of visited states and a FIFO queue filled with states to explore. Each state dequeued is expanded and those of its successors that do not belong to the visited set are inserted into it and enqueued to be later expanded. Applying a FIFO strategy ensures that levels of the state space will be processed one by one: if a state s of level k is dequeued then all states of levels $l < k$ have been processed and belong to the visited set. A transition (s, s') generates a **duplicate** s' if s' already belongs to the visited set when s is processed. Obviously, any backward transition will generate a duplicate when applying BFS.

Related work. Dill and Stern [3] were the firsts to propose the use of external memory as a way to enhance the capabilities of explicit model checkers. Their BFS algorithm stores all the states of the current level in a RAM hashtable while previous levels are stored on disk. The search for duplicates occurs each time a BFS level has been completed or the table becomes full. The disk file is then read and duplicates in memory are deleted. Remaining states are written on disk and inserted in the memory queue. Grouping disk lookups is the strategy of most disk based model checkers. It allows to read or write whole blocks of states at once, while checking for each state individually would likely require to reload a new block from disk. This technique is known as *delayed duplicate detection* (DDD): the resolution is postponed until it may be efficiently conducted.

Della Penna et al. [17,16] proposed two algorithms that benefits from a property usually exhibited by communications protocol: backward transitions are usually short. Hence, in BFS, they usually lead to a recently visited state. The first one, [17], is a cache based algorithm that only keeps recent states in memory while the queue is on disk. The second one, [16] is a variation of the initial disk-based algorithm of [3]. Instead of systematically comparing the set of candidate states to the visited set, it is only checked against some blocks of states chosen randomly according to their age.

Bao and Jones observed in [1] that duplicate detection is the most time consuming operation of the algorithms of [3] and [16]. They proposed a new algorithm, based on a partitioned hash table, that mimics a distributed search and behaves better than these two.

With the same motivation, the algorithm of [2] dynamically estimates the costs of performing/delaying duplicate detection and chooses the alternative that a priori seems preferable. Our algorithm is inspired from this principle but also exploits some usual properties of state graphs and changes its strategy according to the specific characteristics of the model.

Hammer and Weber [5] designed a hybrid algorithm that adapts itself to the graph: as long as memory is sufficient it remains a pure RAM based algorithm and it slowly shifts to a disk based algorithm as memory becomes scarce.

In [7], an I/O efficient solution is presented for directed model checking. States are organized into buckets according to some heuristics. Files are associated to buckets in order to store overflowing states. This distribution greatly eases the search for duplicates and the parallelization of the algorithm [8].

Korf introduced in [9] and [10] the principles of sorting-based (SDDD) and hash-based delayed duplicate detection (HDDD) both based on the generic frontier search algorithm that only stores states to be expanded (the frontier). This algorithm cannot really be applied in the context of model checking as it requires the ability to compute the predecessors of a state, an operation that is impossible when the graph is given implicitly as an initial state and a successor function. After each expansion phase SDDD sorts resulting files in order to detect duplicates while HDDD avoids the complexity of sorting by distributing states onto multiple files using two orthogonal hash functions.

In structured duplicate detection [18] an abstraction of the system is used to determine when to load/unload states from/to disk. This technique has a strong potential but heavily relies on the quality of the abstraction. This problem may be overcome by partitioning the edges [20] or by automatically extracting an appropriate abstraction from the system description [19].

2 A Variation of Hash-Based Duplicate Detection

Hash based delayed duplicate detection (HDDD) is a very successful strategy for external memory graph search, already applied to state spaces with more than 10^{12} states [11]. Unfortunately, in its basic version, it is based on the generic frontier search that cannot be applied in the context of implicitly given graphs that we have in model checking. We present in this section a simple breadth first search (BFS) variation of the algorithm of Korf [9]. We refer to this algorithm as BFS-HDDD. Exploring the state space in breadth-first order has several advantages. The most obvious one is its ability to report safety violations of minimal lengths. Secondly, as opposed to, e.g., depth first search, BFS can be easily parallelized. This requires some synchronizations [8], but if the load is well balanced among processors (or nodes of the network), which is the case in the algorithm of [9], it is likely that latency will be negligible. Last, it is possible with BFS to exploit some interesting properties of state spaces to reduce duplicate detection times and fasten the search. This is perhaps the most interesting property of BFS for us since it is an important component of our new algorithm.

The BFS-HDDD algorithm (see figure 2) partitions the queue of states to visit into N files Q_1, \dots, Q_N and the visited set into N files V_1, \dots, V_N . In a first step, queued states are processed, their successors generated and inserted into a memory cache. If this one becomes full its content is flushed to the candidate set, also partitioned in a set of N files C_1, \dots, C_N . A first hash function is used to map states to the appropriate candidate file ensuring that duplicates will be inserted into the same file. Once all queued states have been expanded the cache is flushed to candidate files and the duplicate detection phase begins. In the second step each partition is processed one by one. The content of a candidate file is hashed to memory using a second hash function, thus detecting duplicates in this file. Then, the states of the corresponding visited file are read one by one and deleted from memory. Remaining states in memory are therefore new and can be written in the visited file as well as in the queue file so that they can be processed by the algorithm at the next iteration. Once all partitions are processed the algorithm can move to the next BFS level. Before that, the candidate set is emptied.

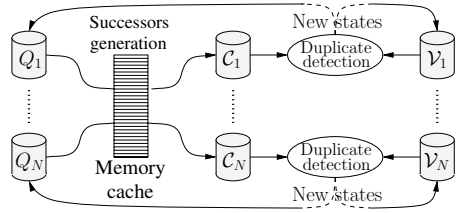


Fig. 2. An iteration of BFS-HDDD

Partitioning the state space has two advantages. First it is helpful to parallelize the algorithm as shown in [11]. Moreover, it virtually multiplies by N the number of candidates that may reside in memory allowing us to perform a single duplicate detection per level. This is in contrast with algorithms of [3] and [16] which also perform a detection when the cache is full. Hence, BFS-HDDD should behave much better with large models for which only a small fraction of the state space can be kept in RAM. However, duplicate detection still remains a costly operation, especially if the graph has a large height. Detections are very cheap at the beginning of the search when the visited set is small but on the last levels each one entails to read from disk a large portion of the state space. More generally, if H is the height of the graph, the number of states read from the visited files during duplicate detections will exactly be $\sum_{s \in S} (H - d(s))$. Some interesting properties usually exhibited by state spaces can however help us to reduce the cost of this operation and design a new algorithm that behaves better than BFS-HDDD.

3 Some Structural Properties of State Spaces

State spaces, as opposed to random graphs, have some typical properties [13] that can be exploited in automated verification. For instance, disk based model checkers can exploit transition locality [17, 16]. Tools can also decide which reduction technique to apply depending on a partial knowledge of the graph [15].

The BEEM database [14] is a precious tool to analyze such properties. It contains more than 50 parametrized models and 300 actual instances of various families. Three observations, that have some consequences in BFS, can be made. The reader may consult [4] for further details on the data provided in this section.

Observation 1: Low proportion of backward transitions. First, as already shown in [17], most transitions of state spaces are forward transitions. The average rate of backward transitions we computed is around 20%. If we only consider communication protocols this rate goes down to approximately 15%.

Observation 2: Few typical lengths for backward transitions. A closer look at the backward transitions also reveals a non uniform distribution of their length as already pointed out in [13]. There are usually a few typical lengths and most backward transitions have one of these lengths. For example, we observed that on 95% of the database instances 5 lengths covered more than 50% of all backward transitions. Even one single length cover more than 50% of backward transitions in 53% of the instances.

Observation 3: Regular evolution of levels. We measured the progression of rate $\frac{|\mathcal{L}(l+1)|}{|\mathcal{L}(l)|}$, that we shall call the *level progression rate* (or more simply progression rate), and found out that the size of levels evolve in a rather regular way and there are usually no huge variations of this rate between close levels. When this is not the case we however noticed that corresponding levels are rather small, meaning that the number of states involved is negligible. Some simple models, e.g, the tower of Hanoi, do not have such a property but if we look at more interesting ones like communication protocols, this observation is often valid. The progression rate generally follows a three step scenario. First levels are characterized by a high rate: levels grow quickly at the beginning of the search. Then the progression rate quickly collapses to a value close to 1 and during a long period stays around this value while tending to decrease. Finally, on last levels, the rate drops down to 0.

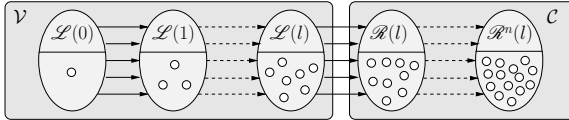
4 Dynamic Delayed Duplicate Detection

We propose in this section, *dynamic delayed duplicate detection* (or DDDD for short), as an alternative to existing duplicate detection schemes. DDDD is based on two key ideas. First, it exploits the structural properties usually exhibited by state spaces that we have discussed in the previous section. We thus obtain a specialized algorithm, especially designed for state spaces having those properties. Second, we dynamically collect data on the graph structure so that the algorithm can adapt itself on-the-fly to its particular characteristics. Thus, even if the model is not, a priori, suited, the algorithm will progressively change its strategy to fit with the model.

4.1 Principle

A breadth first search algorithm based on the DDDD discipline works basically as the algorithm presented in section 2. The only difference is the following one: instead of systematically comparing the candidate set to the visited set at each level, we only perform a duplicate detection when we consider it to be necessary. This decision will mainly be based on data collected by the algorithm during the search. The general principle of DDDD is also the one of [16] and [2] and is motivated by the first observation made in section 3: when we expand the states of level l , it is likely that most of the states reached will not belong to the visited set. Thus, looking for duplicates may be almost useless. Instead we store $\mathcal{R}(l)$ on disk in a candidate set that will be used later during the next duplicate detection. The states expanded at the next BFS level will be those of $\mathcal{R}(l)$ and their successors, i.e., $\mathcal{R}^2(l)$, will also be written in the candidate set, and so on. Only when we decide to perform duplicate detection will the candidate states be hashed to memory and the visited states will be read in order to delete duplicates in memory as done by the BFS-HDDD algorithm. Remaining states in memory are inserted to the visited set and those which are on the “front” of the candidate set, i.e., the states of $\mathcal{R}^n(l)$ (if the detection occurs at level $l + n$) minus the duplicates removed, are later expanded.

The figure below presents a snapshot of visited and candidate sets during the execution of our algorithm. Visited states belongs to $\mathcal{L}(0) \cup \dots \cup \mathcal{L}(l)$ while the candidate set contains all states reachable from level l via a path of length n or less. The latter is actually a multi-set since a state may belong to $\mathcal{R}(l)$, $\mathcal{R}^2(l)$, \dots , and $\mathcal{R}^n(l)$.



Though this strategy is expected to decrease I/Os it has a cost since a state may be reexpanded during the search: any target of a backward transition (or one of its descendant) is likely to be revisited. Since the expansion of a duplicate necessarily leads us to other duplicates, the proportion of duplicates visited may quickly grow even if the graph has few backward transitions. For instance, if 90% of transitions of forward transitions, we can expect to approximately have 10% of duplicates on level $l + 1$, then $1 - 0.9^2 = 19\%$ on level $l + 2$ and more generally, a proportion of $1 - 0.9^n$ duplicates on level $l + n$.

4.2 The Algorithm

The BFS-DDDD algorithm (see figure 3) partitions the visited and candidate sets as well as the BFS queue into N files $\mathcal{V}_1, \dots, \mathcal{V}_N$, $\mathcal{C}_1, \dots, \mathcal{C}_N$ and $\mathcal{Q}_1, \dots, \mathcal{Q}_N$. A unique hash function h is used to map states to these files. The only global data structure to reside in memory is the memory cache $Cache$ implemented by a chained hash table. States overflowing from $Cache$ are stored in some temporary

files $\mathcal{T}_1, \dots, \mathcal{T}_N$ ¹. Those are candidate states and could be directly written in candidate files but we prefer to avoid it as it would involve the possibility to have multiple instances of the same state in a candidate file. Since it is likely that candidate files will be read several times (especially with the optimization described in Section 4.4) this seems preferable.

The search begins with the insertion of the initial state in the appropriate queue file. Each level l is then processed in two steps².

Procedure EXPAND first reads states from the queue and inserts their successors into the cache. If *Cache* becomes full (lines 4-8) a state s'' is chosen, written to the appropriate temporary file if not stored yet and deleted from *Cache*. Once this expansion phase terminates unstored states residing in the cache are written in temporary files (lines 11-14). At this point these will trivially contain all the states of $\mathcal{R}(l)$ and may contain several occurrences of the same state. It may happen if a state is removed from the cache and reached again later within the same expansion phase.

The MERGE procedure decides whether it will perform duplicate detection or postpone it to a future level (line 1) and then processes partitions one by one.

States of the temporary file are first hashed to memory in table \mathcal{H} , hence detecting duplicates in this file (line 4).

If duplicate detection is delayed (lines 5-6), the content of \mathcal{H} is written back to the candidate file in order to be processed during the next duplicate detection.

Otherwise (lines 7-11) the states of the candidate file are also hashed to memory. The visited states of this partition are read from disk and deleted from \mathcal{H} . States in \mathcal{H} are therefore new and written to the visited file. Note that a boolean value is associated to the states of \mathcal{H} in order to identify states in front of the candidate set that will be expanded at the next level (the ones in the temporary file) from those of previous levels which have already been expanded (the ones of the candidate file).

The last step (lines 12-13) consists of writing in the queue file the front states of the candidate set identified as new so that they can be expanded later.

4.3 Deciding When to Perform Duplicate Detection

One question still remains: when should we perform or postpone duplicate detection. Delaying detection comes at the cost of possibly revisiting some duplicates while performing it requires to read the whole visited and candidate sets from disk. The underlying principle of the decision procedure is therefore to delay the detection as long as it estimates that the number of duplicates visited so far is too small to justify a duplicate detection. It is of course impossible to know the number of duplicates in the candidate set as it would require to actually perform the detection, but we can still estimate it from our knowledge of the graph structure.

¹ Thereafter we shall use the term of visited, candidate and temporary sets when speaking of states written in the corresponding files. We write $\mathcal{V} = \cup_i \mathcal{V}_i$, $\mathcal{C} = \cup_i \mathcal{C}_i$ and $\mathcal{T} = \cup_i \mathcal{T}_i$.

² It is actually possible to merge both procedures to save some disk accesses but we separated them for sake of clarity.

```

BFS-DDDD ()
1  for  $i \in 1..N$  do
2     $\mathcal{V}_i := \emptyset$ ;  $\mathcal{C}_i := \emptyset$ ;  $\mathcal{Q}_i := \emptyset$ 
3   $Cache := \emptyset$ ;  $\mathcal{Q}_{h(s_0)}.write(s_0)$ 
4  while  $\exists i \in 1..N$  with  $\mathcal{Q}_i \neq \emptyset$  do
5    EXPAND(); MERGE()
MERGE ()
1   $detection := doDetection()$ 
2  for  $i \in 1..N$  do
3     $\mathcal{H} := \emptyset$ ;  $\mathcal{Q}_i := \emptyset$ 
4    for  $s \in \mathcal{T}_i$  do  $\mathcal{H}.insert(s, true)$ 
5    if  $\neg detection$  then
6      for  $(s, \_) \in \mathcal{H}$  do  $\mathcal{C}_i.write(s)$ 
7    else
8      for  $s \in \mathcal{C}_i$  do  $\mathcal{H}.insert(s, false)$ 
9      for  $s \in \mathcal{V}_i$  do  $\mathcal{H}.delete(s)$ 
10     for  $(s, \_) \in \mathcal{H}$  do  $\mathcal{V}_i.write(s)$ 
11      $\mathcal{C}_i := \emptyset$ 
12     for  $(s, exp) \in \mathcal{H}$  do
13       if  $exp$  then  $\mathcal{Q}_i.write(s)$ 
EXPAND ()
1  for  $i \in 1..N$  do  $\mathcal{T}_i := \emptyset$ 
2  for  $i \in 1..N, s \in \mathcal{Q}_i, s' \in succ(s)$  do
3    if  $s' \notin Cache$  then
4      if  $Cache.isFull()$  then
5         $s'' := Cache.choose()$ 
6        if  $\neg s''.stored$  then
7           $\mathcal{T}_{h(s'')}.write(s'')$ 
8           $Cache.delete(s'')$ 
9           $Cache.insert(s')$ 
10          $s'.stored := false$ 
11     for  $s \in Cache$  do
12       if  $\neg s.stored$  then
13          $\mathcal{T}_{h(s)}.write(s)$ 
14          $s.stored := true$ 

```

Fig. 3. The BFS-DDDD algorithm based on dynamic delayed duplicate detection

Visiting a duplicate is an expensive task as it implies many costly operations. First, the state must be written (read) to (from) the queue file and the candidate file. Then it is expanded³ and its successors are written to temporary files to be later read again. To estimate both alternatives, the algorithm assigns a cost to each of these basic operations: ec for expansions, rc for read accesses and wc for write accesses⁴. We can thus approximate the cost of visiting a duplicate by $ec + (rc + wc) \cdot (2 + deg)$ where deg is the average degree of the graph. Note that, due to cache effect, not all the successors may be written in the temporary files, but we will still use the average degree as an over approximation.

Since duplicate detection implies the read of the whole visited and candidate files we can therefore estimate that delaying detection should be preferred if:

$$(|\mathcal{C}| + |\mathcal{V}|) \cdot rc > |duplicates| \cdot (ec + (rc + wc) \cdot (2 + deg)) \quad (1)$$

where $|duplicates|$ is the total number of duplicates in candidate and temporary files.

This is naturally a rather coarse approximation. What really matters in our sense is that the decision procedure makes its choice on the basis of:

³ An expansion implies several non trivial operations that represent the most time consuming tasks of RAM based model checkers: computation of enabled actions, generation of successors and insertion of the successors into the cache usually via an encoding into bit strings.

⁴ In our implementation we arbitrarily set $rc = 1$, $wc = 2$ and $ec = 2$ which is clearly not the best solution. We propose in section 4.5 a method to set these parameters.

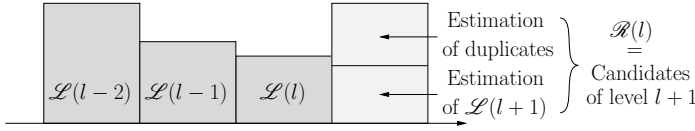


Fig. 4. Estimating the number of duplicates in the candidate set

- the size of the candidate and visited sets: looking for duplicates is very cheap on the first levels and its cost increase as we go deeper into the graph.
- the proportion of backward transitions, which has a direct impact on the number of duplicates: numerous backward transitions will naturally introduce many useless state revisits which in turn mean additional disk accesses.
- the average degree of the graph. This third factor is perhaps less intuitive but still should be considered. A high degree weighs down the cost of visiting duplicates insofar as such a revisit may, in the worst case, lead to approximately deg read/write accesses to temporary files.

Some parameters of formula (II) are available, like $|\mathcal{V}|$, $|\mathcal{C}|$ or the average degree deg that can be estimated from our partial exploration of the graph. Computing $|duplicates|$ is more problematic as it requires to actually perform duplicate detection which is exactly what we want to avoid. Instead, we approximate it using observation 3 from the previous section: the size of levels does usually progress in a regular way. Thus, we can roughly forecast the size of a level from previous ones. We can then reasonably assume that the difference between what we expected and the actual size of the candidate set can be explained by the presence of duplicate candidates.

This forecast process can be illustrated with the help of figure 4. We notice a decrease in previous levels $l-2$, $l-1$ and l . By making the hypothesis that this trend will continue we forecast the size of level $l+1$. We then deduce from this forecast an estimation of the number of duplicates.

Thereafter, we shall denote by $lpr_i = \frac{|\mathcal{L}^{(i+1)}|}{|\mathcal{L}^{(i)}|}$ the level progression rate of level i , l the level of the last detection and $l+n$ the current level. The estimation of some value v will be denoted by \overline{v} . The number of duplicates stored in candidate and temporary files is estimated as follows.

$$\overline{|duplicates|} = \sum_{i=1}^n cr_{l+i} \cdot \min(0, |\mathcal{R}^i(l)| - \overline{|\mathcal{L}(l+i)|}) \quad (2)$$

A correction rate cr_{l+i} , which purpose will be made clear thereafter, is used to over-approximate our estimation.

The size of some level $l+i$ is then estimated from the last level we actually measured, i.e., level l , by combining it with the successive (estimated) progression rates $\overline{lpr}_l, \dots, \overline{lpr}_{l+i-1}$.

$$\overline{|\mathcal{L}(l+i)|} = |\mathcal{L}(l)| \cdot \prod_{j=l}^{l+i-1} \overline{lpr}_j \quad (3)$$

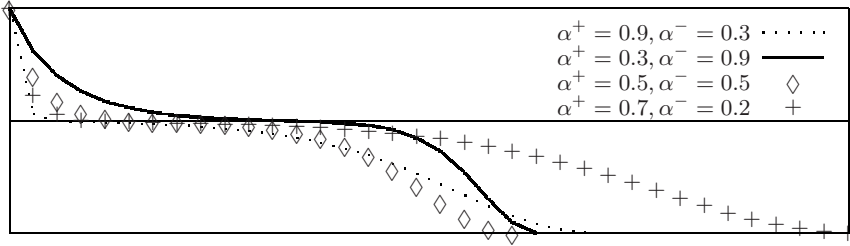


Fig. 5. Curve used to estimate the progression of levels

This simplifies our task since the level progression rate can be estimated from the data previously collected on the graph structure and the size of $\mathcal{R}^i(l)$ is available.

Estimation of $|\mathcal{R}^i(l)|$. For any $i < n$, $\mathcal{R}^i(l)$ is stored in candidate files and we can hence easily know its size. $\mathcal{R}^n(l)$ is stored in temporary files and as the cache may not be large enough, \mathcal{T} is actually a multi-set of $\mathcal{R}^n(l)$. Computing the actual size of $\mathcal{R}^n(l)$ would require to merge temporary files, i.e., remove duplicate in these, before calling *doDetection*, which is a non trivial operation. The solution we implemented is to only merge a few files in order to have an idea of the average multiplicity of each state of $\mathcal{R}^n(l)$ in \mathcal{T} and hence, a more accurate estimation of $|\mathcal{R}^n(l)|$. Note that if the cache is large enough to contain $\mathcal{R}^n(l)$ then $|\mathcal{T}| = |\mathcal{R}^n(l)|$ and this operation is not necessary.

Estimation of the level progression rate. Using observation 3 made in Section 3, we will assume that the progression rate evolves in a regular way and can be captured through the following formula.

$$lpr_{i+1} = \begin{cases} lpr_i \cdot (lpr_i + \epsilon)^{-\alpha^+} & \text{if } lpr_i \geq 1 \\ lpr_i \cdot (lpr_i - \epsilon)^{+\alpha^-} & \text{else} \end{cases} \quad (4)$$

where $\alpha^+ \in [0, 1]$ (resp. $\alpha^- \in [0, 1]$) determines how fast a progression rate greater than 1 (less than 1) drops down to 1 (0); and ϵ is used, first to ensure that the rate will eventually reach 1 and later 0, and second to determine how long the progression rate will stay around 1. In our implementation, we set its value to 0.01.

We use two different values α^+ and α^- as, in general, the progression rate decreases faster when it is above 1 than below. Therefore it is preferable that $\alpha^+ > \alpha^-$. In our implementation, we set $\alpha^+ = 0.7$ and $\alpha^- = 0.2$ as, on the average, these gave us the best results. To give to the reader an idea of the progression induced by this formula, we have plotted in figure 5 the curves for some values of α^- and α^+ .

The progression rate of the current level can then be forecasted from the previous one using formula (4). If a detection occurred on the current level k , the forecast is made from the actual rate since we measured $\mathcal{L}(k - 1)$ and $\mathcal{L}(k)$. Otherwise, it is based on the forecasts made on previous levels.

Correcting the estimation. It is clear that the quality of our estimations will degrade as we move away from the level of the last detection. As a defensive approach we will over-approximate our estimation of the number of duplicates by a correction factor that grows exponentially as we delay detection:

$$cr_{l+i} = (1 + \beta)^{i-1} \quad (5)$$

Hence, we will avoid long series of levels without detection. As on the first levels following a detection the estimation is usually satisfactory (see experience 1 in Section 5), β should be set to a very small value, e.g, 0.02 in our implementation.

4.4 Performing Partial Duplicate Detections

A strategy allowing to reduce revisits is, when detection is delayed, to select a subset of disk states and perform a *partial duplicate detection*. The question is how to select these states in such a way that it helps us to delete many duplicates while still not asking for too much work with respect to a full duplicate detection.

In [16], it was suggested to select states randomly according to the locality principle. As previously suggested by Pelánek in [13], our algorithm rather exploits the second statistical fact described in section 3: backward transitions usually have a few typical lengths. Hence, using our knowledge of the graph we know in which part of the file we should look to delete many duplicates. To this end, the algorithm records for each partition p and level l the position in the visited or candidate file of the first state of level l in partition p . States of a given level can then be recovered using a seek operation in the appropriate file. After each full duplicate detection we then select the k most typical lengths observed so far (k being a user defined parameter) and use these to select stored states during the next partial duplicate detections.

4.5 Extensions

We discuss in this section two possible extensions to the method.

Sampling the state space. BFS-DDDD assigns a cost to each basic operation (wc for a write access, rc for a read access and ec for a state expansion) to decide when to perform duplicate detection. For ideal performance these should be tuned according to the specific characteristics of the model. The value rc and wc should indeed reflect the size of the state vector while ec should be set according to the state generation speed. A possible way to address this problem is to perform a first “training run” using only RAM, as in [6], to collect some data on the model that can be used for a second run using BFS-DDDD. Not only is it useful to tune these parameters, but it can also give us some precious knowledge of the graph structure, e.g., on the length of backward transitions, that could later be exploited by BFS-DDDD.

Profiting from a static analysis of the model. Backward transitions are often triggered by some specific higher level transitions in the system specification.

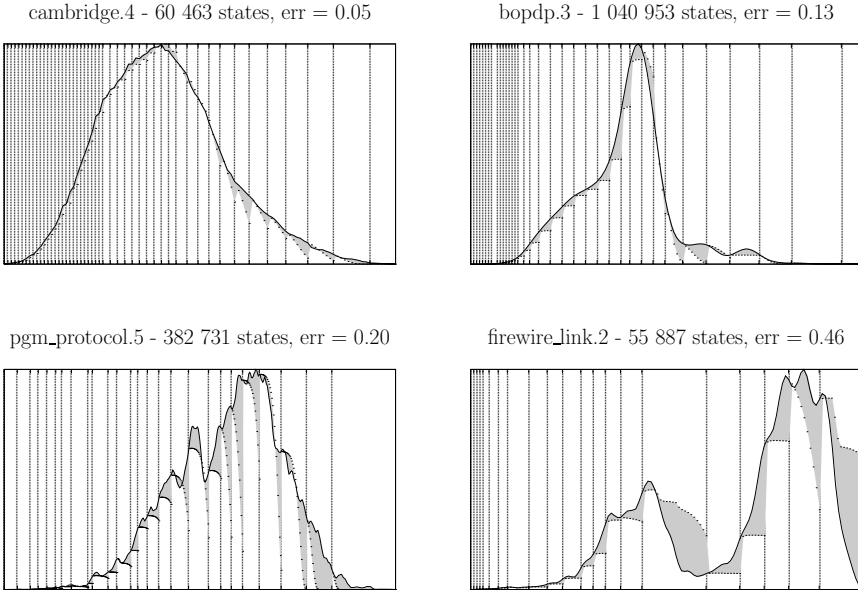


Fig. 6. Experiment 1: comparison of the BFS level graph with our forecast

For instance, `end loop` statements often close cycles and hence are the source of backward transitions. On the opposite, some actions, such as variable incrementations, will never generate backward transitions. It could thus be interesting to perform a static analysis of the model prior to state space exploration to identify actions that could potentially lead to such transitions. These data can then be used by BFS-DDDD to estimate the probability of a newly generated candidate to be a duplicate, depending on the actions associated to the incoming arcs of the state. For states marked as “probably duplicate” it may be interesting to delay their expansion to the next duplicate detection (if the detection revealed that it is actually not a duplicate) rather than expanding them at the next expansion phase. Once again, a first training run may be useful to identify more accurately those actions in the models.

5 Experiments

The BFS-DDDD algorithm has been integrated into the ASAP verification tool [12]. We report in this section the results of a series of experiments. All models are taken from the BEEM database. Some additional data on experience 3 and 4 may be found in [4].

Experiment 1. One may wonder how the algorithm used to estimate the number of duplicates works in practice. To this end, we first selected 204 instances (all

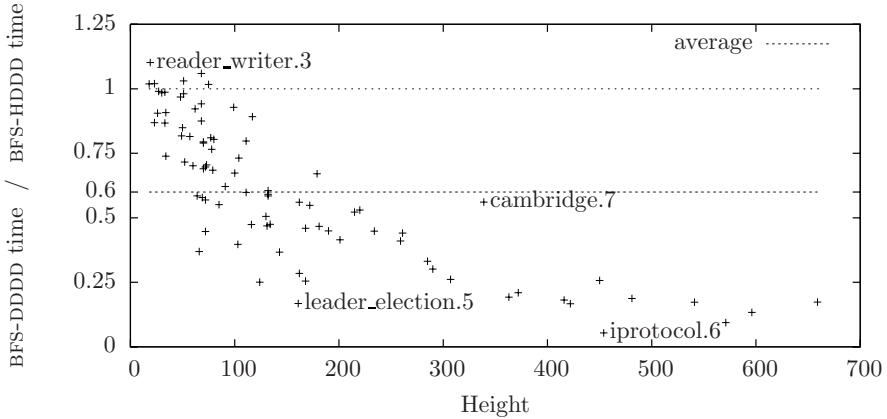


Fig. 7. Experiment 2: comparison of BFS-DDDD and BFS-HDDD

instances with at most 5,000,000 states) and compared their BFS level graphs with the graphs forecasted during the search. We measured the error rate

$$\text{err} = \frac{\sum_{i \in \{1, \dots, n\}} \text{abs} \left(\sum_{j=l_{i-1}+1}^{l_i} |\overline{\mathcal{L}(j)}| - |\mathcal{L}(j)| \right)}{\sum_i |\mathcal{L}(i)|}$$

where $l_0 = -1$ and l_1, \dots, l_n denote levels where detections were performed. The principle of this rate is to observe for each slice $[l_i + 1, l_{i+1}]$ of levels closed by a duplicate detection the distance between the number of states estimated and the number of states actually measured. If detections are performed on each level, using BFS-HDDD, we obtain an average error rate of 0.10 which basically means that our method to evaluate duplicates is viable: it is possible to accurately evaluate a BFS level based on prior levels. With BFS-DDDD the average goes up to around 0.18 which is still rather good. Figure 6 presents some comparisons between the actual BFS level graph (plain curve) and the graph forecasted (dotted curve) using BFS-DDDD. We drew a vertical line for each level with a detection. The graph of `cambridge.4` has a regular bell shape and a large proportion of backward transitions which leads to frequent detections. Our estimation is thus excellent and we can estimate that detection frequency is almost optimal. On the contrary, for `firewire_link.2`, we often over-approximate levels. As a consequence, detections are too largely spaced and too many states are revisited. However, the opposite situation is much more frequent: we generally tend to under-approximate levels (and forecast too much duplicates) and perform unnecessary detections. Hence, our strategy is perhaps not aggressive enough.

Experiment 2. BFS-DDDD was then compared to BFS-HDDD. We selected all the non trivial (with more than 500.000 states) instances of the database and measured the execution times with both algorithms. We plotted the results in figure 7. Each point corresponds to an instance. Data collected confirm our

Table 1. Experiment 3: comparison of BFS-DDDD and PART

Instance	States	PART	BFS-DDDD	BFS-DDDD + partial DD		
				k=2	k=4	k=8
anderson	538 M	17 : 56	0.80	0.28	0.28	0.28
bakery	403 M	7 : 36	0.65	0.47	0.47	0.47
brp2	145 M	20 : 08	0.20	0.20	0.21	0.22
cambridge	255 M	32 : 28	0.62	0.44	0.28	0.30
collision	972 M	25 : 34	0.71	0.67	0.69	0.69
elevator	833 M	17 : 23	1.16	0.86	0.85	0.82
iprotocol	706 M	31 : 01	0.35	0.30	0.29	0.32
lann	421 M	23 : 28	0.47	0.47	0.49	0.52
leader_filters	431 M	7 : 24	0.38	0.38	0.38	0.38
lup	379 M	8 : 21	1.03	0.29	0.31	0.32
peterson	142 M	3 : 20	0.76	0.71	0.73	0.77
rether	151 M	54 : 08	0.07	0.07	0.07	0.08
telephony	534 M	12 : 57	1.01	0.98	0.95	0.95
train-gate	478 M	26 : 34	0.24	0.24	0.25	0.31
			0.60	0.45	0.45	0.46

initial expectations: BFS-DDDD is especially interesting for long graphs, i.e., with a large height. This is however not the only parameter: the proportion of backward transitions also plays an important role. For example, in the case of `cambridge.7`, it is not so interesting to use BFS-DDDD: even though its graph is long, it has a high proportion of backward transitions ($> 50\%$) which leads us to perform many duplicate detections; whereas we observe good performances for `leader_election.5` which has the opposite characteristics. Rates observed go from 1.10 (`reader_writer.3`) to 0.05 (`iprotocol.6`). The average is around 0.6. This does not look like an important improvement but, as we shall see it, the major interest of our algorithm is that it scales much better than BFS-HDDD to large state spaces. Moreover the next experiment shows that partial detections often allow to delete many duplicates at a low cost.

Experiment 3. We then compared our algorithm to PART [11], based on a partitioned hash table. This choice is mostly motivated by the fact that, according to our experiments and the ones of Bao and Jones, it outperforms the algorithms of [3] and [16] that can be considered, roughly speaking, as parents of BFS-DDDD.

To compare these algorithms we tried to select a representative set of models according to these parameters: average degree, width and height, proportion of backward transitions, distribution of backward transition lengths. As we previously saw these may have a certain impact on the performance of our algorithm.

Table 1 presents the results of this experiment. For each instance we performed one run with PART and several runs with BFS-DDDD, first without partial detection and then with partial detections with different values for parameter k (number of lengths considered). Each run was given the same amount of memory, that is, the ability to keep at most $16 \cdot 10^6$ states in memory. This represents, in most

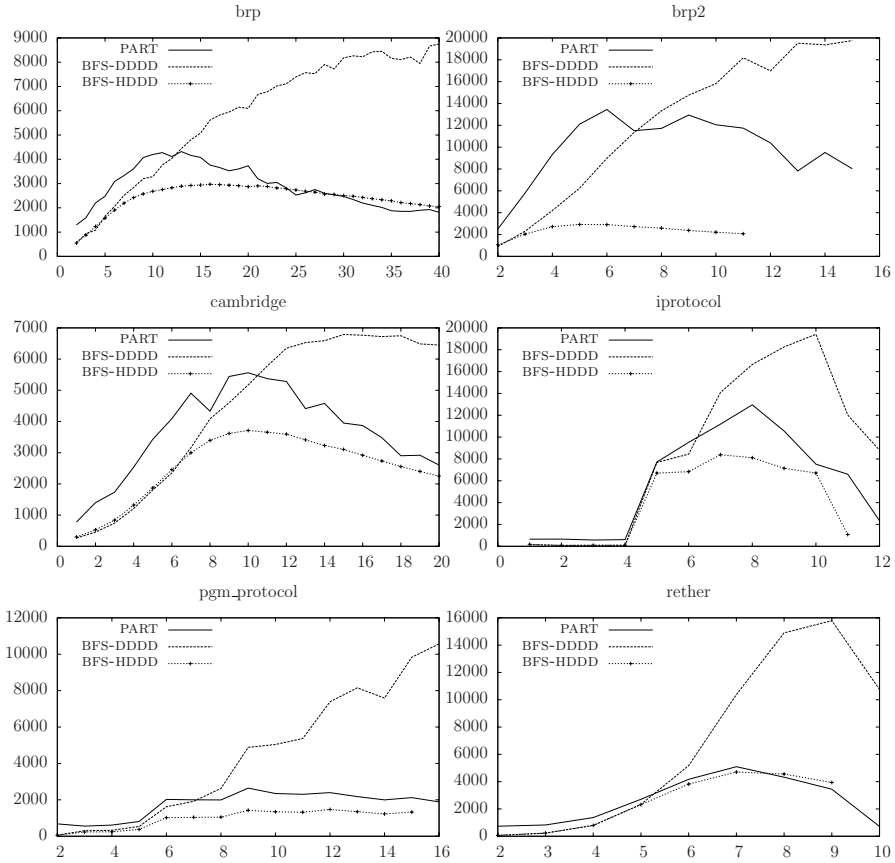


Fig. 8. Experiment 4: Comparison of PART, BFS-HDDD and BFS-DDDD

cases, a small fraction of the state space. Execution times are expressed in the form *hours:minutes* for PART and as a fraction of this time for BFS-DDDD. Best times have been written in bold. The last row indicates average values.

We noticed that PART is very sensitive to the graph structure: it is best suited to wide and short graphs. In this case queues associated with partitions are filled with many states meaning that few partition swaps will occur although disk queues will be accessed more frequently. Therefore, we can basically make the same observations as in the previous experiment: BFS-DDDD is comparatively better on long graphs (e.g., **brp2**, **iprotocol**, **rether**), with preferably, few backward transitions. Loading/unloading partitions is a major time consuming operation of PART for these kinds of graphs, especially if the state vector is large, e.g., for **rether**. **telephony** is typically the worst input we can think of for BFS-DDDD. It is short and has 16 levels with more states than the cache can hold.

Since its average degree is high this leads to a huge amount of disk accesses in temporary files. To a lesser extent, the same remark also applies to `elevator`.

Performing partial detections is especially interesting when backward transitions have very few typical lengths. `anderson` is a caricatural case as all its backward transitions have the same length. Therefore partial detections helped us to divide the execution time by almost three. This also applies to `lup` which graph has two lengths that cover more than 90% of backward transitions. For graphs that do not have typical lengths, e.g., `train-gate` or `brp2`, this optimization does not bring any improvement. Hence, it should always be turned on: at worst, we will not gain anything.

Experiment 4. The last experiments were done with 6 real life protocols: `brp`, `brp2` (timed version of `brp`), `cambridge`, `iprotocol`, `pgm_protocol` and `rether`. Our goal was to evaluate how PART, BFS-HDDD and BFS-DDDD behave as the graph gets larger and the height increases.

Figure 8 presents our results. We gave each algorithm the same amount of memory and for BFS-DDDD, we set parameter k to 4 as it gave us the best results in previous experiment. For some fixed parameters we progressively increased another parameter (on the x-axis, see 4 for details) and recorded the search speed as $\frac{\text{nodes of the graph}}{\text{search time}}$ (on the y-axis). We remark that PART is more efficient for small graphs although there is no huge difference. However, above a certain point BFS-DDDD becomes more interesting. In addition, even though there generally is a moment where the speed decreases for all algorithms, this instant occurs later for BFS-DDDD. At last, many communication protocols have in common that their BFS level graphs are terminated by a long series of very small levels - that possibly correspond to the termination phase of the protocol. This property also explains why BFS-DDDD evolves better on such models.

6 Conclusion

This article proposed an adaptive duplicate detection scheme for external memory model checking that borrows several ideas from the literature: [10,13,16,2]. Its principle is to collect during the search some data that help us to determine when detection should be performed or postponed. We evaluated this method on several models of different families and complexities and find out that the new algorithm is especially well suited to communication protocols with long graphs and few backward transitions that are quite common in model checking.

Besides the extensions described in Section 4.5, we plan to refine the way lengths are selected during partial duplicate detections. Our scheme assumes that these lengths do not evolve during the search. This assumption is apparently invalid in many cases where lengths are function of the level.

Acknowledgments. I thank Lars Michael Kristensen and Michael Westergaard for their comments on earlier versions of this article.

References

1. Bao, T., Jones, M.: Time-efficient model checking with magnetic disk. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 526–540. Springer, Heidelberg (2005)
2. Barnat, J., Brim, L., Simecek, P., Weber, M.: Revisiting resistance speeds up I/O-efficient ltl model checking. In: Proc. of TACAS. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
3. Dill, D.L., Stern, U.: Using magnetic disk instead of main memory in the Mur ϕ verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
4. Evangelista, S.: Dynamic delayed duplicate detection for external memory model checking. Technical report, DAIMI, University of Aarhus, Denmark (2008), <http://daimi.au.dk/~evangelii/doc/ddd.pdf>
5. Hammer, M., Weber, M.: To store or not to store reloaded: Reclaiming memory on demand. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
6. Holzmann, G.J.: State compression in spin: Recursive indexing and compression training runs. In: Proceedings of the Third Spin Workshop (1997)
7. Jabbar, S., Edelkamp, S.: I/O Efficient Directed Model Checking. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 313–329. Springer, Heidelberg (2005)
8. Jabbar, S., Edelkamp, S.: Parallel external directed model checking with linear I/O. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 237–251. Springer, Heidelberg (2005)
9. Korf, R.E.: Delayed duplicate detection: Extended abstract. In: Proc. of IJCAI, pp. 1539–1541. Morgan Kaufmann, San Francisco (2003)
10. Korf, R.E.: Best-first frontier search with delayed duplicate detection. In: Proc. of AAAI, pp. 650–657. AAAI Press/The MIT Press (2004)
11. Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: Proc. of AAAI, pp. 1380–1385. AAAI Press/The MIT Press (2005)
12. Kristensen, L.M., Westergaard, M.: The ascoveco state space analysis platform. In: Proc. of the 8th CPN workshop, DAIMI-PB, pp. 1–6 (2007)
13. Pelánek, R.: Typical structural properties of state spaces. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
14. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
15. Pelánek, R.: Model classifications and automated verification. In: Proc. of FMICS. LNCS. Springer, Heidelberg (2007)
16. Della Penna, G., Intrigila, B., Tronci, E., Venturini Zilli, M.: Exploiting transition locality in the disk based Murphi verifier. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 202–219. Springer, Heidelberg (2002)
17. Tronci, E., Della Penna, G., Intrigila, B., Venturini Zilli, M.: Exploiting transition locality in automatic verification. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 259–274. Springer, Heidelberg (2001)
18. Zhou, R., Hansen, E.A.: Structured duplicate detection in external-memory graph search. In: Proc. of AAAI, pp. 683–689. AAAI Press/The MIT Press (2004)
19. Zhou, R., Hansen, E.A.: Domain-independent structured duplicate detection. In: Proc. of AAAI. AAAI Press/The MIT Press (2006)
20. Zhou, R., Hansen, E.A.: Edge partitioning in external-memory graph search. In: Proc. of IJCAI, pp. 2410–2417 (2007)

State Focusing: Lazy Abstraction for the Mu-Calculus^{*}

Harald Fecher¹ and Sharon Shoham²

¹ Albert-Ludwigs-Universität Freiburg, Germany

fecher@informatik.uni-freiburg.de

² The Technion, Haifa, Israel

sharonsh@cs.technion.ac.il

Abstract. A key technique for the verification of programs is counterexample-guided abstraction refinement (CEGAR). In a previous approach, we developed a CEGAR-based algorithm for the modal μ -calculus, where refinement applies only locally, i.e. *lazy abstraction* techniques are used. Unfortunately, our previous algorithm was not completely lazy and had some further drawbacks, like a possible local state explosion. In this paper, we present an improved algorithm that maintains all advantages of our previous algorithm but eliminates all its drawbacks. The improvements were only possible by changing the philosophy of refinement from *state splitting* into the new philosophy of *state focusing*, where the states that are about to be split are not removed.

1 Introduction

The modal μ -calculus [19] is an expressive modal logic that allows to express safety, reachability, and mixtures of these properties, by using fixpoint constructions. The μ -calculus is a sensible choice for branching time properties, which are relevant whenever nondeterminism occurs from external factors (e.g. user input), from the modeling of faulty systems/channels [9], or from the abstraction of time or arguments [7]. In particular, the μ -calculus can express most of the standard logics, like LTL and CTL. Hence, the μ -calculus is an ideal basis for foundation-examinations.

For automatic verification of properties, the state explosion has to be tackled. One of the most successful techniques to checking correctness of large or even infinite programs is predicate abstraction [12] with *counterexample-guided abstraction refinement* (CEGAR) [5]. This approach consists of three phases: abstraction, model checking, and refinement. Refinement is performed by adding a new predicate that splits the abstract states. A prominent safety-checking tool based on CEGAR is BLAST [16], where refinement is applied locally (called *lazy abstraction*), i.e., only the abstract states of a trace which comprises a spurious counterexample are refined. This avoids the state space doubling obtained when the whole state space is split via a new predicate.

Adapting the idea of lazy abstraction to the μ -calculus is not straightforward. One reason is that in order to preserve branching time properties, an abstract model needs two kinds of transitions (called *may*, respectively *must*, *transitions*). Examples of such models are (Kripke) modal transition systems [20][7]. They also allow to preserve both

^{*} This work is financially supported by the DFG projects (FE 942/2-1) and (SFB/TR 14 AVACS).

validity and *invalidity* from the abstract model to the concrete model, at the cost of introducing a third truth value *unknown*, which means that the truth value in the concrete model is unknown. This leads to a *3-valued semantics*. In this setting, refinement is no longer needed when the result is *invalid*, as in traditional CEGAR approaches. Instead, refinement is needed when the result is *unknown*. As such, the role of a counterexample as guiding the refinement is taken by some cause of the indefinite result.

In [10], we have developed a preliminary algorithm for μ -calculus verification, having the following advantages: (i) Refinement is made lazily. More precisely, some, but not all, *configurations* (abstract states combined with subproperties) having the same abstract state are split during a refinement phase: The state space remains smaller and verification is sped up. (ii) The more expressive *generalized Kripke modal transition systems* [24] are used as underlying abstract models (they have must hypertransitions, i.e. transitions pointing to a set of states rather than to a singleton): A smoother refinement determination is obtained [24] and more properties (in principle, every least fixpoint free μ -calculus formula) can be shown. (iii) Refinement determination is separated from the model checking: Refinement-heuristics can be defined independently. (iv) Configurations and transitions that become irrelevant by newly obtained information of (in)validity are removed: Complexity is reduced.

The algorithm of [10] still has the following disadvantages: (I) A set of configurations rather than the single configuration determined by a refinement-heuristic is split: Verification is unnecessarily slowed down, since often expensive splits that do not contribute to the verification are made. (II) All may-/must-transitions that can arise as a result of a split are calculated even if they are not needed: Avoidable, expensive satisfiability checks are made. (III) An exponential blowup can occur during a refinement phase, since all hypertransitions obtained as the powerset of the may-transitions are calculated. (IV) Some interesting information for defining refinement-heuristics is lost: The split of accompanying configurations along with the one that is determined by the refinement-heuristic obscures the intermediate results obtained after each split, and might divert the refinement into undesired directions. These disadvantages cannot be eliminated with existing techniques, with the exception of (III) that was addressed by [24,25], yet their approaches rely on particular model checking algorithms.

Contribution. We develop the new technique of *state focusing* for refinement: the states that are about to be split are not removed. Instead, the ‘old’ states are connected to the ‘new’ states that result from their split via *focus-transitions*. This allows to encode hypertransitions, but more importantly, it allows to perform a local refinement, in which propagation of a split is deferred until it is called for by a refinement-heuristic. We use this new technique to construct a new lazy, CEGAR-based algorithm for the μ -calculus. Our algorithm uses a configuration structure (where the abstract states are combined with subproperties) to encode the verification problem. In each iteration, (in)valid configurations are determined, the structure is simplified accordingly, a refinement-heuristic is used to determine a refinement step, and refinement is performed *locally* by either splitting (focusing) one configuration, or propagating a previous split to other configurations or components of the structure. Our new algorithm still has all the advantages (i) – (iv) and additionally does not have the disadvantages (I) – (IV). In particular, our algorithm combines the following properties:

- At most two configurations are added during a refinement step.
- No (in)validity is lost during a refinement or simplification calculation.
- Every satisfiability check is made on demand, i.e. unnecessary satisfiability checks are avoided except if called for by the refinement-heuristic.
- The capability of verification with generalized Kripke modal transition systems as the underlying abstract models is preserved while avoiding state explosion, since only the hypertransitions obtained constructively via old configurations are present, as in the non-lazy abstraction approach of [24].
- Improved simplifications of the underlying configuration structure are made by using a 9-valued logic, which is an extension of the 6-valued one used in [3].
- A separation of the refinement determination from the model checking is made, allowing to define refinement-heuristics separately.
- Improved refinement-heuristics (compare with (IV)) are possible, since only elementary updates of the configurations structure are made during a refinement step.
- Other refinements (e.g., that of [10]) can be imitated at no further cost by gathering together several local refinement steps.

Further related work. The state space doubling occurring after splitting the whole state space via a predicate can also be tackled by the usage of BDDs, as in the tool SLAM [4]. There the abstract transition relation is encoded as a BDD, avoiding the explicit calculation of the exponentially large state space. Such a BDD approach is generalized from the safety properties checked by SLAM to μ -calculus properties in the tool YASM [15]. There, the underlying abstract model is equivalent to a Kripke modal transition system, which is less expressive than generalized Kripke modal transition systems.

A CEGAR-approach to branching time properties is given in [23], where, contrary to our approach, only the transition relation is under, resp., over approximated (the state space remains unchanged). In [14,13], CEGAR-based algorithms for the μ -calculus are presented having only Kripke modal transition systems as underlying abstract model. Furthermore, there every configuration for which (in)validity is not yet shown is split, i.e. only a weak form of lazy abstraction is made.

In [22] models are abstracted by *alternating transition systems with focus predicates*. These resemble game-graphs with must-hypertransitions. Refinement is not discussed in this paper. Must-hypertransitions were first introduced in disjunctive modal transition systems [21]. A CEGAR-approach for the more general alternating μ -calculus is given in [1], where must- as well as may-hypertransitions are used in the underlying abstract model. Refinement is made globally (not locally) and the refinement determination depends on the model checking algorithm, i.e. no separation is used. [2] increases expressiveness without using hypertransitions: Backward must-transitions and entry/exit points are used to conclude the existence of transitive must-transitions.

In [11], cartesian abstraction, where ‘previous’ abstract states also remain existent, is used for improving under approximations. However, there, the ‘old’ states are not used to encode hypertransitions, and thus less expressive abstractions are obtained. Moreover, our technique also improves the over approximation by forbidding may-transitions subsumed by may-transitions whose targets are more precise (less abstract).

2 Underlying Structures

Notations. Throughout, functional composition is denoted by \circ . Given a relation $\rho \subseteq B \times D$ with subsets $X \subseteq B$ and $Y \subseteq D$ we write $X.\rho$ for $\{d \in D \mid \exists b \in X : (b, d) \in \rho\}$ and $\rho.Y$ for $\{b \in B \mid \exists d \in Y : (b, d) \in \rho\}$. Let $\text{map}(f, \Phi)$ be the sequence obtained from the sequence Φ by applying function f to all elements of Φ pointwise. $f[b \mapsto d]$ denotes the function that behaves as f except on b , which is mapped to d . Suppose D is ordered, then $\sqsubseteq \subseteq (B \rightarrow D) \times (B \rightarrow D)$ denotes the derived pointwise order between functions in $B \rightarrow D$. Furthermore, for $f, f' : B \rightarrow D$ expression $f \sqcup f'$ denotes the least function (if existent) that is above f and f' w.r.t. \sqsubseteq .

System. Without loss of generality, we will not consider action labels on models in this paper. A *rooted transition system* $T = (S, s^i, \rightarrow, \mathcal{L})$ consists of a (possibly infinite) set S of states, an initial state $s^i \in S$, a transition relation $\rightarrow \subseteq S \times S$, and a *predicate language* \mathcal{L} , which is a set of predicates that are interpreted over the states in S (i.e. each predicate $p \in \mathcal{L}$ denotes a set $\llbracket p \rrbracket \subseteq S$), such that there exists $p^i \in \mathcal{L}$ with $\llbracket p^i \rrbracket = \{s^i\}$. The boolean and exact predecessor closure of \mathcal{L} is denoted by $\overline{\mathcal{L}}$, where $\llbracket _ \rrbracket$ over boolean operators is straightforwardly extended and $\llbracket \text{pre}(\psi) \rrbracket = \rightarrow . \llbracket \psi \rrbracket$ for $\psi \in \overline{\mathcal{L}}$.

Intermediate games. They are a generalization of the three-valued parity games of [10,13] by using a third-kind of states (called intermediate game states) that are not controlled by a unique player, but change the player depending on the type of the play (validity vs. invalidity). The intermediate game states are used to model state-focusing: The states to be split become intermediate game states. Intermediate games also use a more complex validity image to improve refinement and simplification determinations:

Definition 1. An intermediate game $G = (C_0, C_1, C_{\frac{1}{2}}, C^i, R, R^-, R^+, \theta, \omega)$ has

- pairwise disjoint sets of game states for Player 0 (C_0), for Player 1 (C_1), and intermediate game states $C_{\frac{1}{2}}$; the union of all game states is denoted $C = C_0 \cup C_{\frac{1}{2}} \cup C_1$,
- a set of initial game states $C^i \subseteq C$,
- a set of normal transitions $R \subseteq C \times C$,
- a set of must- and a set of may-transitions $R^-, R^+ \subseteq (C_0 \cup C_1) \times C$,
- a parity function $\theta : C \rightarrow \mathbb{N}$ with finite image, and
- a validity function $\omega : C \rightarrow \{\text{tt}, \text{ff}, \perp\} \times \{\text{tt}, \text{ff}, \perp\}$, where we write ω_1 , respectively ω_2 , for applying the projection to the first, respectively second, component of ω .

The values of ω are explained in detail in Sec. 3. In general, ω_1 is used to determine the winner in the validity game, whereas ω_2 is used in the invalidity game:

Definition 2. – Finite validity (resp. invalidity) plays for intermediate game G have the rules and winning conditions as stated in Table 7. An infinite play Φ is a win for Player 0 iff $\text{sup}(\text{map}(\theta, \Phi))$ is even; otherwise it is won by Player 1.

- G is valid (invalid) in $c \in C$ iff Player 0 (resp. Player 1) has a strategy for the corresponding validity (resp. invalidity) game such that Player 0 (resp. Player 1) wins all validity (resp. invalidity) plays started at c with her strategy. G is valid (invalid) iff G is valid (resp. invalid) in all games states of C^i .

Table 1. Moves of (in)validity game at game state c , specified through a case analysis. A player also wins if its opponent has to move but cannot.

Moves of the validity game:

$\omega_1(c) \neq \perp$: Player 0 wins if $\omega_1(c) = \text{tt}$; otherwise Player 1 wins;

$\omega_1(c) = \perp$ and $c \in C_0$: Player 0 picks as next game state $c' \in \{c\} \cdot (R \cup R^-)$;

$\omega_1(c) = \perp$ and $c \in C_{\frac{1}{2}} \cup C_1$: Player 1 picks as next game state $c' \in \{c\} \cdot (R \cup R^+)$;

Moves of the invalidity game:

$\omega_2(c) \neq \perp$: Player 1 wins if $\omega_2(c) = \text{ff}$; otherwise Player 0 wins;

$\omega_2(c) = \perp$ and $c \in C_1$: Player 1 picks as next game state $c' \in \{c\} \cdot (R \cup R^-)$;

$\omega_2(c) = \perp$ and $c \in C_0 \cup C_{\frac{1}{2}}$: Player 0 picks as next game state $c' \in \{c\} \cdot (R \cup R^+)$;

In the validity game, Player 0 has the role of the checker, and Player 1 has the role of the refuter. In the invalidity game their roles switch. In both cases, must-transitions are only used by the checker, whereas may-transitions are only used by the refuter. In addition, the $C_{\frac{1}{2}}$ game states are always controlled by the refuter.

Remark 1. The validity, as well as the invalidity, game obviously correspond to a parity game. Therefore, decidability of validity, resp. invalidity, is in $\text{UP} \cap \text{coUP}$ [18].

Property language. We use an automata representation of the μ -calculus [19]:

Definition 3 (Tree automata). An (alternating tree) automaton $A = (Q, q^i, \delta, \Theta)$ has

- a finite, nonempty set of states $(q \in) Q$ with the initial element $q^i \in Q$
- a transition relation δ mapping automaton-states to one of the following forms, where q, q_1, q_2 are automaton states and $p \in \mathcal{L}$: $p \mid \neg p \mid q_1 \tilde{\wedge} q_2 \mid q_1 \tilde{\vee} q_2 \mid \diamond q \mid \square q$
- an acceptance condition $\Theta: Q \rightarrow \mathbb{N}$ with finite image.

An example of an automaton is depicted in Fig. 1 (α) on page 103, where all automaton-states have acceptance value 0. In the following, $Q_{\mathcal{L}} = \{q \in Q \mid \delta(q) \in \mathcal{L}\}$, $Q_{\text{qua}} = \{q \in Q \mid \delta(q) \in \bigcup_{q' \in Q} \{\diamond q', \square q'\}\}$ and $Q_0 = \{q \in Q \mid \delta(q) \in \bigcup_{q_1, q_2 \in Q} \{q_1 \tilde{\vee} q_2, \diamond q_1\}\}$ resp. $Q_1 = \{q \in Q \mid \delta(q) \in \bigcup_{q_1, q_2 \in Q} \{q_1 \tilde{\wedge} q_2, \square q_1\}\}$ denotes those under control of Player 0 resp. Player 1. Satisfaction of a rooted transition system w.r.t. an automaton is obtained via transformation into an intermediate game:

Definition 4. The property-game for T and A , denoted $P_{T,A}$, is an intermediate game $(S \times Q_0, S \times Q_1, S \times Q_{\mathcal{L}}, \{(s^i, q^i)\}, R, R^-, R^+, \Theta \circ \pi_Q, \omega)$, where π_Q denotes the projection to the second component and

$$\begin{aligned}
 R &= \{((s, q), (s, q')) \mid \exists q'' : \delta(q) \in \{q' \tilde{\wedge} q'', q'' \tilde{\wedge} q', q' \tilde{\vee} q'', q'' \tilde{\vee} q'\}\} \\
 R^- &= R^+ = \{((s, q), (s', q')) \mid \delta(q) \in \{\diamond q', \square q'\} \wedge (s, s') \in \rightarrow\} \\
 \omega(s, q) &= \begin{cases} (\text{tt}, \text{tt}) & \text{if } q \in Q_{\mathcal{L}}, s \in \llbracket \delta(q) \rrbracket \\ (\text{ff}, \text{ff}) & \text{if } q \in Q_{\mathcal{L}}, s \notin \llbracket \delta(q) \rrbracket \\ (\perp, \perp) & \text{otherwise} \end{cases}
 \end{aligned}$$

Furthermore, we write $T \models A$, whenever $P_{T,A}$ is valid, and otherwise, we write $T \not\models A$ (which is equivalent to $P_{T,A}$ being invalid).

Note that our definition of $T \models A$ coincides with the standard definition of satisfaction, and $T \not\models A$ coincides with the satisfaction of the dual formula, i.e. corresponds to negation. Next, special intermediate games derived for automata satisfaction on abstracted systems are introduced. In the following, Z is used to describe subsets of the system's state space. More precisely, $Z = \{z : \overline{\mathcal{L}} \rightarrow \{+, ?, -\} \mid \infty > |z|\}$ with $|z| = |\{\psi \in \overline{\mathcal{L}} \mid z(\psi) \neq ?\}|$. The set $\{+, ?, -\}$ is ordered by $\overset{+}{\setminus} \overset{-}{/}$. The elements of Z can be thought of as abstract states obtained through predicate abstraction, which is made explicit as follows: The derived formula for $z \in Z$, which characterizes the underlying system states, is $\psi_z = ((\bigwedge_{\psi:z(\psi)=+} \psi) \wedge (\bigwedge_{\psi:z(\psi)=-} \neg\psi))$. We say that z is *finer* than z' if $z \sqsupseteq z'$, i.e. $\forall \psi \in \overline{\mathcal{L}} : z(\psi) \geq z'(\psi)$, which ensures that $\llbracket \psi_z \rrbracket \subseteq \llbracket \psi_{z'} \rrbracket$.

Definition 5. An abstract property-game P w.r.t. Q is $(G, R^{-?}, R^{+?}, Z_{\text{sat}}, Z_{\text{unsat}})$, where

- G is an intermediate game such that $C \subseteq Z \times Q$. We write π_Z , respectively π_Q , for the projection from C to its first, respectively second, component.
- $R^{-?}, R^{+?} \subseteq C \times C$ are a set of possible must- and a set of possible not-may-transitions such that $R^{-?} \cap R^{-} = \emptyset$ and $R^{+?} \subseteq R^{+}$.
- $Z_{\text{sat}}, Z_{\text{unsat}} \subseteq Z$ are disjoint sets of abstract states indicating for which abstract state satisfiability (resp. unsatisfiability) is ensured.

The set of all its transitions is $R_{\text{all}} = R \cup R^{-} \cup R^{-?} \cup R^{+} \cup R^{+?}$.

The underlying states of an abstract property-game are also called configurations in the sequel. In our algorithm, we are only interested in abstract property-games where Q is the underlying state space of an automaton and where G obeys the following additional notations and constraints, described only informally: Configurations whose Q -component is in Q_0 (Q_1) belong in general to C_0 (resp. C_1), except that they can also be intermediate game states (in $C_{\frac{1}{2}}$). The parity function θ of G is given by $\Theta \circ \pi_Q$, where Θ is the automaton's acceptance condition.

The normal transitions of G consist of two types; those leaving configurations from $C_{\frac{1}{2}}$ are called *focus-transitions* and those leaving configurations from $C_0 \cup C_1$ and having a property of type $\tilde{\vee}$ or $\tilde{\wedge}$ are called *junction-transitions*. The corresponding configurations are called *junction configurations*. Junction, as well as $C_{\frac{1}{2}}$, configurations, have at most two outgoing transitions. The must- and may-transitions leave $C_0 \cup C_1$ configurations having type \diamond or \square , also called *quantifier configurations*.

Focus-transitions are used to partition an abstract state into several configurations: Each of the targets of the outgoing focus-transitions of an abstract game state $c \in C_{\frac{1}{2}}$ describes a part of the set of concrete states described by c . The automaton component does not change along focus-transitions. Junction-transitions imitate the automaton transitions, but the abstract state of the source of a junction transition might be finer than the abstract state of its target (as a result of refinement). Must and may transitions change both the Q -component of the game state according to the automaton transition relation, and the abstract states according to the system. The must-transitions are used to underapproximate the concrete transitions of T , whereas the may-transitions are used as an overapproximation. Namely, a must-transition exists only if *all* the concrete

system states represented by the source game state have a corresponding transition to *some* concrete state represented by the target game state. This is called the $\forall\exists$ rule. Every must-transition satisfies it, but possibly not all the transitions that satisfy it are included. On the other hand, a may-transition exists (at least) if *some* concrete state represented by the source state has a corresponding transition to *some* concrete state represented by the target state. This is called the $\exists\exists$ rule. Every transition that satisfies it has to be included as a may-transition, but possibly more are contained.

Since the approximations given by the must and may transitions are not always precise, we use $R^{-?}$ to denote the transitions that are candidates to be *added* as must-transitions, as they might satisfy the $\forall\exists$ rule. Dually, $R^{+?}$ denotes transitions that are candidates to be *removed* from the set of may-transitions, as they might not satisfy the $\exists\exists$ rule. The initial abstract property-game is the following:

Definition 6. *The initial abstract property-game $P_{T,A}^I$ for T and A is*

$((C_0, C_1, \{z?\} \times Q_{\mathcal{L}}, \{(z?, q^i)\}, R, \emptyset, R^+, \Theta \circ \pi_Q, \omega), R^+, R^+, \{z?\}, \emptyset)$, where

$$z?(p) = ? \text{ for any } p \in \overline{\mathcal{L}}$$

$$C_0 = (\{z?\} \times Q_0) \cup \{(z?[p \mapsto x], q) \mid (x = -, \delta(q) = p) \text{ or } (x = +, \delta(q) = \neg p)\}$$

$$C_1 = (\{z?\} \times Q_1) \cup \{(z?[p \mapsto x], q) \mid (x = +, \delta(q) = p) \text{ or } (x = -, \delta(q) = \neg p)\}$$

$$R = \{((z?, q), (z?, q')) \mid \exists q'' : \delta(q) \in \{q' \tilde{\wedge} q'', q'' \tilde{\wedge} q', q' \tilde{\vee} q'', q'' \tilde{\vee} q'\}\} \cup$$

$$\{((z?, q), (z?[p \mapsto x], q)) \mid \delta(q) \in \{p, \neg p\}, x \in \{+, -\}\}$$

$$R^+ = \{((z?, q), (z?, q')) \mid \delta(q) \in \{\diamond q', \square q'\}\}$$

$$\omega(z, q) = (\perp, \perp) \text{ for } (z, q) \in C_0 \cup C_1 \cup \{z?\} \times Q_{\mathcal{L}}.$$

Note that the initial abstract property-game does not depend on T . It consists of a single abstract state $z?$, which abstracts any concrete system-state. The may-transitions overapproximate the concrete transitions by including a transition from $z?$ to itself. The underapproximation is empty. All the may-transitions are candidates to be added as must-transitions, or alternatively be removed from the set of may-transitions. Here, all the configurations whose property is in Q_0 are C_0 -states, and all the configurations whose property is in Q_1 are C_1 -states, as in the concrete property-game. The $C_{\frac{1}{2}}$ configurations consist of the combination of $z?$ with the predicate subproperties. Such a game state is divided according to the predicate, via focus-transitions, into two configurations having the same subproperty (predicate) but whose states are less abstract: one where the predicate is added to $z?$, and another where its negation is added to $z?$. The game state where the abstract state agrees with the predicate is a C_1 -state, meaning Player 0 wins in it (since it has no outgoing transition for Player 1 to use). The game state that represents disagreement between the abstract state and the predicate is a C_0 -state, meaning Player 1 wins in it. This makes the $C_{\frac{1}{2}}$ -state, which is controlled by Player 1 in the validity game and by Player 0 in the invalidity game, neither valid nor invalid, indicating that the value of the predicate in $z?$ is unknown. The set of configurations for which satisfiability is ensured (Z_{sat}) is initialized to $\{z?\}$, since system states exist, and the set where unsatisfiability is ensured (Z_{unsat}) is initialized as empty. The initial abstract property-game for the property and system given in Fig. 1 is illustrated in Fig. 2(a). There, the abstract state $z?$ is denoted by \emptyset since no predicate is set in it.

3 CEGAR Via Lazy Abstraction

Validity values. We start by explaining the 9 different validity-values. A configuration with abstract state z and property q of an abstract property-game can only be valid (resp. invalid) if all the underlying concrete states of z satisfy (resp. falsify) q . Thus, validation in abstract property-games is no longer 2-valued, since it is possible that some underlying concrete states satisfy the formula and some do not. This typically leads to a 3-valued setting, where (z, q) can be neither valid nor invalid. In case unsatisfiable abstract states are allowed, as in our case, it even leads to a 4-valued setting, where (z, q) can be both valid and invalid (if z is unsatisfiable). We use further validity values that help us define improved simplifications of the game structure. Namely, we distinguish between (in)validity and existential-(in)validity: (in)validity ensures that *all* the underlying concrete states are (in)valid, possibly vacuously. Existential-(in)validity ensures that there *exists* an underlying (in)valid concrete state. Existential-invalidity thus ensures that the configuration is *not* valid, and dually for existential-validity. These possibilities are recorded by ω_1 w.r.t. validity and by ω_2 w.r.t. invalidity. Namely, we use ω_1 to determine if the configuration is valid (tt), existential-invalid (ff) – meaning it is *not* valid, or its validity is unknown (\perp). Dually for ω_2 .

More precisely, (tt, \perp) stands for valid, (\perp ,tt) stands for existential-valid, and (tt,tt) stands for valid-and-satisfiable. Similarly, (\perp ,ff) stands for invalid, (ff,ff) stands for invalid-and-satisfiable, and (ff, \perp) stands for existential-invalid. Value (tt,ff) stands for unsatisfiable, (ff,tt) stands for existential-mixed, and (\perp , \perp) stands for unknown.

Example. We illustrate how our algorithm works on a toy example. We describe the underlying abstract property-game, the underlying model checking algorithm, the simplifications of the underlying game structure, and the possible refinement steps (which depend on a heuristic). The algorithm is illustrated by checking the μ -calculus formula presented via an automaton in Fig. 1(a) at the system depicted in Fig. 1(b).

Fig. 2(a) presents the initial abstract property-game, defined in Def. 6 and explained thereafter. The unknown existence of may and must transitions is indicated by the symbol ? on the transitions. The focusing of predicates configurations can be viewed as a degenerate split that takes place in the initial abstract property-game. (In)validity is determined via a parity game algorithm. The configurations where Player 0 has a winning strategy in the validity game are labeled as valid (tt, \perp), whereas the ones where Player 1 has a winning strategy in the invalidity game are labeled as invalid (\perp ,ff), as shown in Fig. 2(b). No further validity-label improvements and redundant transitions/configurations removals are possible in Fig. 2(b).

Since the initial configuration in Fig. 2(b) is undetermined, refinement is needed. Thus, a heuristic, determining how to refine the abstraction, is applied. The different possibilities are illustrated by the remainder of the example. Assume that the heuristic determines that the initial configuration, denoted c , needs to be split according to the least precondition of $true$ (the characterizing formula of the abstract state \emptyset), denoted $\tilde{p} \equiv \text{pre}(true)$. Then the structure Fig. 2(c) is obtained, where two initial configurations that correspond to the division of c by \tilde{p} are added, c becomes an intermediate ($C_{\frac{1}{2}}$) configuration, corresponding focus-transitions are added from c to the new configurations, and the outgoing transitions of c are redirected to the new configurations

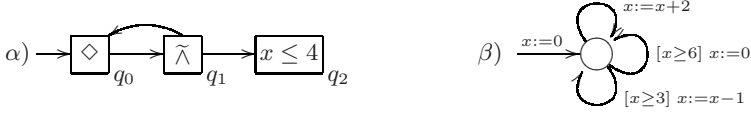


Fig. 1. A μ -calculus formula (α) in terms of automata (see Def. 3), and a system (β). The property of (α) holds if there is an infinite path possible such that always $x \leq 4$ holds. It corresponds to the CTL formula $EG(x \leq 4)$. In (β), the range of x is \mathbb{N} , initialized with 0. The actions of the transitions can be executed whenever the guard, depicted in rectangular brackets, is valid.

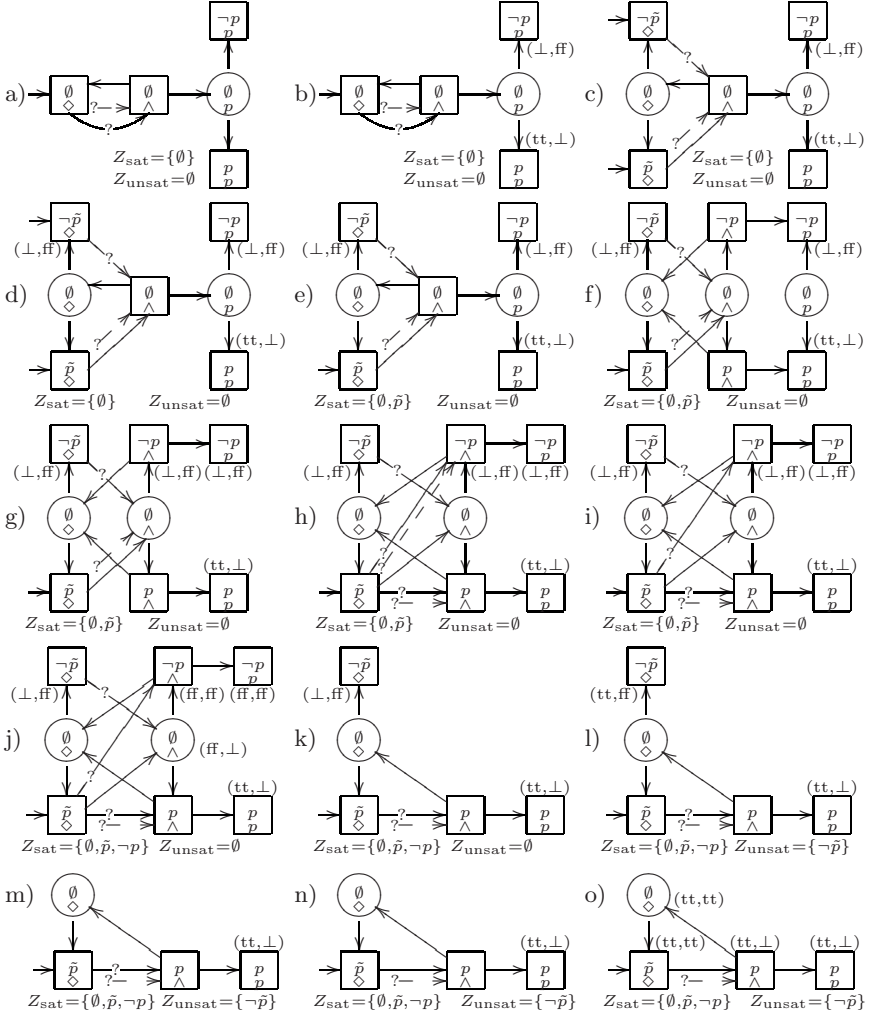


Fig. 2. Example of a property check via lazy abstraction. Here, $p \equiv x \leq 4$ and $\tilde{p} \equiv \text{pre}(\text{true})$. May-transitions are depicted as dashed arrows and must-, as well as focus- and junction-, transitions as solid arrows. May-/must-transition whose existence is unknown contain symbol ?. Intermediate configurations are depicted as circles and the others are depicted as rectangles.

(by doubling them). Note that only c is split. Here we already determine the existence of the must-transition from (\tilde{p}, \diamond) to (\emptyset, \wedge) since \tilde{p} represents the least precondition of \emptyset , thus the $\forall\exists$ condition necessarily holds. By analogous arguments, we determine the absence of the may-transition from $(\neg\tilde{p}, \diamond)$ to (\emptyset, \wedge) (this detection mechanism is not yet included in the algorithm in order to increase readability). The validity check determines the upper left configuration to be invalid, yielding Fig. 2(d).

Next, assume that the heuristic determines that the lower left configuration needs to be checked to determine if it contains the initial concrete state (since two initial configurations currently exist as a result of the previous split). This is indeed the case here. Therefore, the set of initial configurations becomes this singleton set. Moreover, the satisfiability of the corresponding abstract state \tilde{p} is implied, thus Z_{sat} is extended. Fig. 2(e) is obtained.

Now, assume the heuristic determines that the “degenerate” split of the intermediate configuration (\emptyset, p) needs to be propagated backwards along the junction-transition t that points to it. Then the source (\emptyset, \wedge) of t , denoted $\text{sor}(t)$, is divided into two new configurations via the predicate p that determined the split of (\emptyset, p) . This is done by dividing $\text{sor}(t)$ according to the targets of the focus-transitions that leave (\emptyset, p) (these are the configurations to which (\emptyset, p) was split). As a result, $\text{sor}(t)$ becomes an intermediate configuration, corresponding focus-transitions are added from $\text{sor}(t)$ to the new configurations, t is redirected (by doubling it) such that each of its copies connects one of the new configurations directly to the corresponding target of the focus-transitions leaving (\emptyset, p) that agrees with it on the splitting predicate p (instead of connecting it to (\emptyset, p)), and other transitions leaving $\text{sor}(t)$ are redirected (by doubling them) such that their sources become the new configurations. As a result, the original target (\emptyset, p) of t becomes unreachable. Thus Fig. 2(f) is obtained and after (in)validity determination and removal of the unreachable configuration, Fig. 2(g) is obtained.

Assume the heuristic yields the (unique) may-transition t that points to the intermediate configuration (\emptyset, \wedge) in order to redirect it to the configurations that resulted from the previous split of (\emptyset, \wedge) . Then t is replaced by may-transitions (for which existence is not ensured) pointing to the targets of the focus-transitions leaving (\emptyset, \wedge) . Also, possible must-transitions are added to those configurations, but previous must-transitions (to the intermediate configuration (\emptyset, \wedge)) are not removed. These previous must-transitions can be considered as hypertransitions. Thus Fig. 2(h) is obtained and after the removal of irrelevant transitions Fig. 2(i) is obtained. There, the (not ensured) may transition from (\tilde{p}, \diamond) to $(\neg p, \wedge)$ is removed. This is because the source configuration is controlled by Player 0 and may transitions are used by Player 0 in the invalidity game. However, since the target of the transition is labeled by (\perp, ff) , Player 0 will not use this transition in a winning strategy in the invalidity game, since it will make him lose (by reaching a configuration whose ω_2 -value is ff). Thus, removing it does not change the outcome.

Assume the heuristic determines that satisfiability of the abstract state encoded by $\neg p$ needs to be checked. The state is satisfiable and therefore Z_{sat} is extended. Furthermore, the validity-value of the two configurations having this abstract state and that are invalid (\perp, ff) is modified to (ff, ff) , indicating that, beside the fact that *all* underlying concrete states are invalid, there also *exists* an invalid underlying concrete state. Using this information, the intermediate configuration (\emptyset, \wedge) pointing to one of those

configurations via a focus-transition is labeled with a value (ff, \perp) (existential-invalid) indicating that there is an underlying concrete state which is invalid. This is justified by the fact that the intermediate configuration has the same subproperty as the targets of its outgoing focus-transitions and its abstract state represents a superset of their abstract states. Thus Fig. 2(j) is obtained and after the removal of irrelevant transitions and unreachable configurations Fig. 2(k) is obtained. Namely, first both of the (possible-)must transitions pointing to the intermediate configuration (\emptyset, \wedge) are removed. This is because the sources of these transitions are controlled by Player 0 and must transitions are used by Player 0 in the validity game, but the ω_1 -value of the target (\emptyset, \wedge) of the transitions is ff , which makes Player 0 lose. Therefore, Player 0 will not use these transitions in a winning strategy in the validity game, and removing them does not change the outcome. Similar arguments are responsible for the removal of the possible-must transition pointing to $(\neg p, \wedge)$. The removal of these transitions makes (\emptyset, \wedge) , $(\neg p, \wedge)$ and $(\neg p, p)$ unreachable and they are removed with their transitions.

Assume the heuristic yields the abstract state encoded by \tilde{p} for which satisfiability needs to be checked. The state is unsatisfiable and therefore Z_{unsat} is extended. Furthermore, the upper left configuration having this abstract state is labeled as unsatisfiable (tt, ff) . Thus Fig. 2(l) is obtained and after the removal of irrelevant transitions and unreachable configurations Fig. 2(m) is obtained. Namely, the focus-transition pointing to the unsatisfiable configuration is removed (along with the unsatisfiable configuration), since it will never be used as a part of a winning strategy: in the validity game its source is controlled by Player 1, yet, the ω_1 -value of its target is tt , making Player 1 lose. Analogously, in the invalidity game it makes Player 0 who controls it lose, since the ω_2 -value of its target is ff .

Finally, assume the heuristic yields the (unique) possible-must-transition from (\tilde{p}, \diamond) to (p, \wedge) , for which existence needs to be checked. After checking the $\forall\exists$ condition by an unsatisfiability check of $\tilde{p} \wedge \neg \text{pre}(p)$, the must transition is added to the structure. Thus Fig. 2(n) is obtained. The parity-game algorithm determines all the configurations as valid, after which the validity function is adapted to (tt, tt) in the configurations where the states are known to be satisfiable. This yields Fig. 2(o), where the calculation terminates, since the property is verified: the single initial configuration is valid (the first component of its validity-value is tt).

Base algorithm. Table 2 presents the verification algorithm `PropertyCheck` and its used procedure `Validity`, which

determines the (in)valid configurations of a given abstract property-game and adapts the validity function as best as possible. In Line 1, Z_{unsat} is used to determine unsatisfiable states. In Line 2 the validity algorithm is applied and the determined validity is stored in the first component of ω . In Line 3, valid configurations become valid-and-satisfiable if the underlying abstract state is known to be satisfiable, i.e. is in Z_{sat} . In Line 4, a configuration c that points via a chain of focus-transitions to a configuration for which the existence of a concrete state satisfying the corresponding property is known, i.e. where ω_2 is tt , is also updated to have value tt for ω_2 . This is because the concrete states described by c are a superset of those described by the targets of its outgoing focus-transitions. Lines 5-7 make the analogous adaptations

Table 2. A model checking algorithm `PropertyCheck` for the μ -calculus. Its used procedure for validity determination `Validity` is presented here and the procedure for the refinement calculation is given in Table 3. Here, $P = ((C_0, C_1, C_{\frac{1}{2}}, C^i, R, R^-, R^+, \theta, \omega), R^{-?}, R^{+?}, Z_{\text{sat}}, Z_{\text{unsat}})$.

Algorithm. `PropertyCheck` (A : automaton, T : rooted transition system)

Local variables P : an abstract property-game, initialized with $P_{T,A}^I$

- 1: `Validity` (P)
- 2: while $(\exists c^i \in C^i : \omega_1(c^i) \neq \text{tt}) \wedge (\exists c^i \in C^i : \omega_2(c^i) \neq \text{ff})$ do
- 3: Remove all transitions in R_{all} leaving a configuration c with $\omega(c) \in \{(\text{tt}, \text{tt}), (\text{ff}, \text{ff}), (\text{tt}, \text{ff})\}$.
- 4: If $c \in C_0 \wedge (\omega_1(c) = \perp \Rightarrow \omega_1(c') = \text{ff}) \wedge (\omega_2(c) = \perp \Rightarrow \omega_2(c') = \text{ff})$ or $c \in C_1 \wedge (\omega_1(c) = \perp \Rightarrow \omega_1(c') = \text{tt}) \wedge (\omega_2(c) = \perp \Rightarrow \omega_2(c') = \text{tt})$ or $c \in C_{\frac{1}{2}} \wedge \omega(c') = (\text{tt}, \text{ff})$ then Remove (c, c') from R .
- 5: If $c \in C_0 \wedge (\omega_2(c) = \perp \Rightarrow \omega_2(c') = \text{ff})$ or $c \in C_1 \wedge (\omega_1(c) = \perp \Rightarrow \omega_1(c') = \text{tt})$ then Remove (c, c') from $R^+ \cup R^{+?}$.
- 6: If $c \in C_0 \wedge (\omega_1(c) = \perp \Rightarrow \omega_1(c') = \text{ff})$ or $c \in C_1 \wedge (\omega_2(c) = \perp \Rightarrow \omega_2(c') = \text{tt})$ then Remove (c, c') from $R^- \cup R^{-?}$.
- 7: Remove from P all the configurations which are unreachable from the initial configurations via R_{all} .
- 8: `Refine` (P)
- 9: `Validity` (P)
- 10: if $\forall c^i \in C^i : \omega_1(c^i) = \text{tt}$ then return(tt) else return(ff)

Algorithm. `Validity` (P : an abstract property-game)

- 1: Set ω to (tt, ff) on every configuration $c \in C$ where $\pi_Z(c) \in Z_{\text{unsat}}$.
- 2: Determine the valid configurations via a parity-game algorithm and set ω_1 to tt on those.
- 3: Set $\omega_2(c)$ to tt on every configuration $c \in C$ where $\omega_1(c) = \text{tt}$ and $\pi_Z(c) \in Z_{\text{sat}}$.
- 4: Set ω_2 to tt on every configuration from $C_{\frac{1}{2}}$ that points to a configuration where ω_2 is tt.
- 5: Determine the invalid configurations via a parity-game algorithm and set ω_2 to ff on those.
- 6: Set $\omega_1(c)$ to ff on every configuration $c \in C$ where $\omega_2(c) = \text{ff}$ and $\pi_Z(c) \in Z_{\text{sat}}$.
- 7: Set ω_1 to ff on every configuration from $C_{\frac{1}{2}}$ that points to a configuration where ω_1 is ff.

concerning invalidity determination. As a result of Lines 4 and 7, $C_{\frac{1}{2}}$ -configurations may get the “pure existential” values $(\perp, \text{tt}), (\text{ff}, \perp), (\text{ff}, \text{tt})$, which are later used to simplify the game. No other configurations can get these values.

`PropertyCheck` starts by constructing the initial abstract property-game obtained from a given automaton. Then it repeatedly applies `Validity`, makes some simplifications (explained below), and calculates a refinement step until the property is verified or falsified, which is the case if the initial configurations are either all valid or all invalid. Redundant transitions (and configurations) are removed as follows. In Line 3, the outgoing transitions of configurations whose validity value is from $\{(\text{tt}, \text{tt}), (\text{ff}, \text{ff}), (\text{tt}, \text{ff})\}$ are removed. This is reasonable, since any play in one of the games will end at such configurations. The same argument holds for value (ff, tt) that might be given to a $C_{\frac{1}{2}}$ configuration, but here (in)validity is not completely resolved, and therefore the outgoing transitions are not removed, since they might be necessary during refinement.

Transitions that will never be chosen in a winning strategy are removed as follows. In Line 4 junction-transitions leaving \tilde{v} -configurations c , i.e. $c \in C_0$, are removed whenever each not yet determined component of $\omega(c)$ is ff in the target of the transition. This is justified since Player 0 would lose (in both games) by using such transitions. Analogously for $c \in C_1$. Focus-transitions to unsatisfiable configurations are also removed since they are losing for their controlling player (in both games). In Line 5 (resp. Line 6) may- (resp. must-)transitions are removed in similar conditions as for junction-transitions except it is sufficient to consider only one component of $\omega(c)$, since may- (resp. must-)transitions can only be used by Player 0 in the invalidity game or Player 1 in the validity game (resp. Player 0 in the validity game or Player 1 in the invalidity game). Finally, all unreachable configurations are removed in Line 7.

Refinement algorithm. The pseudo code of the refinement algorithm is presented in Table 3. There, a heuristic is used to determine the refinement. The heuristic may depend on further arguments other than the abstract property-game, like the system to be checked or the history of refinement. It can also be probabilistic. We describe each possible refinement-scenario along with the way it is handled algorithmically.

The first two scenarios of the refinement correspond to the local split, better focusing, of a quantifier-, resp. junction-, configuration (Line 1 resp. Line 7). A split always originates either in a configuration whose subproperty is a predicate (such configurations are split in the initial abstract property-game already) or in a quantifier-configuration. It later might be propagated to junction-configurations. A local split is performed by turning the configuration into a $C_{\frac{1}{2}}$ configuration, which serves as an auxiliary configuration, and introducing new subconfigurations. Newly introduced configurations map via ω to (\perp, \perp) and via θ as $\Theta \circ \pi_Q$. Thus, during refinement, C_0 or C_1 configurations can become $C_{\frac{1}{2}}$ configurations. However, a C_0 configuration will never become a C_1 configuration and vice versa. Similarly, $C_{\frac{1}{2}}$ -configurations never become C_0 or C_1 configurations. In addition, the initial configurations always remain $C_0 \cup C_1$ configurations. Furthermore, as a result of the local splitting, the configurations used in the abstract property-game might overlap. However, we have the property that when considering only $C_0 \cup C_1$ configurations having the same property (i.e. the same second component), then their abstract states have disjoint underlying sets of concrete states. The different calculations for the two splitting scenarios are described in more detail below, followed by the other scenarios of the refinement, which are aimed at updating the components of the abstract property-game and making them more precise (e.g. after a split took place). Note that in previous papers the propagation of a split to junction-configurations, as well as the updates of the other components, were all performed together in each refinement step.

Splitting quantifier-configurations (Line 1). A quantifier-configuration $c = (z, q) \in C_0 \cup C_1$ whose validity is unknown, i.e. $\omega(c) = (\perp, \perp)$, is determined together with a predicate $\psi \in \overline{\mathcal{L}}$ performing the split. The predicate is unused in z , i.e. $z(\psi) = ?$ (otherwise no improvement takes place). Two new configurations where z is set to positive, resp. negative, at ψ are added to the configurations of the corresponding player, and c becomes a $C_{\frac{1}{2}}$ configuration that points via focus-transitions to the newly added configurations (Line 2). The transitions outgoing c are redirected (and doubled) such that they leave the two new configurations (Line 5). The redirected must-transitions

Table 3. Here, $P = ((C_0, C_1, C_{\frac{1}{2}}, C^i, R, R^-, R^+, \theta, \omega), R^{-?}, R^{+?}, Z_{\text{sat}}, Z_{\text{unsat}})$. Newly introduced configurations map via ω to (\perp, \perp) and via θ as $\Theta \circ \pi_Q$. $\text{Satisfiable}(\psi)$ denotes a satisfiability check of ψ (checks if $\llbracket \psi \rrbracket \neq \emptyset$) made by a theorem prover, and similarly for $\text{Unsatisfiable}(\psi)$. For $(c_s, c_t) \in R_{\text{all}}$ let $R_{\text{ta}}^{\uparrow(c_s, c_t)} = \{(c_s, c'_t) \in R_{\text{all}} \mid c_t \in \{c'_t\}.R_{\frac{1}{2}}^*\}$ and $R_{\text{ta}}^{\downarrow(c_s, c_t)} = \{(c_s, c'_t) \in R_{\text{all}} \mid c'_t \in \{c_t\}.R_{\frac{1}{2}}^*\}$, where $R_{\frac{1}{2}}^*$ is the transitive closure of the focus transitions $R \cap (C_{\frac{1}{2}} \times C)$.

Algorithm. Refine (P : a property-game)

A heuristic determines one of the following cases including the determination of the corresponding configuration, transition, etc.:

- 1: [determine $((z, q), \psi) \in (C_0 \cup C_1) \times \overline{\mathcal{L}}$ with $q \in Q_{\text{qua}} \wedge z(\psi) = ? \wedge \omega(z, q) = (\perp, \perp)$:
 % Here, $c = (z, q)$ and $C' = \{(\{z[\psi \mapsto +]\}, q), (\{z[\psi \mapsto -]\}, q)\}$ and $j \in \{1, 2\}$ with $q \in Q_j$.
- 2: $C_j := (C_j \setminus \{c\}) \cup C'$ and $C_{\frac{1}{2}} := C_{\frac{1}{2}} \cup \{c\}$ and $R := R \cup (\{c\} \times C')$
- 3: $R^{+?} := \{(c', c'') \mid c' \in C' \wedge (c, c'') \in R^+\} \cup R^{+?} \setminus (\{c\} \times C)$
- 4: $R^{-?} := \{(c', c'') \mid c' \in C' \wedge (c, c'') \in R^+ \setminus R^-\} \cup R^{-?} \setminus (\{c\} \times C)$
- 5: $R^u := \{(c', c'') \mid c' \in C' \wedge (c, c'') \in R^u\} \cup R^u \setminus (\{c\} \times C)$ with $u \in \{+, -\}$
- 6: if $c \in C^i$ then $C^i := C' \cup C^i \setminus \{c\}$
- 7: [determine $(c, c') \in R \cap ((C_0 \cup C_1) \times C_{\frac{1}{2}})$ with $\omega(c) = (\perp, \perp) \wedge \omega(c') \notin \{(\text{tt}, \text{tt}), (\text{ff}, \text{ff}), (\text{tt}, \text{ff})\}$:
 % Here, $c = (z, q)$ and $c' = (z', q')$ and $C'' = \{(\tilde{z}, q) \mid \exists (z', q') \in \{c'\}.R \wedge \tilde{z} = (z' \sqcup z)\}$.
- 8: $R := R \setminus \{(c, c')\} \cup \{(\tilde{z}, q), (z', q')\} \mid (z', q') \in \{c'\}.R \wedge \tilde{z} = (z' \sqcup z)\}$
- 9: if $c \notin C''$ then % otherwise $C'' = \{c\}$
- 10: $C_j := (C_j \setminus \{c\}) \cup C''$ and $C_{\frac{1}{2}} := C_{\frac{1}{2}} \cup \{c\}$ % Here, $j \in \{0, 1\}$ with $q \in Q_j$.
- 11: $R := (\{c\} \times C'') \cup \{(c'', \tilde{c}) \mid c'' \in C'' \wedge (c, \tilde{c}) \in R\} \cup (R \setminus (\{c\} \times C))$
- 12: if $c \in C^i$ then $C^i := C'' \cup C^i \setminus \{c\}$
- 13: [determine $(c, c') \in R^+$ with $c' \in C_{\frac{1}{2}} \wedge \omega(c') \notin \{(\text{tt}, \text{tt}), (\text{ff}, \text{ff}), (\text{tt}, \text{ff})\}$:
 % Here $R' = \{c\} \times (\{c'\}.R)$
- 14: $R^{-?} := R^{-?} \cup R'$ and $R^u := (R^u \setminus \{(c, c')\}) \cup R'$ with $u \in \{+, +?\}$
- 15: [determine $(c, c') = ((z, q), (z', q')) \in R^{+?}$: if **Unsatisfiable** $(\psi_z \wedge \text{pre}(\psi_{z'}))$
- 16: then $R^u := R^u \setminus \{(c, c')\}$ with $u \in \{+, +?\}$
- 17: else $R^{+?} := R^{+?} \setminus \{(c, c')\}$
- 18: [determine $(c, c') = ((z, q), (z', q')) \in R^{-?}$: if **Unsatisfiable** $(\psi_z \wedge \neg \text{pre}(\psi_{z'}))$
- 19: then $R^- := (R^- \setminus R_{\text{ta}}^{\uparrow(c, c')}) \cup \{(c, c')\}$ and $R^{-?} := R^{-?} \setminus R_{\text{ta}}^{\uparrow(c, c')}$
- 20: else $R^{-?} := R^{-?} \setminus R_{\text{ta}}^{\downarrow(c, c')}$
- 21: [determine $(z, q) \in C^i$: if **Satisfiable** $(p^i \wedge \psi_z)$
- 22: then $C^i := \{(z, q)\}$ and $Z_{\text{sat}} := Z_{\text{sat}} \cup \{z\}$
- 23: [determine $z \in Z$: if **Satisfiable** (ψ_z)
- 24: then $Z_{\text{sat}} := Z_{\text{sat}} \cup \{z\}$
- 25: else (if **Unsatisfiable** (ψ_z) then $Z_{\text{unsat}} := Z_{\text{unsat}} \cup \{z\}$)

remain ensured. The redirected may-transitions, on the other hand, might be overly approximated due to the specialization of the source. Thus, they are added as possible not-may-transitions (Line 3). Furthermore, the may-transitions which are not subsumed by ensured must-transitions are also added as possible must-transitions (Line 4), since the specialization of the source can cause them to fulfill the $\forall\exists$ rule. Finally, if c is an initial configuration, it is replaced by the new ones (Line 6). In Fig. 2 ‘splitting quantifier-configuration’ is performed from (b) into (c).

Splitting junction-configurations (Line 7). Here, the purpose is to propagate a previous split to the antecedent junction-configurations of the split configuration. Therefore, a transition (c, c') from a junction-configuration whose validity is unknown, i.e. $\omega(c) = (\perp, \perp)$, to a $C_{\frac{1}{2}}$ -configuration is determined, i.e. $(c, c') \in R \cap ((C_0 \cup C_1) \times C_{\frac{1}{2}})$. The idea is to split c via the two configurations reachable from c' by focus-transitions, which are the configurations to which c' was split earlier. Thus, c' is determined such that its validity is not from $\{(tt, tt), (ff, ff), (tt, ff)\}$, since otherwise either all the underlying concrete states of c' are in agreement or none exist, and in both cases no improvement will result from splitting c according to the split of c' .

Then c is (possibly) split as follows. For each of the two configurations \tilde{c}' , reachable via a focus-transition from c' , we consider in C'' a configuration that corresponds to the least upper bound $z \sqcup \tilde{z}'$ of the abstract state of c and the abstract state of \tilde{c}' , if it exists. The concrete states encoded by $z \sqcup \tilde{z}'$ correspond to the intersection of the underlying concrete states of z and \tilde{z}' . Note that while \tilde{z}' is finer than the abstract state z' of c' , it is not guaranteed that \tilde{z}' is finer than z , since z could become finer than z' by previous splits based on other outgoing junction-transitions of c . In particular, z and \tilde{z}' might give contradictory values (+ vs. -) to some predicate (meaning they represent disjoint sets of concrete states), in which case $z \sqcup \tilde{z}'$ does not exist. Still, at least for one focus-transition target such an upper bound exists, since by an invariant z is finer than z' . Moreover, exactly one additional predicate $\psi \in \overline{\mathcal{L}}$ is set (either to + or to -) in \tilde{z}' compared to z' (along the focus-transition). ψ is the predicate that c' was split by. This means that z is finer than \tilde{z}' w.r.t. all predicates, except for possibly ψ . Now, if ψ is not set in z , then ψ does not introduce contradictions as well, thus (1) upper bounds exist for both of the focus-transitions targets. Otherwise (ψ is already set in z as a result of a previous split), then (2) the least upper bound exists (only) for the one focus-transition target in which ψ is set the same as in z .

In case (2), the only existing least upper bound is equal to z , since in this case z is already finer than \tilde{z}' ($\tilde{z}' \sqsubseteq z$), i.e., it was already split by ψ (and therefore the abstract state of the other focus-transition target is disjoint from z). Thus, $C'' = \{c\}$. In this case, c is not split but the transition (c, c') is redirected to point directly to the configuration \tilde{c}' for which $\tilde{z}' \sqsubseteq z$ (Line 8).

In case (1), the least upper bounds are $z[\psi \mapsto +]$ and $z[\psi \mapsto -]$, each of which represents the intersection of the underlying states of z with the states satisfying ψ or $\neg\psi$, resp., meaning z is split by ψ . In this case, identified by the condition $c \notin C''$ in Line 9, c is split into the two new configurations collected in C'' . These are added to the configurations of the corresponding player (Line 10) and c becomes a $C_{\frac{1}{2}}$ configuration (Line 10) pointing via focus-transitions to the new configurations (Line 11). Furthermore, the outgoing transitions of c are redirected to the new configurations

as follows. First, instead of the transition (c, c') , each of the new configurations points directly to the focus-transition target c' that “created” it, i.e., whose least upper bound (intersection) w.r.t. z it represents (Line 8). Additionally, the outgoing transitions of c pointing to a target different than c' are redirected (by doubling them) to leave the new configurations (Line 11 combined with Line 8). Finally, if c is an initial configuration, it is replaced by the new ones (Line 12). In Fig. 2 ‘splitting junction-configuration’ (based on case (1)) takes place from (e) into (f).

Focusing may-transitions (Line 13). Here, the purpose is to propagate a previous split to the incoming may-transitions of the split configuration. Therefore, a may-transition $(c, c') \in R^+$ with $c' \in C_{\frac{1}{2}}$ is determined. Such a transition models a hypertransition. It is redirected (by doubling it) such that it points directly to the focus-transition targets of c' , i.e. to the configurations to which c' was ‘split’ earlier. The determined transition is such that $\omega(c') \notin \{(tt,tt), (ff,ff), (tt,ff)\}$, ensuring that these focus-transition targets were not removed during simplification. The new may-transitions also become possible not-may-transitions, since they might be overly approximated. Furthermore, they are also added as possible must-transitions. Note that unlike the removal of may-transitions pointing to c' , (possible) must-transitions that point to c' (if exist) remain intact, since hypertransitions are needed in the case of must transitions to increase expressiveness. In Fig. 2 ‘focusing may-transition’ is performed from (g) into (h).

Ascertaining may-transitions (Line 15). A possible not-may-transition $((z, q), (z', q')) \in R^{+?}$ is determined and it is checked if it is overly approximated. This is done by a theorem prover call of **Unsatisfiable** $(\psi_z \wedge \text{pre}(\psi_{z'}))$. If this call is successful, meaning the $\exists\exists$ condition does not hold w.r.t. ψ_z and $\psi_{z'}$, the transition is removed as a may-transition. Otherwise, it is only removed from $R^{+?}$. Note that here and in the next scenario an unsatisfiability call is made instead of a satisfiability call in order to remain sound if incomplete satisfiability checks are applied.

Ascertaining must-transitions (Line 18). A possible must-transition $((z, q), (z', q')) \in R^{-?}$ is determined and it is checked if it is a real must transition. This is done by a theorem prover call of **Unsatisfiable** $(\psi_z \wedge \neg \text{pre}(\psi_{z'}))$. If this call is successful, meaning the $\forall\exists$ condition holds w.r.t. ψ_z and $\psi_{z'}$, the transition is added as a real must-transition and all (possible) must-transitions that have the same source but a less precise target are removed, since their existence does not increase precision. Otherwise, the transition and all possible must-transitions that have the same source but a more precise target are removed (they cannot become real must-transitions). The less (resp. more) precise targets are given by the configurations that are backwards (resp. forwards) reachable from (z', q') via focus-transitions. This is justified by the invariant that the abstract state of the target of a focus-transition is always finer, i.e. more precise, than the abstract state of its source. In Fig. 2 ‘ascertaining must-transition’ is performed from (m) into (n).

Ascertaining initial configuration (Line 21). Splitting of configurations might result in multiple initial configurations. However, recall that initial configurations are always in $C_0 \cup C_1$ and thus their abstract states are disjoint. This ensures that only one of them abstracts the concrete initial configuration, and the rest are merely overly approximated. Thus an initial configuration $(z, q) \in C^i$ is determined and it is checked if it contains the concrete initial state. This is done by a theorem prover call **Satisfiable** $(p^i \wedge \psi_z)$.

If successful, C^i becomes $\{(z, q)\}$ and z is added to Z_{sat} (since its satisfiability is ensured). In Fig. 2 ‘ascertaining initial configuration’ is performed from (d) into (e).

Checking satisfiability of abstract states (Line 23). An abstract state $z \in Z$ is determined and its (un)satisfiability is checked. This is done by a theorem prover call $\text{Satisfiable}(\psi_z)$, resp. $\text{Unsatisfiable}(\psi_z)$. If the call is successful, z is added to Z_{sat} , resp. to Z_{unsat} . Both theorem prover calls are necessary for soundness if incomplete satisfiability checks are applied. In Fig. 2 ‘checking satisfiability of abstract states’ is performed from (i) into (j) and from (k) into (l).

Properties of the algorithm. Satisfiability checks are said to be *sound* if $\text{Satisfiable}(\psi)$ implies that ψ is satisfiable, i.e. $\llbracket \psi \rrbracket \neq \emptyset$, and if $\text{Unsatisfiable}(\psi)$ implies that ψ is not satisfiable, i.e. $\llbracket \psi \rrbracket = \emptyset$. Satisfiability checks are *complete* if the reverse implications of the above constraints hold. Let $\mathcal{O} = \{P3, \dots, P7\} \cup \{V1, \dots, V7\}$ denote the execution lines of PropertyCheck, resp. of Validity, in which simplifications of the game structure as well as (in)validity determinations are made. In the following, PropertyCheck also denotes a more liberal version of it where the lines from \mathcal{O} are not always applied after every refinement step as long as (in)validity determinations are applied infinitely often (more precisely, the bundle of the three Lines V1, V2, V5 are infinitely often calculated after a refinement step). This means that more than one refinement step is calculated at once. We have that our algorithm is correct and no validity information is lost during an execution (even if the more liberal version is used):

Theorem 1 (Soundness). *Let satisfiability checks be sound. If $\text{PropertyCheck}(A, T)$ based on any heuristic returns tt (resp. ff), then $T \models A$ (resp. $T \not\models A$) holds.*

Theorem 2 (Incremental). *Suppose satisfiability checks are sound, P is a property-game obtained during the execution of $\text{PropertyCheck}(A, T)$, and P is valid (resp. invalid) in $c \in C$. Then the execution of any line of PropertyCheck or Validity yields a property-game that is valid (resp. invalid) in c or that does not contain c anymore.*

The algorithm is relatively complete for least fixpoint free formulas (and is often also successful for formulas containing least fixpoints). Note that this statement is not implied by the relative completeness of generalized Kripke modal transition systems, since not all hypertransitions are calculated.

Theorem 3 (Relative completeness). *Suppose satisfiability checks are sound and complete and \mathcal{L} can describe every subset of S . If the acceptance function of A maps always to zero (i.e. a least fixpoint free μ -calculus formula is encoded) and $T \models A$, then any heuristic applied for the first, say n , refinement determination steps, can be extended to a (not necessarily computable) one such that $\text{PropertyCheck}(A, T)$ returns tt.*

Theorem 3 does not hold if we restrict to computable refinement heuristics, since otherwise the halting problem would be decidable. Furthermore, Theorem 3 does not hold for automata with arbitrary acceptance function, since the underlying class of abstract models is not expressive enough. To handle arbitrary functions, fairness constraints, as in [6,8], are needed.

Remark 2. Previous CEGAR-algorithms, including ours [10], usually need less refinement steps than our new algorithm, since it has a very fundamental laziness. Nevertheless, our algorithm can mimic the refinement steps of the other algorithms without increasing its computation time by calculating all refinements made by the other algorithms in a single step before calling *Validity*. We expect our algorithm to be in general faster than the others, since we can use improved heuristics that avoid the expensive cost of refinement-calculations by restricting to the relevant calculations.

4 Conclusion

We presented a new CEGAR-based algorithm for μ -calculus verification, which is based on the lazy abstraction technique. We obtained the high level of laziness by developing a new philosophy of a refinement step, namely *state focusing*: The to be split configuration is not removed and is, e.g., used to model hypertransitions. Our algorithm avoids state explosion and, at the same time, remains complete for least fixpoint free formulas. The heuristics presented in [10] can be straightforwardly adapted to our setting. Determination of heuristics that better support the finer lazy abstraction approach of our new algorithm, and a prototype implementation, are topics for future work.

References

1. Ball, T., Kupferman, O.: An abstraction-refinement framework for multi-agent systems. In: LICS, pp. 379–388 (2006)
2. Ball, T., Kupferman, O., Sagiv, M.: Leaping loops in the presence of abstraction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 491–503. Springer, Heidelberg (2007)
3. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 67–81. Springer, Heidelberg (2005)
4. Ball, T., Podolski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
6. Dams, D., Namjoshi, K.S.: The existence of finite abstractions for branching time model checking. In: LICS, pp. 335–344 (2004)
7. Fecher, H., Huth, M.: Model checking for action abstraction. In: VMCAI. LNCS, vol. 4905, pp. 112–126 (2008)
8. Fecher, H., Huth, M.: Ranked predicate abstraction for branching time: Complete, incremental, and precise. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 322–336. Springer, Heidelberg (2006)
9. Fecher, H., Huth, M., Schmidt, H., Schönborn, J.: Refinement sensitive formal semantics of state machines with persistent choice. In: AVoCS (2007) (will appear in ENiTCS)
10. Fecher, H., Shoham, S.: Local abstraction-refinement for the mu-calculus. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 4–23. Springer, Heidelberg (2007)
11. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)

12. 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)
13. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: When not losing is better than winning: Abstraction and refinement for the full μ -calculus. *Information and Computation* 205(8), 1130–1148 (2007)
14. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: Don't know in the μ -calculus. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 233–249. Springer, Heidelberg (2005)
15. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
17. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
18. Jurdzinski, M.: Deciding the winner in parity games is in $UP \cap co-UP$. *Inf. Process. Lett.* 68(3), 119–124 (1998)
19. Kozen, D.: Results on the propositional μ -calculus. *Theor. Comput. Sci.* 27, 333–354 (1983)
20. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210 (1988)
21. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: LICS, pp. 108–117 (1990)
22. Namjoshi, K.S.: Abstraction for branching time properties. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 288–300. Springer, Heidelberg (2003)
23. Pardo, A., Hachtel, G.D.: Incremental CTL model checking using BDD subsetting. In: DAC, pp. 457–462 (1998)
24. Shoham, S., Grumberg, O.: Monotonic abstraction-refinement for CTL. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 546–560. Springer, Heidelberg (2004)
25. Shoham, S., Grumberg, O.: 3-Valued Abstraction: More Precision at Less Cost. In: LICS, pp. 399–410 (2006)

Efficient Modeling of Concurrent Systems in BMC

Malay K. Ganai and Aarti Gupta

NEC Labs America, Princeton, NJ, USA

Abstract. We present an efficient method for modeling multi-threaded concurrent systems with shared variables and locks in Bounded Model Checking (BMC), and use it to improve the detection of safety properties such as data races. Previous approaches based on synchronous modeling of interleaving semantics do not scale up well due to the inherent asynchronism in those models. Instead, in our approach, we first create independent (uncoupled) models for each individual thread in the system, then explicitly add additional synchronization variables and constraints, incrementally, and only where such synchronization is needed to guarantee the (chosen) concurrency semantics (based on sequential consistency). We describe our modeling in detail and report verification results to demonstrate the efficacy of our approach on a complex case study.

1 Introduction

The growth of cheap and ubiquitous multi-processor systems and concurrent library support are making concurrency programming very attractive. On the other hand, verification of concurrent systems remains a daunting task especially due to complex and unexpected interactions between asynchronous threads, and various architecture-specific memory consistency models [1]. In this work, we focus on concurrency semantics based on sequential consistency [2]. In this semantics, the observer has a view of only the local history of the individual threads where the operations respect the program order. Further, all the memory operations exhibit a common *total order* that respect the *program order* and has the *read value property*, i.e., the read of a variable returns the last write on the same variable in that total order. In the presence of synchronization primitives such as locks/unlocks, the concurrency semantics also respects the mutual exclusion of operations that are guarded by matching locks. Sequential consistency is the most commonly used concurrency semantics for software development due to ease of programming, especially to obtain race-free, i.e, correctly synchronized threads. A *data race* corresponds to a global state where two different threads access the same shared variable, and at least one of them is a write.

Bounded Model Checking (BMC) [3] has been successfully applied to verify real-world designs. Strengths of BMC are manifold: First, expensive quantification used in symbolic model checking [4] is avoided. Second, reachable states are not stored, avoiding blow-up of intermediate state representation. Third, modern SAT solvers are able to search through the relevant paths of the problem even though the paths get longer with the each BMC unrolling. We focus on verifying concurrent systems through efficient modeling in BMC.

1.1 Related Work

We discuss various model checking efforts, both explicit and symbolic, for verifying concurrent systems with shared memory. The general problem of verifying a concurrent system with even two threads with unbounded stacks is undecidable [5]. In practice, these verification efforts use *incomplete* methods, or *imprecise* models, or sometimes both, to address the scalability of the problem. The verification model is typically obtained by composing individual thread models using interleaving semantics, and model checkers are applied to systematically explore the global state space. Model checkers such as Verisoft [6], Zing [7] explore states and transitions of the concurrent system using explicit enumeration. Although several state space reduction techniques based on partial order methods [8] and transactions-based methods [9, 10, 11, 12] have been proposed, these techniques do not scale well due to both state explosion and explicit enumeration.

Symbolic model checkers such as BDD-based SMV [4], and SAT-based Bounded Model Checking (BMC) [3] use symbolic representation and traversal of state space, and have been shown to be effective for verifying synchronous hardware designs. There have been some efforts [13, 14, 15, 16] to combine symbolic model checking with the above mentioned state-reduction methods for verifying concurrent software. However, they still suffer from lack of scalability. To overcome this limitation, some researchers have employed sound abstraction [7] with bounded number of context switches [17], while some others have used finite-state model [15, 18] or Boolean program abstractions with bounded depth analysis [19]. This is also combined with a bounded number of context switches known *a priori* [15] or a proof-guided method to discover them [18]. To the best of our knowledge, all these model checking methods use synchronous modeling of interleaving semantics. As we see later, our focus is to move away from such synchronous modeling in BMC in order to obtain significant reduction in the size of the BMC instances.

Another development is the growing popularity of Satisfiability-Modulo Theory (SMT)-solvers such as [20]. Due to their support for richer expressive theories beyond Boolean logic, and several latest advancements, SMT-based methods are providing more scalable alternatives than BDD-based or SAT-based methods. In SMT-based BMC, a BMC problem is translated typically into a quantifier-free formula in a decidable subset of first order logic, instead of translating it into a propositional formula, and the formula is then checked for satisfiability using an SMT solver. Specifically, with several acceleration techniques, SMT-based BMC has been shown [21] to scale better than SAT-based BMC for finding bugs.

There have been parallel efforts [22, 23, 24] to detect bugs for weaker memory models. As shown in [25], one can check these models using axiomatic memory style specifications combined with constraint solvers. Note, though these methods support various memory models, they check for bugs using given test programs. There has been no effort so far, to our knowledge, to integrate such specifications in a model checking framework that does not require test programs. There have been other efforts using static analysis [26, 27] to detect static races. Unlike these methods, our goal is to find true bugs and to not report false warnings.

1.2 Our Approach: Overview

We present an efficient modeling for multi-threaded concurrent systems with shared variables and locks in BMC. We consider C threads under the assumption of a bounded heap and bounded stack. Using this modeling, we augment SMT-based BMC to detect violations of safety properties such as data races. *The main novelty of our approach is that it provides a sound and complete modeling with respect to the considered concurrency semantics, without the expensive synchronous modeling of interleaving semantics.* Specifically, we do not introduce *wait-cycles* to model interleaving of the individual threads, and do not model a scheduler explicitly. As we see later, these wait-cycles are detrimental to the performance of BMC. Instead, we first create *independent* (decoupled) individual thread models, and add memory consistency constraints *lazily, incrementally, and on-the-fly* during BMC unrolling to capture the considered concurrency semantics. Our modeling preserves with respect to a property the set of all possible executions up to a bounded depth that satisfy the sequential consistency and synchronization semantics, without requiring an *a priori* bound on the number of context switches. We have implemented our techniques in a prototype SMT-based BMC framework, and demonstrate its effectiveness through controlled experiments on a complex concurrency benchmark. For experiments, we contrast our *lazy modeling* approach with an *eager modeling* [13,14,15,16] of the concurrent system, i.e., a monolithic model synchronously composed with interleaving semantics (and possibly, with state-reduction constraints) enforced by an explicit scheduler, capturing all concurrent behaviors of the system eagerly.

1.3 Our Contributions

The main idea of our modeling paradigm for concurrent systems is to move away from expensive modeling based on synchronous interleaving semantics. We focus primarily on reducing the size of the BMC problem instances to enable deeper search within the limited resources, both time and memory. Features and merits of our approach are:

1. *Lazy modeling constraints:* By adding the constraints *lazily*, i.e., as needed for a bounded depth analysis, as opposed to adding them *eagerly*, we reduce the BMC problem size at that depth. The size of these concurrency-modeling constraints depends *quadratically* on the number of shared memory accesses at any given BMC depth in the worst case. Since the analysis depth of BMC bounds the number of shared memory accesses, these constraints are typically smaller than the model with constraints added eagerly, in practice.
2. *No wait-cycle:* We do not allow local wait cycles, i.e., there are no self-loops in read/write blocks with shared accesses. This enables us to obtain a reduced set of statically reachable blocks at a given BMC depth d , which dramatically reduces the set of pair-wise concurrency constraints that we need to add to the BMC problem.
3. *Deeper analysis:* For a given BMC depth D and n concurrent threads, we guarantee finding a witness trace (if it exists), i.e., a sequence of global interleaved transitions, of length $\leq n \cdot D$, where the number of local thread transitions is at most D . In contrast, an eager modeling approach using BMC [14], an unrolling depth of $n \cdot D$ is needed for such a guarantee. Thus, we gain in memory use by a factor of n .

4. *Using static analysis:* We use property preserving model transformations such as path/loop balancing, and context-sensitive control state reachability to reduce the set of blocks that are statically reachable at a given depth. Again, this potentially reduces the lazy modeling constraints. We also use *lockset* [10,9,14] analysis to reduce the set of constraints, by statically identifying which block pairs (with shared accesses) are simultaneously unreachable.
5. *SMT-based BMC:* We use an SMT solver instead of a traditional SAT solver, to exploit the richer expressiveness, in contrast to bit-blasting. We effectively capture the *exclusivity* of the pair-wise constraints, i.e., for a chosen shared access pair, other pairs with a common access are implied invalid immediately.

Outline: We provide a short background in Section 2, motivation in Section 3, illustrate our basic approach with an example in Section 4, formal description of our modeling in Section 5, correctness theorems and discussion on size complexity in Section 6, BMC size-reduction techniques in Section 7, followed by experiments in Section 8, and conclusions in Section 9.

2 Preliminaries

2.1 Concurrent System: Model and Semantics

We consider a concurrent system comprising a finite number of deterministic bounded-stack threads communicating with shared variables, some of which are used as synchronization objects such as locks. Each thread has a finite set of control states and can be modeled as an extended finite state machine (EFSM). An EFSM model is a 5-tuple (s_0, C, I, D, T) where, s_0 is an initial state, C is a set of control states (or blocks), I is a set of inputs, D is a set of data state variables (with possibly infinite range), and T is a set of 4-tuple (c, g, u, c') transitions where $c, c' \in C$, g is a Boolean-valued enabling condition (or *guard*) on state and input variables, u is an update function on state and input variables.

We define a concurrent system model \mathcal{CS} as a 4-tuple $(\mathcal{M}, \mathcal{V}, \mathcal{T}, s_0)$, where \mathcal{M} denotes a finite set of EFSM models, i.e., $\mathcal{M} = \{M_1, \dots, M_n\}$ with $M_i = (s_{0i}, C_i, I_i, D_i \cup \mathcal{V}, T_i)$, \mathcal{V} denotes a finite set of shared(or global) variables i.e., $\mathcal{V} = \{g_1, \dots, g_m\}$, \mathcal{T} denotes a finite set of transitions, i.e., $\mathcal{T} = \bigcup_i T_i$, s_0 denotes the initial global state. Note, for $i \neq j$, $C_i \cap C_j = \emptyset$, $I_i \cap I_j = \emptyset$, $D_i \cap D_j = \emptyset$, and $T_i \cap T_j = \emptyset$, i.e., except for shared variables \mathcal{V} , each M_i is disjoint. Let VL_i denote a set of tuples values for local data state variables in D_i , and VG denote a set of tuple values for shared variables in \mathcal{V} . A global state s of \mathcal{CS} is a tuple $(s_1, \dots, s_n, v) \in \mathcal{S} = (C_1 \times VL_1) \cdots \times (C_n \times VL_n) \times VG$ where $s_i \in C_i \times VL_i$ and $v \in VG$ denotes the values of the shared global variables. Note, s_i denotes the local state tuple (c_i, x_i) where $c_i \in C_i$ represents the local control state, and $x_i \in VL_i$ represents the local data state. A global transition system for \mathcal{CS} is an interleaved composition of the individual EFSM models, M_i . Each global transition consists of firing of a local transition $t_i = (a_i, g_i, u_i, b_i) \in \mathcal{T}$. In a given global state s , the local transition t_i of model M_i is said to be *scheduled* if $c_i = a_i$, where c_i is the local control state component of s_i . Further, if enabling predicate g_i evaluates to true in s , we say that t_i is

enabled. Note, in general, more than one local transition of model M_i can be scheduled but exactly one of them can be enabled (M_i is a deterministic EFSM). The set of all transitions that are enabled in a state s is denoted by $enabled(s)$.

We can obtain a synchronous execution model for \mathcal{CS} by defining a scheduling function $E : \mathcal{M} \times \mathcal{S} \mapsto \{0, 1\}$ such that t is said to be executed at global state s , iff $t \in enabled(s) \cap T_i$ and $E(M_i, s) = 1$. Note, in interleaved semantics, at most one enabled transition can be executed at a global state s . In this synchronous execution model, each thread local state s_i (with shared access) has a wait-cycle, i.e., a self-loop to allow all possible interleavings.

Semantics of a sequentially consistent memory model [2, 25] are as follows:

- *Program Order Rule*: Shared accesses, i.e. read/write to shared variables, should follow individual program thread semantics.
- *Total Order Rule*: Shared accesses across all threads should have a total order.
- *Read Value Rule*: A read access of a shared variable should observe the effect of the last write access to the same variable in the total order.
- *Mutual Exclusion Rule*: Shared accesses in matched locks/unlock operations should be mutually exclusive.

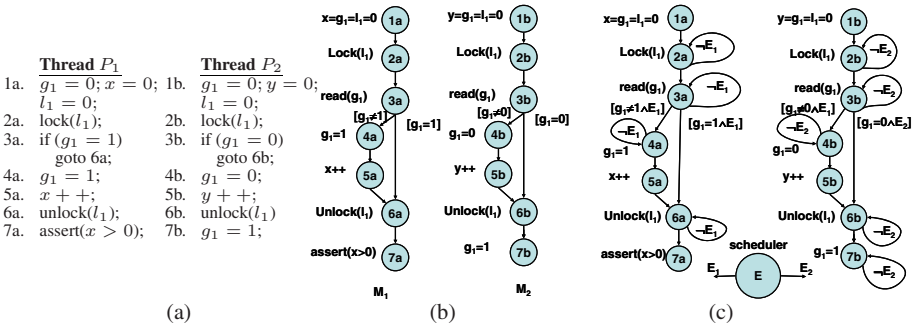


Fig. 1. (a) Concurrent system with threads P_1 and P_2 with local variables x and y respectively, communicating with lock l_1 and shared variable g_1 . (b) CFG of P_1 and P_2 , and (c) CFG of concurrent system with scheduler E .

Example: We illustrate a concurrent system comprising threads P_1 and P_2 with local variables x and y , respectively, interacting through lock l_1 and shared variable g_1 , as shown in Figure 1(a). Each numbered statement is *atomic*, i.e., it cannot be interrupted. EFSM models M_1 and M_2 of the two threads P_1 and P_2 are shown as control flow graphs (CFG) in Figure 1(b). Note, M_1 is the tuple $(c_{01}, C_1, I_1, D_1, T_1)$ with $c_{01} = 1a$, $C_1 = \{1a, \dots, 7a\}$, $I_1 = \{\}$, $D_1 = \{x\} \cup \{g_1, l_1\}$. The transitions are shown by directed edges with enabling predicates (if not a tautology) shown in square brackets and update functions are shown on the side of each control state. The model M_2 is similarly defined. An synchronously interleaved model for the concurrent system with threads P_1 and P_2 , i.e., $\mathcal{CS} = (\{M_1, M_2\}, \{g_1, l_1\}, \{T_1, T_2\}, ((1a, x), (1b, y), (g_1, l_1)))$, with global shared variable g_1 and lock variable l_1 , and a scheduler E is shown in Figure 1(c). It is obtained by inserting a wait-cycle, i.e., a self-loop at each control state

of model M_i and associating the edge with a Boolean guard E_i such that $E_i = 1$ iff $E(M_i, s) = 1$. To understand the need for such wait-cycles, consider a global state s with thread control states at $2a$ and $6b$, respectively. To explore both the interleaving $2a \rightarrow 3a$ and $6b \rightarrow 7b$ from s , each thread needs to wait when the other makes the transition. By noting that the transitions at control states $5a$, $7a$, and $5b$ correspond to non-shared memory accesses, one can remove the self-loops at these control states. In general, however, all self-loops can not be removed.

2.2 Building EFSMs from C Threads

For brevity, we highlight the essentials in building a thread model (EFSM) from a C thread (using the F-Soft framework [28]) under the assumption of a bounded heap and a bounded stack. First we obtain a simplified CFG by creating an explicit memory model for (finite) data structures and heap memory, where indirect memory accesses through pointers are converted to direct accesses by using auxiliary variables. We model arrays and pointer arithmetic precisely using a sound pointer analysis. We model loops in the CFG without unrolling them. We handle non-recursive procedures by creating a single copy (i.e., not inlining) and using extra variables to encode the call/return sites. Recursive procedures are inlined up to some user-chosen depth. We perform merging of control nodes in CFG involving parallel assignments to local variables into a basic block, where possible, to reduce the number of such blocks. We, however, keep each shared access as a separate block to allow context-switches.

From the simplified CFG, we build an EFSM with each basic block identified with a unique *id* value, and a control state variable PC denoting the current block *id*. We construct a symbolic transition relation for PC , that represent the guarded transitions between the basic blocks. For each data variable, we add an update transition relation based on the expressions assigned to the variable in various basic blocks in the CFG. We use *Boolean* expressions and *arithmetic* expressions to represent the guarded and update transition functions, respectively.

2.3 Control State Reachability (CSR) and CSR-Based BMC Simplification

Control state reachability (CSR) analysis is a breadth-first traversal of the CFG (corresponding to an EFSM model), where a control state b is one step reachable from a iff there is a transition edge $a \rightarrow b$. At a given sequential depth d , let $R(d)$ represent the set of control states that can be reached *statically*, i.e., ignoring the guards, in one step from the states in $R(d - 1)$, with $R(0) = s_0$. Computing *CSR* for the CFG of M_1 shown in Figure 1(b), we obtain the set $R(d)$ for the first six depths as follows: $R(0) = \{1a\}$, $R(1) = \{2a\}$, $R(2) = \{3a\}$, $R(3) = \{4a, 6a\}$, $R(4) = \{5a, 7a\}$, $R(5) = \{6a\}$, $R(6) = \{7a\}$. For some d , if $R(d - 1) \neq R(d) = R(d + 1)$, we say that the *CSR saturates* at depth d , and $R(t) = R(d)$ for $t > d$.

CSR can be used to reduce size of a BMC instance significantly [21]. Basically, if a control state $r \notin R(d)$, then the unrolled transition relation of variables that depend on r can be simplified. We define a Boolean predicate $B_r \equiv (PC = r)$, where PC is the program counter that tracks the current control state. Let v^d denote the unrolled variable v at depth d during BMC unrolling. Consider the thread model M_1 , where

the next state of variable g_1 is defined as $next(g_1) = B_{1a} ? 0 : B_{4a} ? 1 : g_1$ (using C language notation $?:$ for *cascaded if-then-else*). At depths $k \notin \{0, 3\}$, $B_{1a}^k = B_{4a}^k = 0$ since $1a, 4a \notin R(k)$. Using this unreachability control state information, we can *hash* the expression representation for g_1^{k+1} to the existing expression g_1^k , i.e., $g_1^{k+1} = g_1^k$. This hashing, i.e., reusing of expressions, considerably reduces the size of the logic formula, i.e., the BMC instance.

3 Motivation: Why Wait-Cycles Are Bad?

The scope of CSR-based BMC simplification is reduced considerably by a large cardinality of the set $R(d)$, i.e., $|R(d)|$, and hence, the performance of BMC also gets affected adversely. In general, re-converging paths of different lengths and different loop lengths are mainly responsible for enlarging set R , due to inclusion of all control states in a loop, and ultimately leading to saturation [21]. For example, computing CSR on the concurrent synchronous model (Figure 1(c)), we obtain $R(d)$ as follows:

$$R(0) = \{1a, 1b\}, R(1) = \{2a, 2b\}, R(2) = \{2a, 3a, 2b, 3b\}, R(3) = \{2a, 3a, 4a, 6a, 2b, 3b, 4b, 6b\}, \\ R(4) = \{2a, 3a, 4a, 5a, 6a, 7a, 2b, 3b, 4b, 5b, 6b, 7b\}, R(t) = R(4) \text{ for } t > 4 \text{ (Saturates at 4)}$$

Clearly, saturation is inevitable due to the presence of self-loops. At $t \geq 4$, the BMC unrolled transition relation cannot be simplified further using unreachable control states, i.e., not in $R(t)$. Thus, the scope of reusing the expression for next state logic expression is also reduced heavily. In general, saturation can also be caused by program loops. To overcome that we use a *Balancing Re-convergence* strategy [21] effectively to balance the lengths of the re-convergent paths and loops by inserting *NOP* states. An *NOP* state does not change the transition relation of any variable. However, this approach does not work well in the presence of self-loops.

In our experience, synchronous models with self-loops for modeling interleaving semantics are not directly suitable for verifying concurrent systems using BMC. Instead, we propose a modeling paradigm that eliminates self-loops with the goal of reducing the size of BMC instances. However, there are many challenges in doing so.

- We would like to have soundness and completeness, i.e., neither to miss true witnesses nor to report spurious witnesses (up to some bounded depth). We do so by decoupling the individual thread models, and add the required memory consistency constraints on-the-fly to the unrolled BMC instances. This allows us to exploit advances in SMT-based BMC, without the expensive modeling of interleaving semantics. Note that for bounded analysis the number of shared memory accesses are also bounded, and therefore, these constraints are typically smaller than the model with constraints added eagerly, in practice.
- We also would like to formulate and solve iterative BMC problems incrementally, and integrate seamlessly state-of-the-art advancements in static analysis and BMC. We achieve our goals in a lazy modeling paradigm that simultaneously facilitates the use of several static techniques such as context-sensitive CSR and lockset analysis [10, 9, 14] and model transformations [21] to improve BMC significantly.

4 Lazy Modeling Paradigm: Overview

We illustrate the main idea of our modeling using an example in Section 4.1 and highlight novelties in our approach in Section 1.3. A formal exposition of the modeling and its soundness is provided in Section 5.

4.1 Basic Approach

As a first step in our modeling, we construct abstract and independent (i.e. decoupled) thread models LM_1 and LM_2 corresponding to the threads P_1 and P_2 as shown in Figure 2. We introduce atomic thread-specific procedures `read_sync` and `write_sync` before and after every shared access. In Figure 2, the control states r_i and w_i correspond to calls to procedures `read_synci` and `write_synci`, respectively. We also introduce global variables TK , CS_1 , and CS_2 described later. For each thread, we make the global variables *localized* by renaming. As shown in Figure 2, for P_1 (and similarly for P_2), we rename g_1 , l_1 , TK , CS_1 , and CS_2 to local variables g_{11} , l_{11} , TK_1 , CS_{11} , and CS_{21} , respectively. The localized shared variables get non-deterministic (ND) values in the control state r_i . (Refer Section 5.1 for detailed instrumentation.) The models LM_i obtained after annotations are *independent* since the update transition relation for each variable now depends only on the local state variables. Note, there are no self-loops in these models. However, due to ND read values for shared variables in r_i control state, these models have additional behaviors, which we eliminate by adding concurrency constraints as described below.

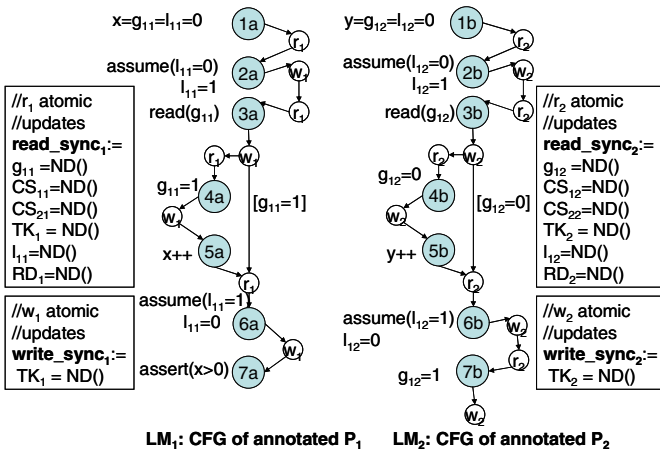


Fig. 2. CFG of threads P_1 and P_2 with annotations

We unroll each model LM_i independently during BMC (i.e. with possibly different unroll depths). To each BMC instance, we add concurrency constraints on-the-fly between each pair of control states with accesses to shared variables, that are statically reachable in unrolled CFG at the corresponding thread-specific depths. For sequential

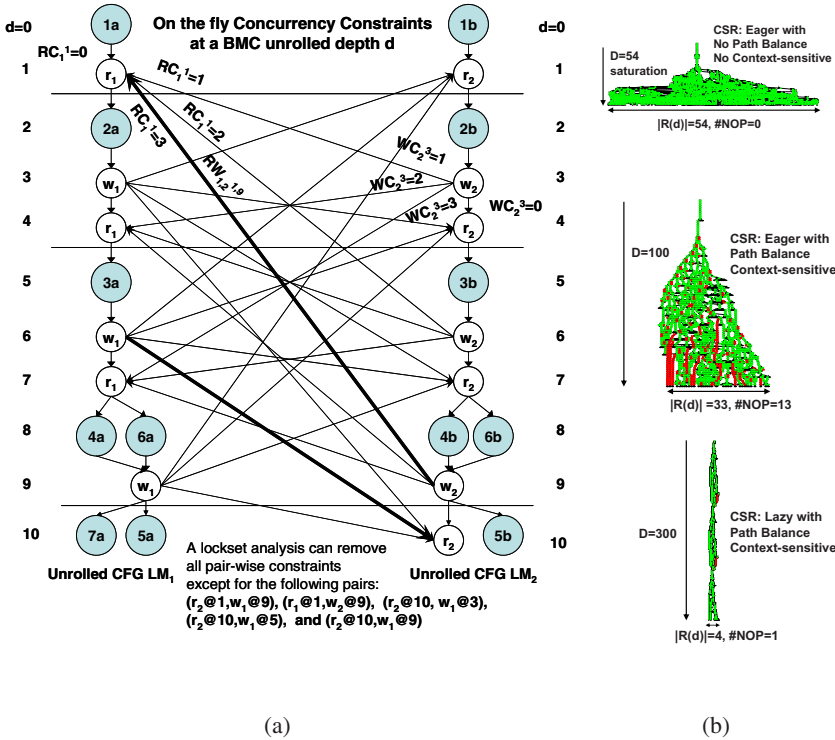


Fig. 3. (a) Unrolled CFG of thread models LM_1 and LM_2 with concurrency constraints added on-the-fly during BMC unrolling. The dark arrows show the token passing events (i.e., context switches) leading to a data race between source lines 3a and 7b (in Figure 1(a)). (b) CSR graphs with path balancing (PB), context-sensitive (CXT) for eager/lazy on a thread-model (Green/Red dots denote non-NOP/NOP blocks, resp.)

consistency, these constraints allow sufficient context-switching to maintain the *read value property*, and sequentialize the context-switches to enforce a common *total order*. Note, in addition to these constraints, a BMC instance comprises transition relation of all thread models and the property constraints. The transition relation of each thread model ensures that memory accesses within the thread follow the *program order*.

To capture context-switching events, we specifically added a Boolean shared variable referred to as *token* (TK). The semantics of a token asserted by a thread is equivalent to a guarantee that all *visible* operations, i.e., shared memory accesses, issued so far have been *committed* for other threads to see. Initially, only one thread (chosen non-deterministically) is allowed to assert its token. To track the sequentiality of the global execution and maintain *total order*, we also add two global clock variables (one per thread) i.e., CS_1 and CS_2 to timestamp [2] the token passing events. The pair-wise constraints, added between shared access states, allow *passing* of the token. Whenever the token is passed from thread LM_i (post-access shared state) to LM_j (pre-access

shared state) the concurrency constraints ensure that each localized shared variable of LM_j gets the current state value of the corresponding localized shared variable of LM_i , and the clock variables get synchronized. We call these pair-wise constraints as *Read-Write Synchronization Constraints* as described in Section 5.2 with more details.

We illustrate the concurrency constraints added in Figure 3(a). Let $c@k$ denote the control state c reached statically at depth k . Computing CSR, we obtain $R(0) = \{1a@0, 1b@0\}$, $R(1) = \{r_1@1, r_2@1\}$, $R(2) = \{2a@2, 2b@2\}$ and so on. The concurrency constraints added between a pair $(r_i@k, w_j@h)$ are shown as an arrow from w_j (of LM_j) at depth h to r_i (of LM_i) at depth k . Note, r_i corresponds to pre-access shared state of LM_i , and w_j corresponds to post-access shared state of LM_j . These constraints also capture the exclusivity of the pair, i.e., constraint for pair $(r_i@k, w_j@h)$, excludes other pairs $(r_i@k, *)$ and $(*, w_j@h)$. For BMC at depth $d < 3$ we do not add any pair-wise constraints. For BMC at $d = 3$, we add concurrency constraints corresponding to pairs $(r_1@1, w_2@3)$ and $(r_2@1, w_1@3)$. For BMC at $d = 4$, we add concurrency constraints corresponding to pairs $(r_1@1, w_2@3)$, $(r_2@1, w_1@3)$, $(r_1@4, w_2@3)$, and $(r_2@4, w_1@3)$. Note, in incremental formulation of BMC, we only need to add the constraints corresponding to the last two pairs. In general, the constraints grow quadratically with the analysis depth (ref. Section 6). In addition, we use various static analyses and model transformations to reduce the set of pair-wise constraints added that are redundant (ref. Section 7). For example, using the lockset analysis [10,9,14], we can remove constraints for all pairs other than $(r_2@1, w_1@9)$, $(r_1@1, w_2@9)$, $(r_2@10, w_1@3)$, $(r_2@10, w_1@5)$ and $(r_2@10, w_1@9)$. This is because other pairs are not reachable simultaneously in the mutually exclusive region, protected by the lock.

In our approach, a data race condition is detected, if there exists a witness trace where a token passing event occurs between the pair $(r_i@k, w_j@h)$ with shared accesses on the same variable, with at least one access being a write. In Figure 3(a), we indicate the witness trace corresponding to the data race between source lines 3a and 7b (from Figure 1(c)) as a sequence of the token passing events highlighted in **bold arrows** between the pairs $(r_1@1, w_2@9)$ and $(r_2@10, w_1@5)$. Note, the control state pair $(r_2@10, w_1@5)$ corresponds to simultaneous accesses of shared variable g_1 with $r_2@10$ being a pre-write access state. We obtain the witness trace by unrolling each model in BMC up to depth 11. In a synchronously interleaved model (i.e., with wait-cycles), we would have obtained the trace at an unrolled depth of $11 + 7 = 18$. Note, we sum the two depths, as thread P_2 has to wait (self-loop) after context switch to P_1 .

For a thread model (example described in Section 8), we compare the reachability graphs on the lazy and eager models, as shown in Figure 3(b), with and without using model transformation such as path/loop balancing (PB) [21] and context-sensitive CSR (CXT) [29]. Note, the width of the graph is proportional to $|R(d)|$. It is desirable that the width is small for greater BMC simplification. We observe that path/loop balancing is more effective on our lazy models (i.e., without wait-cycles) as $R(d)$ is reduced significantly compared to eager models (i.e., with wait-cycles).

5 Lazy Modeling of Concurrent Systems

We present details of our modeling in Sections 5.1 and 5.2

5.1 Sound Abstraction: Independent Thread Models

We perform source-to-source transformations to directly use a model builder (F-Soft [28]) for sequential programs and obtain sound abstraction.

Token: We introduce a global Boolean variable, a token TK , to signify that the thread with the token can execute a shared access operation and commit its current shared state to be *visible* to the future transitions. Initially, only one thread, chosen non-deterministically, is allowed to assert TK . Later, this token is *passed*, from one thread to another, i.e., de-asserted in one thread and asserted by the other thread, respectively.

Logical Clock: To obtain a total ordering on token *passing* events, we use the concept of *logical clocks* and *timestamp* [2]. We add a global clock variable CS_i for each thread P_i , so that the tuple $(CS_1 \cdots CS_n)$ represents the logical clock. These variables are initialized to 0. Whenever a token TK is acquired by a thread P_i , CS_i is incremented by 1 in P_i . The variable CS_i keeps track of the number of occurrences of token passing events wherein thread P_i acquires the token from another thread P_j , $j \neq i$.

Race Detector: We add a race detector local Boolean variable RD_i for each thread P_i . This variable is set to 1 (initially, 0) whenever a shared variable is accessed in thread P_i and written in another thread P_j while token is passed from P_j to P_i .

Localization: For each thread, we make the global variables *localized* by renaming.

Atomic Procedures: We add two atomic thread-specific procedures before and after every shared access, i.e., `read_sync` and `write_sync`, respectively. In the `read_sync` procedure, each *localized* shared and race detector variable get a non-deterministic value, (ND), while in the `write_sync` procedure only TK gets an ND value.

Synchronization primitives: Operations `lock(lk)` and `unlock(lk)` are modeled as atomic operations `assume(lk = 1)` and `assume(lk = 0)`, respectively. To maintain synchronization semantics, we only consider wait-free execution [15] where the acquisition of the same lock twice is disallowed in a row without an intermediate unlock. Note, this consideration is sufficient to find all data races.

5.2 Concurrency Constraints

Given independent abstract models, obtained as above, we add concurrency constraints *incrementally*, and *on-the-fly* to each BMC instance, in addition to the transition constraints of the individual thread models and property constraints. The concurrency constraints capture *inter-* and *intra-* thread dependencies due to interleavings, and thereby, eliminate additional behaviors in the models up to a bounded depth. Specifically, these constraints comprise (a) pair-wise, i.e., inter-model constraints (shown in Table 1), (b) single-threaded, i.e., intra-model constraints (shown in Table 2), and (c) global constraints (shown in Table 3). In the following, we use $R_i(d)$, $0 \leq d \leq D$, to denote the set of control states reachable at depth d for each thread model LM_i , for a given BMC bound D . (Note, we compute CSR on each of the models LM_i separately before starting BMC.) Also, we use x_i^k to denote the expression for the variable x in the unrolled model LM_i at depth k .

Table 1. Pair-wise concurrency constraints added in each BMC instance

<p>P1. Read-Write Synchronization Enabling Constraint: For every pair of <i>read_sync</i> control state in $LM_i, r_i \in R_i(k)$ and <i>write_sync</i> control state in $LM_j, j \neq i, w_j \in R_j(h)$, we introduce a Boolean variable RW_{ij}^{kh}, and add the following enabling constraint:</p> $RW_{ij}^{kh} \iff (B_{r_i}^k \wedge \neg TK_i^k \wedge B_{w_j}^h \wedge TK_j^h \wedge CS_{ii}^k = CS_{ij}^h) \quad (1)$ <p>If $RW_{ij}^{kh} = 1$, we say, the <i>token passing condition</i> is enabled.</p>
<p>P2. Read-Write Synchronization Exclusivity Constraint: Let RS_i^k define the set $\{RW_{ij}^{kh} \mid i \neq j, 0 \leq h \leq d\}$ for a <i>read_sync</i> control state of LM_i at depth k. To allow at most one <i>write_sync</i> (from a different thread) to match with this <i>read_sync</i>, we assign a unique id $a_j^h \neq 0$ to each element of RS_i^k. We add a new variable RC_i^k for the <i>read_sync</i> control state of LM_i at depth k, <i>require</i> that it takes value $a_j^h \neq 0$ iff $RW_{ij}^{kh} = 1$. Similarly, we introduce a new variable WC_j^h for the <i>write_sync</i> control state of LM_j at depth h, and <i>require</i> that it takes value $b_i^k \neq 0$ iff $RW_{ij}^{kh} = 1$. The constraints added are:</p> $RW_{ij}^{kh} \iff (RC_i^k = a_j^h), a_j^h \neq 0 \quad (2)$ $RW_{ij}^{kh} \iff (WC_j^h = b_i^k), b_i^k \neq 0 \quad (3)$ <p>Thus, if $RW_{ij}^{kh} = 1$, we require that both $RC_i^k \neq 0$ and $WC_j^h \neq 0$; and vice-versa.</p>
<p>P3. Read-Write Synchronization Update Constraint: For every RW_{ij}^{kh} variable introduced, we add the following update constraints:</p> $RW_{ij}^{kh} \implies \bigwedge_{p=1}^m g_{pi}^{k+1} = g_{pj}^h \quad (4)$ $RW_{ij}^{kh} \implies (TK_i^{k+1} \wedge \neg TK_j^{h+1}) \quad (5)$ $RW_{ij}^{kh} \implies (CS_{ii}^{k+1} = CS_{ii}^k + 1) \wedge \left(\bigwedge_{q=1, q \neq i}^n CS_{qi}^{k+1} = CS_{qj}^h \right) \quad (6)$
<p>P4. Data Race Detection Property Constraint: We define two predicates <i>will_access</i> and <i>just_written</i> statically, where $will_access(r_i, g) = 1$ iff shared variable g is accessed in the next local control state reachable from r_i, and $just_written(w_i, g) = 1$ iff shared variable g was written in the previous local control state reachable to w_i. We add the following race detection constraint only if $will_access(r_i, g) = 1$ and $just_written(w_j, g) = 1$:</p> $RW_{ij}^{kh} \implies RD_i^{k+1} \quad (7)$

We give the intuition and description for each of the following pair-wise (i.e., inter-model) constraints *P1-P4* added between unrolled models LM_i and LM_j at depths k and h , respectively as shown in Table [1](#).

P1. Read-Write Synchronization Enabling Constraint: The constraints (Eqn [1](#)) capture the enabling of *token passing* condition. This happens exactly when: a) thread model LM_i is in *read_sync* control state (i.e., pre-access shared state) at depth k and does not hold the token, b) thread model LM_j is in *write_sync* control state (i.e., post-access shared state) at depth h and holds the token, and c) thread

Table 2. Single threaded concurrency constraints added in each BMC instance

S1. No Sync Update Constraint: When none of the token passing events is triggered for a *read_sync* control state of LM_i at depth k , we force the next state values to be *unchanged* for each localized shared and race detector variable in LM_i by adding:

$$RC_i^k = 0 \implies \left(\bigwedge_{p=1}^m g_{pi}^{k+1} = g_{pi}^k \right) \wedge (RD_i^{k+1} = RD_i^k) \quad (8)$$

$$RC_i^k = 0 \implies TK_i^{k+1} = TK_i^k \quad (9)$$

$$RC_i^k = 0 \implies \bigwedge_{q=1}^n CS_{qi}^{k+1} = CS_{qi}^k \quad (10)$$

Similarly, for every *write_sync* control state of LM_j at depth h , we force the next state token value to be *unchanged* by adding a similar constraint:

$$WC_j^h = 0 \implies TK_j^{h+1} = TK_j^h \quad (11)$$

S2. Lock/Unlock Synchronization Constraint: To model *assume*($lk = 0$) in *lock* control state l_i of LM_i at depth k , and similarly, for *unlock* control state ul_i , we add

$$B_{l_i}^k \implies (\neg lk_i^k); \quad B_{ul_i}^k \implies (lk_i^k) \quad (12)$$

S3. Write Commit Constraint: We make only a write operation commit its current shared state to be *visible* to the future transitions by adding the following constraint in *write_sync* control state w_j of LM_j at depth h corresponding to write operation only, i.e.,

$$B_{w_j}^h \implies TK_j^h \quad (13)$$

S4. Single Control State Reachability Property Constraint: For checking reachability of a local control state $a \in C_i$, we add constraint:

$$B_a^k \implies TK_i^k \quad (14)$$

Table 3. Global concurrency constraints added in each BMC instance

Single Token Constraint: Initially, exactly one thread model has the token. We add,

$$\left(\bigvee_{1 \leq i \leq n} TK_i^0 \right) \wedge \left(\bigwedge_{i \neq j} TK_i^0 \implies \neg TK_j^0 \right) \quad (15)$$

Multiple Race Detections: To check multiple data races *incrementally*, we add the following blocking clause corresponding to the *token passing events* seen in the last witness trace.

$$\neg (RW_{ij}^{kh} \wedge \dots \wedge RW_{i'j'}^{k'h'}) \quad (16)$$

model LM_j has the latest value of clock variable of LM_i , and both threads agree on that. Note, this constraint *per se* is not enough for *token passing*, and we require the following exclusivity constraint as well.

- P2. Read-Write Synchronization Exclusivity Constraint:** Exclusivity constraints (Eqn 23) ensure that for a chosen pair for token passing $(r_i@k, w_j@h)$ with $i \neq j$, other pairs $(r_i@k, w_{j'}@h')$ with $h \neq h'$ or $j \neq j'$ and $(r_{i'}^k, w_j^h)$ with $i \neq i'$ or $k \neq k'$ are *implied invalid*. As shown in Figure 3(a), the pair $(r_1@1, w_2@3)$ excludes other pairs $(r_1@1, w_2@6)$, $(r_1@1, w_2@9)$, $(r_1@4, w_2@3)$, and $(r_1@7, w_2@3)$ due to specific values of variables RC_1^1 and WC_2^3 chosen. Note, Eqn 1 together with Eqn 2 and 3, define RW_{ij}^{kh} . We say a *token passing event* is triggered iff $RW_{ij}^{kh} = 1$.
- P3. Read-Write Synchronization Update Constraint:** If the token passing event is triggered, each localized shared variable of LM_i at depth k gets the current state value of the corresponding localized shared variable of LM_j at depth h (Eqn 4), the next state value of token of LM_i is constrained to 1, while it is constrained to 0 for LM_j , indicating a transfer of the token (Eqn 5), the next state value of the clock variable of LM_i is incremented by 1, while the remaining clock variables are *sync*-ed with that of LM_j (Eqn 6).
- P4. Data Race or Pair-wise Reachability Property Constraints:** A data race is detected, i.e., $RD_i^{k+1} = 1$ at thread model i at depth $k+1$ whenever for a shared variable g read-write synchronization enabling constraint RW_{ij}^{kh} holds (Eqn 7) with at least one write access on g . To check whether control states $a \in C_i$ (of LM_i) and $b \in C_j$ (of LM_j) are reachable simultaneously, we reduce the reachability problem to a token passing event detection, i.e., $RW_{ij}^{kh} = 1$ by adding control states *read_sync* and *write_sync* before and after control states a and b .

Similarly, we give the intuition and description for the following single-threaded (i.e., intra-model) constraints *S1-S4* added for each thread model LM_i (LM_j) at depth k (h) as shown in Table 2.

- S1. No Sync Update Constraint:** The constraints (Eqn 8-11) keep the states of thread-specific localized global variables, and newly introduced variables unchanged when there is no token passing event (i.e. no context switching) occurs.
- S2. Lock/Unlock Synchronization Constraint:** The constraints (Eqn 12) assert/deassert the locking predicate variables in the respective lock/unlock states.
- S3. Write Commit Constraint:** The constraints (Eqn 13) make the write operation of a thread visible to others by asserting the token.
- S4. Single Control State Reachability Property Constraint:** Like *S3*, the constraints (Eqn 14) make the reachability of a control state visible by asserting the token.

As shown in Table 3, we add a single token constraint (Eqn 15) needed for total order and blocking clauses (Eqn 16) to detect multiple data races.

6 Correctness and Size Complexity

Theorem 1 (Correctness) (A) *The lazy modeling constraints allow only those traces that respect the sequential consistency of memory model and synchronization semantics up to the bound D , i.e., our modeling is complete.* (B) *Further, if there exists a witness*

for a reachability property, such that the global trace length is $\leq n \cdot D$ and each local trace length $\leq D$, there exists an equivalent trace allowed by our model corresponding to the witness trace. In other words, our modeling is sound in that it does not miss any witness up to these bounds.

Proof: Here is an outline. (Details upon request.)

- *Completeness:* Our modeling captures the requirements for sequential consistency (a) *program order:* using the transition model of each thread, *No Sync Update Constraint* (Eqn 9-11), and *Write Commit Constraint* (Eqn 13), (b) *total order of shared accesses:* using logical clock along with *Read-Write Synchronization* (Eqn 5 and 6) and *No Sync Update Constraint* (Eqn 9 and 10), (c) *read value rule:* using *Read-Write Synchronization Constraint* (Eqn 4), and *No Sync Update Constraint* (Eqn 8), and (d) *mutual exclusion rule:* using *Lock/Unlock Synchronization Constraint* (Eqn 12).
- *Soundness:* We add pair-wise constraints for *all* pairs of shared accesses that are *statically reachable* up to the bounded depth. Thus, we capture all possible interleavings of shared accesses up to the bound, and hence, we cannot miss any witness up to the bound. \square

6.1 Size Complexity

We now discuss the size of constraints and variables *incrementally* added at each depth d for n concurrent threads. We consider thread specific `read_sync` and `write_sync` procedure calls, without inlining. This implies that at each unrolled depth k of LM_i , at most one `read_sync` control state r_i and at most one `write_sync` control state w_i belong to the reachable set $R_i(k)$. Thus, at depth d , r_i (or w_i) block is paired with at most $(n \cdot d) w_j$ (or r_j) blocks. Since there are n threads, we have at most $(n^2 \cdot d)$ pairs. Thus, at depth d , the number of pair-wise constraints added, and variables introduced are $O(d)$, and number of non-pair constraints added is $O(1)$. Overall, the size of constraints and variables added up to depth d is $O(d^2)$. Thus, the concurrency constraints grow *quadratically* with unrolling in the worst case.

For X memory accesses up to depth d , the size complexity can be shown to be $O(X^2)$. To compare, the previous approaches [23, 24], incur a *cubic* cost, i.e., $O(X^3)$, for a given memory model and a test program. \square

7 Removal of Redundant Concurrency Constraints

We use following static analyses to reduce the number of context switches to consider; thereby, remove redundant concurrency constraints corresponding to them:

- *CFG transformation* (PB): We use path/loop balancing transformations [21] on each thread model independently to obtain a reduced set of statically reachable blocks in *CSR*. This reduces concurrency and unrolled transition constraints.
- *Lockset-based analysis* (MTX): We determine statically pair-wise unreachability of `read_sync` and `write_sync` control states using *lockset* [10, 9, 14] analysis. For such

pairs of `read_sync` and `write_sync` control states that are mutually exclusive (e.g., due to matching locks/unlocks), we do not add pair-wise constraints as the concurrency semantics forbids context-switching between those thread states.

- *Context-sensitive CSR (CXT)*: We handle non-recursive procedures by creating a single copy and using extra variables to encode the call/return sites. (Recursive procedures are inlined upto some user-chosen depth). However, not inlining a procedure can cause *false* loops in the CFG of each thread, due to unmatched calls/returns of a procedure. We avoid saturation in CSR due to *false* loops in CSR by determining reachability in a context-sensitive manner, i.e., by matching the call/return sites for each procedure call. We observe in our experimental results that such analysis gives a dramatically reduced set of reachable `read_sync` and `write_sync` control states, and hence, a reduction in the set of pair-wise constraints added.

Discussion: The above mentioned static analysis techniques are not required to be precise as the imprecision does not affect completeness and soundness (Theorem 1) and size complexity of the overall analysis. However, less precise (but conservative) static analysis may result in less simplification and thereby poorer BMC performance. Furthermore, we can utilize transactions and/or partial order reductions [8,14], to eliminate some pair-wise concurrency constraints.

8 Experiments

We experimented on the Daisy file system [30], a public benchmark used to evaluate and compare various concurrent verification techniques for concurrent threads communicating with shared memory and locks/unlocks. It is a 1KLOC Java implementation of a representative file system, where each file is allocated a unique inode that stores the file parameters, and a unique block which stores data. Each access to a file, inode or block is guarded by a dedicated lock. Since a thread does not guarantee exclusive operations on byte access, potential race conditions exist. This system has some known data races. For our experiments, we used a C version of the code [14] with a two-threaded concurrent system. Each thread model comprises 215 control states and 80 localized variables. Overall system model has 3 lock and 3 global variables.

We conducted our experiments on an SMT-based BMC framework similar to [21]. We used the `yices-1.0` [31] SMT solver at the back-end. We compared our *lazy modeling* with an *eager modeling*. In eager modeling approach, we add the pair-wise constraints in the model itself between the states with shared access, that also have wait-cycles. We applied BMC simplification using CSR as discussed in Section 2.3 for all cases, referred to as the baseline strategy. We then combined this baseline strategy with other static analysis techniques such as path balancing/loop CFG transformations (PB) on CFG [21], context-sensitive analysis (CXT), and *lockset* analysis (MTX) [10,9,14]. We conducted controlled experiments with various combinations of these techniques. Time taken for these static analyses, and for adding concurrency constraints are insignificant compared to solving BMC instances. We combine these times with the solve times, and do not report them separately.

8.1 Comparing BMC Results

We now compare the performance of SMT-based BMC on detecting *multiple* data races, on both eager/lazy models, in various combinations of strategies. Note, multiple race detection is an optional feature. We detect *multiple data races* incrementally, by adding a blocking clause corresponding to the token passing events *seen* in the last witness trace to the satisfiable BMC instance, and then continuing the search. We conducted our experiments on a workstation with dual Intel 2.8 GHz Xeon Processors with 4GB physical memory running Red Hat Linux 7.2, using a 6 hrs (≈ 20 Ks) time limit and unroll bound limit of 300 for each BMC run. The results are summarized in Table 4(a). Column 1, shows the modeling approach (eager/lazy); Columns 2-6 show BMC results for various combinations of static analysis methods. Each *data point* ($d : t, m$) corresponds to a performance summary of BMC up to depth d , with t and m representing the cumulative run time and memory used, respectively. Note, cumulative time includes the solve time incurred in the previous depths for the same run. We show a selected few *data points* for comparison. Specifically, Column 2 shows data for CSR with no PB and no CXT; Column 3 shows data for CSR with PB and no CXT; Column 4 shows data for CSR with PB and CXT; Column 5 shows data for CSR with PB, CXT and MTX; and Column 6 shows data for CSR with PB and MTX, but no CXT. For eager modeling, due to non-inlining of procedure calls, we did not obtain any useful lockset information to reduce the constraints statically, and therefore, results in the columns (CSR+PB+CXT+MTX) and (CSR+PB+MTX) are the same as (CSR+PB+CXT) and (CSR+PB), respectively. As an example, consider BMC at unroll depth 64. BMC on eager model with CSR times out (TO) requiring 66 Mb, while on lazy model with CSR it takes 26s and 39Mb.

In general, PB and CXT help BMC go deeper in both the eager and lazy models. However, CXT has a pronounced effect on the BMC performance. We also observed that the *lockset* analysis helps in improving the BMC performance, but not significantly. BMC on eager model, in general, does not go very deep, and times out in all cases without detecting any data races. In contrast, BMC on lazy model with (CSR+PB+CXT) or (CSR+PB+CXT+MTX) is able to find 50 data races in a single BMC run. Note, less precise static analysis CSR and CSR+PB show poorer performance compared to CSR+PB+CXT and CSR+PB+CXT+MTX.

In Table 4(b), we provide details of BMC performance on lazy models on the first five data races using CSR+PB+CXT+MTX. Column 1 shows the data races listed in the order of detection; Columns 2-4 show the BMC depth, cumulative time, and memory used, respectively. Column 5 shows the context-switches in the trace, each denoted as $(P_i : k_i, l_i) \rightarrow (P_j : k_j, l_j)$ where model P_i executes *uninterrupted* from depth k_i to l_i , and then switches the context to P_j at depth k_j . The Daisy example intentionally included many data races as a benchmark. Each race reported here corresponds to a unique set of context-switches in the witness trace. As an example, the first data race is detected at depth 143 taking 12s and 10Mb. There are 3 context switches: a P_1 run from depth 0 to 127, followed by a P_2 run from depth 0 to 127, followed by another P_1 run from 128 to 142, followed by a data race detection when P_2 accesses the same variable at depth 128. Note, the length of the trace is 271(= 143 + 128).

8.2 Comparison with Related Work

In a related effort [14], two write-write data races were detected for the same benchmark, i.e. Daisy file system [30], using 1283s and 122Mb, and 5925s and 902Mb, respectively. Note, our eager modeling (used in experiments) differs from them mainly in the back-end solver, i.e., SMT vs SAT, and in use of static reduction methods, which was the crux of their approach. We believe that the orders of magnitude improvement in BMC performance (as reported in Table 4(b)), are attributed mainly to our lazy modeling paradigm that facilitates dramatic size-reduction of BMC instances.

In contrast to a closely related work TCMBC [15], we do not bound the number of context switches. Further, we add constraints lazily and incrementally during BMC unrolling. TCBMC, built over CBMC [32], translates concurrent C threads into static single assignment (SSA) form, and adds constraints for a bounded number of context-switches for a bounded depth. TCMBC approach requires full inlining of functions and unwinding of loops like CBMC. This CBMC-based approach, therefore, is not scalable to large piece of code or code with reactive behavior, as shown previously [28].

We also contrast our work with [23, 24], where weaker memory models are considered. However, these approaches only check given test programs. Further, the number of concurrency constraints they add are *cubic* [24] in the number of shared accesses, while we add a *quadratic* number of constraints.

Table 4. SMT-based BMC (a) Comparison Results (b) Sample Witness Traces

(a) Comparing SMT-based BMC on Lazy/Eager Models						(b) First 5 Data Race Traces using CSR+PB+CXT+MTX on lazy model					
Model	Static Analysis Strategies					#depth	BMC	Time	Mem	$(P_i : k_i, l_i) \rightarrow (P_j : k_j, l_j)$ P_i executes from depths k_i to l_i uninterrupted and context-switches to P_j at depth k_j .	
	CSR (1)	CSR+PB (2)	CSR+PB +CXT(3)	CSR+PB +CXT+MTX(4)	CSR+PB +MTX(5)						sec
	$d: t, m$ with $d \equiv$ BMC Depth, $t \equiv$ Cum. Time(s), $m \equiv$ Mem(Mb)										
Eager	64: TO,66	64: 132,21	64: 10,14	same as (3)	same as (2)	1	143	12	10	(1: 0,127) \rightarrow (2: 0,127) \rightarrow (1: 128,142) \rightarrow (2: 128,-)	
		95: TO,59	95: 2K,31								
			124: TO,49								
race?	N	N	N	N	N						
Lazy	64: 26,39	64: 6,10	64: 2,6	64: 1,6	64: 3,10						2
	73: 8K,101	95: 35,10	95: 5,8	95: 4,8	95: 17,28						
	Yices	118: TO,114	118: 8,11	118: 8,11	118: 8,11	118: 15K,108					
			124: 16, 10	124: 9, 10	119: TO,112						
	aborted		287: 2.7K,34	287: 2.4K,32							
race?	N	N	50 races	50 races	N						
						3	180	30	15	(1: 0,127) \rightarrow (2: 0,179) \rightarrow (1: 15,-)	
											4
						5	211	99	18	(1: 0, 127) \rightarrow (2: 0,210) \rightarrow (1: 128,-)	

Note: N \equiv No Race Detected, * \equiv Yices Aborted, TO \equiv Time Out

9 Conclusions and Future Work

We described a novel lazy modeling paradigm for shared memory multi-threaded concurrent systems, that is more suitable for BMC compared to synchronous modeling of interleaving semantics proposed previously. Such direct modeling of concurrency semantics in BMC is geared toward reducing the size of BMC instances, and thereby, improving the performance of BMC for deeper analysis. We add concurrency constraints lazily, incrementally and on-the-fly during BMC unrolling that preserve the concurrency semantics up to the bounded depth. By avoiding wait-cycles, our modeling allows

greater scope for reduction in size of a BMC instance. In addition, we use various static analyses to reduce the number of context-switches to consider, which further reduces the size of the constraints. We demonstrated the efficacy of our approach on a complex example. In future, we would like to combine *partial-order reduction* methods [8, 14], and add deadlock detection.

Acknowledgement

We thank Vineet Kahlon for helpful discussions and providing the C version of the Daisy benchmark.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer, Los Alamitos (1996)
2. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM (1978)
3. 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)
4. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
5. Ramalingam, G.: Context sensitive synchronization sensitive analysis is undecidable. In: ACM Transactions on Programming Languages and Systems (2000)
6. Godefroid, P.: Model checking for programming languages using verisoft. In: Proc. of POPL (1997)
7. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: ZING: Exploiting program structure for model checking concurrent software. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 1–15. Springer, Heidelberg (2004)
8. Godefroid, P.: Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem. PhD thesis (1995)
9. Flanagan, C., Qadeer, S.: Transactions for software model checking. In: Proc. of TACAS (2003)
10. Stoller, S.D.: Model-checking multi-threaded distributed Java programs. Journal on STTT (2002)
11. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 489–504. Springer, Heidelberg (2003)
12. Levin, V., Palmer, R., Qadeer, S., Rajamani, S.K.: Sound transaction-based reduction without cycle detection. In: Proc. of SPIN Workshop (2003)
13. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Proc. of CAV, pp. 340–351 (1997)
14. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Proc. of CAV (2006)
15. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Proc. of CAV (2005)
16. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. Electronic Notes Theoretical Computer Science (2003)
17. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

18. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided Underapproximation-Widening for Multi-process Systems. In: Proc. of POPL (2005)
19. Cook, B., Kroening, D., Sharygina, N.: Symbolic Model Checking for Asynchronous Boolean Programs. In: Proc. of SPIN Workshop (2005)
20. 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)
21. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: Proc. of IC-CAD (2006)
22. Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B.: Detecting data races on weak memory systems. In: Proc. of ISCA (1991)
23. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory-model-sensitive data race analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 30–45. Springer, Heidelberg (2004)
24. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: Proc. of PLDI (2007)
25. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: Proc. of IPDPS (2004)
26. Voung, J.W., Jhala, R., Lerner, S.: Relay: static race detection on millions of lines of code. In: ESEC/SIGSOFT FSE, pp. 205–214 (2007)
27. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proc. of POPL, pp. 327–338 (2007)
28. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based Bounded Model Checking for Software Verification. In: Margaria, T., Steffen, B. (eds.) ISO/LA 2004. LNCS, vol. 4313. Springer, Heidelberg (2006)
29. Ganai, M.K., Gupta, A.: Completeness in SMT-based BMC for software programs. In: Proc. of DATE (2008)
30. Joint CAV/ISSTA Special Event. Specification, Verification, and Testing of Concurrent Software (2004), <http://research.microsoft.com/~quadeer/cav-issta.htm>
31. SRI. Yices: An SMT solver, <http://fm.csl.sri.com/yices>
32. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: Proc. of DAC (2003)

Tackling Large Verification Problems with the Swarm Tool*

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce

Laboratory for Reliable Software (LaRS)
Jet Propulsion Laboratory, California Institute of Technology

Abstract. The range of verification problems that can be solved with logic model checking tools has increased significantly in the last few decades. This increase in capability is based on algorithmic advances, but in no small measure it is also made possible by increases in processing speed and main memory sizes on standard desktop systems. For the time being, though, the increase in CPU speeds has mostly ended as chip-makers are redirecting their efforts to the development of multi-core systems. In the coming years we can expect systems with very large memory sizes, and increasing numbers of CPU cores, but with each core running at a relatively low speed. We will discuss the implications of this important trend, and describe how we can leverage these developments with new tools.

1 Introduction

The primary resources in most software applications are time and space. It is often possible to make an algorithm faster by using more memory, or to reduce its memory use by allowing the run time to grow. In the design of SPIN, a reduction of the run time requirements has almost always taken precedence.

For an exhaustive verification, the run time requirements of SPIN are bounded by both the size of the reachable state space and by the size of available memory. If M bytes of memory are available, each state requires V bytes of storage, and the verifier on average records S new states per second, then a run can last no longer than $M/(S*V)$ seconds. If, for example, M is 64 MB, V is 64 bytes, and S is 10^4 states per second, the *maximum* runtime would be 10^2 seconds. If there are more than 10^6 reachable states, the search will remain incomplete – being limited by the size of memory.

The verification speed depends primarily on the average size of the state descriptor, which is typically in the range of 10^2 to 10^3 bytes. On a system running at 2 or 3 GHz the processing speed is normally in the range of 10^5 to $5 \cdot 10^5$ states per second. This means that in about one hour, the model checker can explore 10^8 to 10^9 states, provided sufficient memory is available to store them. This means that some 10^{11}

* The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was supported in part by NASA's Exploration Technology Development Program (ETDP) on Reliable Software Engineering.

bytes, or 100 GB, can be used up per hour of runtime.¹ On an 8 GB system, that means that the model checker can (in exhaustive storage mode) normally run for no more than about 5 minutes. If we switch to a 64 GB system running at the same clock-speed, the worst-case runtime increases to 40 minutes.

An interesting effect occurs if we switch from exhaustive verification mode to *bitstate* mode, where we can achieve a much higher coverage of large state spaces by using less than a byte of memory per state stored [H87]. The exact number of bytes stored per state is difficult to determine accurately in this case. The current version of SPIN by default uses three different hash-functions, setting between one and three new bit-positions for each state explored. We will assume conservatively here that each state in this mode consumes 0.5 bytes of memory, and that the speed of the model checker is approximately 10^8 states per hour. Under these assumptions, the model checker will consume maximally $10^8 * 0.5$ bytes of memory per hour of run time, or roughly 50 MB. To use up 8 GB will now take about a week (6.8 days) of non-stop computation. In return, we cover significantly more states, but both time and space should be considered limited resources, so the greater number of states is not always practically achievable. To make the point more clearly, if we increase the available memory size to 64 GB, a bitstate search could consume close to two months of computation, which is clearly no longer a feasible strategy, no matter how many states are explored in the process.

We are therefore faced with a dilemma. The applications that we are trying to verify with model checkers are increasing in size, especially as we are developing methods to apply logic model checkers directly to implementation level code [H00, HJ04]. As state descriptors grow in size from tens of bytes to tens of kilobytes, processing speed drops and is no longer offset by continued CPU clock-speed increases. For very large applications, a bitstate search is typically the only feasible option as it can increase the problem coverage (i.e., the number of reachable states explored) by orders of magnitude. Exhaustive coverage for these applications is generally prohibitively expensive, given the enormous size of both the state descriptors and the state spaces, no matter which algorithm is used. In these cases we have to find ways to perform the best achievable verification. Technically, the right solution in these cases is always to apply strong abstraction techniques to reduce the problem size as much as possible. We will assume in the remainder of this paper that the best abstractions have already been applied and that the resulting state space sizes still far exceed the resource limits in time and/or memory.

2 Leveraging Search Diversity

To focus the discussion, we will assume that there is always an upper bound on the time that is available for a verification run, especially for large problem sizes. We will assume that this bound is *one hour*. With a fixed exploration rate this means that we cannot use more than a few Gigabytes of memory in exhaustive verification and no more than about 50 to 500 MB in bitstate exploration. Given that for very large verification problems we have to accept that the search for errors within a specific time

¹ Smaller state descriptors normally correlate with higher processing speeds.

constraint will generally be incomplete, it is important that we do not expend all our resources on a single strategy. Within the limited time available, we should approach the search problem from a number of different angles – each with a different chance of revealing errors.

A good strategy is to leverage both *parallelism* and *search diversity*. The types of applications that then become most promising fall in the category of “embarrassingly parallel” algorithms.

To illustrate our approach, we will use a simple model that can generate a very large state space, where we can easily identify every reachable state and predict when in a standard depth-first search each specific state will be generated. The example is shown in Figure 1.

```

byte pos = 0;
int val = 0;
int flag = 1;

active proctype word()
{
end: do
  :: d_step { pos < 32 -> /* leave bit 0 */ flag = flag << 1; pos++ }
  :: d_step { pos < 32 -> val = val | flag; flag = flag << 1; pos++ }
od
}

never { do :: assert(val != N) od } /* check if number N is reached */

```

Fig. 1. Model to generate all 32-bit values, to illustrate the benefits of search diversification

The model executes a loop with two options, from which the search engine will non-deterministically select one at each step. Each option will advance an index into a 32-bit integer word named *val* from the least significant bit (at position one) to the most significant bit (at position 32). The first option leaves the bit pointed to set to its initial value of zero, and merely advances the index. The second option sets the bit pointed at to one. Clearly, there will be 2^{32} (over 4 billion) possible assignments to *val*. Each state descriptor is quite small, at 24 bytes, but storing all states exhaustively would still require over 100 GB.

If we perform the state space exploration on a machine with no more than 2 GB, an exhaustive search cannot reach more than 2% of the state space. A bitstate search on the other hand could in principle store all states, but only under ideal conditions. For this model, with a very small state descriptor, we reach a processing speed of close to 2 million states per second, on a standard 2.3 GHz system. We will, however, artificially limit the amount of memory that we make available for the bitstate hash array to 64 MB and study what we can achieve in terms of state coverage by exploiting parallelism and search diversification techniques. In terms of SPIN options, this means the selection of a *pan* runtime flag of maximally $-w29$. In practice this means that for this example only about 148 million states are reached in bitstate mode, or no more than 3.5% of the total state space.

For this example it is also easy to check if a specific 32-bit value is reached in SPIN's exploration, by defining the corresponding value for N when the model is generated. For instance, checking if the value negative one is reached in the maximal bitstate search can be done as follows:

```
$ spin -DN=-1 -a model.pml
$ cc -DMEMLIM=2000 -DSAFETY -o pan pan.c
$ ./pan -w29
```

This particular search fails to produce a match. It is easy to understand why that is. Note that the value negative one is represented in two's complement as a series of all one bits, which in the standard depth-first search is the *last* number that would be generated by the verifier. Performing the same search for the value positive one will produce an almost immediate match, for the same reason. If we reverse the order of the two options in the model of Figure 1, the opposite effect would occur: the search for negative one would complete quickly and the search for positive one would fail.

If the number to be matched is randomly chosen, we could not devise a search strategy that can optimize our chances of matching it, which is more representative of a real search problem. After all, if we know in advance where the error might be, we would not need a model checker to find it.

In the experiments that we will describe we will use a list of 100 randomly generated numbers, and compare different methods for matching as many of them as possible. If the random number generator behaves properly, the random numbers will be distributed evenly over the entire state space of over 4 billion reachable states. Statistically, the best we could expect to do in any one run would be to uncover no more than 3 or 4 of those states (given that with runtime flag `-w29` we can explore at most 3.5% of the reachable state space). We will see that with a diversified search strategy, we can do significantly better and identify 49% of the randomly generated states. We will also show that even when using only 4 MB (a tiny fraction of the 100 GB that would be required to store the full state space) we can already identify 10% of the target numbers.

3 Algorithms

To make our method work we have to be able to use as many different search methods as there are processing cores available to us. If each search is setup to use only a small fraction of the total memory that is available on our system, we can run all searches in parallel. In the description that follows we describe a number of different search algorithms. Several of these algorithms can be modified to form any number of additional searches, each of which able to search a different part of the state space. The base algorithms we use can be described as follows.

1. (*dfs*) The first method is the standard depth-first search that is the default for all SPIN verifications.
2. (*dfs_r*) The next method reverses the order in which a list of non-deterministic choices within a process is explored, using the compiler directive `-D_TREVERSE` (new in SPIN version 5.1.5).

3. (*r_dfs*) The next method uses a search randomization strategy on the order in which a list of transitions is explored, using the existing compiler directive `-DRANDOMIZE` (first introduced in SPIN version 4.2.2). With this method the verifier will randomly select a starting point in the transition list, and start checking transitions for their executability in round-robin order from the point that was randomly selected.
4. (*pick*) The next method uses a user defined selection method, using embedded C code, to permute the transitions in a list.²
5. The last method reverses the order in which process interleavings are explored, using the compiler directive `-DREVERSE` (introduced in SPIN version 5.1.4).

Because our example uses just a single process, we will not use the last variation of the search. Alternative methods for modifying process scheduling decisions during a search can be found in [MQ08].

Algorithms 3 and 4 can be used to define a range of search options, using different seeds for the random number generator. To illustrate this, we will use two versions of algorithm 3, called *r_dfs1* and *r_dfs2*. Each algorithm from the set can be used in a series of runs. In our tests we repeated each run 100 times, once for each number from the list of 100 target numbers to match. We then repeat each of these 100 runs 24 times, while varying the size of the hash-array used from our limit value of `-w29` (64 MB) down to a minimum of `-w6` (64 bytes). This is an application of *iterative search refinement*, as first discussed in [HS00].

About 2,000 runs from this series of experiments, for hash array sizes from `-w6` up to and including `-w23` take less than a second of runtime each, so despite the large number of runs, they can be completed very quickly. For the larger hash array sizes (2MB and up) the runtime and the number of states covered becomes more notable, with the longest runs taken 84 seconds each on a 2.4 GHz system. All 24 runs combined take no more than about 3 minutes of real time when run sequentially, which means that all 2,400 runs can be completed in about 5 hours on a single CPU core, or in about 37 minutes total on the 8-core machine that was used for these experiments. The results are shown in Figure 2.

When used separately, and performing one verification run alone, none of the search methods identify more than about **9%** of the target values set for this experiment. If we look at the cumulative effectiveness of the iterative search refinement method, using 24 runs of a each algorithm and increasing the size of the hash array step by step, this coverage increases, with the best performing search method (*r_dfs2*) identifying **15%** of all targets. The performance of all five search strategies combined in our proposed diversified multi-core search strategy increases the coverage to the identification of **49%** of all targets.

The top curve in Figure 2 shows the *cumulative* number of matches (out of 100) as the memory arena is increased from 64 bytes (`-w6`) to 64 MB (`-w29`). The other curves show the performance of the individual search algorithms.

² Available from the authors.

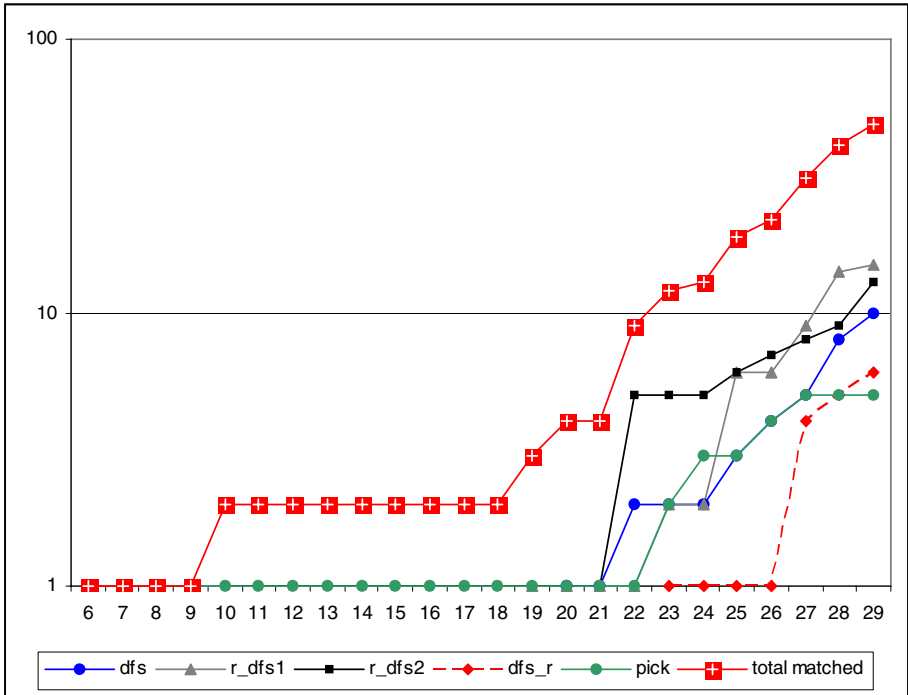


Fig. 2. Results of 2,400 Verification Runs for the Model in Figure 1 (logscale)

Each different search method identifies a different set of targets, thus boosting the overall effectiveness of the use of all methods combined. Adding more variations of the searches could increase the problem coverage still further. It is evident from these data that the performance of the diversified multi-core search is significantly better than any one search method in isolation.

The diversified multi-core search promises to be a very valuable addition to the range of techniques that we can use to tackle very large software verification problems. It builds directly on the availability of systems with increasing numbers of CPU cores and rapidly growing memory sizes, that we can expect in the coming years and perhaps decades. There may also be a direct application of this strategy of swarm verification in grid computing, using large numbers of standard networked computers. In this paper, though, we focus on the application to multi-core systems only.

4 The Swarm Tool

Based on the observations above, we developed a tool that allows us to leverage the effect of search diversification on multi-core machines. The Swarm tool, written in about 500 lines of C, can generate a large series of parallel verification runs under precise user-defined constraints of time and memory. The tool takes parameters that

define a time-constraint, the number of available CPUs, and the maximum amount of memory that is available.³

Swarm first calculates how many states could maximally be searched within the time allowed, and then sets up a series of bitstate runs. By limiting the hash arena in a bitstate run, Swarm controls what the maximal time for each run will be. Parallelism is used to explore searches with different search strategies to provide diversity. The commands that are generated include standard, randomized, and reverse depth-first search orders, varying depth-limits, and using a varying numbers of hash-functions per run. In a relatively small amount of time, hundreds of different searches can be performed, each different, probing different parts of an oversized state space.

A typical command line invocation of the Swarm tool is as follows:

```
$ swarm -c4 -m16G -t1 -f model.pml > script
swarm: 33 runs, avg time per cpu 3593.8 sec
```

In this case we specify that we want to use 4 CPU cores for the verifications, and have up to 16 GB of memory available for these runs. The `-t` parameter sets the time limit for all runs combined to one hour. Swarm writes the verification script onto the standard output, which can then be written into a script file. Executing the script performs the verification.

Application. An example of a large problem that cannot be handled with standard search methods is a SPIN model of an experimental Fleet processor architecture. The details of the design itself are not of interest to us here, but the verifiability of the model is.⁴ One version of this model has a known assertion violation that can be triggered through a manually guided simulation in about 350 steps.

The model is over one thousand lines of PROMELA.⁵ Each system state is 1,440 bytes. An attempt to perform a full verification on a machine with 32 GB of memory runs at roughly 10^5 states per second, and exhausts memory in 195 seconds without reporting the error. At this point the search explored 23.4 million states, corresponding to an unknowable fraction of the reachable state space. A search using `-DCOLLAPSE` compression (a lossless state compression mode) reaches 327.6 million states before running out of memory after 3,320 seconds. A run with `hash-compact` (a stronger, but not lossless, form of compression) runs out of memory after 1,910 seconds and increases the coverage to 537 million states. A bitstate run, using all 32 GB of memory, runs for 34 days, and explores over 10^{11} system states. None of these search attempts succeed in locating the assertion failure.

The full reachable state space for this problem is likely to be orders of magnitude larger than what can be searched or stored by any verification method. The bitstate run can be performed in parallel on 8 CPUs, shrinking the run time from 34 days to about 5 days [HB07], but without change in result. An alternative would be to run the verification with `-DMA` compression, which is lossless and often extremely frugal in memory use. Such a run could in principle be able to complete the verification and reveal the error, but it would likely take at least a year of computation to do so.

³ For a manual page see: <http://spinroot.com/swarm/>

⁴ The Spin model of the Fleet Architecture Design was built by Rhishikesh Limaye and Naran Sundaram under the guidance of Sanjit Sehia from UC Berkeley.

⁵ The specification language of the SPIN model checker.

A Swarm run for this application is quickly setup. Swarm generates 74 small jobs in 8 scripts that can be executed in parallel on the 32 GB machine, when given a time limit of one hour (the default). Executing the script finds the assertion violation within a few seconds. In this case by virtue of the inclusion of the reversed depth-first search. The assertion violation, as it turns out, normally happens towards the end of the standard depth-first order— which means that it is encountered near the very beginning of the search if the depth-first search order is reversed.

For a different test of the performance of Swarm we also studied a series of large verification models from our benchmark set, most of which were also used in [HB07]. EO1 is a verification model of the autonomous planning software used on NASA’s Earth Observer-1 mission [C05]. The Fleet architecture model was discussed above. DEOS is a model of an operating system kernel developed at Honeywell Laboratories, that was also discussed in [P05]. Gurdag is a model of an *ad hoc* network structure with five nodes, created by a SPIN user at a commercial company. CP is a large model of a telephone switch, extracted from C source code with the Modex tool. DS1 is a large verification model with over 10,000 lines of embedded C code taken from NASA’s Deep Space 1 mission, as described in [G02]. NVDS is a verification model of a data storage module developed at JPL in 2006, with about 6,000 lines of embedded C code, and NVFS is non-volatile flash file system design for use on an upcoming space mission, with about 10,000 lines of embedded C.

For each of these models we first counted the number of local states in the automata that SPIN generates for the model checking process. This is done by inspecting the output of command “pan -d.” Next, we measured the number of these local states that is reported as *unreached* at the end of a standard depth-first search with a bitstate hash-array of 64 MB (using runtime flag -w29, as before). Next, we used the Swarm tool to generate a verification script for up to 6 CPUs and 1 hour of runtime. We then measured how many of the local states remained unreached in *all* runs.

Table 1. Swarm Coverage Improvement for Eight Large Verification Models

Verification Model	Number of Control States			Percent of Control States Reached	
	Total	Unreached Control States		standard dfs	dfs + swarm
		standard dfs	dfs + swarm		
EO1	3915	3597	656	8	83
Fleet	171	34	16	80	91
DEOS	2917	1989	84	32	97
Gurdag	1461	853	0	41	100
CP	1848	1332	0	28	100
DS1	133	54	0	59	100
NVDS	296	95	0	68	100
NVFS	3623	1529	0	58	100

The last two columns of **Table 1** show the percentage of control states reached, respectively in the original depth-first search using a 64 MB hash-array, and in that search plus all Swarm verification runs. In all cases the coverage increases notably. For the EO1 model performance increases from 8% to 83%. In the next two cases, coverage increases to over 90%. In the last five large applications we see coverage by this metric reach 100% percent of the control states. It should be noted that this last result does not mean that the full reachable state space was explored. For the models

considered here achieving the latter would be well beyond our resource limitations, which is precisely why we selected them as candidates for the evaluation of Swarm verification.

All measurements were performed on a 2.3 GHz eight-core desktop system with 32 GB of main memory, of which no single run consumed more than 64 MB in these tests. The state vector size for the models ranges from a180 (NVDS) to 3426 (DS1) bytes of memory. The number of Swarm jobs that can be executed within our 1 hour limit ranged from 86 (EO1) to 516 (NVDS).

Swarm unexpectedly succeeded in uncovering previously unknown errors in both the CP model and the NVFS applications. The NVFS application is relatively new, but the CP verification model was first subjected to thorough verification eight years ago, and has since been used in numerous tests without revealing any errors.

5 Conclusion

It is often assumed that the best way to tackle large verification problems is to use all available memory in a maximal search, possibly using multi-core algorithms, e.g., [HB07], to reduce the runtime. As memory sizes grow, most search modes that would allow us to explore very large numbers of states take far too much time (e.g., months) to remain of practical value. In this paper we have introduced a new approach that allows us to perform verifications within strict time bounds (e.g., 1 hour), while fully leveraging multi-core capabilities. The Swarm tool uses parallelism and search diversity to optimize coverage.

We have measured the effectiveness of Swarm in several different ways. In each case we could determine that the new approach could defeat the standard method of a single depth- or breadth first search by a notable margin, both by dramatically reducing runtime and by increasing coverage.

A similar approach to the verification problem was explored in [D07], where it was applied to the verification of Java code with the Pathfinder tool, though without considering run-time constraints.

The use of embarrassingly parallel approaches, like Swarm, becomes increasingly attractive as the number of processing cores and the amount of memory on desktop systems continues to increase rapidly.

An often underestimated aspect of new techniques is the amount of training that will be required to fully leverage them. This is perhaps one of the stronger points in favor of the Swarm tool. It would be hard to argue that the use of Swarm requires more training than a cursory reading of the manual page.

The swarm tool has meanwhile proven to be so remarkably effective that it has become the default interface to the Spin tool that we use for all large verification efforts in our group.

Acknowledgements

The authors are grateful to Sanjit Seshia, Rhishikesh Limaye, Narayan Sundaram, for providing access to and insight in the Fleet Architecture models, and to Madan Musuvathi

and Klaus Havelund for inspiring discussions about search strategies. Doron Peled proposed the introduction of the `-DRANDOMIZE` option in SPIN version 4.2.2.

References

- [C05] Chien, S., Sherwood, R., Tran, D., et al.: Using Autonomy Flight Software to Improve Science Return on Earth Observing One (EO1). *Journal of Aerospace Computing, Information, and Communication* (April 2005)
- [D07] Dwyer, M.B., Elbaum, S.G., et al.: Parallel Randomized State-Space Search. In: *Proc. ICSE 2007*, pp. 3–12 (2007)
- [M69] Moore, G.E.: Cramming more components onto integrated circuits. *Electronics* 38(8) (April 9, 1965)
- [G02] Gluck, P.R., Holzmann, G.J.: Using Spin Model Checking for Flight Software Verification. In: *Proc. 2002 Aerospace Conf.*, March 2002. IEEE, Big Sky (2002)
- [H87] Holzmann, G.J.: On limits and possibilities of automated protocol analysis. In: Rudin, H., West, C. (eds.) *Proc. 6th Int. Conf. on Protocol Specification, Testing, and Verification*, INWG IFIP, Zurich, Switzerland (June 1987)
- [H00] Holzmann, G.J.: Logic verification of ANSI-C Code with Spin. In: Havelund, K., Penix, J., Visser, W. (eds.) *SPIN 2000*. LNCS, vol. 1885, pp. 131–147. Springer, Heidelberg (2000)
- [HS00] Holzmann, G.J., Smith, M.H.: Automating software feature verification. *Bell Labs Technical Journal* 5(2), 72–87 (2000)
- [HJ04] Holzmann, G.J., Joshi, R.: Model-driven software verification. In: Graf, S., Mounier, L. (eds.) *SPIN 2004*. LNCS, vol. 2989, pp. 76–91. Springer, Heidelberg (2004)
- [HB07] Holzmann, G.J., Bosnacki, D.: The design of a multi-core extension to the Spin model checker. *IEEE Trans. On Software Engineering* 33(10), 659–674 (2007)
- [MQ08] Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, June 7–13 (2008)
- [P05] Penix, J., Visser, W., Pasareanu, C., Engstrom, E., Larson, A., Weininger, N.: Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in Systems Design Journal* 26(2) (2005)

Formal Verification of a Flash Memory Device Driver – An Experience Report*

Moonzoo Kim¹, Yunja Choi², Yunho Kim¹, and Hotae Kim³

¹ CS Dept. KAIST, Daejeon, South Korea
moonzoo@cs.kaist.ac.kr
kimyunho@kaist.ac.kr

² School of EECs, Kyungpook National University, Daegu, South Korea
yuchoi76@knu.ac.kr

³ Samsung Electronics, Suwon, South Korea
hotae.kim@samsung.com

Abstract. Flash memory has become virtually indispensable in most mobile devices. In order for mobile devices to operate successfully, it is essential that flash memory be controlled correctly through the device driver software. However, as is typical for embedded software, conventional testing methods often fail to detect hidden flaws in the complex device driver software. This deficiency incurs significant development and operation overhead to the manufacturers.

In order to compensate for the weaknesses of conventional testing, we have applied NuSMV, Spin, and CBMC to verify the correctness of a multi-sector read operation of the Samsung OneNANDTM flash device driver and studied their relative strengths and weaknesses empirically. Through this project, we verified the correctness of the multi-sector read operation on a small scale. The results demonstrate the feasibility of using model checking techniques to verify the control algorithm of a device driver in an industrial setting.

1 Introduction

Flash memory has become a crucial component for mobile devices. Accordingly, in order for mobile devices to operate successfully, it is essential that the device driver of the flash memory operates correctly. However, as is typical of embedded software, conventional testing methods often fail to detect hidden bugs in the device driver software for flash memory, since it is infeasible to test all possible scenarios generated from the complex control structure of the device driver. This deficiency incurs significant overhead to the manufacturers. For example, Samsung spent more project time and resources to test flash software than in developing the software. Limitations of conventional testing were manifest in the development of flash software for Samsung OneNANDTM flash memory [1]. For example, a multi-sector read function was added to the flash software to optimize the reading speed (see Section 3). However, this function caused numerous errors in spite of extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature.

* This work was supported by KAIST Institute for Information Technology Convergence and Samsung Electronics.

In this project, we have verified the correctness of a multi-sector read (MSR) operation of the Samsung OneNAND flash device driver by using NuSMV [6], Spin [14], and CBMC [10] in an exhaustive analysis of a small size flash. These three model checkers employ three different techniques, namely, BDD-based model checking, explicit model checking, and SAT-based bounded model checking.

The contributions of this project are three-folds. First, this project addresses an interesting industrial problem to verify the *functional correctness* of a controller with a *large data structure*. To date, there have been studies on verification of device drivers [13][22][25] and handling of large data structures such as arrays and linked lists [4][8][24]. Nevertheless, they either focus on debugging the compatibility of the hardware-software interface or checking the correctness of standard operations for the data structure itself, rather than verification of the functional correctness of the device driver algorithm. The verification problem we have encountered with MSR is unique in that (1) MSR operates over a large set (potentially millions) of structured data and (2) It is necessary to verify MSR as it is, since our goal is to verify the correctness of existing code – optimization or aggressive abstraction of the algorithm should be minimized.

Second, the model checking results provide a practical alternative verification technique to testing. Although the model checking results guarantee the correctness of MSR on a small size flash only, the exhaustive analysis results provide higher confidence compared to testings previously performed by Samsung. Furthermore, since MSR is a core logic used for most flash software with variations, the verification framework and strategy used in this project can be applied to other flash software with only modest efforts. Samsung highly valued the verification results and started to apply model checking to a flash file system as a subsequent project.

Finally, we conducted a series of experiments to verify MSR by using three popular model checkers: NuSMV, Spin, and CBMC. The results of the experiments show that the selection of a model checking technique has a significant effect on the performance of verifying MSR. It is further demonstrated that effort to create and maintain a target model is also a crucial factor for successful application of model checking in industry. Although a number of case studies involving comparisons between different model checkers have been reported [26][9][12], comparison from the view point of data-intensive applications has not seen intensive study thus far. We believe that this issue is crucial to the success of verifying flash software. Furthermore, the present empirical studies involving the three different model checkers can provide valuable insight into the relative strengths and weaknesses of these popular model checking techniques.

2 Overview of the OneNAND Verification Project

Our team for this project consists of two professors, one graduate student, and one senior engineer at Samsung. We worked on this verification project for six months. We spent the first three months reviewing the design and code of the device driver software and the characteristics of OneNAND flash. Most parts of the device driver software are written in C (~30000 lines) and a small portion of the software is written in ARM assembly language.

2.1 Overview of the Device Driver Software for OneNAND Flash Memory

A unified storage platform (USP) is a software solution to operate OneNAND device. Figure 1 presents an overview of USP. USP allows applications to store and retrieve data on OneNAND through a file system. USP contains a flash translation layer (FTL) through which data and programs in the OneNAND device are accessed. FTL consists of three layers - a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. A prioritized read request for executing a program is made through demand paging manager (DPM) and this request goes to BML directly. A prioritized read request from DPM can preempt generic I/O operations requested from STL.

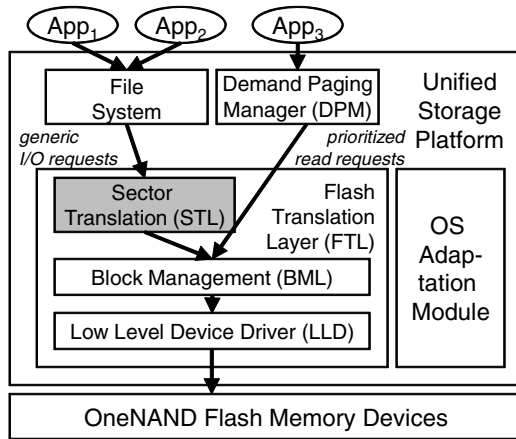


Fig. 1. An overview of USP

2.2 Overview of the Sector Translation Layer (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*. When new data is written to flash memory, rather than overwriting old data directly, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages this mapping from logical sectors (LS) to physical sectors (PS). This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM.

Figure 2 illustrates a mapping from logical sectors to physical sectors where 1 unit contains 4 sectors. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of

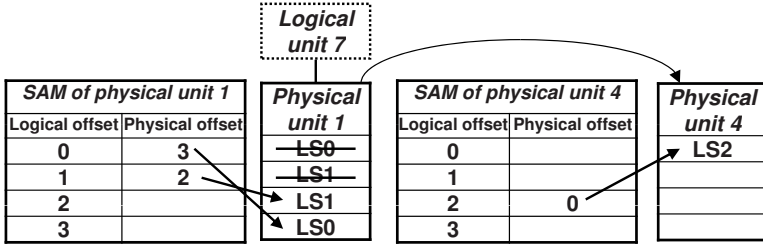


Fig. 2. Mapping from logical sectors to physical sectors

PU1 (SAM1[1]=1). The user then updates the LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3. Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7 and yields SAM4[2]=0.

3 Multi-sector Read Operation

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. The core logic of this mechanism is implemented in a single function in STL. Due to the non-trivial traversal of data structures for logical-to-physical sector mapping (see Section 2.2), the function for MSR is 157 lines long and highly complex, having 4-level nested loops. Figure 3 describes simplified pseudo code of these 4-level nested loops. The outermost loop iterates over LUs of data (line 2-17). The second outermost loop iterates until the LS's of the current LU are completely read (line 4-15). The third loop iterates over PUs mapped to the current LU (line 6-14). The innermost loop identifies consecutive PS's that contain consecutive LS's in the current PU (line 7-10). This loop calculates `conScts` and `offset`, which indicate the number of such consecutive PS's and the starting offset of these PS's, respectively. Once `conScts` and `offset` are obtained, `BML_READ` reads these consecutive PS's as a whole fast (line 11).

For example, suppose that the data is “ABCDEF” and each unit consists of four sectors and PU0, PU1 and PU2 are mapped to LU0 (“ABCD”) in order and PU3 and PU4 are mapped to LU1 (“EF”) in order as depicted in Figure 4(a). Initially, MSR accesses SAM0 to find which PS of PU0 contains LS0(‘A’). Then, it finds SAM0[0]=1 and reads PS1 of PU0. Since SAM0[1] is empty (i.e., PU0 does not have LS1(‘B’)), MSR moves to the next PU, which is PU1. For PU1, MSR accesses SAM1 and finds that LS1(‘B’) and LS2(‘C’) are stored in PS1 and PS2 of PU1 consecutively. Thus, MSR reads PS1 and PS2 of PU1 altogether through `BML_READ` and continues its reading operation.

The requirement for MSR is that the content of the read buffer should correspond to the original data in the flash memory when MSR finishes reading, as given by the following invariant formula for `INV`.

$$INV : after_MSR \rightarrow (\forall i. logical_sectors[i] = buf[i])$$

```

01:curLU = LU0;
02:while(curLU != NULL ) {
03:  readScts = # of sectors to read in the current LU
04:  while(readScts > 0 ) {
05:    curPU = LU->firstPU;
06:    while(curPU != NULL ) {
07:      while(...) {
08:        conScts = # of consecutive PS's to read in curPU
09:        offset = the starting offset of these consecutive PS's in curPU
10:      }
11:      BML_READ(curPU, offset, conScts);
12:      readScts = readScts - conScts;
13:      curPU = curPU->next;
14:    }
15:  }
16:  curLU = curLU->next;
17:}

```

Fig. 3. Loop structures of MSR

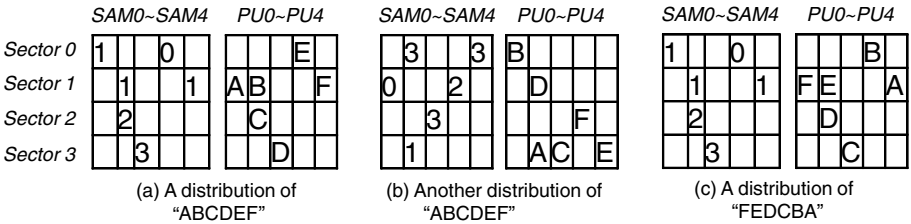


Fig. 4. Possible distributions of data “ABCDEF” and “FEDCBA” to physical sectors

In our verification tasks, we assume that each sector is 1 byte long and each unit has four sectors. Also, we assume that data is a fixed string of distinct characters (e.g., “ABCDE” if we assume that data is 5 sectors long, and “ABCDEF” if we assume that data is 6 sectors long). We apply this data abstraction since the values of logical sectors should not affect the reading operations of MSR, but distribution of logical sectors into physical sectors does. For example, for the same data “ABCDEF”, the reading operations of MSR are different for Figure 4(a) and Figure 4(b), since they have different SAM configurations (i.e. different distributions of “ABCDEF”). However, for “FEDCBA” in Figure 4(c) which has the same SAM configuration of Figure 4(a), MSR operates exactly the same way as for Figure 4(a). Thus, if MSR reads “ABCDEF” in Figure 4(a) correctly, MSR reads “FEDCBA” in Figure 4(c) correctly too.

In addition, we assume that data occupies 2 logical units. The number of possible distribution cases for l LS’s and n physical units, where $5 \leq l \leq 8$ and $n \geq 2$, increases exponentially in terms of n , and can be obtained by

$$\sum_{i=1}^{n-1} \binom{(4 \times i)}{4} C_4 \times 4! \times \binom{(4 \times (n-i))}{4} C_{(l-4)} \times (l-4)!$$

For example, if a flash has 1000 physical units with data occupying 6 LS’s, there exist a total of 3.9×10^{22} distributions of the data.

As you have seen from Figure 4, the operations of MSR depend on the values of SAM tables and the order of PUs linked to LU. Therefore, MSR has characteristics of control-oriented program (4-level nested loops) and data-oriented program (large data structure consisting of SAMs and PUs) at the same time, although the values of PS's are not explicitly manipulated.

4 Model Checking MSR Using NuSMV

NuSMV [6] is an open-source symbolic model checker branched from SMV. Although the SMV family has been widely used in the hardware industry, its application to industrial software has been limited to a couple of case studies [5,20], mostly in the area of software specifications. Despite that explicit model checking has been favored in software verification, MSR has the following attractive characteristics, which have motivated the present authors to verify it with symbolic model checking.

1. MSR operates with a semi-random environment – sector writing is assumed to be random except for having some constraints.
2. MSR's data structure can be abstracted in a simple array form with a couple of simple operations, such as assignments and equality checking.
3. MSR is a single-threaded program that can be verified independently from other modules.

Unlike explicit model checking, which requires that the system environment be modeled explicitly, symbolic model checking allows free-variables to represent environmental inputs whose possible values are exhaustively evaluated through symbolic computation; this is one of the advantages of checking MSR using NuSMV. Whereas it is generally known that symbolic model checking performs poorly on applications with a large data structure and arithmetic operations, the main data structure in MSR is relatively simple two 2-dimensional integer arrays (PUs and SAMs) with no arithmetic operations. MSR's single-threaded structure is also suitable for using NuSMV, which is known to be inefficient in handling interleaves.

4.1 Model Translation

We manually specified a NuSMV model for MSR after reading corresponding design documents and C code. The first challenge in creating a NuSMV model for MSR arises from the different modeling paradigms used in C and NuSMV; the NuSMV modeling language is dataflow-based, whereas C is a control-flow based language. Thus, translation of a C program into a NuSMV model requires introduction of control points to reflect control-dependent changes of data variables.

The second challenge is to model the data structure in NuSMV. Even though SAMs and PUs can be abstracted into simplified two-dimensional integer arrays, modeling such data structure and operations for NuSMV is not a trivial task, especially because NuSMV does not support index variables for arrays. Circumventing the expressional limitations of NuSMV, the resulting translated MSR model consists of more than 1000 lines of code; the original C-code is 157 lines long.

The third challenge involves setting the operational environment of MSR. MSR assumes randomly written logical data on PUs and a corresponding SAM records the actual location of each LS. Unfortunately, however, the writing is not purely random, which means the open environment of the symbolic model checking has to be constrained according to several rules; the followings are some of the representative rules applied to the random writing.

1. One PU is mapped to at most one LU.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i \bmod m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k , where m is a number of sectors per unit (4 in our experiments).
3. The PS number of the i_{th} LS must be written in *only* one of the $(i \bmod m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

For example, for imposing the last two rules, we use the following weaker invariants, which include spurious value combinations in SAMs, to reduce the complexity of imposing invariants. Note that this weakening of invariants does not produce false positives when checking the *INV* property specified in Section 3.

$$\begin{aligned} \forall i, j, k \ (logical_sectors[i] = PU[j].sect[k] \rightarrow (SAM[j].valid[i \bmod m] = true \\ \& SAM[j].offset[i \bmod m] = k \\ \& \forall p.(SAM[p].valid[i \bmod m] = false) \\ \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to } \lfloor \frac{i}{m} \rfloor_{th} \text{ LU})) \end{aligned}$$

4.2 Performance Analysis

We have performed a series of experiments in order to assess the feasibility and scalability of model checking the invariant property *INV*.

Experimental settings and verification results. We verified MSR using a workstation equipped with Xeon 5160 (dual core 3 GHz) and 32 gigabytes memory. The workstation runs 64 bit Fedora Linux 7 and uses NuSMV 2.4.3. The scalability of NuSMV model checking is assessed by measuring the amount of time and memory required to verify the *INV* property as the number of physical units increases from 5 to 8 and the size of logical data increases from 5 to 7 sectors. Figure 5 shows the growth of time and memory consumption from these experiments; the verification time grows exponentially both with the number of physical units and the number of logical sectors (Figure 5(a)). Note that the memory consumption shows better scalability than that of time consumption (Figure 5(b)).

Although NuSMV succeeds in verifying that the MSR satisfies the *INV* property, the exponential time complexity limits the applicability of NuSMV model checking to only a small flash memory. As it requires about 11 hours and 550 megabytes of memory to verify the *INV* property for the MSR model with 7 physical units and 7 logical sectors, we conclude that further experiments with larger numbers of physical units and logical sectors would not be feasible due to a large amount of verification time.

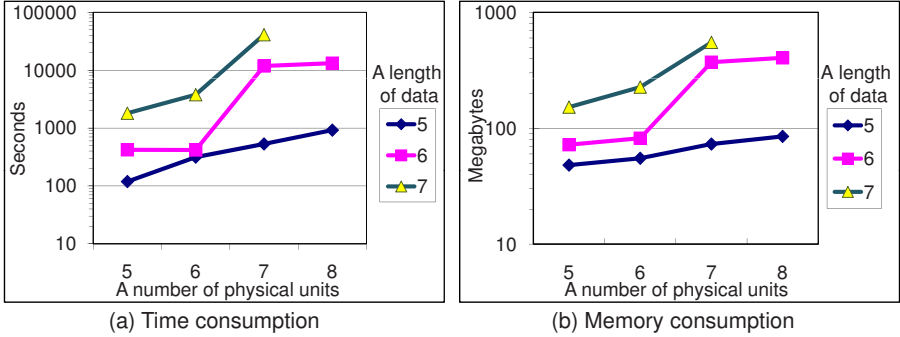


Fig. 5. Time and space complexities of NuSMV model checking

Dynamic reordering and time complexity. The exponential growth of verification time is mainly due to the dynamic reordering of BDD variables to keep the symbolic representation of the state space as compact as possible. OBDD representations for a Boolean formula can be quite different in terms of the number of nodes representing the formula. Since finding optimal BDD variable orderings is an intractable problem [3], NuSMV periodically attempts to improve the orderings by moving each variable through the ordering to find its best location using the sifting algorithm [23]. While this ordering process is known to be effective in terms of reducing the state-space, it is time-consuming, as is clearly seen from our experiments. We observed that more than 90% of verification time was consumed for dynamic reordering.

The NuSMV version of the MSR model (with 5 LS's, 5 PUs) requires 365 BDD variables for its symbolic representation and generates more than 1,182,300 BDD nodes during the verification process, which is 12 times larger in size than the rule-of-thumb-limit for effective reordering, i.e., 100,000. Note that the number of BDD variables encoding the MSR is undesirably large, mainly due to the encodings of data variables; it is necessary to encode at least 20 data variables for 5 PUs with 4 PS's each. Even with a restricted domain, e.g., with the size of the data domain 5, each PS needs 6 (=3+3) Boolean variables, resulting in a total of 120. Since MSR also maintains a SAM of approximately the same size, we can deduce that more than 240 BDD variables are used for encoding the main data structure alone.

Search depth and performance. From the experiments, we note that the long search depth for checking MSR constitutes a major performance bottleneck. In [26], a case study on model checking a flight guidance system (FGS) is reported. FGS is encoded in a larger number of BDD variables, but shows better performance than that of MSR; one of the FGS models was encoded with 839 BDD variables and the peak number of nodes was 3,213,168. The major difference is that FGS requires less than 10 iterations for symbolic fixed-point computation, whereas MSR models require 37 - 53 iterations, because of the nested loops used in the MSR algorithm. The long search depth exacerbates memory and time complexity since reordering is performed at each search iteration.

To assess the effect of long search depth on verification performance, we remodeled the MSR environment such that the random writing environment is explicitly modeled

with non-deterministic random value assignments. Note that adding a random writing routine increases the complexity of symbolic model checking especially in terms of the search depth; for 5 LS's written in 5 PUs, each with 4 sectors, at least $5 \times 5 \times 4 = 100$ iterations are required to simulate random writing. Experiments show that a model with a random writing routine requires 169 iterations for fixed-point computation, spending 10 times more verification time than the model with invariants.

4.3 Data Abstraction

In our experiments, we have applied a simple data abstraction on the domain of LS's to avoid the state-space explosion problem. Though the original LS's range over integer domain, we can reduce the domain to a set of integer values where the total number of distinct values in the set equals to the number of logical sectors to read, because the MSR model as well as the *INV* property requires only equality checking between LS's and the read buffer. For example, 6 distinct values, e.g., $\{0,1,2,3,4,5\}$, would be enough for checking the *INV* property for MSR with 6 logical sectors.

In fact, two distinct values, $\{0,1\}$, for each data variable may be enough to check an equality condition $a = b$ since all possible combinations of values $\{(a, b) | a, b \in \mathcal{N}\}$ can be partitioned into two equivalent classes, $A = \{(a, b) | a = b\}$, $B = \{(a, b) | a \neq b\}$, and one representative value pair per each equivalent class, e.g., $(0,0)$ for A and $(0,1)$ for B, is enough to cover all possible cases. Nevertheless, we cannot reduce our data domain uniformly to $\{(0,0), (0,1)\}$ because of the constraints we have imposed on the MSR models; as mentioned in Section 4.1 we have imposed several constraints for setting the operational environment of MSR instead of explicitly modeling the sector writing routine for performance reasons. One example of such invariants, as introduced in Section 4.1, constrains that no two logical sectors contain the same data. For example, suppose $logical_sectors[0] = logical_sectors[1]$ and $logical_sectors[0] = PU[0].sect[0] \ \& \ logical_sectors[1] = PU[0].sect[1]$, then we can imply that $logical_sectors[0] = PU[0].sect[0] = PU[0].sect[1]$. This implies to a contraction, $SAM[0].offset[0] = 0 \ \& \ SAM[0].valid[0] = 1$ and $SAM[0].offset[0] = 1 \ \& \ SAM[0].valid[0] = 1$, by the invariants introduced in Section 4.1. Therefore, we need at least k distinct values to distinguish k logical sectors.

5 Model Checking MSR Using Spin

Due to common characteristics of Promela and C, creation of a formal Promela model from MSR is more convenient compared to NuSMV. Furthermore, the Promela model of MSR was semi-automatically generated by using Modex [15] which is a general purpose translation tool from C to Promela.

5.1 Model Translation

In a Promela model, the MSR environment (i.e., logical sectors, physical sectors and SAM) is specified such that all possible distributions of data into physical sectors are generated exhaustively through non-deterministic guarded commands.¹

¹ Note that this operational environment for model checking can be used to set up the actual testing environment. For more detail, see [21].

Modex [15] translates the control structure of MSR such as `if` and `while` into corresponding Promela control structures. Other C statements are inserted as embedded C code into the Promela model starting with a keyword `c_expr{...}` for Boolean expressions and `c_code{...}` for assignments and function calls [15]. As a result, the Promela model generated by Modex has the same 4-level nested loops as MSR does. The embedded C codes are blindly copied from the text of the Promela model into the code of the verifier that Spin generates.

In addition, we modified the original MSR C code to work correctly and efficiently under the Spin verification environment. For example, the linked lists of PUs and SAMs in the original MSR C code were replaced with arrays of PUs and SAMs. These modifications were performed through an explicit translation table given to Modex. Modex textually replaces C patterns in the table with the corresponding Promela codes specified in the table. The requirement property is specified by the assert statement `assert(logical_sectors[0]==buf[0] && logical_sectors[1]==buf[1]...)` located at the end of the MSR process.

5.2 Data Abstraction

In explicit model checking, all system states are stored in a huge hash table explicitly. Thus, data structures in the model do not incur extra overhead, since data structures are stored into the state vector as they are, not through complex BDD encoding.

MSR traverses a large amount of memory, most of which are taken by PUs and SAMs. The PUs and SAMs in embedded C code are *tracked* throughout the verification process, but are *not* stored in the state vector.² Instead, we add a new *signature* that represents the state of PUs and SAMs and put that signature into a state vector so that verification produces the correct result with this data abstraction. The signature is an ordered list of the physical locations of logical sectors (i.e. pairs of a PU number and a PS number). For example, the signature for Figure 4(a) is $\langle(0,1),(1,1),(1,2),(2,3),(3,0),(4,1)\rangle$ since ‘A’ is located at PU0’s PS1 and ‘B’ is located at PU1’s PS1 and so on. Therefore, there exists an one-to-one relation between signatures and states of PUs and SAMs and this abstraction preserves logical soundness according to the soundness theorem in [15].

Considering that the size of PUs is much larger than that of data, a significant amount of memory is reduced through this abstraction. For example, if 5 sectors long data is distributed over 10 physical units, the state vector should contain at least 80 ($=2 \times (10 \text{ PUs} \times 4 \text{ sectors})$) bytes for PUs and SAMs. The corresponding signature is 4 bytes long ($=5 \times ((\lceil \log_2 10 \rceil \text{ bits} + \lceil \log_2 4 \rceil \text{ bits}))$), resulting in a 95% reduction of the state vector size for PUs and SAMs.

5.3 Performance Analysis

We used Spin 4.3.0 on the same computing platform where NuSMV experiments were performed. We have performed two series of experiments with different lengths of data as well as different numbers of physical units. The first series of experiments are performed without the data abstraction described in Section 5.2 and the second series of

² This data abstraction on embedded C data is achieved through `c_track` keyword with `Unmatched` parameter. This feature has been available since Spin 4.1.

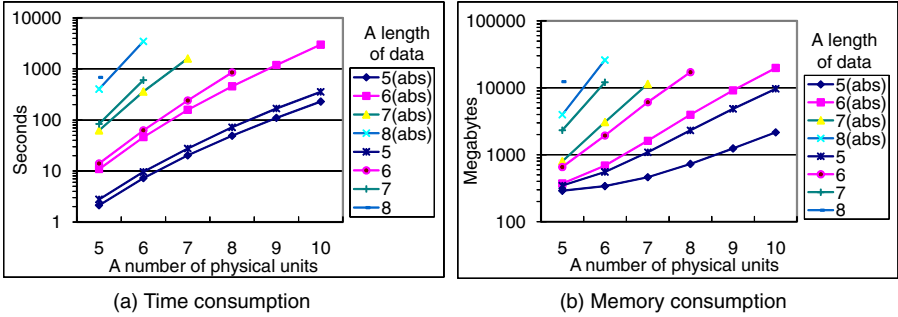


Fig. 6. Time and space complexities of Spin model checking

experiments are performed with the data abstraction³ Figure 6 illustrates performance data for checking the requirement property. In all of the experiments, Spin shows that the requirement property is satisfied.

For experiments without the data abstraction, Spin verified a flash containing 10 PUs and 5 logical sectors in 356 seconds, consuming 9.6 gigabytes of memory. In comparison, experiments with the data abstraction, denoted as “ $n(\text{abs})$ ” in Figure 6. Spin consumes 2.1 gigabytes of memory in 230 seconds, which reduces the memory consumption by 78% and the verification time by 35%, as shown in Table 1. Data abstraction reduces not only memory consumption but also verification time, since the time taken to store and retrieve state space is reduced as well.

Table 1. Memory and time reductions for 5 logical sectors due to the data abstraction

# of physical units	5	6	7	8	9	10
Memory reduction	17%	38%	57%	68%	74%	78%
Time reduction	23%	24%	26%	32%	34%	35%

As can be seen in Figure 6, the memory consumption and verification time increase exponentially in relation to the number of physical units. The bottleneck in this verification task is its memory consumption. Spin handles states explicitly, and thus the exponentially increasing number of possible distribution cases accordingly causes an exponential increase of memory. Compared to NuSMV, however, Spin is significantly faster for the verification tasks of this type. For example, for a case of 7 logical sectors and 7 physical units, Spin takes 27 minutes with 11.4 gigabytes with the data abstraction, while NuSMV takes more than 11 hours with 550 megabytes. In Spin, scalability on memory consumption is a larger problem while verification time is a more serious problem in NuSMV.

³ These experiments were performed without lossy compression such as bitstate hashing. For experiments without data abstraction, the `-DCOLLAPSE` option was used.

6 Model Checking MSR Using CBMC

CBMC [10] automatically translates a target C program into a corresponding SAT formula, then checks whether the C program satisfies a given requirement or not by solving the SAT formula through an external SAT solver. Although SAT is a NP-complete problem and a generated SAT formula can be huge including millions of Boolean variables and clauses, many structured problem instances can be solved in an acceptable time [2] with help of heuristics such as VSIDS or random restart [18].

6.1 Model Translation

CBMC does not need an explicit model translation, since it can directly analyze MSR C code. However, to obtain meaningful verification result, we have to build an environment that provides only valid configurations of a flash memory to MSR, as we did for NuSMV experiments. We specified an environment model using assume statements (`_CPROVER_assume (Boolean expression)`) and the environment model is similar to that of NuSMV experiments since both of them share most of the invariants (e.g. invariants in Section 4.1). In addition, a target C code was modified to use an array representation of SAMs and PUs for fair performance comparison with the other model checkers.

For the numbers of loop unwindings for bounded model checking, we can get a valid upper bound of each loop from the loop structures described in Section 3.

- The outermost loop iterates at most L times, where L is a number of LUs.
- The second outermost loop iterates at most 4 times, since one LU contains 4 LS's (i.e., $readScts \leq 4$) and at least one LS is read at each iteration.
- The third loop iterates at most M times where M is a total number of PUs.
- The innermost loop iterates at most 4 times, since one PU contains 4 PS's.

For example, $L = 2$ and $M = 5$ for Figure 4(a).

6.2 Performance Analysis

We used CBMC 2.6.0 (with MiniSAT 1.1.4 [11]) on the same computing platform where NuSMV and Spin experiments were performed. We performed a series of experiments with different lengths of data as well as different numbers of PUs. In all of these experiments, the requirement property specified by the assert statement `assert(logical_sectors[0]==buf[0] && logical_sectors[1]==buf[1]...)` is satisfied.

Figure 7 illustrates the performance results of this series of experiments. In addition, Table 2 enumerates the sizes of SAT instances of the MSR problems. For example, if 8 sectors long data is distributed over 10 PUs, then the corresponding SAT formula contains 8.6×10^5 Boolean variables and 2.9×10^6 clauses.

Compared to the results of the Spin experiments, CBMC demonstrates better performance in both verification time and memory consumption for large problem instances. For example, when data is 7 sectors long and the number of PUs is 7, CBMC takes 203 seconds and consumes 148 megabytes of memory while Spin with the data abstraction takes 1604 seconds and consumes 11.4 gigabytes of memory. CBMC demonstrates

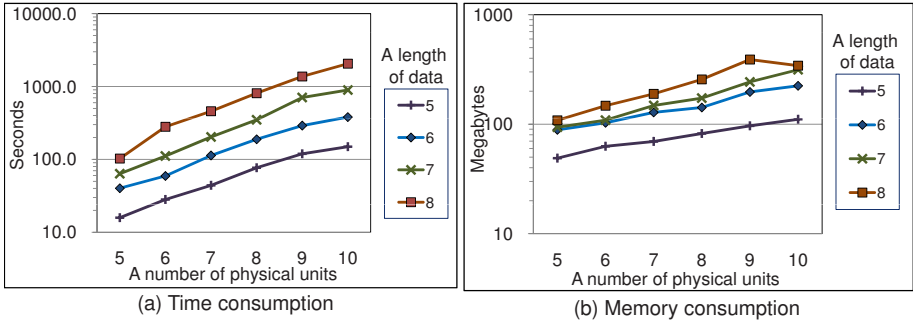


Fig. 7. Time and space complexities of CBMC model checking

Table 2. The sizes of the SAT CNF instances of MSR with different configurations

$\times 10^5$	5		6		7		8		9		10	
	var	clause	var	clause	var	clause	var	clause	var	clause	var	clause
5	1.9	6.3	2.4	7.7	2.8	9.2	3.2	11	3.7	12	4.2	14
6	3.6	12	4.4	15	5.3	17	6.2	21	7.1	24	8.0	27
7	3.8	12	4.6	15	5.5	18	6.4	21	7.4	25	8.3	28
8	3.9	13	4.8	16	5.7	19	6.7	22	7.6	26	8.6	29

better performance compared to NuSMV for large problem instances, too⁴; NuSMV takes 11 hours and consumes 550 megabytes of memory for the problem of the same size. Detailed analysis on these CBMC based experiments will be reported in a separate article.

7 Discussion

In this section, several issues are discussed on the basis of our experience in this project.

7.1 Application of Model Checking in Industrial Software Projects

Formal verification techniques are being evaluated for experimental purposes for software projects in major electric device manufacturers such as Samsung, as a complement of software testing, which has thus far been *the* software verification technique. Samsung performed the majority of testing for the OneNAND device driver randomly, which does not provide sufficient coverage for detecting bugs even with a huge number of test cases, since there are astronomically many possible scenarios (see Section 3). Even with a scalability limitation, this OneNAND verification project was evaluated as a success as it confirmed the correctness of the MSR, which could not be assumed

⁴ We could not use the SAT-based bounded model checking capability of NuSMV in this project, since the smallest problem instance took more than 3 hours.

through testing, for a small flash memory; exhaustive exploration through model checking provides high confidence in the correctness of MSR.

7.2 Advanced Abstraction Techniques

We have applied a basic data abstraction (e.g. a fixed string “ABCDE” serves for all 5 sectors long strings) described in Section 3 to reduce state space of MSR. In addition, we used weaker invariants to model the environment of MSR with reduced complexity, which does not produce false positives (see Section 4.1). However, more aggressive abstractions based on symmetry turned out to be hard to apply for this project, since we have to validate several assumptions on the MSR code to exploit symmetry, which requires inductive proof techniques which are beyond the scope of the project.

For the experiments in NuSMV, only primitive data type reductions are applied to reduce the search space. Other aggressive abstraction techniques, such as predicate abstraction [16], counter-example-guided abstraction-refinement (CEGAR) [7], and temporal case splitting [19], might be applied to enhance the performance of the NuSMV verification. Nevertheless, their effectiveness on MSR is questionable due to the following reasons:

1. A simple trial experiment shows that case-splitting does not always improve the verification performance for MSR. For example, we have checked each sub-case of the *INV* property, $after_MSR \rightarrow logical_sector[i] = buf[i]$, separately for each i , where $0 \leq i \leq 4$ and a flash has 7 PUs. The verification time required for the case $i = 1$ is about 840 seconds. Note that the time required for checking the original *INV* property for the same setting is only 530 seconds.
2. The predicate abstraction or CEGAR approach can be effective when checking the safety or compatibility of a software module whose unrelated branches of control flow and/or large data domains can be safely abstracted away. In the case of MSR, however, each control branch contributes to its functional correctness, leaving small chances for reducing complexity by abstraction. We also note that the data domain in MSR has already been reduced to a minimal set (e.g. $\{0,1,2,3,4\}$ for 5 sectors long data) for the verification using NuSMV.

For the Spin experiments, besides the one-to-one data abstraction described in Section 5.2, there exist other data abstraction techniques that have yet to be applied. For example, [17] proposes a memory-efficient hash table that implements a sophisticated hash table structure which uses less memory at the cost of operation time overhead. To apply this technique, however, we should modify the Spin source code to change its hash table structure, which is beyond the scope of the project.

For the CBMC experiments, there exist not much room to apply aggressive abstraction to MSR itself, since the MSR C code should not be modified much for abstraction purpose. However, the environment model of MSR can be built in an efficient way. For example, the environment model can be built using weaker invariants to reduce complexity (see Section 4.1).

7.3 Scalability of Model Checking

Even with the abstraction techniques noted in Section 7.2, it is not clear that model checking can be an overall solution for verifying software similar to MSR. For software handling a large data structure, model checking has an inherent scalability limitation, since the size of the environment to be modeled and analyzed is extremely large, even with symmetry reduction or state partitioning. Thus, for software similar to MSR, it still seems appropriate to utilize human expertise through a theorem prover, constraint solving, or inductive proofs. For this aspect, invariant based modeling is preferable since invariant based model can be adapted to specify a target system for theorem proving. We are planning to study this problem further by using a theorem prover.

8 Conclusion and Future Work

We have shown that a difficult verification problem in industrial software can be tackled using automated formal verification tools. Though the project was conducted on a small-scale, Samsung highly valued the verification result. It was also confirmed that comprehensive verification techniques have feasible use and that they can complement testing or provide an alternative solution. This has motivated our next project; we plan to analyze a flash file system to check data consistency at the events of random power-off.

At the same time, we could understand relative strengths and weaknesses of the three popular model checking techniques empirically - BDD-based one, explicit one, and SAT-based one. We will analyze the experimental results of CBMC further, since CBMC demonstrated the best verification performance in this project and requires minimal verification effort due to the capability of verifying C code. The experience gained in this project also led the authors to realize the practical limitations on the scalability of model checking and the necessity of conducting further research to address the issue through the use of smart abstraction techniques and/or by utilizing human expertise.

References

1. Samsung OneNAND fusion memory, http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html
2. SAT competition 2007: a satellite event of the SAT 2007 conference (2007), <http://www.satcompetition.org/2007/>
3. Bollig, B., Wegener, I.: Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers* 45(9) (September 1996)
4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: 13th International Static Analysis Symposium, pp. 52–70 (2006)
5. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. *IEEE Transactions on Software Engineering* 24(7), 498–520 (1998)
6. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: *Proceeding of International Conference on Computer-Aided Verification* (2002)

7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the 12th International Conference on Computer Aided Verification, July 2000, pp. 154–169 (2000)
8. Darga, P.T., Boyapati, C.: Efficient software model checking of data structure properties. In: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (2006)
9. Dong, Y., Du, X., Holzmann, G.J., Smolka, S.A.: Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer* (4) (2003)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
11. 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)
12. Eisner, C., Peled, D.: Comparing symbolic and explicit model checking of a software system. In: SPIN Workshop (2002)
13. Ball, T., et al.: Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review* 40(4), 73–85 (2006)
14. Holzmann, G.J.: *The Spin Model Checker*. Wiley, New York (2003)
15. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: Spin Workshop (2004)
16. 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)
17. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Spin Workshop (2003)
18. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: *Computer Aided Verification* (2002)
19. McMillan, K.: Verification of infinite state systems by compositional model checking. In: *Conference on Hardware Design and Verification Methods* (1999)
20. Miller, S.P., Tribble, A.C., Whalen, M.W., Heimdahl, M.P.E.: Proving the shalls: Early validation of requirements through formal methods. *International Journal on Software Tools for Technology Transfer* 8(4), 303–319 (2006)
21. Kim, M., Kim, Y., Choi, Y., Kim, H.: Pre-testing flash device driver through model checking techniques. In: *IEEE Int. Conf. on Software Testing, Verification and Validation* (2008)
22. Monniaux, D.: Verification of device drivers and intelligent controllers: A case study. In: 7th ACM and IEEE international conference on Embedded Software (2006)
23. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: *International Conference on Computer-Aided Design (ICCAD)* (November 1993)
24. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. *Software Tools for Technology Transfer* (2008)
25. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: *Automated Software Engineering* (November 2007)
26. Choi, Y.: From NuSMV to SPIN: Experiences with model checking flight guidance systems. *Formal Methods in System Design*, 199–216 (2007)

Layered Duplicate Detection in External-Memory Model Checking

Peter Lamborn and Eric A. Hansen

Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
pcl116@msstate.edu, hansen@cse.msstate.edu

Abstract. This paper presents a disk-based explicit-state model checking algorithm that uses an approach called layered duplicate detection. In this approach, states encountered during a breadth-first traversal of the graph of the transition system are stored in memory according to the layer of the graph in which they are first encountered. With this layered organization of memory, transition locality is exploited by checking only the most recent layers for duplicates. In RAM, exploiting transition locality in this way saves time. In external memory, it saves space. In addition, a layered structure allows an easy method of counterexample reconstruction in disk-based model checking. We prove a worst-case linear bound on the redundant work performed by our approach. Experimental results indicate that average case redundant work is much better than the worst-case. The implemented model checker has been used to verify a transition system that required more than 275 GBs of disk storage.

1 Introduction

Explicit-state model checking is a methodology for verifying the properties of a system by a systematic search for error states in a state graph that represents the behavior of the system. An “on-the-fly” algorithm for searching the state graph begins with a queue that contains the start state, removes states from the queue in an order that depends on the search strategy (e.g., depth-first, breadth-first, or best-first), generates the successors of each state and adds them to the queue. Generated states are also stored in a hash table that is used for duplicate detection, that is, for determining whether or not a newly-generated state is a duplicate of a previously-generated state before adding it to the queue. A complete search of the graph is needed to verify a system. Therefore, the memory required is proportional to the number of states in the model. Thus, memory is a bottleneck for model checking.

One solution is to use external memory. When RAM is full, states can be moved to an external memory device. Delayed duplicate detection (DDD) is an approach to external memory graph search that writes duplicates temporarily to disk and eventually eliminates them [1,2,3]. Because disk is accessed most quickly

in a sequential manner, allowing temporary duplicates can save time compared to accessing disk randomly to eliminate the duplicates immediately. But DDD still incurs significant overhead.

When searched in a breadth first manner, most graphs exhibit transition locality [4]. Transition locality means that transitions tend to be in local levels of a breadth first visit [5]. In other words, newly-generated states are more likely to be duplicates of a state in a recent layer than a state in a more distant layer. We introduce *layered duplicate detection* as an approach to exploiting transition locality in eliminating duplicates in external-memory search. By layered duplicate detection, we mean storing the search graph in layers and only saving and checking the most recent layers for duplicates. This approach is useful both in deciding which states to store in a hash table in RAM, and in deciding which states to store on disk. We show that layered duplicate detection makes it possible to improve the running time of an external-memory model checker, as well as decrease the amount of disk storage it needs.

The paper is organized as follows. Section 2 gives an overview of relevant background and related work. Section 3 presents the new algorithm as an extension of the Mur ϕ model checker [1]. Section 4 reports experimental results. Section 5 concludes the paper and discusses potential future directions.

2 Background

There are many approaches to addressing the state explosion problem in model checking. Two of the most relevant approaches for this paper are partial storage methods and external memory model checking.

2.1 Partial Storage Methods

Graph-search algorithms avoid redundant search of the same parts of a state-space graph by storing already-visited states in a hash table and checking each newly-generated state to see whether it is a duplicate of an already-visited state. The set of states that has already been visited is referred to as the *closed set* of the search algorithm. When the closed set is too large to fit in RAM, one solution is to store only part of it. Since this allows redundant search, it presents a time-space tradeoff. The amount of redundant search can be limited by an intelligent choice of which states to keep in the hash table and which to remove.

We distinguish three types of partial storage methods: depth-first search (DFS) approaches, breadth-first search (BFS) without generation of duplicates, and BFS with generation of duplicates. Depth-first search has the advantage of always having a complete counterexample at the time of discovering an error state, but it is not guaranteed to find the shallowest error state [6]. Breadth-first search has the advantage of always finding the shallowest error state, if an error exists. But if the closed set is only partially stored in the hash table, part of the counterexample may not be stored in RAM. Because of this, partial storage methods were originally developed for use with depth-first search. But reconstruction of the counterexample may be possible in other ways. For example, if

states that do not fit in the hash table are stored on disk, reconstruction of the counterexample is still possible.

Depth-First Approaches. Godefroid et al. [7] describe a model checker that uses depth-first search combined with *state caching*, which refers to partial storage of the closed set in RAM. When a new state is generated and the hash table is full, an already stored state is selected for removal to make room for the new state. Geldenhuys [8] extends this approach with a technique called *stratified caching* that caches states in certain ‘strata’ based on distance from the start state. This makes it possible to bound the number of duplicates generated during the search.

Breadth-First Search without Generation of Duplicates. Using breadth-first search, there are methods that delete states from the hash table in such a way that no duplicates can be generated. These methods exploit special properties of certain search graphs to determine which states are necessary to cache, and which can be deleted from the hash table without risking redundant search. The sweep-line method developed by Kristensen and Mailund exploits ordering in models where states can be ordered, either with a total ordering [9] or a partial ordering [10]. Parshkevov and Yantchev [11] consider graphs where all states have the same number of parent states, and remove a state from the hash table once all its parents have been observed. These methods are domain-dependent because not all graphs have properties that can be exploited in this way.

Breadth-First Search with Generation of Duplicates. A more general approach to partial storage of the closed set in breadth-first search allows generation of duplicate states, but uses various methods to limit the number of duplicates.

Hash Collisions. Tronci et al. [5] use state caching in a breadth-first search algorithm for model checking. When two states (s, s') collide in the hash table, the older state (s) is removed and the newer state (s') is remembered. This replacement strategy tries to keep the most recently-generated states in the hash table and thus exploits the property of transition locality [4]. Because the state cache is partial, states may enter the open queue multiple times. Therefore, the open queue tends to be larger than a standard BFS queue. To accommodate a larger open queue, Tronci stores the queue on disk and swaps it in and out as needed.

Layered duplicate detection. Although they do not consider model checking, Zhou and Hansen [12] describe a partial storage method for breadth-first search called layered duplicate detection. Instead of storing all of the layers of a breadth-first search graph in a hash table, they store only the most recent k layers, where k is determined by the structure of the graph and available memory. Although some duplicates may be generated, their approach allows a theoretical bound of the number of duplicates which is $\frac{d}{k}$, where d is the height of the graph and k is the

number of layers cached. In this paper, we extend their approach and apply it to model checking.

2.2 External-Memory Search

Using external memory dramatically increases the maximum number of states that can be stored for use in duplicate detection, and allows much larger state spaces to be completely explored. Because it is impractical to implement a hash table on disk and immediately check whether each newly-generated state is a duplicate, external-memory search relies on an approach called delayed duplicate detection (DDD) [1].

External-Memory Bucketing. Bao and Jones [13] found that state comparisons, not disk I/O, takes the majority of time in external-memory search. In a brute-force external-memory search, DDD takes $O(mn)$ time, where m is the number of states on the disk and n is the number of *candidate* states to be checked. One way to improve the efficiency of external-memory model checking is to split the state space into several chunks called buckets. All duplicate states are assigned to the same bucket, and therefore DDD only needs to be performed within buckets. This reduces the time complexity of DDD to $O(\sum m_i n_i)$, where m_i is the number of states previously assigned to bucket i and n_i is the number of *candidate* states in bucket i . In most cases, $O(\sum m_i n_i) \ll O(mn)$. Several different methods of bucketing the states can be employed, including hash functions [2], heuristics [14], transition locality [15,16,17] and graph structure [18]. All of these algorithms speed the search by reducing the number of state comparisons that need to be performed in DDD.

Transition Locality in External-Memory Model Checking. Some external-memory model checkers perform DDD on only the most recent states, in an attempt to exploit transition locality.

Locality in state caching. Hammer and Weber [19] describe an external-memory search algorithm that takes its cue from partial storage algorithms. The algorithm uses a function to guess which states are less likely to be duplicated and swaps them to external memory. They tried several selection functions and the ones that performed best used a state's age as part of the criteria, with older states more likely to be sent to disk. However Hammer and Weber's algorithm is not a true BFS because of its special treatment of single successor states.

Locality in DDD. Della Penna et al. [15,16,17] extend Tronci's [5] cached-memory model checker so that it uses external memory. When a successor state is generated, it is checked immediately for membership in the hash table. When there is a hash collision, the oldest of the two colliding states is written to disk. External memory is partitioned by the order states are written. DDD is usually performed on only the most recent portion of the states in external memory. Occasionally a full DDD is performed to removed duplicates missed in the partial DDD. By

checking the *candidate* states against only the most recently written parts of the closed set, Della Penna et al. attempt to exploit transition locality in external memory.

3 Algorithm

The starting point for our implementation is the Mur ϕ model checker, a domain-independent tool that takes a description of a model as input and uses breadth-first search to verify that the model is correct [20]. If an error is found, it returns an error trace. The Mur ϕ model checker uses breadth-first search to explore the state-space graph of the model. In the following, we describe an implementation of external-memory breadth-first search that we implemented in Mur ϕ .

3.1 Breadth-First Search with Layered Duplicate Detection

Figure 1 gives the pseudocode of an implementation of breadth-first search that uses layered duplicate detection in checking for duplicates in both the hash table in RAM and on disk. The algorithm partitions the closed set of states into buckets. The buckets have two indices. The first is a hash function that distributes states approximately equally among buckets. The second is the depth of the layer of the breadth-first search graph in which the state first appears. The second index makes it possible to use layered duplicate detection. As discussed in Section 2.2, partitioning the closed set into buckets improves the speed of DDD, although there is some overhead for increased usage of external memory.

In the pseudocode, *ClosedRAM* denotes the portion of the closed set that is stored as a hash table in RAM and *ClosedDisk* denotes the closed set that is stored on disk. *Open* denotes the queue of generated states in the current layer of the search graph waiting to be expanded, that is, waiting to have their successor states generated. *Candidates* is the set of newly-generated states in the next layer of the search graph. Both *Open* and *Candidates* may be too large to fit in RAM, in which case they are partially stored on disk. *Open* is read sequentially from disk and *Candidates* is written sequentially to disk, and so storing them partially on disk is no problem. Since duplicate detection on disk is delayed, *Candidates* typically contains duplicates that are removed later.

The algorithm proceeds as follows. While the search graph has not yet been fully explored (line 8), it loops through all buckets (line 9) and through all states in each queue of the current layer of the search graph (line 10). For each dequeued state, its successor states are generated (line 12). If RAM is full, the shallowest layer of states in the hash table is deleted (line 16). The reason for this is that, assuming transition locality, states generated earlier are less likely to have duplicates in the layer of the search graph currently being generated. When layers are large, however, it is sometimes necessary to delete states from the hash table that are in the layer of the search graph that is currently being generated (line 17). If a generated state is not in the hash table, it is added to both the hash table and to the candidate list (line 19). If a violation of the

```

1  externalBreadthFirstSearch(startState, ErrorStates)
2    depth := 0
3    cacheStartingLayer := 0
4    s := startState
5    ClosedRAM[hash(s)][depth].add(s)
6    ClosedDisk[hash(s)][depth].add(s)
7    Open[hash(s)][depth].enqueue(s)
8    while( $\neg \forall_b$  Open[b][depth].empty())
9      for(each bucket b)
10       while( $\neg$ Open[b][depth].empty())
11         s := Open[b][depth].dequeue()
12         for( each successor s' of s )
13           if(s'  $\in$  ErrorStates) return counterExampleReconstruction(s')
14           ;If RAM is full, delete shallowest layer of state cache
15           if(RAM.FULL())
16             for ( each bucket b )
17               delete ClosedRAM[b][cacheStartingLayer]
18               if(depth + 1 > cacheStartingLayer) cacheStartingLayer ++
19               ;If node is not in state cache, add to state cache and disk
20               if( $\forall_{i=cacheStartingLayer}^{depth+1}$  s'  $\notin$  ClosedRAM[hash(s')][i])
21                 ClosedRAM[hash(s')][depth+1].add(s')
22                 Candidates[hash(s')][depth+1].add(s')
23           depth++
24       removeDuplicatesOnDisk(depth, cacheStartingLayer)
25   return Model is valid

```

Fig. 1. Pseudocode for external-memory breadth-first search algorithm that uses layered duplicate detection to check for duplicates of states stored in a hash table or disk

property is found, the counterexample is returned (line [13](#)). After all the nodes in the currently layer of the search graph are expanded, delayed duplicate detection is performed to remove duplicate states from *Candidates* (line [22](#)).

3.2 Delayed Duplicate Detection

The pseudocode of a procedure that performs delayed duplicate detection to remove duplicates of states already stored on disk is shown in Figure [2](#). The procedure removes states from *Candidates* that are duplicates of other states in *Candidates* (line [11](#)) or duplicates of states that are already stored on disk. States that are not detected as duplicates are put into the *Open* queue and are also stored in *ClosedDisk* (line [12](#)). The pseudocode of a Boolean function that checks whether a state is a duplicate of a state already stored on disk is given in Figure [3](#). The number of previous layers checked for duplicates is controlled by the parameter L .

Delayed duplicate detection is the most time-consuming part of the search algorithm, and takes even more time than disk I/O, which is notoriously slow [13](#). One way to reduce the time it takes is to sort all states stored in external memory. The complexity of an external sort is $O(n * (1 + \lceil \log_{B-1} \lceil \frac{n}{B} \rceil \rceil))$, where n is

```

1  removeDuplicatesOnDisk(depth, cacheStartingLayer)
2  for ( all buckets  $b$  )
3     $l :=$  cacheStartingLayer -  $L$ 
4    while (  $l <$  cacheStartingLayer )
5      offset[ $l$ ]=0
6       $l ++$ 
7    sort Candidates[ $b$ ][depth]
8     $e := 0$ 
9    while (  $e <$  Candidates[ $b$ ][depth].size()
      ;Next line skips duplicate nodes in Candidates
10     if(Candidates[ $b$ ][depth].element( $e$ ) $\neq$ Candidates[ $b$ ][depth].element( $e - 1$ ))
      ;Next line skips duplicates of nodes already stored on disk
11     if( $\neg$  inPreviousLayers(Candidates[ $b$ ][depth].element( $e$ ),  $b$ , offset)
12       Open[ $b$ ][depth].enqueue(Candidates[ $b$ ][depth].element( $e$ ))
13       ClosedDisk[ $b$ ][depth].add(s)
14      $e ++$ 
15    delete Candidates[ $b$ ][depth]

```

Fig. 2. Pseudocode for procedure that removes duplicate nodes on disk using delayed duplicate detection

```

1  boolean inPreviousLayers( $s$ ,  $b$ , offset)
2  layer := cacheStartingLayer - 1
3  while ( layer  $>$  cacheStartingLayer -  $L$  )
4    while  $s <$  ClosedDisk[ $b$ ][layer].element(offset[layer])
5      offset[layer] ++
6    if  $s ==$  ClosedDisk[ $b$ ][layer].element(offset[layer])
7      return true
8    layer - -;
9  return false

```

Fig. 3. Pseudocode for function that checks whether a state is stored in the L most recent layers in external memory

the number of *candidate* states, m is the number of states in the bucket and B is the number of states cached by the external sort. But when the buckets are sorted, DDD takes $O(n + m)$ time instead of $O(nm)$.

Sorting the states in each bucket makes DDD more efficient for three reasons. First, all duplicates within a bucket are stored consecutively once sorted. Thus, duplicates in the current bucket can be found by simply comparing a state to its neighbors. (See Figure 2 line 10.) Second, when comparing *Candidates* against previous layers one state at a time (Figure 3 lines 4 and 6), if a state is reached that sorts greater than the *candidate* state, we know the *candidate* has no match in that layer and do not have to check any more states in that layer. Third, because *Candidates* is sorted, the next *candidate* state checked will sort greater than the current *candidate*. Thus we can resume checking the layer from the same location, recorded in the offset array (line 5), rather than starting at

the beginning of the layer. In the delayed duplicate detection procedure, we load a state from external memory into RAM no more than once.

3.3 Counterexample Reconstruction

The pseudocode for reconstructing an error trace is given in Figure 4. In a standard RAM-only search, the data structure for each state includes a pointer back to its parent state, that is, the state that generated it. Reconstructing an error trace is simply a matter of tracing these pointers back to the start state. In external-memory search, some states on the error trace may no longer be in RAM. This makes reconstructing the error trace more complicated. Instead of including a parent pointer in the data structure for each state, we include two items of information: the parent's hash, which determines which bucket the parent is in, and the parent's location in the bucket (line 8). With these two items of information, we can find the parent state whether it is located in RAM or disk.

```

1  counterexampleReconstruction(error)
2  trace:=∅
3  traceDepth:=depth
4  cur:=error
5  while(traceDepth>cacheStartingLayer)
6    trace:=trace & cur
7    traceDepth - -
8    cur:=ClosedRAM[cur.parentHash][traceDepth].element(cur.parentPosition)
9  while(traceDepth>0)
10   trace:=trace & cur
11   traceDepth - -
12   cur:=ClosedDisk[cur.parentHash][traceDepth].element(cur.parentPosition)
13  trace:=trace & cur
14  return trace

```

Fig. 4. Pseudocode for procedure that reconstructs error trace

3.4 Termination

Because the search is breadth-first, if an error is found, the error trace is guaranteed to be the shortest path to the error state. If an error exists, our algorithm will always find it. If no error is found, the search terminates when there are no more states of the graph to explore, i.e., the open queue is empty (Figure 1, line 8). If the search terminates in this way, the model is verified. However, when only the most recent layers are checked for duplicates, the search is no longer guaranteed to terminate with an empty open queue, even if the graph is finite. It is possible, when no error states exist, for the search to continue indefinitely because missing duplicate states result in regeneration of parts of the search space. Thus, our algorithm, like many partial-memory algorithms, is not guaranteed to terminate. Answers produced are always correct, but in some cases no answer may be produced.

This suggests another way to detect termination and verify a model. If the model is verified for a depth that is equal to or greater than the diameter of the graph, called the *completeness threshold*, verification is complete. This method of detecting termination is similar to that used in bounded model checking, which uses satisfiability testing to verify that a model does not have any errors up to some depth k [21]. Various methods for determining a completeness threshold have been explored in the literature on bounded model checking, and can be applied in our layered approach.

4 Results and Analysis

We implemented layered duplicate detection in the Mur ϕ model checker, which uses a state cache to immediately eliminate as many duplicates as possible and uses delayed duplicate detection to eventually eliminate duplicates that are written to disk. Our experiments were performed on a 3.0 GHz dual-core Xeon processor with two gigabytes of RAM and one terabyte of disk storage.

Table 1. Performance results using layered delayed duplicate detection, with a maximum of 50 layers stored on disk. The column *Num states* shows the number of unique states of the model that are visited during the search. The column *Num layers* shows the depth of the search. The column *RAM* records the number of duplicates eliminated in RAM, and the column *DDD* shows the number of duplicates written temporarily to disk and later eliminated by DDD. The column *Missed* shows the number of duplicates missed by our algorithm because not all layers of the search graph are stored on disk. The column *BFS* shows the CPU time for the search, and the column *Trace* shows the CPU time to reconstruct the error trace. Time is reported as Days:Hours:Minutes:Seconds. The column *Disk* shows the amount of external memory required to store the complete state space, reported in gigabytes.

Model	Num States	Num Layers	Duplicates			Time		Disk GB	
			RAM	DDD	Missed	BFS	Trace		
Verified									
newlist	80,109,359	110	346,175,576	116,341,341	288,695	14:35:00	N/A	22	
ldash	254,935	64	2,391,696		7	2	3:28:42	N/A	41
directory	1,071,401,426	113	479,609,483	78,501,871		28	1:22:16:29	N/A	279
Error Detected									
adashe	2,156,280	16	3,329,518	169,981	0	4:02:17	7:39	5	
arbiter	90,913,223	31	378,993,079	64,426,284	0	6:17:41	1	26	

The results reported in Table 1 show the performance of the algorithm in verifying five models. Three models were verified. For the other two models, an error was found and the error trace was returned. For all of the models except *ldash*, the largest layer of the search graph exceeded RAM capacity; this means that the state cache contained less than one layer at times. A maximum of 50 layers of the search graph were stored on disk at a time to check for duplicates.

Table 1 shows the number of unique states of the model. Because the algorithm checks a bounded number of layers for duplicates, and thus could miss some duplicates, the number of unique states of the model was determined by a post-processing step.

Despite often caching less than a layer in RAM, our results show that more duplicates are detected in RAM than there are unique states in the model. Fewer duplicates are eliminated in DDD; this is good, since DDD is relatively expensive. The number of duplicates missed because not all layers are stored on disk is typically small, and much less than the number eliminated by DDD. We report the amount of time taken by the algorithm, broken down by how much time was spent on the search and how much time was spent reconstructing the error trace if an error state was discovered. Counterexample reconstruction usually takes a small portion of the total time. The counterexample reconstruction for *arbiter* is unusually fast; in this case, the error trace is sorted near the beginning of the disk files (i.e., the buckets in *ClosedDisk*). Finally, we report the amount of gigabytes required to completely store the state space. While all these models exceed the RAM we have available, the *directory* model is particularly large, requiring 279 GB to completely store the state space.

4.1 Effective State Caching Saves Time and Space

Having an effective state cache saves both time and space. Table 2 shows that without a state cache, there is a large increase in the number of duplicates written temporarily to disk and eliminated during DDD, as well as a corresponding increase in the running time of the model checker. In addition, writing these duplicates to disk increases the amount of disk storage required by the algorithm.

Table 2. Performance with and without a layered state cache. The column *Time* shows the overall time to perform the search, and is reported in Hours:Minutes:Seconds. The column *Delayed Duplicates* shows the number of duplicates eliminated during DDD.

Model	Cache	Time	Delayed Duplicates
ldash	state Cache	1:02:42	7
	No Cache	9:03:25	1,536,918
adashe	state Cache	4:02:17	172,184
	No Cache	6:12:45	3,459,785
arbiter	state Cache	6:17:41	66,980,652
	No Cache	22:15:19	413,937,369

4.2 Layer Sizes

Figure 5 shows the size of each layer of the search graph for the *directory* model. As Pelánek [4] earlier observed, the distribution of layers usually has a near bell curve shape, with most states in the middle layers. For this model, 40% of the states are in layers 62 to 72, out of a total of 113 layers. In fact, layer 66 alone

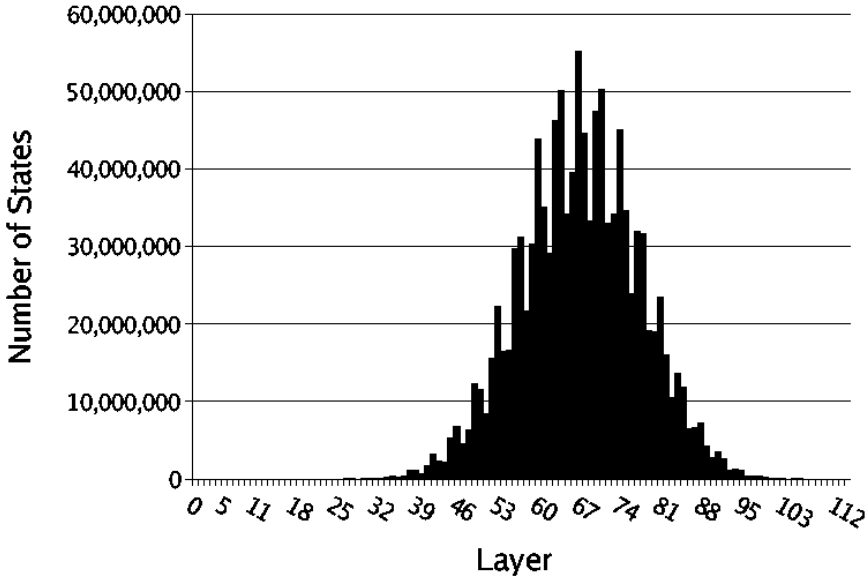


Fig. 5. The number of states at each depth of the *directory* model

contains almost 5% of the states in the model. Although disk provides much more storage than internal memory, it is also limited. In our experiments, we have not yet reached the limits of our available disk storage, but increasingly large models will test these limits. For example, Korf and Schultze [3] describe a complete breadth-first search of the *Fifteen Puzzle* that requires 1.4 terabytes of disk just to store the largest single layer of the graph. When model checking considers problems that approach this size, it will become important to conserve external memory. One way to economize in the use of external memory is by using our partial DDD algorithm. Since our algorithm only checks a subset of all previous layers during DDD, it would use less disk storage if it only keeps this subset of layers on disk.

4.3 Transition Locality in Partial Delayed Duplicate Detection

Table 3 shows the result of running our algorithm on three large models and varying the number of layers checked in DDD. If too few layers are checked, large numbers of duplicates are never eliminated, increasing the running time of the algorithm. It is possible to check more layers than necessary. This usually doesn't increase the running time of the algorithm much since most duplicates are caught before checking the older layers. For these models, the best space-time performance is achieved by checking only a bounded number of layers, but the optimal number of layers is model-dependent and cannot be known *a priori*.

Table 3. Tradeoffs for checking a bounded number of previous layers for duplicates, as the number of layers is varied. The column *Layers* gives the number of previous layers used in DDD. The column *CPU Time* gives the amount of time the algorithm took, reported in Days:Hours:Minutes:Seconds. The column *Disk Storage* gives the maximum amount of external memory required for the algorithm to complete the search, reported in gigabytes.

Layers	CPU Time	Disk Storage
adashe		
15	4:02:17	5.00
10	3:48:25	4.99
5	3:53:46	4.56
0	4:04:00	2.04
ldash		
46	1:07:15	40.88
26	1:03:51	37.58
10	1:03:34	20.50
8	1:01:43	17.09
6	1:11:05	13.39
5	1:12:46	11.41
4	2:21:06	9.44
2	>8:34:22	>6.23
newlist		
88	19:22:47	22.00
66	19:15:24	21.96
44	19:51:39	21.70
22	>2:19:58:30	>8.41

4.4 Model Descriptions

All of our models are based on complex protocols. The following explains the protocols used to create the models.

- 1 *arbiter*: model of a mutual exclusion algorithm with an arbiter that allocates resources [22]. We used a model with 13 resources.
- 2 *dash*: model of the Dash Communication Protocol [23]. This protocol has two variant models.
 - *adashe*: abstraction of the dash protocol. This variant has been changed to include error states. *Adashe* was tested on a model that had 1 cluster with memory, 2 clusters without memory, 2 address in the memory cluster, and 2 value types to be saved.
 - *ldash*: concrete model of the dash protocol. We tested a model with 1 cluster with memory, 4 clusters without memory, 1 address per cluster could be locked, with message buffer of size 30.
- 3 *directory*: model of a directory-based cache protocol [24]. This is a model created by IBM and used for evaluating the protocol for use in servers. We tested a model with 14 clients.

- 4 *newlist*: protocol for a distributed linked list that covers maintaining coherency in the list and controlling adjustments as the list is sorted. We used a system with 8 distributed nodes.

4.5 Theoretical Results

We next prove some bounds on the number of duplicates that can be generated when not all all layers of the search graph are stored and checked for duplicates. First, we bound the number of duplicates that are never eliminated in the search. We call these redundant states. Then we bound the number of duplicates that will be eliminated in delayed duplicate detection. In our experiments, observed performance is much better than these worst-case bounds.

Bounds on Redundant States. Zhou and Hansen [12] give some conditions under which no duplicates will be generated, using layered duplicate detection. In the worst-case, the number of times a state can be regenerated is bounded by the depth of the search (d) and the number of layers checked in DDD (L), and no state can be duplicated more than d/L times.

Theorem 1. *In layered external memory search, the worst-case number of times a state can be regenerated is bounded by $\frac{d}{L}$.*

Proof. Let $L \geq 1$ be the number of layers checked by the algorithm. No duplicates can exist in these L layers. New states are checked against those L layers before being inserted into the open list. The least layers a duplicate state s can reappear is at layer $g^*(s) + L$ where $g^*(s)$ is the first layer at which state s is encountered. State s may then appear every L layers for the rest of the search. If the depth of the search is d , the state s can appear a maximum of j times where $g^*(s) + j * L \leq d$. Thus, state s may recur at most $\frac{d}{L}$ times, which assumes the state appears in layer 0 and every L layer thereafter.

Theorem 2. *In layered breadth-first search, the worst-case number of number of states that can be regenerated is bounded by $\frac{S*d}{L}$.*

Proof. This is a small step from Theorem 1. Since a state s can be duplicated no more than $\frac{d}{L}$ times and there are S states, no more than $\frac{S*d}{L}$ states can be searched. Since searching $\frac{S*d}{L}$ states would mean that every state in the graph is duplicated the maximum times allowed by the bound in Theorem 1.

Note as well if $L = d$ then S states are searched, i.e. each state is searched once. Since $L = d$ in a traditional BFS this result is appropriate.

Theorem 3. *In layered breadth-first search, the worst-case redundant work factor is bounded by $\frac{d}{L}$.*

Proof. The redundant work factor is a metric used by Geldenhuis [8]. The metric is the ratio of reported states to actual states. Actual states are S . Reported states are bounded by Theorem 2 to $\frac{S*d}{L}$. Thus, the redundant work factor is bounded by $\frac{\frac{S*d}{L}}{S}$ which reduces to $\frac{d}{L}$.

Therefore, according to Theorem 3, if $L = d$, the redundant work factor is 1, which is, again, what we would expect from a traditional search. In the worst-case $L = 1$ making $\text{rwf} = d$.

Bounds on Delayed Duplicates. Temporary duplicates are duplicates that are not eliminated immediately in state cache, but are eventually eliminated during DDD. Each temporary duplicate adds to the overhead of our search. We can bound the number of temporary duplicates by similar reasoning as in Theorem 1. We simply switch the number of layers checked in DDD (L) for the number of layers cached in RAM (k).

Theorem 4. *In layered external memory search, the worst-case number of times a state can be temporally duplicated is bounded by $\frac{d}{k}$.*

Proof. If k is the number of layers cached by the algorithm. No duplicates will be temporarily missed in these k layers. New states will be checked against those k layers immediately. When a layer is larger than our state cache, i.e. $k < 1$, states may be repeated more than once per layer. Still, the least layers a duplicate state s can reappear is at layer $g^*(s) + k$ where $g^*(s)$ is the first layer at which state s is encountered. State s may then appear every k layers for the rest of the search. If the depth of the search is d , the state s can appear a maximum of j times where $g^*(s) + j * k \leq d$. Thus, state s may recur at most $\frac{d}{k}$ times, which assumes the state appears in layer 0 and every k layer thereafter. This bound is applicable whether $k \geq 1$ or $1 > k$.

Theorem 5. *In layered breadth-first search, the worst-case number of number of states that can be temporary duplicates is bounded by $\frac{S*d}{k}$.*

Proof. This is a small step from Theorem 4. Since a state s can be a temporary duplicate no more than $\frac{d}{k}$ times and there are S states, no more than $\frac{S*d}{k}$ states can be seen during the search. Since $\frac{S*d}{k}$ temporary duplicates states would mean that every state in the graph is duplicated the maximum times allowed by the bound in Theorem 4.

5 Conclusion and Future Work

External layered duplicate detection has the potential to improve on previous work in several ways. First, layered duplicate detection is easy to implement in a domain-independent way. Second, the layered framework allows us to exploit transition locality in the hash table. The vast majority of duplicates are identified instantly in RAM because the most recent states are cached in the hash table. This is especially true when only a small portion of the state space is cached. For our largest model, *directory*, we eliminated 86% of duplicates in RAM while caching approximately 1% of the state space. Transition locality is clearly a good metric for caching. Third, transition locality can also be exploited in external memory. In external memory we can still rely on most duplicates occurring in

the most recent layers. If we only perform DDD on the L most recent layers, we can save large amounts of space in external memory while insignificantly increasing time requirements. In the case of *ldash*, we require approximately one quarter of the disk space needed to store the entire state space, with a delay of 11 minutes, out of a search taking three and a half hours. Fourth, we generate several worst-case linear bounds on the amount of redundant work performed by our algorithm. Results are well below the bound. Finally, a layered approach lends itself to a speedier counterexample reconstruction.

Besides testing this approach on larger and more varied models, we hope to eventually extend it in several ways. We pointed out that only the L layers checked in DDD must be retained in our search. But up to this point we do not delete old layers because they are used during counterexample reconstruction. We will extend our tool to allow complete search of graphs that do not fit on disk. This will include reconstructing parts of the counter example that have been removed from disk using a divide-and-conquer method [12].

We would also like to use a better hashing function. Jabbar and Edelkamp [14] use a hashing function that limits a bucket's successors to a few other buckets. This makes it easier to parallelize the search, since completed buckets can be refined by delayed duplicate detection in parallel with bucket expansions. Their hash is domain-dependent, however. We would like a hash that limits the number of successor buckets but works in a domain-independent way.

Another valuable addition would be to bound the diameter of the graphs. This bound could be used to prune the search deeper than the bound and guarantee termination of the algorithm.

A final addition would be a system for dynamically varying the number of layers checked in DDD. Since the ideal number of layers checked differs for every model, it would be desirable to determine the number of layers to check automatically. One possibility is to vary the number of layers checked based on the maximal back edge length observed.

References

1. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Mur ϕ verifier. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
2. Bao, T., Jones, M.: Time-efficient model checking with magnetic disk. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 526–540. Springer, Heidelberg (2005)
3. Korf, R., Schultze, P.: Large-scale parallel breadth-first search. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 1380–1385 (2005)
4. Pelánek, R.: Typical structural properties of state spaces. In: [25], pp. 5–22
5. Tronci, E., Penna, G.D., Intrigila, B., Zilli, M.V.: Exploiting transition locality in automatic verification. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 259–274. Springer, Heidelberg (2001)

6. Gastin, P., Moro, P.: Minimal counterexample generation for spin. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 24–38. Springer, Heidelberg (2007)
7. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. *Formal Methods in System Design* 7(3), 227–241 (1995)
8. Geldenhuys, J.: State caching reconsidered. In: [25], pp. 23–38
9. Kristensen, L.M., Mailund, T.: A compositional sweep-line state space exploration method. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 327–343. Springer, Heidelberg (2002)
10. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
11. Parashkevov, A.N., Yantchev, J.: Space efficient reachability analysis through use of pseudo-root states. In: *Tools and Algorithms for Construction and Analysis of Systems*, pp. 50–64 (1997)
12. Zhou, R., Hansen, E.: Breadth-first heuristic search. *Artificial Intelligence* 170, 385–408 (2006)
13. Bao, T.: Empirical comparison of algorithms for model checking with magnetic disk. Technical Report VV-0402, Department of Computer Science, Brigham Young University (2004)
14. Jabbar, S., Edelkamp, S.: Parallel external directed model checking with linear I/O. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 237–251. Springer, Heidelberg (2005)
15. Penna, G.D., Intrigila, B., Tronci, E., Zilli, M.V.: Exploiting transition locality in the disk based Mur ϕ verifier. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 202–219. Springer, Heidelberg (2002)
16. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Integrating RAM and disk based verification within the Murphi verifier. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 277–282. Springer, Heidelberg (2003)
17. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite-state concurrent systems. *International Journal on Software Tools for Technology Transfer* 6(4), 320–341 (2004)
18. Zhou, R., Hansen, E.: Structured duplicate detection in external-memory graph search. In: McGuinness, D.L., Ferguson, G. (eds.) *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, San Jose, CA, pp. 683–688. AAAI Press / MIT Press (2004)
19. Hammer, M., Weber, M.: To Store or not to Store Reloaded: Reclaiming memory on demand. In: *11th International Workshop on Formal Methods for Industrial Critical Systems*. Springer, Heidelberg (2006)
20. Dill, D.L.: The Mur ϕ verification system. In: Alur, R., Henzinger, T. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
21. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19(1), 7–34 (2001)
22. Kumar, R., Mercer, E.G.: Load balancing parallel explicit state model checking. *Electr. Notes Theor. Comput. Sci.* 128(3), 19–34 (2005)
23. Lenoski, D.: DASH Prototype System. PhD thesis, Stanford University (1992)
24. Emerson, A.E., German, S., Havlicek, J., Venkataramani, A.: Model checking a parameterized directory-based cache protocol (2002)
25. Graf, S., Mounier, L. (eds.): SPIN 2004. LNCS, vol. 2989. Springer, Heidelberg (2004)

Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes

Stefan Leue¹, Alin Ștefănescu^{2,*}, and Wei Wei¹

¹ Department of Computer and Information Science
University of Konstanz

D-78457 Konstanz, Germany

{Stefan.Leue,Wei.Wei}@uni-konstanz.de

² SAP Research CEC Darmstadt

Bleichstr. 8, D-64283 Darmstadt, Germany

alin.stefanescu@sap.com

Abstract. The execution of a reactive system amounts to the repetitions of executions of control flow cycles in the component processes of the system. The way in which cycle executions are combined is not arbitrary since cycles may depend on or exclude one another. We believe that the information of such dependencies is important to the design, understanding, and verification of reactive systems. In this paper, we formally define the concept of a *cycle dependency*, and propose several static analysis methods to discover such dependencies. We have implemented several strategies for computing cycle dependencies and compared their performance with realistic models of considerable size. It is also shown how the detection of accurate dependencies is used to improve a livelock freedom analysis that we developed previously.

1 Introduction

The main purpose of a concurrent reactive system is to maintain an ongoing interaction with its environment [15]. The execution of the system is therefore expected to last forever. Since each component process in the system has a finite control structure, any infinite execution of the system is essentially an infinite repetition of a certain set of control flow cycles in the concurrent processes that form the system. The way in which cycle executions are combined is certainly *not* arbitrary. For instance, the repetition of one cycle may rely on the repetitions of some other cycles; and the execution of one cycle may also eliminate the possibility of executing some other cycles.

We believe that the information of such cycle dependencies is important to the design, understanding, and verification of reactive systems. As one example, the knowledge of cycle dependencies may reveal potential design errors. In a reactive system, let us suppose that the repetition of a control flow cycle C relies on the repetitions of other cycles. When we expect C to be executed infinitely often,

* The work was done while this author was working at the University of Konstanz.

we may want to check statically whether there is any other cycle in the system on which C relies. The fact that no such cycles can actually be found hints at an incompleteness in the design or implementation of the system.

The knowledge of cycle dependencies is also useful in the verification of concurrent reactive systems. In our precursory work, we proposed an efficient system verification framework based on integer linear program (ILP) solving [13,12]. Our verification methods abstract the original verification problem into an ILP problem that describes a necessary condition for the violation of the property under scrutiny. Any solution to the ILP problem corresponds to a counterexample in the form of a set of cycles. A counterexample is spurious if it is impossible to repeat the cycles in the counterexample forever without other cycles also being repeated infinitely often. Consequently, the dependency among cycles stands at the very core of the refinement procedure based on the detection of spurious counterexamples in [14,12].

The central contribution of this paper is a formal framework capturing a notion of dependency between the control flow cycles of the concurrent processes. We also inspect different causes of dependencies, and develop techniques for discovering dependencies with respect to each cause. In this paper we choose Promela [9] as modeling language for the systems that we analyze. This choice is motivated by convenience since a large number of Promela models are available in the public domain [21] and some of the features of the SPIN tool environment, which interprets Promela, greatly facilitate our static analysis. We conjecture that applying our analysis ideas to other modeling and programming languages based on communicating finite state machines, such as UML-RT, could easily be accomplished.

Related Work. To the best of our knowledge, there is currently no work addressing control flow cycle dependencies. Control flow graphs of general programs were extensively studied in the area of static program analysis [20] with applications, e.g., in the area of compiler optimization. Slicing of programs [25,7,18,23] checks dependences between statements but not cycles. The “may happen in parallel” [19] and “non-concurrency” [16] analyses also consider dependences between statements. Finally, the INCA verification framework [4,24] studies the relation between acyclic paths and control flow cycles but not relations among cycles. Moreover, the above techniques are applied to either sequential programs or synchronous communication settings, while we also address an asynchronous setting where exchanging messages via buffers is the dominant way of communication.

Structure of the Paper. Section 2 introduces the Promela modeling language, define cycles and some related concepts. Section 3 defines the concept of cycle dependencies. We propose in Sections 4 and 5 several static analysis methods for cycle dependency discovery. Section 6 briefly shows how the discovery of cycle dependencies can help improve the precision of a livelock freedom test. Section 7 reports the experimental results, before Section 8 concludes the paper. All the proofs of the theoretical results of the paper can be found in Appendix A.

2 Preliminaries

Promela. Promela is the input language of the SPIN explicit state model checker [9]. It has been successfully used for the modeling and analysis of many concurrent systems [10,6]. The Promela language supports asynchronous communication as well as synchronous rendez-vous communication and synchronization via shared variables. The subset of the Promela language that we consider includes the definition of concurrently running processes (“**proctype**”), communication channels (“**chan**” declarations), message sending (“**!**”) and receiving (“**?**”), assignments, condition statements, nondeterministic branching (“**if ... fi**”), looping (“**do ... od**”), and arithmetics. For the sake of simplicity we do not consider arrays and structured data types in this paper.

```

1  active proctype client()
2    int x = 0;
3    do
4      :: (x < 3) -> toServer!request; x++;
5      :: (x == 3) ->
6         fromServer?reply; x--;
7    od
8
9  active proctype server()
10 do
11  :: toServer?request -> fromServer!reply;
12  od

```

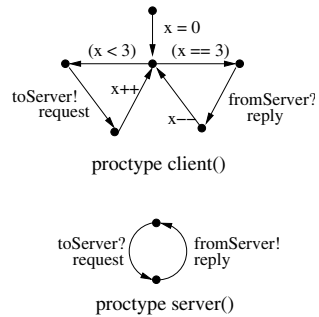


Fig. 1. An example Promela model and its control flow graphs

Figure 1 shows a simple Promela model consisting of two processes: `client` and `server`, whose behavior is described by the sequential Promela code within the respective `proctype` definition. The process `client` may send a `request` message to the buffer `toServer` if $x < 3$ (Line 4). Otherwise, it waits until a `reply` message is available in the buffer `fromServer` and then receives the message (Line 6). A condition statement such as $(x < 3)$ is a boolean expression enclosed in parentheses, which acts as a guard to the subsequent statements. It is executable if and only if the enclosed expression evaluates to true. We can construct a control flow graph from each of the `proctype` definitions (see Figure 1). Each transition corresponds to one statement in the code, and its *source* state and *target* state respectively denote the control points before and after the execution of the statement.

Control flow cycles. We define a *control flow cycle* (or simply *cycle*) in a control flow graph as a sequence of consecutive transitions in the graph such that the source state of the first transition in the sequence is the same as the target state of the last transition. A cycle is *elementary* (or *simple*) if no two transitions in the defining sequence of the cycle have a same source state. Informally, an elementary cycle cannot be decomposed further into smaller cycles. In the control flow graph of the process `client` in Figure 1, there are two elementary cycles.

Even though a finite control flow graph may contain infinitely many cycles, the number of elementary cycles is always finite and in the worst case exponential in the number of transitions. Since any non-elementary cycle can be decomposed into elementary cycles, our analysis considers only elementary cycles. Unless otherwise specified, all the cycles mentioned in the following are elementary.

If two cycles share states, then they are *neighbors* of each other. Any shared state is an *exit state* of the cycles that contain it, because one can exit one cycle at that state and enter another cycle. The two cycles in the process `client` in Figure 1 are neighbors sharing one exit state.

Cycle executions. An infinite run of a Promela model amounts to the repeated executions of cycles in some processes of the model. For an infinite run r of a Promela model, let r/p denote the projection of r on the set of transitions in a process p . Thus, r/p corresponds to the local execution of p in r . Any r/p can be decomposed into two parts: (1) an acyclic path from the initial state, and (2) repeated executions of cycles. Given a cycle c in p , one execution of c in r/p may be interrupted by the executions of other cycles in p : Some part of c is executed until some exit state s is reached where it starts to execute other cycles. The execution of c is later resumed from s after the executions of those interrupting cycles are completed. Since r is an infinite run, at least one cycle in the model is repeated infinitely often. We denote by $IRC(r)$ (*infinitely repeated cycles*) the set of cycles that are executed infinitely often in r . For a process p , $IRC(r/p)$ is the subset of $IRC(r)$ consisting of only cycles in p . It is easy to see that $IRC(r/p)$ is either empty or forms a strongly connected subgraph of the control flow graph of p .

3 Cycle Dependencies

We now define the concept of cycle dependencies. Intuitively, a cycle c depends on a set of cycles S if the infinite execution of c must be accompanied by the infinite executions of some cycles in S .

Definition 1. *Given a Promela model, a cycle c and a set of cycles S in the model, we call the pair (c, S) a cycle dependency if they satisfy the following conditions: a) $c \notin S$; and b) for any infinite run r of the model where $c \in IRC(r)$, there exists a cycle $c' \in S$ such that $c' \in IRC(r)$. In this case, we say that c depends on S .*

In the above definition, if all the cycles in S are in the same process as c is, then (c, S) is a *local* dependency. Otherwise, (c, S) is a *global* dependency. Moreover, if c does not depend on any subset of S , then we say that (c, S) is a *minimal* dependency. In the model in Figure 1, we denote by c_l (resp. c_r) the left (resp. right) cycle in the process `client` and by c_s the only cycle in the process `server`. $(c_r, \{c_l, c_s\})$ is a cycle dependency, while $(c_r, \{c_l\})$ and $(c_r, \{c_s\})$ are two minimal cycle dependencies. In particular, $(c_r, \{c_l\})$ is a local dependency, and $(c_r, \{c_s\})$ is a global dependency.

If we interpret all message buffers in a Promela model to have only finite capacities, then the Promela model possesses a finite global state space. In this case, we show as follows that it is decidable whether (c, S) is a cycle dependency: We construct the global state space for the model and then look for any elementary or non-elementary cycle in the global state space that contains c but no cycles from S . If no such global cycles exist, then (c, S) is a cycle dependency. However, we are more interested in infinite state models. If we assume that buffers in Promela models have infinite capacities and variables may have infinite domains such as integer variables, then a Promela model may have an infinite global state space, for which we show in the following theorem that the above problem becomes undecidable.

Theorem 1. *Given a cycle c and a set S of cycles, it is undecidable in general whether (c, S) is a cycle dependency.*

3.1 The Causes of Cycle Dependencies

The root cause for cycle dependencies lies in the executability of Promela statements. Given a cycle, if the executability of every statement along the cycle is unconditional, then the cycle can be repeated without interruption forever once the cycle is entered. Such a cycle does not depend on any other cycles. On the contrary, consider a cycle c that contains a statement s whose executability is conditional. If s cannot be continuously enabled forever by only repeating c , then some other cycles need to be executed in order to re-enable s by, e.g., modifying the values of some variables, sending a message etc. In Promela there are two kinds of statements with conditional executability: condition statements and message receiving statements, when we take the assumption that message buffers have unbounded capacities and message sending statements are therefore always enabled. In the following we explain how cycle dependencies may be imposed by these two kinds of statements.

Condition statements. Consider the right cycle c_r in the process `client` in Figure 1. c_r contains a condition statement (`x == 3`). The condition $x = 3$ cannot remain true after c_r is executed because x is decremented by 1 in the cycle. Then, c_r can be repeated infinitely often only if the left cycle c_l is also repeated infinitely often to modify the value of x such that x can always acquire the value 3 again. This is one example that a cycle is terminating on a condition statement along the cycle. Since we focus on discovering cycle dependencies in this paper, it is out of scope how to determine whether a cycle is terminating, which is a well-known undecidable problem. In [14] we proposed an incomplete procedure to prove termination for control flow cycles. There are also many existing techniques [22, 3, 5, 1] to prove termination for certain kinds of loops in programs, which can be adapted to prove termination for control flow cycles. In Section 4 we will show how to determine cycle dependencies from a condition statement on which a cycle is terminating.

Message receiving statements. The above mentioned cycle c_r contains a message receiving statement `fromServer?reply`. Thus, the cycle c_s sending `reply` messages has to be repeated infinitely often when c_r is to be repeated infinitely often. In Section 5 we will present a method to determine cycle dependencies from message receiving statements, which are usually global dependencies.

4 Discovering Dependencies from Condition Statements

We show some types of cycle dependencies imposed by condition statements on which a cycle is terminating. In order to derive them, we need to discriminate between different ways in which the variables in a condition statement are modified in the cycle. A variable is *local* if its value can be referenced and modified only by one process. Otherwise, it is a *global* variable. However, the runtime value of a local variable may still depend on the executions of other processes. For instance, given a local variable x , if there is an assignment $x = e(y)$ where e is an arithmetic expression containing a global variable y , then the runtime value of x may depend on how y is modified in other processes.

Definition 2. *For a cycle c and a variable x , x is globally modified in c if one of the following is satisfied: a) x is global, or b) there is a message receiving statement `b?msg(x_1, \dots, x_n)` in c where x is some x_i , or c) there is an assignment $x = e(y)$ in c where y is globally modified in c . Otherwise, x is locally modified in c .*

Note that in the above definition we disregard the dependency of the runtime value of a local variable on a condition statement. The reason is that a control flow cycle contains only one branch of a condition statement. Therefore, the impact of the condition statement is fixed in the cycle. Note that we are only interested in how a variable is modified inside a particular cycle when the cycle is repeated without interruption.

For a boolean condition B in a cycle c , we denote by $var(B)$ the set of variables occurring in B . If all the variables in $var(B)$ are locally modified in c , then B is a *locally determined* condition. Otherwise, it is *globally determined*. In Subsection 4.1 and 4.2, we show how to determine cycle dependencies from these two kinds of conditions.

4.1 Locally Determined Conditions

First, we can easily see that, if a cycle is terminating on a locally determined condition, then it depends on some of the cycles in the same process for an infinite number of executions. In particular, the cycle must depend on one of its neighbors.

Proposition 1. *Given a cycle c in a process p such that c is terminating on a locally determined condition B , if c is repeated infinitely often in a run r , then one of the neighbors of c is also repeated infinitely often in r .*

Let C_p denote the set of cycles in p , and N_c denote the set of the neighbors of c . The above discussion gives two cycle dependencies, namely $(c, C_p - \{c\})$ and

(c, N_c) . The cycle $(c, C_p - \{c\})$ is usually coarse because not necessarily all the cycles in p contribute to the re-satisfaction of B . In the following, we propose several methods to refine the dependency $(c, C_p - \{c\})$.

Refinement 1. In general, it is impossible to determine which cycles make a contribution to the re-satisfaction of the condition B . We define $E_c(B)$ as the set of variables occurring in c such that at least one of the variables in $E_c(B)$ must be modified in order to make B true again. The set $E_c(B)$ subsumes but not necessarily equals $var(B)$. In the example in Figure 2, an infinite number of repetitions of the left cycle relies on an infinite number of repetitions of the right one that resets the value of y . However, the enabling condition of the left cycle contains only the variable x which is not modified by the right cycle. We propose the following recursive method to compute $E_c(B)$. A variable v is in $E_c(B)$ if one of the following is satisfied: a) $v \in var(B)$, or b) there is an assignment $v' = e(v)$ in c such that $v' \in E_c(B)$. For a set S of variables, we denote by $MC_p(S)$ the set of cycles in p which modify at least one variable in S . We obtain a finer dependency $(c, MC_p(E_c(B)) - \{c\})$ by disregarding all cycles that do not modify any variables in $E_c(B)$.

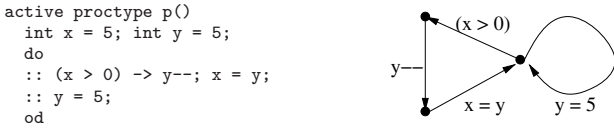


Fig. 2. An example Promela model and its control flow graph

Refinement 2. The above cycle dependency may still be coarse. Consider the control flow graph in Figure 3. Note that, whenever leaving $C1$ to execute $C3$, $C2$ is always executed. So, in any run in which $C1$ is repeated infinitely often, no matter whether $C3$ is repeated infinitely often or not, $C2$ is always repeated infinitely often. Based on this observation we can refine the cycle dependency $(C1, \{C2, C3\})$ by safely removing $C3$. The above simple example leads to the following definition.

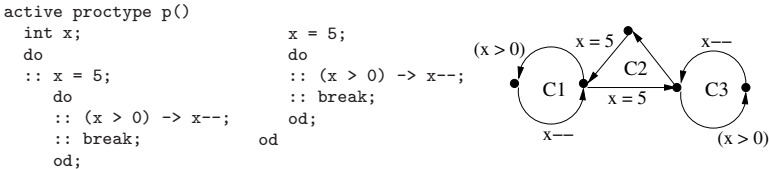


Fig. 3. An example Promela Model and its control flow graph

Definition 3. Given a cycle c in a process p such that c is terminating on a locally determined condition B , and a cycle $c' \in MC_p(E_c(B))$ such that c and c' are reachable from each other, c' is preemptive with respect to c and B if there

exist one exit state s in c and one exit state s' in c' such that a) there is an acyclic path from s to s' that does not modify any variables in $E_c(B)$, and b) there is an acyclic path from s' to s that does not modify any variables in $E_c(B)$. Otherwise, c' is preempted.

In the previous example, $C2$ is preemptive and $C3$ is preempted. It is easy to prove that, on the way from any cycle c to execute one of its preempted cycles and then back to c , at least one preemptive cycle must be executed. We can therefore refine the cycle dependency $(c, MC_p(E_c(B)) - \{c\})$ by removing all the preempted cycles from $(MC_p(E_c(B)) - \{c\})$.

```

1  proc compute_cd(cycle c_0, condition B_0)
2    set[cycle] visited = {}
3    set[cycle] ccs = {}
4    queue[cycle] open = {}
5    search_for_preemptive_cycles(c_0, B_0)
6    return (c_0, ccs) // return the determined cycle dependency
7
8  proc search_for_preemptive_cycles(cycle c, condition B)
9    add c to visited
10
11   for each nc in neighbors(c)
12     if (nc not in visited) and (nc not in open)
13       if (nc modifies some variables in E_c(B))
14         then
15           add nc to visited
16           add nc to ccs
17         else
18           enqueue(open, nc)
19
20   if (open not empty)
21     c' = dequeue(open)
22     search_for_preemptive_cycles(c', B)

```

Fig. 4. An algorithm to determine cycle dependencies from locally determined conditions

Whereas Definition 3 can be used to determine whether a cycle is preempted, Figure 4.1 gives an efficient algorithm to collect preemptive cycles during the computation of cycle dependencies. In a Breadth First Search manner, the algorithm visits each cycle at most once, and thus is linear in the number of cycles. This is a generalization of the so-called “next door” strategy first mentioned in [12]. In Appendix A.3 the termination and soundness of the algorithm are proved.

4.2 Globally Determined Conditions

If a cycle is terminating on a globally determined condition, then it may not only depend on cycles in the same process, because cycles in other concurrent processes can possibly influence the runtime value of the condition. This can be illustrated in the example in Figure 5. The cycle in Process p is actually the only cycle in p , and it depends on the cycle in q . We will not consider any globally determined condition whose value is influenced by a message receiving statement, which will be discussed in the next section.

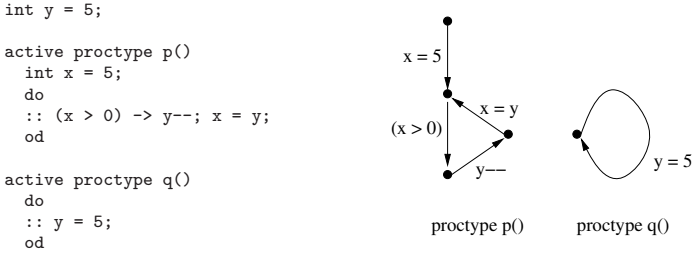


Fig. 5. An example Promela model and its control flow graphs

For a Promela model M , we denote by $proc(M)$ the set of processes in M . Suppose a cycle c in a process in M such that c is terminating on a globally determined condition B . We can easily derive that c depends on $(\bigcup_{q \in proc(M)} MC_q(E_c(B)) - \{c\})$. We may refine this cycle dependency by using the algorithm in Figure 4.1 to rule out all the preempted cycles in $MC_p(E_c(B))$ if c is in the process p .

5 Discovering Dependencies from Message Receiving Statements

When a cycle c contains a message receiving statement $b?msg(x_1, \dots, x_n)$, it needs an infinite number of msg messages to be repeated infinitely often. Consequently, c depends on some cycles that send such messages. Let $SC_{b,msg}$ be the set of the cycles sending messages msg to b . If $c \notin SC_{b,msg}$, then $(c, SC_{b,msg})$ is cycle dependency. In the remainder of the section, we assume that a cycle never receives messages sent by itself.

A cycle that receives messages may contain a condition statement in which the condition contains some variables used to store components of received messages. Usually, the cycle can be executed only if the received message contains such components that make the condition true. Consider a cycle that contains a message receiving statement s_1 and a condition statement s_2 such that the condition in s_2 contains variables used in s_1 . The following pattern for s_1 and s_2 are observed in most real life Promela models: (1) all the variables in s_1 are local; (2) the condition in s_2 contains only variables used in s_1 ; (3) no variable in the condition is modified between s_1 and s_2 in the cycle. We call such a condition a *message determined* condition. Figure 6 shows two processes `GIOPClient` and `GIOPAgent`. In the control flow graph of `GIOPClient`, there is a cycle depicted using only solid lines that contains a message determined condition `reply_status = 4`. We show in the remainder of the section how to derive cycle dependencies from such a message determined condition.

In Figure 6, let c_1 denote the solid-lined cycle in Process `GIOPClient`, and c_2 and c_3 denote the cycles that respectively assign 4 and 5 to `rs` in Process `GIOPAgent`. We have a dependency $(c_1, SC_{toClientL,Reply})$ and both c_2 and c_3 are

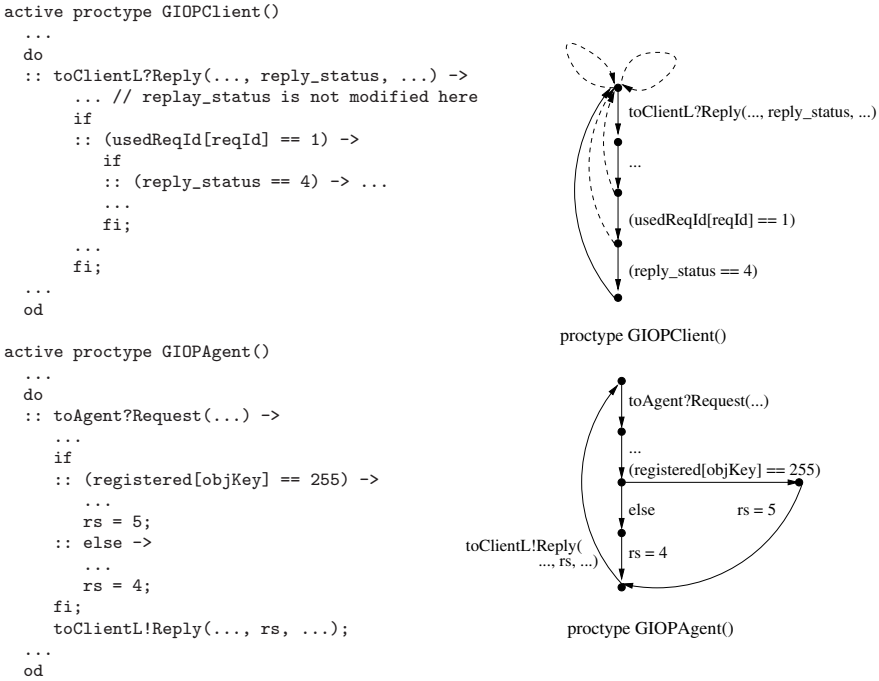


Fig. 6. An excerpt from a Promela model for CORBA GIOP [10]

in $SC_{toClientL, Reply}$. However, this dependency is coarse because not necessarily every cycle in $SC_{toClientL, Reply}$ may send a `Reply` message to make `reply_status = 4` true in c_1 . As an example, c_3 assigns 5 to `rs` whose value is passed to `reply_status` in c_1 through message passing. Thus, c_3 cannot make `reply_status = 4` true, and it can be safely removed from $SC_{toClientL, Reply}$ to obtain a finer dependency. Now the question is how to determine which cycle *cannot* send messages to make `reply_status = 4` true.

First, we need to determine which kind of `Reply` messages must be received by c_1 to make `reply_status = 4` true. More precisely, we need to know which condition must be satisfied by the components of such a message. According to the definition of message determined conditions, `reply_status` is not modified in c_1 between the message receiving statement and `(reply_status == 4)`. However, after a message is received, `reply_status` may still be modified before `(reply_status == 4)` is reached. This is because the execution of c_1 can be interrupted, e.g., at the source state of the transition corresponding to `(usedReqId[reqId] == 1)`. Then, when the execution of c_1 is resumed, `reply_status` may be already modified by other cycles. However, in this concrete example, if c_1 is interrupted, then before c_1 is resumed the last completed interrupting cycle always receives a `Reply` message. Moreover, this message contains a component whose value is passed to `reply_status`. The value of `reply_status` is afterward unchanged before reaching the message determined condition. This is because c_1 and its neighbors satisfy the

following structural property named *fastened cycles*: Given a cycle c that contains a message receiving statement s_1 and a condition statement s_2 , we denote by t_1 the transition corresponding to s_1 , by t_2 the transition corresponding to s_2 , and by p the path from the source state of t_1 to the source state of t_2 . For each neighbor c' of c , if c' and c contain a common state s within p , then c' contains also the path in c from the source state of t_1 to s . The pattern in the fastened cycles property results from nested `if` statements inside `do` loops which are a common control structure of concurrent processes in an asynchronous reactive system.

Proposition 2. *Let c be a cycle that contains a condition statement (B) in which the condition B is determined by messages received via the statement `b?msg(x_1, \dots, x_n)` in c . If the fastened cycles property is satisfied by c and all of its neighbors, then one execution of c needs a `msg(d_1, \dots, d_n)` message such that $B[x_i \leftarrow d_i]$ ¹ is true.*

Using Proposition 2, if we can determine that the execution of c requires a message `msg(d_1, \dots, d_n)` such that $B[x_i \leftarrow d_i]$ is true, then we can use the following method to determine whether a cycle c' may *not* send such a message. Given a cycle c' that contains a message sending statement `b!msg(d_1, \dots, d_n)`, if all d_i 's are constant values, then we directly evaluate $B[x_i \leftarrow d_i]$ which is a constant truth value. If it is false, then we can exclude c' from $SC_{b,msg}$. When some d_i is a variable, we traverse backward in c' from the source state s' of the transition t corresponding to `b!msg(d_1, \dots, d_n)`, and locate the first state $s \neq s'$ such that s has an incoming transition outside c' but within other cycles. If no such s exists, then we take as s the predecessor of s' in c' . The path p from s to s' is then the longest acyclic path within c that must be consecutively executed immediately before reaching the message sending statement. We compute the postcondition $Post(p)$ of p by Floyd-Hoare-style forward inference starting with the precondition `true`². This assumes that all the variables initially contain arbitrary values before p is consecutively executed. If $Post(p) \wedge B[x_i \leftarrow d_i]$ is unsatisfiable, then c' can be removed from $SC_{b,msg}$. If the Promela model contains only linear arithmetic expressions in assignments and conditions, then the satisfiability of $Post(p) \wedge B[x_i \leftarrow d_i]$ can be decided fully automatically using either an automated theorem prover or a linear programming solver. In the example in Figure 6, we illustrate how to determine that c_3 cannot send a message to satisfy `reply_status = 4`. The longest consecutively executed path p in this example starts from the source state of the transition corresponding to the message receiving statement, i.e., the topmost state in the control flow graph of `GIOPAgent`. Then $Post(p) = (\dots \wedge (rs = 5))$. Since $Post(p) \wedge (reply_status = 4)[reply_status \leftarrow rs]$ is false, c_3 can be safely removed from $(c_1, SC_{toClientL,Reply})$.

¹ $B[x_i \leftarrow d_i]$ is a boolean expression obtained from B by substituting simultaneously each occurrence of x_i with d_i .

² Since the path p is acyclic, $Post(p)$ can be computed fully automatically.

6 The Refinement of a Livelock Freedom Test

We show how the discovery of cycle dependencies can be used to improve the precision of a livelock freedom test that we developed [12]. We sketch this test using the example in Figure 1.

In a Promela model we may label a set of statements as *progress* statements. Let us assume the message receiving statement (Line 6) in the process `client` is the only progress statement in Figure 1. A model is said to be *free of livelock* if and only if at least one of the progress statements must be repeated infinitely often in any infinite run of the model. Therefore, our example model is free of livelock if the client always receives replies from the server infinitely often. Moreover, we define a cycle to be a *progress cycle* if it contains at least one progress statement. So, the right cycle c_r of `client` is the only progress cycle. We have shown in [12] that livelock freedom is undecidable for infinite state systems.

The basic idea of our livelock freedom test in [12] is to check whether there is any infinite run of a model in which no progress cycle is repeated infinitely often. If no such run exists, then the model is livelock free. In our test, we first abstract from arbitrary program code in the model and retain only the message sending and receiving statements. Next, we abstract from message orders and denote the message passing effect of a statement by an integer vector called an *effect vector*. Each component of an effect vector corresponds to one type of messages. A positive component represents the number of messages of the corresponding type being sent by the statement. A negative component represents the number of messages being received. We abstract further from the activation conditions and dependencies of cycles. The resulting abstract system is a set of cycles with their summary effect vectors. In our example, there are three cycles: c_l with the effect vector $(1, 0)$, c_r with $(0, -1)$, and c_s with $(-1, 1)$.

We now give a necessary condition for the existence of a livelocked run, i.e., a run in which no progress cycle is repeated infinitely often, in the form of an integer linear programming (ILP) problem. The ILP problem is shown in Figure 7. It can be solved in polynomial time. Intuitively, any solution to this ILP problem represents a combination of cycle effects that (1) can be repeated forever since it does not consume any type of messages (Inequalities 1-2); and (2) does not include any progress cycle (Inequality 3). The last inequality 4 restricts the number of times that a cycle is repeated to be non-negative. If the ILP problem has no solutions, then such cycle combination does not exist, which proves livelock freedom for the model. Unfortunately, the ILP problem has a solution: $x_1 = 1, x_2 = x_3 = 0$. In this case, we do not know whether the model is livelock free or not because the abstraction used in our test is over-approximating and may introduce spurious behavior.

The above obtained ILP solution represents a counterexample suggesting the scenario that only the cycle c_l is repeated infinitely often in some runs. However, by the help of our cycle dependency discovery, we can see that the cycle c_l depends on c_r . Since c_r is not included in the counterexample, the counterexample is spurious. Furthermore, we can use the cycle dependency to refine the

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} x_1 + \begin{pmatrix} 0 \\ -1 \end{pmatrix} x_2 + \begin{pmatrix} -1 \\ 1 \end{pmatrix} x_3 \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (1)$$

$$x_1 + x_2 + x_3 > 0 \quad (2)$$

$$x_1 = 0 \quad (3)$$

$$x_i \geq 0 \quad (4)$$

Fig. 7. The livelock freedom determination ILP problem for the model in Figure 1

abstraction by adding the following inequality to the ILP problem in Figure 7: $x_1 \leq 3x_2$. Intuitively, the new constraint says that the cycle c_r must be executed at least once for every 3 times that c_l is repeated. The determination of this constraint also relies on the estimation of the maximal iteration counts of the cycle c_l , which we discussed in 14.

The quality of the above described refinement procedure largely relies on the accuracy of the cycle dependency discovery techniques. The smaller a detected cycle dependency is, the more spurious behavior can be excluded through refinement.

7 Experimental Results

We have implemented different strategies to detect both local and global cycle dependencies for the models 8 listed in Table 1. The experimental results were obtained on a Pentium IV 1.60GHz machine with 1GB memory.

Table 1. Test models

Model	# cycles	# detected cycle dependencies
i-Protocol	22	30
MVCC	30	51
GIOP	66	203
SMCS	171	541

We detected three types of dependencies: (1) dependencies on neighbors (see Corollary 1); (2) dependencies on cycles that may render the considered condition to be re-satisfied; (3) dependencies caused by message receiving statements. Table 1 lists the total number of dependencies of all three types that were detected for each model. Different strategies are used to over-approximate dependencies of Type 2 and 3, and their performances are compared as explained below.

³ MVCC 8 models the *Model View and Concurrent Control* protocol used in the *Clock* toolkit for the development of groupware applications; *i-Protocol* 6 models a sliding-window protocol for *Unix-to-Unix-Copy*; *GIOP* 10 models inter-ORB message exchange and server object migration in the *CORBA* architecture; *SMCS* 17 models the *T.122* and *T.125* multi-point communication service protocol.

Table 2. The comparison of different strategies to detect dependencies of Type 2. By the size of a cycle dependency (c, S), we refer to the size of the set S . For each model and each strategy, we list the sum of the sizes of all the detected dependencies of this type.

Model	summary size of dependencies			% reduction w.r.t. MC		runtime (secs.)		
	MC	ND	PC	ND	PC	MC	ND	PC
i-Protocol	63	63	43	0	31.7	13.02	13.27	13.66
MVCC	60	59	41	1.7	31.7	3.55	3.59	3.52
GIOP	837	788	714	5.9	14.7	29.25	30.12	33.17
SMCS	5200	5200	3424	0	34.2	136.63	143.75	175.05

Table 7 compares three different strategies for the detection of dependencies of type 2: *MC* is the coarsest one that includes a cycle in the dependency as long as it may influence at least one variable in the considered condition; *ND* is the next-door strategy; *PC* is the finest one that collects only preemptive cycles for computing dependencies. We observe that *ND* leads to only a minor improvement of the accuracy of the detected dependencies. *PC* reduces the sizes of dependencies much more effectively at the expense of a modest or even no runtime penalty (see the results for *MVCC* 4). In particular, *PC* reduces the sizes of dependencies by more than one third for the model *SMCS* while *ND* does not reduce the cycle number at all.

Table 3. The comparison of different strategies to detect global dependencies caused by message receiving statements

Model	summary size of dependencies		% reduction w.r.t. SC	runtime (secs.)	
	SC	FC	FC	SC	FC
i-Protocol	62	33	46.8	0.01	0.05
MVCC	35	35	0	0.01	0.66
GIOP	274	242	11.7	0.10	10.78
SMCS	410	338	17.5	1.37	45.17

Table 3 shows two strategies to detect global dependencies caused by message receiving statements: *SC* is coarser and includes any cycle in the dependency as long as it may send the same type of messages as received by the considered receiving statement. *FC* checks the fastened cycles property in order to exclude cycles that cannot contribute a desired message. We observe that *FC* can reduce the sizes of dependencies quite considerably at the expense of a moderate to significant runtime penalty. The fact that *FC* did not reduce the dependency sizes for *MVCC* is expected because few variables are used in the model to store components of incoming messages. Moreover, those component storing variables are not used to control the behavior of the model, i.e., there are no message determined conditions. The extra runtime required by *FC* on *MVCC* was spent

⁴ The reason is that *ND* and *PC* sometimes check only a small number of cycles for computing dependencies for one cycle while *MC* has to check all the cycles in the same process.

on checking the existence of message determined conditions. If we know a priori that no such conditions exist in a model, which can be achieved by a manual scan of the Promela code, then FC is unnecessary.

To illustrate the benefit of our analysis we applied our approach to the counterexample refinement of our livelock freedom analysis for Promela models [12]. We have mentioned that, by obtaining smaller and more dependencies, we stand a better chance to determine spuriousness for the counterexample. The previous version of our prototype livelock freedom checker *aLive* used the ND strategy to discover local dependencies and was not able to detect global dependencies. In [12] we reported that the local cycle dependency detection helped to remove 7 counterexamples for a model of the Group Address Registration protocol for which livelock freedom was successfully proved. For the GIOP model, 8 counterexamples were found and *aLive* failed to prove spuriousness for one of them. The spuriousness of this counterexample is caused by abstracting away a global dependency. After we employed the FC strategy proposed in this paper in *aLive*, the one remaining counterexample in GIOP was determined to be spurious and subsequently excluded from the abstraction. The same was observed during the checking of the i-Protocol model for which 4 more spurious counterexamples were discovered due to the detection of global dependencies.

We also performed experiments in which we used the cycle dependency analysis in the spuriousness determination of counterexamples found during our buffer boundedness analysis [13]. The increase in precision that we achieved lies within the range of increase that we obtained for the livelock freedom analysis.

8 Conclusion

The first contribution of our work is a formalization of the concept of control flow cycle dependencies. The second contribution is that we presented several incomplete but efficient static analysis methods for the detection of both local and global cycle dependencies for reactive systems of concurrent processes. Furthermore, we conducted experiments that show the precision of this analysis when applied to a set of models of real-life systems. We also show that the precision of our approach compared to naive cycle dependency detection techniques improves the precision of our livelock freedom and buffer boundedness analyses since more spurious counterexamples can be detected.

Future work will include improving our analysis by incorporating data flow analysis. As an example, consider the computation of $E_c(B)$ as the set of variables that may influence the run-time values of the variables in B along the cycle c (Sec. 4.1). If a variable in $E_c(B)$ does not occur in B , then it must appear in the right hand side of an assignment statement that directly or indirectly changes the value of some variable v in B . However, the effect of such an assignment may be killed later by an assignment to v before the condition statement (B) is reached. Therefore, the use of reachable definition analysis may improve the precision of $E_c(B)$. We will also consider broadening the approach to a wider range of programming and modeling languages. Finally, we see a potential for

the application of cycle dependency analyses to other application areas, such as the prediction of temporal conflicts and spatial localities of code blocks for the improvement of instruction cache hit rates [11].

Acknowledgment. The work of the second author was supported by the DFG-funded research project IMCOS (Grant No. LE 1342/1-/2). We thank Daniel Butnaru for his assistance in programming the implementation prototype. Finally, we thank the anonymous referees for their valuable suggestions.

References

1. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
2. Brand, D., Zafropulo, P.: On communicating finite-state machines. *Journal of the ACM* 30(2), 323–342 (1983)
3. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
4. Corbett, J.C., Avrunin, G.S.: Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design* 6(1), 97–123 (1995)
5. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
6. Dong, Y., Du, X., Holzmann, G.J., Smolka, S.A.: Fighting livelock in the GNU i-Protocol: a case study in explicit-state model checking. *Int. Journal on Software Tools for Technology Transfer (STTT)* 4(4), 505–528 (2003)
7. Dwyer, M.B., Hatcliff, J.: Slicing software for model construction. In: Proc. of PEPM 1999, pp. 105–118 (1999)
8. Graham, T.C.N., Urnes, T., Nejabi, R.: Efficient distributed implementation of semi-replicated synchronous groupware. In: ACM Symposium on User Interface Software and Technology, pp. 1–10 (1996)
9. Holzmann, G.J.: The SPIN model checker: Primer and reference manual. Addison Wesley, Reading (2004)
10. Kamel, M., Leue, S.: Formalization and validation of the general Inter-ORB protocol (GIOP) using PROMELA and SPIN. *Int. Journal on Software Tools for Technology Transfer (STTT)* 2(4), 394–409 (2000)
11. Kumar, R., Tullsen, D.: Compiling for instruction cache performance on a multi-threaded architecture. In: Proc. of MICRO 2002, pp. 419–429. ACM/IEEE (2002)
12. Leue, S., Ștefănescu, A., Wei, W.: A livelock freedom analysis for infinite state asynchronous reactive systems. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 79–94. Springer, Heidelberg (2006)
13. Leue, S., Mayr, R., Wei, W.: A scalable incomplete test for the boundedness of UML RT models. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 327–341. Springer, Heidelberg (2004)
14. Leue, S., Wei, W.: Counterexample-based refinement for a boundedness test for CFSM languages. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 58–74. Springer, Heidelberg (2005)

15. Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, Heidelberg (1992)
16. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: PPOPP 1993, pp. 129–138. ACM Press, New York (1993)
17. Merino, P., Troya, J.M.: Modeling and verification of the ITU-T multipoint communication service with SPIN. In: Proc. of SPIN 1996 (1996)
18. Millett, L.I., Teitelbaum, T.: Issues in slicing Promela and its applications to model checking, protocol understanding, and simulation. *STTT* 2(4), 343–349 (2000)
19. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In: SIGSOFT FSE 1998, pp. 24–34. ACM Press, New York (1998)
20. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of program analysis*, 2nd edn. Springer, Heidelberg (2005)
21. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bořnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
22. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
23. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.* 29(5) (2007)
24. Siegel, S.F., Avrunin, G.S.: Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Trans. Software Eng.* 28(2), 115–128 (2002)
25. Tip, F.: A survey of program slicing techniques. *Journal of Programming Languages* 3(3), 121–189 (1995)

A Appendix

A.1 The Proof of Theorem 1

We prove the theorem by a reduction from the following undecidable problem [2]: the executability of a message reception in a system of communicating finite state machines (CFSM) [3].

Instance: A CFSM M and a local state s of M having an outgoing transition t labeled by the receive action ‘ $?a$ ’

Question: Does there exist a run of M such that the message reception ‘ $?a$ ’ is executed at s ?

We construct a CFSM system M' from M by (1) introducing a new state s' in the same state machine as s is; (2) adding at s' a self-transition labeled with ‘! b ’ where b is a newly introduced type of message; (3) changing the target state of the transition t to the newly introduced state s' ; and finally (4) adding a new state machine consisting of a single state s'' and a self-transition at s'' .

⁵ The proof is actually for the undecidability of the same problem for communicating finite state machines (CFSM). However, Promela models with unbounded buffers can simulate CFSM systems. Thus, the undecidability result also holds for Promela models.

Moreover, let c be the self-loop at s' and S be the singleton cycle set consisting of the self-loop at s'' .

We prove that “ $?a$ ” can be executed at s in M if and only if c does *not* depend on S in M' .

For the “if” part, assume that c does *not* depend on S . Then, there exists an infinite run of M' in which c is executed infinitely often while the self-loop at s'' is not. From the construction of M' , c can be executed only if “ $?a$ ” can be executed at s in M' , which means that “ $?a$ ” can be executed also in M .

For the “only if” part, assume that “ $?a$ ” can be executed at s in M . Then, “ $?a$ ” can be also executed in M' . After “ $?a$ ” is executed, c can be repeated alone forever, which means c does not depend on any other cycles for an infinite number of executions. \square

A.2 The Proof of Proposition 1

If c is repeated an infinite number of times, then some other cycle c' in p must be repeated also infinitely often. On every path from a state in c to a state in c' , there must be a transition t from an exit state of c such that t is not contained in c . There are only finitely many such transitions, so one of them is taken an infinite number of times and it belongs to at least one of the neighbors of c . \square

A.3 The Termination and Soundness of the Algorithm in Figure 4.1

Proposition 3 (Termination). *The algorithm in Figure 4.1 always terminates.*

Proof. It is easy to see that no cycle can be added to `visited` more than once. Hence, each call to `search_for_preemptive_cycles` results in a new cycle being added to `visited` (Line 9). Note that our algorithm never removes any cycle from `visited`. Since there are only finitely many cycles, the algorithm must terminate. \square

Proposition 4 (Soundness). *Given as an input a cycle c in process p such that c is terminating on a locally determined condition B , the algorithm in Figure 4.1 returns a cycle dependency (c, S) such that a cycle $c' \in MC_p(E_c(B))$ is preemptive if and only if $c' \in S$.*

Proof. We assign a natural number $level(d)$ to each cycle d that is added to `visited` as follow: (1) $level(c) = 0$; (2) if c_1 is enqueued (Line 18) or added to `ccs` (Line 16) inside the call to `search_for_preemptive_cycles(c_2, B)` and $level(c_2) = n$, then $level(c_1) = n + 1$. In the second case, we say that c_2 is the *parent* of c_1 and c_1 is a *child* of c_2 . Then, we can build a *parent-child tree* (PCT).

For the “if” part, we prove that if $c' \in S$ then it is preemptive. It is easy to see that, in the path from the root c to c' in the PCT, no cycle except c and c' modifies any variable in $E_c(B)$. From this path, we can easily construct an acyclic path θ from an exit state t in c to an exit state t' in c' , and an acyclic path θ' from t' to t . θ and θ' apparently do not modify any variable in $E_c(B)$.

For the “only if” part, assume that c' is preemptive. Then, there is an exit state t in c , an exit state t' in c' , an acyclic path θ from t to t' , and an acyclic

path θ' from t' to t such that θ and θ' do not modify any variable in $E_c(B)$. The path $\langle \theta, \theta' \rangle$ can be decomposed into a set of cycles, from which we can construct a sequence of pairwise distinct cycles c_1, \dots, c_n such that (1) c_1 is a neighbor of c , (2) c_n is a neighbor of c' , and (3) each c_i and c_{i+1} are neighbors. It is easy to see no cycle in such a sequence modifies any variable in $E_c(B)$. Let SEQ be the set of shortest sequences of cycles as constructed in this way. Assume that the sequences in SEQ are of length n . For each sequence in SEQ , we add c to its head and attach c' to the end. We prove that there is one sequence $seq \in SEQ$ that is a path in the PCT, which implies that c' is added to ccs . The proof is by showing that, for any $k \leq n$, there is a sequence $seq \in SEQ$ such that its prefix of length i is a path in the PCT (*), by induction on the length i of prefixes of sequences in SEQ .

Induction base: The prefix of length 1 of any sequence in SEQ is c , which is a path in the PCT.

Induction step: Assume that (*) holds for k . Let P be the set of sequences in SEQ such that their prefixes of length k are paths in the PCT. Let C_k be the set of cycles $\{d \mid d \text{ is the } k\text{-th element in a sequence in } P\}$, and C_{k+1} be $\{d \mid d \text{ is the } (k+1)\text{-th element in a sequence in } P\}$. By contradiction, we assume that there is no sequence in P such that its prefix of length $(k+1)$ is a path in the PCT. Then, inside the call to `search_for_preemptive_cycles(c_k, B)` for each $c_k \in C_k$, none of the neighbors of c_{k+1} in C_{k+1} is enqueued or added to `visited`. This happens only when c_{k+1} is already in `open` or in `visited`. Let p be the parent of c_{k+1} . So, $p \notin P$. We have either that (1) $level(p) = k$, or that (2) $level(p) < k$. When $level(p) = k$, the path from c to p must be the prefix of length k of some sequence in SEQ , which means that $p \in P$. This leads to a contradiction. When $level(p) < k$, we construct a sequence of cycles from any sequence in P whose $(k+1)$ -th element is c_{k+1} , by replacing the prefix of length k by p . The new sequence is shorter than the sequences in SEQ , which contradicts that SEQ contains the shortest sequences of pairwise distinct cycles connecting c and c' . \square

A.4 The Proof of Proposition 2

Lemma 1. *Using the notation in the definition of the fastened cycles property in Section 5, the following is satisfied: For any path p_1 that ends at an exit state s within p , the path p_2 in c from the source state of t_1 to s is consecutively executed⁶ in the end of p_1 .*

Proof. We suppose that there are q exit states in p : es_1, \dots, es_q . We prove the lemma by induction on the index k of es_k .

Induction base: es_1 is the source state of t_1 . The path from es_1 to es_1 is an empty path which is always consecutively executed.

⁶ Given two paths p_1 and p_2 , we say that p_2 is executed in p_1 if p_2 is a subsequence of p_1 . If p_2 is a consecutive subsequence of p_1 , then we say that it is consecutively executed in p_1 . In particular, an empty path is always consecutively executed.

Induction step: Assume the lemma holds for es_m where $m < k$. Let p' denote the path from the source state of t_1 to es_k . Suppose that es_j is the last exit state at which the execution of p' is interrupted. We have that $j < k$. From the induction assumption, immediately before the execution p' is resumed at es_j , the path from the source state of t_1 to es_j is consecutively executed. Furthermore, after the execution of p' is resumed, the remaining part of p' is also consecutively executed. So, p' is consecutively executed. \square

In the following, we prove Proposition [2](#) using the above lemma.

Proof. We denote by s_1 the statement $\text{b?msg}(x_1, \dots, x_n)$, by s_2 the statement (B) , by t_1 the transition corresponding to s_1 , by t_2 the transition corresponding to s_2 , and by p the path from the source state of t_1 to the source state of t_2 .

We denote by s_l the exit state within p at which the execution of c is interrupted at the last time in a run. We denote by p' the path from the source state of t_1 to s_l in c , and by p'' the path from s_l to the source state of t_2 in c . So, $p = \langle p', p'' \rangle$. Following Lemma [1](#), before the execution of c is resumed, p' is consecutively executed. Because s_l is the last state at which c is exited, p'' is also consecutively executed after c is re-entered. So, p is consecutively executed before the condition statement s_2 is reached. In this consecutive execution of p a message $\text{msg}(d_1, \dots, d_n)$ is received and each variable $x_i \in \text{var}(B)$ is assigned with d_i . After p is executed, the execution of c can continue if and only if B is true. Since any variable $x_i \in \text{var}(B)$ is not modified in p after receiving the message, we have that $B[x_i \leftarrow d_i]$ is true. \square

Improved On-the-Fly Equivalence Checking Using Boolean Equation Systems

Radu Mateescu and Emilie Oudot*

INRIA/VASY project-team

Faculté des Sciences Mirande, bât. LE2I, F-21000 Dijon, France

{Radu.Mateescu,Emilie.Oudot}@inria.fr

Abstract. Equivalence checking is a classical verification method for ensuring the compatibility of a finite-state concurrent system (*protocol*) with its desired external behaviour (*service*) by comparing their underlying labeled transition systems (LTSS) modulo an appropriate equivalence relation. The local (or *on-the-fly*) approach for equivalence checking combats state explosion by exploring the synchronous product of the LTSS incrementally, thus allowing an efficient detection of errors in complex systems. However, when the two LTSS being compared are equivalent, the on-the-fly approach is outperformed by the global one, which completely builds the LTSS and computes the equivalence classes between states using partition refinement. In this paper, we consider the approach based on translating the on-the-fly equivalence checking problem in terms of the local resolution of a boolean equation system (BES). We propose two enhancements of the approach in the case of equivalent LTSS: a new, faster encoding of equivalence relations in terms of BESS, and a new local BES resolution algorithm with a better average complexity. These enhancements were incorporated into the BISIMULATOR 2.0 equivalence checker of the CADP toolbox, and they led to significant performance improvements w.r.t. existing on-the-fly equivalence checking algorithms.

1 Introduction

Equivalence checking is a classical verification method for finite-state concurrent systems that consists in comparing the behaviour of the system under design (typically, a *protocol* or a low-level hardware description) with its desired external behaviour (typically, a *service* or a high-level hardware description) modulo a suitable equivalence relation. Protocol and service behaviours are usually represented as labeled transition systems (LTSS), and the relations most used for comparing them are the *bisimulations* defined in the framework of process algebras, such as CCS [37], CSP [9], or ACP [6] and of the formal specification languages inspired from them, such as LOTOS [26] or CHP [30]. In practice, LTSS are often represented in two complementary ways, which also determine the nature of equivalence checking algorithms: either *explicitly*, by their list of states

* This research was partially funded by the EC-MOAN project no. 043235 of the FP6-NEST-PATH-COM European program.

and transitions, or *implicitly*, by their “successor function” returning the set of transitions going out of a given state. The implicit and explicit LTS representations are suitable for protocols (which are usually large) and services (which are usually small), respectively.

There are basically two approaches for equivalence checking: the *global* one [15], which operates on explicit LTSS, computes the equivalence classes of states by using partition refinement and then checks whether the initial states of the two LTSS fall into the same equivalence class; and the *local* one [12], which operates on implicit LTSS, explores the synchronous product between the two LTSS and searches for mismatches indicating the non equivalence of their initial states. Global algorithms are more effective when the two LTSS are equivalent, but require their complete construction, which is limited for large systems by the amount of memory available. Local (or *on-the-fly*) algorithms are more effective when the LTSS are not equivalent, allowing the detection of errors in complex systems even when the global approach would fail. Therefore, on-the-fly algorithms are useful at the beginning of the verification process, when errors occur frequently and must be detected quickly, whereas global algorithms are more suitable at a later stage, once the formal descriptions of the protocol and the service are stable and their underlying LTSS become equivalent.

Our objective is to improve the performance of on-the-fly equivalence checking algorithms when the LTSS to be compared are equivalent, which is the worst case for this class of algorithms because it forces them to explore the synchronous product of the two LTSS entirely. This would combine the advantages of global and local verification, making the on-the-fly approach suitable throughout the verification process. We consider here the technique relying on the translation of on-the-fly equivalence checking to the local resolution of a boolean equation system (BES) [2,32]. This technique involves two clearly separated aspects, namely the BES encodings of bisimulation relations and the local BES resolution algorithms, which can be developed and optimized independently. To improve performance, we seek to enhance both these aspects.

First, we devise new BES encodings of the branching [44] and weak [37] bisimulations, obtained by migrating a part of the computation of transitive reflexive closures over internal steps (τ -closures) into the boolean equations. This simplifies the structure of BES equations considerably and reveals to be faster than computing τ -closures separately by using specialized algorithms [31]. Second, we propose a new local BES resolution algorithm, which exhibits a smaller average complexity than previously published algorithms [1,45,17,32]. Our algorithm is based on a suspend/resume depth first search (sr-DFS) of the dependencies between boolean variables, and stops as soon as the BES portion explored contains a single example or counterexample for the boolean variable to be solved, therefore being optimal from this point of view.

These two enhancements led to version 2.0 of the BISIMULATOR [32] equivalence checker of the CADP [22] verification toolbox. The tool was developed using the generic OPEN/CÆSAR [21] environment for on-the-fly manipulation of LTSS, and uses as verification engine the CÆSAR_SOLVE [32] library for on-the-fly

resolution of BESS. The enhancements led to a significant performance increase w.r.t. BISIMULATOR 1.0, as we observed on LTSS generated from protocol and hardware descriptions or taken from the VLTS benchmark suite [46].

Related work. On-the-fly equivalence checking algorithms [12] received relatively little attention from the verification community, the research being mainly focused on optimizing global algorithms based on partition refinement [15,20]. Among the first on-the-fly equivalence checking algorithms were those proposed in [18] and subsequently implemented in the ALDÉBARAN tool [19]. Two different algorithms were elaborated: the first one compares deterministic LTSS by searching their synchronous product for a pair of non equivalent states, and the second one handles nondeterministic LTSS by assuming that certain couples of states are equivalent and by backtracking in the synchronous product whenever such an assumption turns out to be wrong. The verification technique based on BES resolution allows one to reproduce the first algorithm by observing that the BESS corresponding to bisimulations between deterministic LTSS are conjunctive, and by devising a specialized local resolution algorithm for this case [32]. The algorithm for the nondeterministic case is outperformed in practice by local BES resolution algorithms, as it was observed experimentally [5].

Another approach of checking the equivalence of two LTSS is to rephrase the problem as the model checking on one LTS of a *characteristic formula* [25] in modal μ -calculus derived from the other LTS. This approach was elegantly implemented in the Concurrency Workbench [13,11], but was hampered in practice for large LTSS by the prohibitive size of characteristic formulas, which is at least of the same order as the LTS size. The quest for performance was pursued by considering other intermediate formalisms suitable for representing equivalence checking, such as the BESS, which are lower-level than the modal μ -calculus and therefore are likely to require less computation effort.

Encodings of branching and weak bisimulation using BESS of alternation depth two were proposed in [2]. These BESS contain two mutually recursive equation blocks, a maximal fixed point one encoding the bisimulation relation, and a minimal fixed point one encoding the τ -closures to be computed in the input LTSS. The local resolution algorithms underlying this class of BESS have a quadratic complexity w.r.t. the BES size [45], which makes them impractical for large LTSS; no implementation of this approach was reported as far as we know. Although a subquadratic algorithm for solving BESS with disjunctive/conjunctive equation blocks of arbitrary alternation depth was proposed in [24], it does not seem to capture the BESS for branching and weak bisimulations, which consist of a disjunctive block encoding τ -closures and a general block encoding the equivalence relation. Simpler encodings of weak equivalences using alternation-free BESS, obtained by leaving the computation of τ -closures (possibly enhanced with on-the-fly τ -confluence reduction [38]) outside the BES, proved to be practically effective [32]. The resulting BESS can be solved using the many local resolution algorithms available [28,145,29,17,32].

An alternative approach consists in formulating equivalence checking by means of Horn clauses [40], which can be solved using classical HORNSAT

resolution algorithms [16,4]. We believe that BES encodings provide a more direct way of connecting on-the-fly equivalence checking to graph exploration algorithms. In fact, local BES resolution algorithms, such as the one presented in this paper, can also be used for solving HORNSAT efficiently, by applying the translation from Horn clauses to BESs proposed in [29].

Paper outline. Section 2 defines the class of BESs we use and illustrates the functioning of local resolution algorithms. Section 3 proposes new, faster BES encodings for branching and weak bisimulations. Section 4 describes our new local resolution algorithm, and Section 5 shows experimentally its performance when applied to equivalence checking. Finally, Section 6 gives some concluding remarks and directions for future work.

2 Background

A boolean equation system (BES) is a set of possibly recursive equations $B = \{X_i \stackrel{\sigma}{=} X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}}\}_{1 \leq i \leq n}$, where $X_i \in \mathcal{X}$ are boolean variables, $\text{op}_i \in \{\vee, \wedge\}$ are disjunctive or conjunctive connectors, and $\sigma \in \{\mu, \nu\}$ is a minimal or maximal fixed point sign. An empty disjunction (resp. conjunction) is equivalent to the false (resp. true) constant. Each boolean variable occurring in the right-hand side of an equation must be defined by some equation of the BES. A variable X_i is said to be disjunctive (resp. conjunctive) iff $\text{op}_i = \vee$ (resp. \wedge). BESs of this kind are called *simple*, because each of their equations contains a single type of boolean connector (either \vee , or \wedge) in its right-hand side. Any BES containing arbitrary combinations of boolean connectors in the right-hand sides of its equations can be brought to the simple form with at most a linear blow-up in size, by introducing new equations to factor subformulas [3]. We focus our attention on BESs with a single equation block (i.e., set of equations having the same fixed point sign), since they are suitable for encoding equivalence checking problems [32]; more general BESs with multiple blocks are used for encoding model checking problems [14,33]. In-depth presentations of the theory and applications of BESs can be found in [1,34].

For each equation i of a BES, the evaluation of the formula in its right-hand side yields a boolean value defined as follows:

$$\llbracket X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}} \rrbracket \delta = \delta(X_{i_1}) \text{ op}_i \cdots \text{op}_i \delta(X_{i_{m_i}}).$$

where the context $\delta : \mathcal{X} \rightarrow \text{Bool}$ is a partial function assigning boolean values to all variables occurring in the formula. The solution of a BES is a vector $\langle v_1, \dots, v_n \rangle \in \text{Bool}^n$ equal to the fixed point $\sigma\Phi$ of the functional $\Phi : \text{Bool}^n \rightarrow \text{Bool}^n$ associated to the BES:

$$\Phi(b_1, \dots, b_n) = \langle \llbracket X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}} \rrbracket [b_1/X_1, \dots, b_n/X_n] \rangle_{1 \leq i \leq n}$$

where $[b_1/X_1, \dots, b_n/X_n]$ is the context assigning the boolean value b_i to variable X_i for $1 \leq i \leq n$. Since the boolean formulas in a BES do not contain negation

operators, the functional Φ is monotonic, which ensures the existence of its minimal and maximal fixed points on $\langle \text{Bool}^n, \text{false}^n, \text{true}^n, \vee^n, \wedge^n \rangle$, the pointwise extension of the boolean lattice [27]. In the sequel, we consider only maximal fixed point BESS (i.e., with $\sigma = \nu$), which allow to encode equivalence checking.

The local resolution of a BES B , which underlies on-the-fly verification (based on a forward exploration of LTSS), amounts to computing the solution v_i of a particular variable X_i by solving as few equations of B as possible. Local resolution algorithms are easier to devise and understand by representing BESS as *boolean graphs* [1]. Given a BES $B = \{X_i \stackrel{\sigma}{=} X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}}\}_{1 \leq i \leq n}$, its associated boolean graph $G = (V, E, L)$ is defined as follows: $V = \{X_1, \dots, X_n\}$ is the set of vertices (boolean variables), $E = \{(X_i, X_j) \mid 1 \leq i \leq n \wedge j \in \{i_1, \dots, i_{m_i}\}\}$ is the set of edges (dependencies between variables), and $L : V \rightarrow \{\vee, \wedge\}$, $L(X_i) = \text{op}_i$ for $1 \leq i \leq n$ is the labeling of vertices as disjunctive or conjunctive. The constant **false** (resp. **true**) is represented as a sink \vee -vertex (resp. \wedge -vertex). The local resolution of a vertex X_i consists in two activities performed simultaneously [1,45,32]: a forward exploration of the boolean graph along its edges, starting at X_i ; and a backward propagation of the *stable* variables found, i.e., whose boolean value has been computed. An example of local BES resolution is shown on Figure 1. The local resolution algorithm used is based on a depth-first search (DFS) of the boolean graph, starting at the variable of interest X_1 . The light grey area delimits the boolean subgraph explored during resolution. Black (resp. white) vertices correspond to variables whose solution is **true** (resp. **false**).

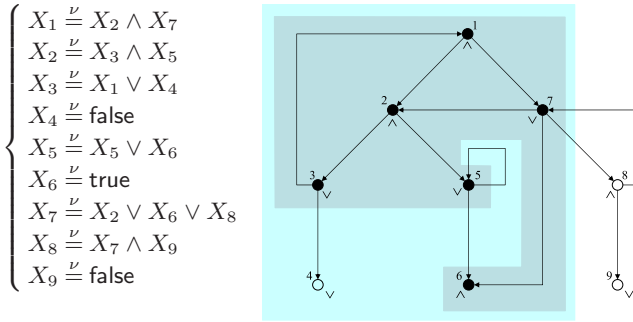


Fig. 1. Boolean graph-based local resolution of variable X_1

The solution of a BES can also be characterized by interpreting on its boolean graph the following *example formula* [35] written in modal μ -calculus:

$$\phi_{ex} = \nu X. (P_{\vee} \wedge \langle - \rangle X) \vee (P_{\wedge} \wedge [-] X)$$

where the atomic propositions P_{\vee} and P_{\wedge} denote \vee -vertices and \wedge -vertices, respectively. Formula ϕ_{ex} expresses that every \vee -vertex (resp. \wedge -vertex) satisfying ϕ_{ex} must have one (resp. all) of its successors satisfying ϕ_{ex} . The solution v_i of

a variable X_i is true iff the corresponding vertex X_i satisfies ϕ_{ex} in the boolean graph. A positive diagnostic (or *example*) for vertex X_i is a boolean subgraph that contains X_i and is a model for ϕ_{ex} . Dually, a negative diagnostic (or *counterexample*) for X_i is a boolean subgraph containing X_i and being a model for the counterexample formula $\phi_{cx} = \neg\phi_{ex}$. The dark grey area shown on Figure 1 delimits an example for X_1 , generated by traversing again the boolean subgraph explored during resolution [35].

3 Encoding Bisimulation Relations as BESs

Labeled transition systems (LTSS) are the semantic model underlying process algebras [7] and the related languages, such as LOTOS [26] and CHP [30]. An LTS is a quadruple $M = \langle Q, A, T, q_0 \rangle$, where Q is the set of states, A is the set of actions (including the internal action τ), $T \subseteq Q \times A \times Q$ is the transition relation, and $q_0 \in Q$ is the initial state. A transition $\langle p, a, q \rangle \in T$ (also written $p \xrightarrow{a} q$) indicates that the system can move from state p to state q by performing action a . The notation is extended to transition sequences: $p \xrightarrow{l} q$ denotes the existence of a sequence going from p to q and whose concatenated labels form a word of the language $l \subseteq A^*$.

To compare the LTSS modeling the behaviour of concurrent systems, various equivalence relations were proposed (see [43] for a survey), among which *bisimulations* are most useful in practice due to their congruence properties w.r.t. the parallel composition operators of process algebras. We consider here three widely-used bisimulations: strong [39], branching [44], and weak [37], the last two being originally proposed as native equivalence relations for ACP [6] and CCS [37], respectively. Given two LTSS $M_i = \langle Q_i, A_i, T_i, q_{0i} \rangle$ with $i \in \{1, 2\}$, a bisimulation $\approx \subseteq Q_1 \times Q_2$ is a relation such that $p \approx q$ iff $\forall p \xrightarrow{a} p'. \exists q \xrightarrow{a} q'. p' \approx q'$ and $\forall q \xrightarrow{a} q'. \exists p \xrightarrow{a} p'. p' \approx q'$, where $p, p' \in Q_1$, $a \in A_1 \cup A_2$, and $q, q' \in Q_2$. Bisimulations are closed under union, and the strong bisimulation \approx_s is defined as the greatest one, i.e., the union of all bisimulations. M_1 is strongly equivalent to M_2 (notation $M_1 \approx_s M_2$) iff $q_{01} \approx_s q_{02}$.

A basic encoding of this mathematical definition as a maximal fixed point BES is shown in Table 1 (upper part, first row). The fact that $p \approx_s q$ is encoded as a boolean variable X_{pq} defined by an equation whose right-hand side boolean formula is directly derived from the two bisimulation conditions. The correctness of this encoding scheme (which reformulates the definition of strong bisimulation in propositional logic instead of first-order logic), relies on a bijection between the set of bisimulations and the set of fixed point solutions of the functional associated to the BES. To obtain a simple BES compliant with the definition given in Section 2, we introduce the new variables $Y_{p'qa}$ and $Z_{pq'a}$ such that each right-hand side formula contains a single type of boolean connector (second row). The BES for the strong preorder relation \preceq_s is obtained by keeping only the coloured parts of the equations. Checking the strong bisimilarity of M_1 and M_2 amounts to solving the variable $X_{q_{01}q_{02}}$ of this BES, which can be carried out using a local resolution algorithm. The evaluation of the boolean formulas

in the right-hand sides of the equations defining X_{pq} , $Y_{p'qa}$, and $Z_{pq'a}$ triggers a forward exploration of transitions in M_1 and M_2 , which enables an incremental construction of both LTSS, and therefore an on-the-fly verification.

Table 1. Basic and full BES encodings of three widely-used bisimulations

Strong bisimulation	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} \bigvee_{q \xrightarrow{a} q'} X_{p'q'} \wedge \bigwedge_{q \xrightarrow{a} q'} \bigvee_{p \xrightarrow{a} p'} X_{p'q'}$	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{p'qa} \wedge \bigwedge_{q \xrightarrow{a} q'} Z_{pq'a}$	$Y_{p'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} X_{p'q'}$ $Z_{pq'a} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} X_{p'q'}$
Branching bisimulation	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} ((a = \tau \wedge X_{p'q}) \vee \bigvee_{q \xrightarrow{\tau^*} q'} \bigvee_{q' \xrightarrow{a} q''} (X_{pq'} \wedge X_{p''q''})) \wedge \bigwedge_{q \xrightarrow{a} q'} ((a = \tau \wedge X_{pq'}) \vee \bigvee_{p \xrightarrow{\tau^*} p'} \bigvee_{p'' \xrightarrow{a} p'''} (X_{p'q} \wedge X_{p''q''}))$	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{pp'qa} \wedge \bigwedge_{q \xrightarrow{a} q'} Z_{pp'qa}$	$Y_{pp'qa} \stackrel{\nu}{=} (a = \tau \wedge X_{p'q}) \vee U_{pp'qa}$
$Z_{pp'qa} \stackrel{\nu}{=} (a = \tau \wedge X_{pq'}) \vee V_{pp'qa}$	$U_{pp'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} W_{pp'qq'} \vee \bigvee_{q \xrightarrow{\tau} q'} U_{pp'q'a}$
$V_{pp'qa} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} W_{pp'qq'} \vee \bigvee_{p \xrightarrow{\tau} p'} V_{p'qq'a}$	$W_{pp'qq'} \stackrel{\nu}{=} X_{pq} \wedge X_{p'q'}$
Weak bisimulation	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} ((a = \tau \wedge \bigvee_{q \xrightarrow{\tau^*} q'} X_{p'q'}) \vee \bigvee_{q \xrightarrow{\tau^*} q'} \bigvee_{q' \xrightarrow{a} q''} X_{p'q''}) \wedge \bigwedge_{q \xrightarrow{a} q'} ((a = \tau \wedge \bigvee_{p \xrightarrow{\tau^*} p'} X_{p'q'}) \vee \bigvee_{p \xrightarrow{\tau^*} p'} \bigvee_{p'' \xrightarrow{a} p'''} X_{p'q''})$	
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{p'qa} \wedge \bigwedge_{q \xrightarrow{a} q'} Z_{pq'a}$	
$Y_{p'qa} \stackrel{\nu}{=} (a = \tau \wedge U_{p'q}) \vee V_{p'qa}$	$U_{p'q} \stackrel{\nu}{=} X_{p'q} \vee \bigvee_{q \xrightarrow{\tau} q'} U_{p'q'}$
$V_{p'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} U_{p'q'} \vee \bigvee_{q \xrightarrow{\tau} q'} V_{p'q'a}$	$Z_{pq'a} \stackrel{\nu}{=} (a = \tau \wedge W_{pq'}) \vee T_{pq'a}$
$W_{pq'a} \stackrel{\nu}{=} X_{pq'} \vee \bigvee_{p \xrightarrow{\tau} p'} W_{p'q'}$	$T_{pq'a} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} W_{p'q'} \vee \bigvee_{p \xrightarrow{\tau} p'} T_{p'q'a}$

Similar encoding schemes hold for the branching (\approx_b) and weak (\approx_w) bisimulations, as shown in Table 1 (middle and lower parts, first rows). The important difference w.r.t. strong bisimulation is the presence of transitive reflexive closures over τ -transitions, which correspond to the abstraction of internal activity done by these two bisimulations. The simple BESs derived from these encodings by introducing new variables (similarly to strong bisimulation as shown above) were successfully used as basis for on-the-fly equivalence checking in conjunction with linear-time local BES resolution algorithms [32]. For LTSS with a high percentage of τ -transitions, the encodings of weak bisimulations yield relatively small BESs, but shift the computation effort to the evaluation of right-hand side boolean formulas, which involve various forms of τ -closures ($p \xrightarrow{\tau^*} p' \xrightarrow{a} p''$, $p \xrightarrow{\tau^*} p' \xrightarrow{a} p''$, and $p \xrightarrow{\tau^*} p'$) having a quadratic worst-case complexity. Practical usage confirmed that computation of τ -closures, even using optimized algorithms [31], is the most time-consuming part of the verification process.

An alternative solution for computing τ -closures would be to encode them directly using boolean equations, yielding the BESs shown on Table 1 (middle

and lower part, second rows); however, this works only for LTSS without τ -cycles. To see this, consider the two LTSS $M_1 = \langle \{p_0, p_1\}, \{a, \tau\}, \{p_0 \xrightarrow{\tau} p_0, p_0 \xrightarrow{a} p_1\}, p_0 \rangle$ and $M_2 = \langle \{q_0, q_1\}, \{b, \tau\}, \{q_0 \xrightarrow{\tau} q_0, q_0 \xrightarrow{b} q_1\}, q_0 \rangle$, which are obviously not equivalent modulo any of the three bisimulations considered since they have different action sets. The comparison of M_1 and M_2 modulo weak bisimulation yields the BES below:

$$\begin{array}{llll}
 & X_{p_0q_0} \stackrel{\nu}{=} Y_{p_0q_0\tau} \wedge Y_{p_1q_0a} \wedge Z_{p_0q_0\tau} \wedge Z_{p_0q_1b} & & \\
 Y_{p_0q_0\tau} \stackrel{\nu}{=} U_{p_0q_0} \vee V_{p_0q_0\tau} & U_{p_0q_0} \stackrel{\nu}{=} X_{p_0q_0} \vee U_{p_0q_0} & V_{p_0q_0\tau} \stackrel{\nu}{=} V_{p_0q_0\tau} & \\
 Z_{p_0q_0\tau} \stackrel{\nu}{=} W_{p_0q_0} \vee T_{p_0q_0\tau} & W_{p_0q_0} \stackrel{\nu}{=} X_{p_0q_0} \vee W_{p_0q_0} & T_{p_0q_0\tau} \stackrel{\nu}{=} T_{p_0q_0\tau} & \\
 Y_{p_1q_0a} \stackrel{\nu}{=} V_{p_1q_0a} & V_{p_1q_0a} \stackrel{\nu}{=} V_{p_1q_0a} & Z_{p_0q_1b} \stackrel{\nu}{=} T_{p_0q_1b} & T_{p_0q_1b} \stackrel{\nu}{=} T_{p_0q_1b}
 \end{array}$$

We can easily compute the maximal fixed point solution of this BES by using the iterative characterization [27], which consists in initializing all variables to **true** and repeatedly evaluating the right-hand sides of equations until the values of all variables become stable; the process converges in one iteration and all variables remain **true**, erroneously indicating that $M_1 \approx_w M_2$. The problem here is that τ -closures express the existence of *finite* τ -sequences in the LTSS, and hence they correspond to *minimal* fixed point computations, which cannot be done accurately by solving the equations of a *maximal* fixed point BES. On the other hand, if we eliminate the two τ -loops in M_1 and M_2 , the BES becomes:

$$X_{p_0q_0} \stackrel{\nu}{=} Y_{p_1q_0a} \wedge Z_{p_0q_1b} \quad Y_{p_1q_0a} \stackrel{\nu}{=} \text{false} \quad Z_{p_0q_1b} \stackrel{\nu}{=} \text{false}$$

and yields the correct solution **false** for the variable $X_{p_0q_0}$. This is a consequence of the fact that minimal and maximal fixed points have the same interpretation on acyclic models, as shown in [36] for modal μ -calculus formulas. Thus, if the LTSSs being compared do not contain τ -cycles, the encoding of τ -closures using maximal fixed point equations is correct. The elimination of τ -cycles by collapsing their states (also called τ -compression), which preserves both branching and weak bisimulation, can be performed in linear-time during an on-the-fly LTS exploration [31], using an adaptation of Tarjan’s algorithm [41] for detecting strongly connected components (SCCs). To make the BES encodings in Table 1 correct, it is therefore sufficient to reduce both LTSS on-the-fly by applying τ -compression simultaneously with the local BES resolution.

The new BESS obtained in this way for branching and weak bisimulation have a size comparable (see Figure 3 (a), (b)) with the BESS resulting from the previous encodings in which τ -closures were computed separately by specialized algorithms [31]; however, we observed experimentally that their resolution (using the same algorithms) is about one order of magnitude faster. This is due to the fact that intermediate results of τ -closure computations are stored as values of the boolean variables used to encode τ -closures (e.g., variables $U_{pp'qa}$ and $V_{ppq'a}$ of the BES for branching bisimulation), which are retrieved immediately if needed again during resolution; the only risk with this scheme was a too important quantity of such variables, which was not observed in practice. We also encoded as BESS, using similar schemes, the $\tau^*.a$ [18] and safety [8] equivalences, which are weaker than branching bisimulation and slightly less used in practice.

4 Local BES Resolution Based on Suspend/Resume DFS

Several local BES resolution algorithms with a linear-time complexity are available [145,17,32], typically based on DFS or breadth-first search (BFS) strategies for exploring the dependencies between boolean variables, i.e., the edges of the boolean graph. Here we aim to satisfy the following optimality criterion for local BES resolution algorithms, based on the notion of diagnostic [35]: the resolution must stop as soon as the boolean subgraph already explored contains exactly one diagnostic (example or counterexample) for the variable of interest. To our knowledge, all existing algorithms satisfy only a half of this criterion, i.e., they detect optimally either the presence of examples, or of counterexamples, but not of both of them. The LMC algorithm proposed in [17], based on a DFS traversal of the boolean graph with computation of SCCs, detects counterexamples optimally and speeds up the search of examples (in maximal fixed point BESS) without attempting their optimal detection.

In the BESS produced from equivalence checking problems, false constants (sink \vee -vertices) denote couples of non equivalent states; if their backward propagation along edges in the boolean graph is done as soon as these vertices are encountered, it leads to an optimal detection of counterexamples, as in the A0 algorithm proposed in [32]. However, when the LTSS being compared are equivalent, the variable of interest is true and the associated diagnostic is an *example*, which must be detected as soon as possible during resolution. Using the characterization of examples induced by the μ -calculus formula ϕ_{ex} given in Section 2, we can draw an alternative graph-based characterization: an example for vertex X is a subgraph containing X in which every \vee -vertex (resp. \wedge -vertex) must have exactly one successor (resp. all its successors) contained in the example. Each example can be split into maximal SCCs, which are connected acyclically; in the sequel, we denote them as *pseudo-SCCs*, since they are special cases of SCCs in the boolean graph (for instance, a trivial SCC containing a single sink \vee -vertex denotes a false constant, which is neither an example, nor a pseudo-SCC). These pseudo-SCCs are the smallest “building blocks” of the examples, and therefore their presence in the boolean subgraph already explored must be determined as soon as possible in order to achieve an optimal detection of examples.

To detect pseudo-SCCs, one can adapt Tarjan’s algorithm [41], which relies on a DFS traversal. The problem is that a classical DFS of the boolean graph does not allow the detection of pseudo-SCCs as soon as they occur, because Tarjan’s algorithm identifies SCCs only when their root vertex is popped from the DFS stack, meaning that the subgraph reachable from the root has been entirely explored; this subgraph may very well contain other pseudo-SCCs, which could make several examples to be contained in the boolean subgraph explored at the end of the resolution, i.e., when the variable of interest will be popped in turn from the DFS stack (if it evaluates to true, this variable is the root of the last pseudo-SCC identified). In order to detect the first pseudo-SCC encountered, it is necessary to *suspend* the DFS for each \vee -vertex when one of its successors was already visited; if this successor turns out to be false at some later stage (and thus not part of a pseudo-SCC), and the \vee -vertex is encountered again, it is

necessary to *resume* the DFS by considering some other successor of the \vee -vertex that may belong to a pseudo-SCC. The exploration of \wedge -vertices is done as in the classical DFS, since for each \wedge -vertex, all its successors must be visited before attempting to detect a pseudo-SCC containing it.

The local resolution algorithm sr-DFS that we propose, based on this suspend/resume DFS, is illustrated below. Taking as input a boolean graph $G = (V, E, L)$ represented implicitly (i.e., by its successor function) and a variable of interest x , the algorithm performs iteratively a forward search of G starting at vertex x . Visited vertices are stored in a set $A \subseteq V$. The DFS stack is stored in a variable dfs and the stack used for detecting pseudo-SCCs is stored in a variable scc . The variable $count$ keeps a global counter allowing the assignment of unique DFS numbers to visited vertices. To each vertex v are associated the following fields:

- a counter $c(v)$ which counts the number of remaining successors of v to visit in order to stabilize v ;
- a number $p(v)$ recording the index of the next successor of v to be visited (the successors in $E(v)$ are supposed to be indexed from 0 to $|E(v)| - 1$);
- a number $n(v)$ representing the DFS number of v ;
- a number $l(v)$ representing the “lowlink” number [41] of v , used to detect if a vertex is the root of a pseudo-SCC;
- a set $d(v)$ containing the vertices that currently depend upon v ;
- a boolean $on_scc(v)$ which is true if v is on the scc stack and false otherwise;
- a boolean $stop(v)$ which is true if the DFS must be suspended for v (i.e., v is a \vee -vertex and one of its successors has been visited);
- a boolean $stable(v)$, which is true if v is stable;
- a boolean $value(v)$, which represents the value of v (this field is of interest only if v is stable).

At each iteration of the main while-loop (lines 20–122), the vertex y at the top of the dfs stack is explored. If y is stable, or the DFS must be suspended for y (that is, y is a \vee -vertex and one of its successors has already been visited), the value of y is back-propagated along its predecessors d (lines 30–62). For each vertex w which is not stabilized by the back-propagation, the algorithm must keep on visiting its successors, if w will be visited again during the DFS (the variable $stop(w)$ becomes false). Due to the suspend/resume principle, this propagation phase may influence the contents of the pseudo-SCC currently stored on the scc stack. Indeed, each \vee -vertex which is visited during the propagation phase is stored on the scc stack. The definition of pseudo-SCCs requires that each \vee -vertex must have exactly one successor contained in its pseudo-SCC. But, as the vertex was not stabilized by the value of the successor which was propagated, it does not meet any more the definition of the pseudo-SCC. Since the scc does not contain a pseudo-SCC anymore, it must be cleared. A variable called *purge* is used for this purpose and becomes true when the scc stack must be cleared (two variables *min* and *max* are used to determine if scc must be cleared: *min* represents the least DFS number among all the DFS numbers of vertices

Algorithm 1. Local BES resolution based on suspend/resume DFS

```

1: function sr-DFS ( $x, (V, E, L)$ ) : Bool is
2: var  $A, B : 2^V$ ;  $d : V \rightarrow 2^V$ ;
3:    $u, w, y, z : V$ ;  $dfs, scc : V^*$ ;
4:    $c, p, n, l : V \rightarrow \text{Nat}$ ;
5:    $stop, stable : V \rightarrow \text{Bool}$ ;
6:    $value, on\_scc : V \rightarrow \text{Bool}$ ;
7:    $count, max, min : \text{Nat}$ ;
8:    $purge : \text{Bool}$ ;
9: if  $L(x) = \wedge$  then
10:   $c(x) := |E(x)|$ 
11: else
12:   $c(x) := 1$ 
13: end if
14:  $p(x) := 0$ ;  $stable(x) := \text{false}$ ;
15:  $d(x) := \emptyset$ ;  $value(x) := \text{false}$ ;
16:  $A := \{x\}$ ;  $count := 0$ ;
17:  $dfs := \text{push}(x, nil)$ ;
18:  $scc := \text{push}(x, nil)$ ;
19:  $on\_scc(x) := \text{true}$ ;  $stop(x) := \text{false}$ ;
20: while  $dfs \neq nil$  do
21:   $y := top(dfs)$ ;
22:   $n(y) := count$ ;
23:   $l(y) := n(y)$ ;
24:   $count := count + 1$ ;
25:   $max := 0$ ;
26:   $min := n(y)$ ;
27:  if  $stable(y) \vee stop(y)$  then
28:   if  $d(y) \neq \emptyset$  then
29:     $B := \{y\}$ ;
30:    while  $B \neq \emptyset$  do
31:     let  $u \in B$ ;
32:      $B := B \setminus \{u\}$ ;
33:     for all  $w \in d(u)$  do
34:      if  $\neg stable(w)$  then
35:       if  $((L(w) = \vee) \wedge value(u)) \vee ((L(w) = \wedge) \wedge \neg value(u))$  then
36:         $c(w) := 0$ 
37:       else
38:         $c(w) := c(w) - 1$ 
39:       end if
40:       if  $c(w) = 0$  then
41:         $stable(w) := \text{true}$ ;
42:         $value(w) := value(u)$ ;
43:         $B := B \cup \{w\}$ ;
44:        if  $n(w) < min$  then
45:          $min := n(w)$ 
46:        end if
47:       else
48:         $stop(w) := \text{false}$ ;
49:        if  $L(w) = \wedge$  then
50:          $c(w) := 1$ ;
51:          $p(w) := 0$ 
52:        else
53:          $E(w) := E(w) \setminus \{u\}$ ;
54:         if  $n(w) > max$  then
55:           $max := n(w)$ 
56:         end if
57:        end if
58:       end if
59:     end for
60:      $d(u) := \emptyset$ 
61:
62:     end while;
63:     if  $max > min$  then
64:       $purge := \text{true}$ 
65:     end if
66:   else
67:     $dfs := pop(dfs)$ ;
68:    if  $dfs \neq nil$  then
69:      $l(top(dfs)) := \min(l(top(dfs)), l(y))$ 
70:    end if
71:   end if
72: else
73:  if  $purge$  then
74:   while  $top(scc) \neq top(dfs)$  do
75:     $scc := pop(scc)$ 
76:   end while;
77:    $purge := \text{false}$ 
78:  end if
79:  if  $p(y) < |E(y)|$  then
80:   if  $L(y) = \vee$  then
81:     $stop(y) := \text{true}$ 
82:   end if
83:    $z := (E(y))_{p(y)}$ ;
84:    $p(y) := p(y) + 1$ ;
85:    $d(z) := d(z) \cup \{y\}$ ;
86:   if  $z \in A$  then
87:    if  $on\_scc(z)$  then
88:     if  $n(z) < n(y)$  then
89:       $l(y) := \min(n(z), n(y))$ 
90:     end if
91:    else
92:      $dfs := \text{push}(z, dfs)$ ;
93:      $scc := \text{push}(z, scc)$ ;
94:      $on\_scc(z) := \text{true}$ 
95:    end if
96:   else
97:    if  $L(z) = \wedge$  then
98:      $c(z) := |E(z)|$ 
99:    else
100:      $c(z) := 1$ 
101:    end if;
102:     $p(z) := 0$ ;
103:     $A := A \cup \{z\}$ ;
104:     $dfs := \text{push}(z, dfs)$ ;
105:     $scc := \text{push}(z, scc)$ ;
106:     $on\_scc(z) := \text{true}$ 
107:   end if
108:  else
109:   if  $(l(y) = n(y)) \wedge (top(scc) \neq y)$  then
110:    repeat
111:      $z := top(scc)$ ;
112:      $c(z) := 0$ ;
113:      $scc := pop(scc)$ 
114:    until  $top(scc) = y$ 
115:   end if
116:    $dfs := pop(dfs)$ ;
117:   if  $dfs \neq nil$  then
118:     $l(top(dfs)) := \min(l(top(dfs)), l(y))$ 
119:   end if
120:  end if
121: end if
122: end while;
123: return  $value(x)$ 

```

stabilized during the propagation phase and max represents the greatest DFS number among all \vee -vertices towards which a false value has been propagated).

If the vertex y at the top of the dfs stack is unstable or that the exploration must continue for this vertex, its next unexplored successor $z = E(y)_{p(y)}$ is visited. Before that, the scc stack is cleared if needed (i.e., if a back-propagation of a false value took place at some previous iteration of the main while-loop). If z is a new vertex (lines 96-107), it is pushed on the dfs stack. If z is an already explored vertex, two cases may appear. Either z is on the scc stack (lines 87-90) and therefore its lowlink number must be updated, or it is not on the stack (lines 91-95), and therefore it must be explored as if it was a new vertex (i.e., z was popped from the scc stack after a propagation phase which induced a clearing of this stack). Finally, if y is unstable and all its successors have been visited, the algorithm watches if y is the root of a pseudo-SCC (lines 108-120). If this is the case, the scc stack is cleared from its top until y and each vertex of the pseudo-SCC is stabilized. Then, y is popped from the dfs stack. Finally, after termination of the main while-loop, the value computed for x is returned.

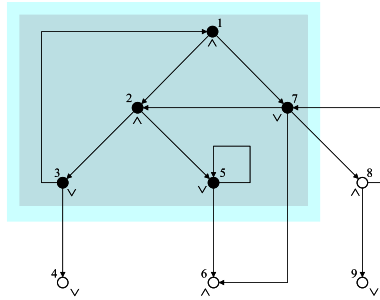


Fig. 2. Local resolution of variable X_1 using the sr-DFS algorithm

The execution of algorithm sr-DFS on the boolean graph considered in Section 2 is illustrated on Figure 2. We observe on this example an optimal behaviour of sr-DFS: due to the suspension of the DFS for the \vee -vertices X_3 , X_5 , and X_7 when one of their successors was visited, the subgraph explored by the algorithm (light grey area) coincides with the example found for vertex X_1 (dark grey area), made of the pseudo-SCCs $\{X_1, X_2, X_3, X_7\}$ and $\{X_5\}$. The resolution previously shown on Figure 1 was done using the algorithm A0 [32], which is based on a classical DFS without computation of SCCs, and therefore explores a larger subgraph than sr-DFS in order to find another, larger example for X_1 .

Complexity. For boolean graphs $G = (V, E, L)$ without sink \vee -vertices (i.e., for maximal fixed point BESS without false constants in the right-hand sides of their equations), the sr-DFS algorithm has a linear-time complexity $O(|V| + |E|)$. The presence of false constants could trigger the reexploration of some vertices (those present on the portions of the scc stack that were cleared after back-propagation

of false constants), increasing the complexity of the algorithm towards quadratic-time $O((|V| + |E|)^2)$, which is the theoretical worst-case. This is the price to pay for achieving an optimal detection of examples and counterexamples in the boolean graph. However, the behaviour of the sr-DFS algorithm that we observed in practice for equivalence checking (by measuring the number of variables explored and reexplored) shows that its complexity is close to linear-time.

5 Implementation and Experiments

The `CÆSAR_SOLVE` [32] library of CADP [22] provides a generic implementation of several local BES resolution algorithms. The library was developed using the `OPEN/CÆSAR` [21] generic environment for LTS manipulation, which offers many graph exploration primitives (stacks, hash tables, edge lists, etc.). BESs are handled by `CÆSAR_SOLVE` by means of their corresponding boolean graphs, represented implicitly in a way similar to LTSS in `OPEN/CÆSAR`. This representation is application-independent, allowing to employ the resolution algorithms as computing engines for several on-the-fly verification tools of CADP: the model checker `EVALUATOR` [33], the equivalence checker `BISIMULATOR` [5,32], and the `REDUCTOR` tool for LTS generation equipped with partial order reductions.

Table 2. Algorithms of `CÆSAR_SOLVE` and their application to equivalence checking

Alg.	BES type	Strategy	Time	Memory	Condition	
A0	general	DFS	$O(V + E)$	$O(V + E)$	nondeterministic LTSS	
A1		BFS				
A2	acyclic	DFS		$O(V)$	one LTS acyclic	
A3	disjunctive				—	
A4	conjunctive				one LTS deterministic, τ -free	
A5	general	BFS		$O(V)$	$O(V + E)$	nondeterministic LTSS
A6	disjunctive				—	
A7	conjunctive		one LTS deterministic, τ -free			

Table 2 summarizes the local resolution algorithms currently available in the `CÆSAR_SOLVE` library and their application for equivalence checking within `BISIMULATOR`. All algorithms have a linear complexity w.r.t. the size of boolean graphs (number of vertices and edges). Algorithms A0, A1, and A5 can solve general BESs (without constraints on the structure of equations), A1 being BFS-based and thus able to produce small-depth diagnostics. When one LTS is deterministic (for strong equivalence) and τ -free (for weak equivalences), the resulting BES is conjunctive and can be solved using the memory-efficient algorithm A4 [32], which stores only the vertices of the boolean graph (and not its edges), i.e., only the states of the LTSS (and not their transitions). Also, when one LTS is acyclic, the resulting BES is also acyclic (i.e., it has an acyclic boolean graph) and can be solved using the memory-efficient algorithm A2. The BFS-based algorithm A7,

recently added to the library, can be applied to conjunctive BESs and combines the advantages of algorithms A1 (small-depth diagnostics) and A4 (low memory consumption) when one LTS is deterministic and τ -free.

The version `BISIMULATOR 2.0` includes the new BES encodings of weak equivalences defined in Section 3 and the new resolution algorithm `sr-DFS` given in Section 4 which was recently added to `CESAR-SOLVE` with the number A8. In the sequel, we present various performance measures showing the effect of these two enhancements. The LTSS considered were generated from the demo examples of CADP (specifications of communication protocols and asynchronous circuits) or taken from the VLTS benchmark suite [46].

New encodings of weak equivalences. The new BES encodings of weak equivalences that we proposed in Section 3 compute τ -closures by means of BES equations instead of relying on external, dedicated graph algorithms as the previous encodings used in `BISIMULATOR 1.0`. Figure 3(a)–(b) compares the performance of the two encodings for branching bisimulation as regards the size of the underlying BESs and their resolution time using algorithm A0. As expected, the BESs produced by the new encoding are larger (more variables but less operators) because intermediate results of τ -closure computations are stored as boolean variables, but they are solved faster due to the simpler structure of boolean equations. Of course, what matters from the end-user point of view is the overall performance of using `sr-DFS` in conjunction with the new BES encoding; this is illustrated below.

Resolution using the `sr-DFS` algorithm. The series of experiments shown in Figure 3(c)–(f) compare the behaviour of `BISIMULATOR 1.0` (algorithm A0 and previous BES encoding) w.r.t. version 2.0 (algorithm `sr-DFS` and new BES encoding) for branching bisimulation. To improve readability, we separated the LTSS in two groups according to their number of transitions. When applying version 2.0, we observed reductions of both the number of vertices visited and edges explored, which determine the memory consumption and the execution time, respectively. These reductions become more important as the LTS size increases, as indicated by curves (e) and (f); in particular, the number of transitions traversed can decrease by a factor 8. It is worth noticing that some of the LTSS compared were not equivalent (e.g., certain erroneous variants of a leader election protocol examined in [23]), showing that version 2.0 of the tool exhibits a good behaviour also for counterexample detection. These experimental results indicate that the increase in BES size induced by the new encoding of weak equivalences is compensated by the reduction achieved using `sr-DFS`, leading to an overall improvement of the on-the-fly verification procedure. As regards strong bisimulation, `sr-DFS` reduces the number of variables explored by up to 25%, as shown in Figure 3(g).

Complexity w.r.t. theoretical worst-case. As pointed out in [18], the on-the-fly comparison of two nondeterministic LTSS $M_1 = \langle Q_1, A_1, T_1, q_{01} \rangle$ and $M_2 = \langle Q_2, A_2, T_2, q_{02} \rangle$ has a worst-case complexity $O((|Q_1| \cdot |T_2|) + (|Q_2| \cdot |T_1|))$. Considering the BES formulation of the problem, this complexity can be estimated

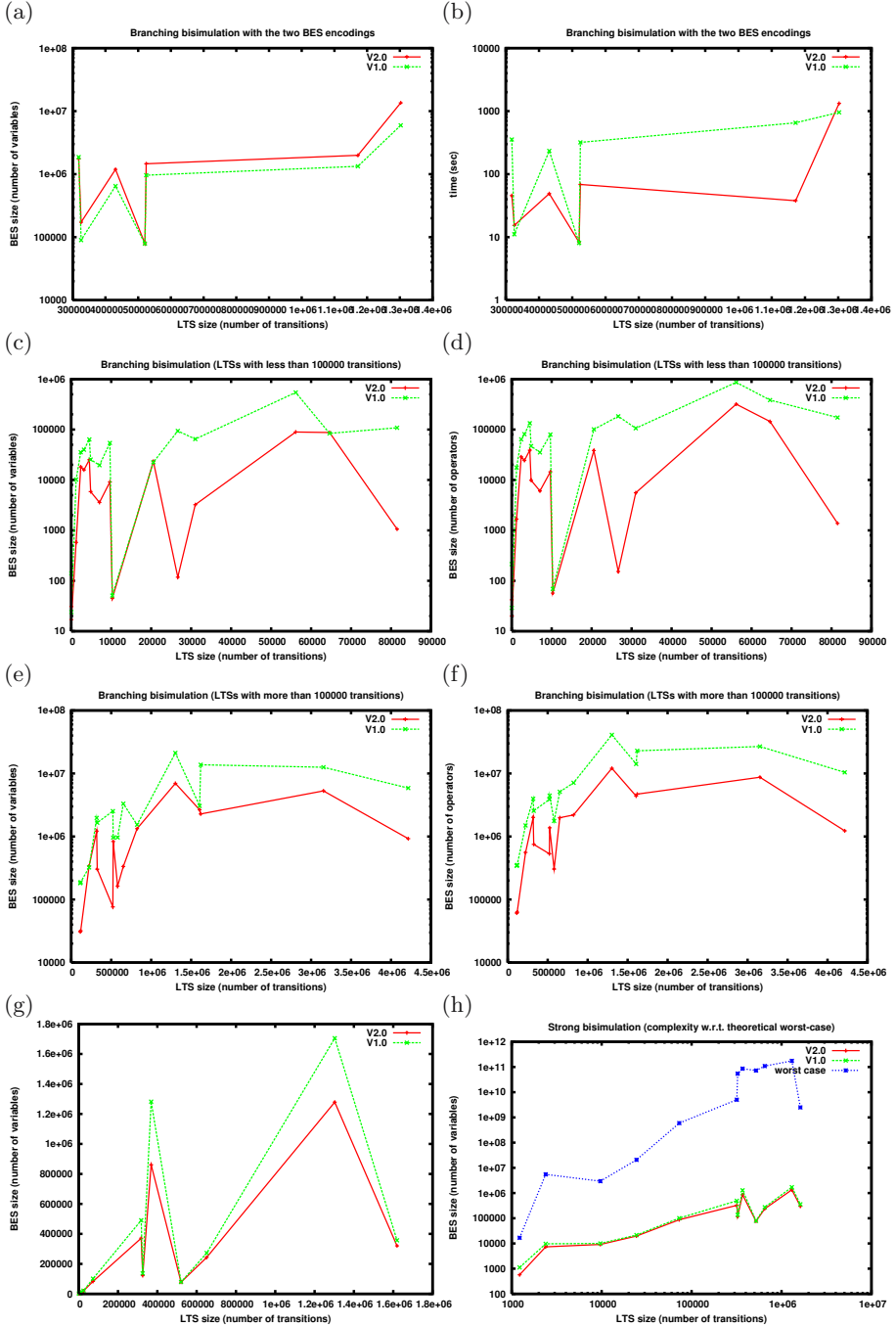


Fig. 3. Performance of equivalence checking using BISIMULATOR 1.0 and 2.0

in terms of BES size: the BESS given in Table 11 have a number of boolean variables proportional to the size of the synchronous product between the two LTSS. However in practice, the BESS produced from equivalence checking have a much smaller size (several orders of magnitude) than the theoretical worst-case, as it is illustrated in Figure 3(h) for strong bisimulation. This also holds for weak equivalences, in particular for branching bisimulation.

6 Conclusion and Future Work

Building efficient software tools for on-the-fly equivalence checking between LTSS is a difficult and time-consuming task. The usage of intermediate formalisms, such as BESS, allows one to separate the concerns of phrasing the verification problem and of solving it, leading to highly modular verification tools [33,5]. The two optimizations we proposed, namely the new encodings of weak equivalences by applying τ -compression on the input LTSS and computing τ -closures using boolean equations (Section 3) and the new sr-DFS local BES resolution algorithm (Section 4) significantly increased the performance of on-the-fly equivalence checking w.r.t. existing approaches.

These optimizations underlie the new version 2.0 of the BISIMULATOR equivalence checker [32] of the CADP toolbox [22]. The sr-DFS algorithm was integrated to the generic CÆSAR_SOLVE library [32] for on-the-fly BES resolution, which is part of the generic OPEN/CÆSAR environment [21] for LTS manipulation. Local BES resolution proved to be a suitable alternative way for computing τ -closures on LTSS produced from protocols and distributed systems, competing favourably with general transitive closure algorithms. The sr-DFS algorithm is able to detect optimally the presence of both examples and counterexamples in the boolean graph, and appears to be quite effective for comparing LTSS modulo weak equivalences.

We plan to continue our work along two directions. First, the range of equivalences and preorders already available in BISIMULATOR 2.0 (strong, branching, weak, $\tau^*.a$, safety, trace, and weak trace) could be extended by devising BES encodings for other weak equivalences, such as CFFD [42] and testing equivalence [10], following the scheme in Section 3. Next, we will pursue experimenting the sr-DFS algorithm and study its applicability for solving BESS coming from other verification problems, such as the model checking of alternation-free modal μ -calculus and the on-the-fly LTS reduction modulo partial order relations (e.g., τ -confluence, τ -inertness, etc.) as formulated in [38].

References

1. Andersen, H.R.: Model checking and boolean graphs. TCS 126, 3–30 (1994)
2. Andersen, H.R., Vergauwen, B.: Efficient checking of behavioural relations and modal assertions using fixed-point inversion. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 142–154. Springer, Heidelberg (1995)
3. Arnold, A., Crubillé, P.: A linear algorithm to solve fixed-point equations on transition systems. Information Processing Letters 29, 57–66 (1988)

4. Ausiello, G., Italiano, G.F.: On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming* 10, 69–90 (1991)
5. Bergamini, D., Descoubes, N., Joubert, C., Mateescu, R.: Bisimulator: A modular tool for on-the-fly equivalence checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 581–585. Springer, Heidelberg (2005)
6. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Information and Computation* 60 (1984)
7. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): *Handbook of Process Algebra*. Elsevier, Amsterdam (2001)
8. Bouajjani, A., Fernandez, J.C., Graf, S., Rodríguez, C., Sifakis, J.: Safety for branching time semantics. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, Springer, Heidelberg (1991)
9. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. *Journal of the ACM* 31, 560–599 (1984)
10. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing* 5, 1–20 (1993)
11. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: A semantics-based verification tool for finite state systems. *ACM TOPLAS* 15, 36–72 (1993)
12. Cleaveland, R., Sokolsky, O.: Equivalence and preorder checking for finite-state systems. In: *Handbook of Process Algebra*, pp. 391–424. North-Holland, Amsterdam (2001)
13. Cleaveland, R., Steffen, B.: Computing behavioural relations, logically. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 127–138. Springer, Heidelberg (1991)
14. Cleaveland, R., Steffen, B.: A linear-time model-checking algorithm for the alternation-free modal μ -calculus. *FMSD* 2, 121–147 (1993)
15. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. *TCS* 311, 221–256 (2004)
16. Dowling, W., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming* 3 (1984)
17. Du, X., Smolka, S.A., Cleaveland, R.: Local model checking and protocol analysis. *STTT* 2, 219–241 (1999)
18. Fernandez, J.C., Mounier, L.: Verifying bisimulations on the fly. In: *Proc. of FORTE 1990* (1990)
19. Fernandez, J.C., Mounier, L.: A tool set for deciding behavioral equivalences. In: Groote, J.F., Baeten, J.C.M. (eds.) *CONCUR 1991*. LNCS, vol. 527, Springer, Heidelberg (1991)
20. Fislser, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. *FMSD* 21, 39–78 (2002)
21. Garavel, H.: Open/cæsar: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) *TACAS 1998*. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
22. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: Cadp 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
23. Garavel, H., Mounier, L.: Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *SCP* 29, 171–197 (1997)
24. Groote, J.F., Keinänen, M.: Solving disjunctive/conjunctive boolean equation systems with alternating fixed points. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 436–450. Springer, Heidelberg (2004)

25. Ingólfssdóttir, A., Steffen, B.: Characteristic formulae for processes with divergence. *Information and Computation* 110, 149–163 (1994)
26. ISO/IEC: Lotos — a formal description technique based on the temporal ordering of observational behaviour. ISO Standard 8807, Genève (1989)
27. Kleene, S.C.: *Introduction to Metamathematics*. North-Holland, Amsterdam (1952)
28. Larsen, K.: Efficient local correctness checking. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 30–43. Springer, Heidelberg (1993)
29. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 53–66. Springer, Heidelberg (1998)
30. Martin, A.J.: Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing* 1, 226–234 (1986)
31. Mateescu, R.: On-the-fly state space reductions for weak equivalences. In: Proc. of FMICS 2005, pp. 80–89. ACM Computer Society Press, New York (2005)
32. Mateescu, R.: Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *STTT* 8, 37–56 (2006)
33. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *SCP* 46, 255–281 (2003)
34. Mader, A.: Verification of Modal Properties Using Boolean Equation Systems. In: VERSAL 8, Bertz Verlag, Berlin (1997)
35. Mateescu, R.: Efficient diagnostic generation for boolean equation systems. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 251–265. Springer, Heidelberg (2000)
36. Mateescu, R.: Local model-checking of modal mu-calculus on acyclic labeled transition systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 281–295. Springer, Heidelberg (2002)
37. Milner, R.: *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs (1989)
38. Pace, G., Lang, F., Mateescu, R.: Calculating τ -confluence compositionally. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 446–459. Springer, Heidelberg (2003)
39. Park, D.: Concurrency and automata on infinite sequences. In *Theoretical Computer Science*. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
40. Shukla, S.K., Hunt III, H.B., Rosenkrantz, D.J.: Hornsat, model checking, verification and games. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 99–110. Springer, Heidelberg (1996)
41. Tarjan, R.E.: Depth first search and linear graph algorithms. *SIAM Journal of Computing* 1, 146–160 (1972)
42. Valmari, A., Tienari, M.: Compositional failure-based semantics models for basic lotos. *Formal Aspects of Computing* 7, 440–468 (1995)
43. van Glabbeek, R.: The linear time — branching time spectrum I. In: *Handbook of Process Algebra*, pp. 3–100. Elsevier, Amsterdam (2001)
44. van Glabbeek, R.J., Weijland, W.P.: Branching-time and abstraction in bisimulation semantics (extended abstract). In: Proc. of 11th IFIP World Computer Congress (1989)
45. Vergauwen, B., Lewi, J.: Efficient local correctness checking for single and alternating boolean equation systems. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 304–315. Springer, Heidelberg (1994)
46. VASY. The VLTS benchmark suite,
<http://www.inrialpes.fr/vasy/cadp/resources/benchmark.html>

Resource-Aware Verification Using Randomized Exploration of Large State Spaces

Nazha Abed¹, Stavros Tripakis², and Jean-Marc Vincent^{1,*}

¹ LIG, 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France

² Cadence Laboratories, 2150 Shattuck Avenue, Berkeley, CA 94704

Abstract. Exhaustive verification often suffers from the state-explosion problem, where the reachable state space is too large to fit in main memory. For this reason, and because of disk swapping, once the main memory is full very little progress is made, and the process is not scalable. To alleviate this, partial verification methods have been proposed, some based on randomized exploration, mostly in the form of random walks. In this paper, we enhance partial, randomized state-space exploration methods with the concept of *resource-awareness*: the exploration algorithm is made aware of the limits on resources, in particular memory and time. We present a memory-aware algorithm that by design never stores more states than those that fit in main memory. We also propose criteria to compare this algorithm with similar other algorithms. We study properties of such algorithms both theoretically on simple classes of state spaces and experimentally on some preliminary case studies.

1 Introduction

To verify system correctness, one can proceed by exhaustive verification (e.g. model checking) or testing. Model checking [1,2,3] has gained wide acceptance within the hardware and protocol verification communities, and is witnessing increasing application in the domain of software verification. When the state space of the system under investigation is finite, model checking may proceed in a fully automatic, push-button fashion. Moreover, should the system fail to satisfy the formula, a counter example trace to the error state is produced. Model checking however is not without its drawbacks, the most prominent of which is state explosion: the phenomenon where the size of a system's state space grows exponentially in the size of its specification. State explosion can render the model-checking problem intractable for many applications of practical interest.

Testing, on the other hand, is typically performed directly on the implemented system. This has the advantage of checking the “real” system instead of a model of it. The disadvantage is that anomalies are detected often too late, resulting in high costs to correct them. Testing is inherently incomplete, as there is no guarantee of covering the state space even after several experiments.

* This work is partially supported by the ANR SETIN Check-Bound and the Region Rhône-Alpes, France.

Researchers have developed a plethora of techniques aimed at curtailing state explosion, by reducing the amount of memory necessary for states storage or reducing the state space to explore. Examples of the approaches made to reach the first goal are hash compaction [4] and bit-state hashing [5] which consists of encoding the graph states by the memory bits via a hash function. The methods that aim to reduce the state space include partial-order reduction methods [6]; which are based on the observation that executing two independent events in either order results in the same global state and symmetry reduction [7]; which uses the existence of nontrivial permutation group that preserves the state transition graph. There is also *symbolic* model checking techniques that operate on sets of states rather than individual states, and represent such sets symbolically, for instance, using binary decision diagrams (BDDs) [8]. In this paper we focus on explicit enumerative state space exploration methods.

Other techniques aim at a partial, i.e., incomplete, exploration of the state space, in particular, using randomized algorithms, which make decisions based on outcomes of random experiments such as tossing a fair coin or generating a random number. Randomized algorithms are extensively used, basically for two reasons: simplicity and speed [9]. A consequence of using randomization is that correctness or termination can often be asserted only with some controlled probability.

The randomized algorithms proposed in the literature are mostly based on random walks. A random walk on a graph starts from the initial state, and at each step, chooses with uniform probability a successor of the current state and visits it. This choice is independent from the traversal history, which is characteristic of a Markov chain. When the random walk encounters a deadlock point, it restarts from the initial state. The algorithm terminates when a target state is reached or when the expected number of the visited states reaches a certain limit. This method stores only an actual state and does not keep any information about previously visited states, thus it has very little memory requirements.

This simple form of random walk was applied first to model-checking by West [10] and more recently in [11,12,13]. Because it is completely memory free, the random walk method cannot distinguish between visited and not visited states, and so it may spend a large amount of time repeatedly visiting the same states: we call this *redundancy*. Because of this, covering the entire graph (or a high portion of it) may need a prohibitively large amount of time. Also, the frequency (probability) of visits may be very variable from one state to another (some states are more frequently visited than others). This frequency depends on the graph structure as well as the algorithm behavior.

Several methods have been proposed to avoid these drawbacks. Some of these methods try to force exploration direction, like the re-initialization methods that restart the random walk process periodically to avoid blocking in a small closed components for a long time. The re-initialization can be made from a random state of the previous walk and not necessary from the initial state. This has the advantage to minimize redundancy and reach deep states [14]. The local

exhaustive search combined to random walk [15] explores better some regions of interest (dense regions for example) which cannot be well explored with only simple random walk. This may be the case for example if one knows that it is near a target node. Guided search decides of the next exploration direction based on general information about the graph and system semantics. In [16], the authors use a metric to estimate reachability probability of a target node. To gain in memory and time, the parallelization method of random walk seems to be very useful and efficient. It explores more states [15] and reduces significantly the error probability [12].

Other methods use some additional memory to keep a subset of the visited states. These states are used to report the counter example trace as done in tracing methods or to limit revisits of same nodes and improve the coverage as done in caching methods [17] [18]. Caching is an exploration algorithm that focuses on the strategy of nodes storing and deletion from the cache. The exploration scheme can be deterministic, as in a breadth-first or depth-first search (BFS, DFS), or random. In [19], the proposed algorithm uses BFS with a randomized partial storage. When the memory is full, the algorithm proceeds at a lower speed but does not give up. As reported in [19], this algorithm can save 30% of the memory with an average time penalty of 100%. Other methods that use randomization in a verification context include [20,21,22,23,24,25].

All the methods mentioned above that are based on random walk improve the redundancy of exploration but the cover time can still be very large. In this paper, we propose methods that aim to further improve exploration by avoiding redundancy and reducing the cover time. First, we propose a generic scheme that aims to encompass special instances of algorithms. Then, we propose the Uniform Random Search (URS) algorithm, which is based on a different selection function than random walk (RW). While RW is a depth-oriented algorithm, URS can go in depth, in breadth or in a uniform fashion. We can also control the rate of depth or breadth exploration by tuning a mixing parameter.

A major novelty of our exploration scheme lies in the fact that it explicitly uses a parameter N that represents the maximum number of states that can be stored in main memory at any given time. Thus, our algorithms are *resource-aware*, and in particular, *memory-aware*. Main memory is the main bottleneck in exhaustive verification, for reasons we explain below. Our scheme tries to overcome this by explicitly taking into account the resource constraints and using them to make decisions during the exploration.

The randomized algorithms proposed are sound, which means that if a bug is found then the model is indeed incorrect. As in [12,13], they are probabilistically complete, in the sense that if after several iterations no bugs are found, then the system is correct with some probability which depends on the number of iterations and visited states.

The rest of the paper is organized as follows: The proposed scheme and algorithm are detailed in section 2. Section 3 gives some general theoretical results that are projected on two cases of regular graphs. Experimental results are summarized in section 4, while section 5 contains our conclusion.

2 Context and Algorithms

We model a system as a directed transition graph $G(M, v_0, Succ)$, where, M is a finite set of nodes representing the system states, v_0 is the initial node ($v_0 \in M$) and $Succ$ is the transition function: it takes as input a node v and returns as output the set of all successors of v . We do not dispose of the entire transition graph. We can, however, construct and explore it gradually by means of the initial state and the transition function $Succ$. We assume that the available main memory can store at most N states. N can be computed by dividing the size of the memory, by the size of the memory representation of each state. To generate randomized algorithms, a pseudo-random numbers generator is given. The generated numbers can be considered as uniformly distributed in $[0, 1]$, based on which, other distribution laws can be generated if necessary.

To verify a given safety property stated as an invariant ϕ , the simplest method is to explore the graph G and verify ϕ for each state $s \in G$. If we use an exhaustive deterministic exploration, the computer's memory will be rapidly filled by the N first reachable states (N depends on the available memory as said above). Then, the computer will typically spend most of its time *swapping* memory to/from disk with very few additional states explored. This is clearly non-scalable: running the model-checker for several days may result in only a few additional visited states than running it only a few hours. Instead, we choose a memory-aware, randomized, partial exploration, and repeat it several times with different paths (consequence of randomization) to cover as many reachable states as possible.

One wishes, naturally, that the randomized algorithm explores the state space efficiently, i.e., quickly and using reasonable memory resources. Since the memory size is given and finite, a good exploration is defined mainly according to the time it takes: one can hope to cover with a randomized algorithm a considerable percentage of the reachable graph in less time than with the exhaustive algorithm which will be quickly blocked because of the swapping.

2.1 A Generic Randomized Exploration Scheme

A random exploration algorithm can be cast into the generic scheme shown in Figure 1. P represents the algorithm parameters, for example the memory size N , the number of initial parallel runs in the case of a parallel random walk [15], ect. This last parameter, among others, can be modified during the algorithm execution according to the available resources and exploration needs. The set I contains global information on the graph structure, for instance, mean number of successors per node, mean number of loops, strongly connected components, etc. Note that this type of information can be collected on the fly and used to guide and optimize the exploration [16].

A specific algorithm that fits the above scheme is defined by specifying the *stop condition* and the two functions *select* and *update*. With these three parameters, one can define many variants of the general algorithm, including many found in the literature. The *stop condition* can be, for example, the presence of

```

V : set of stored nodes;
P : algorithm parameters;
I : global information;
v : node;

V ←  $V_0$ ; //Set of initial nodes
P ←  $Par$ ; //Algorithm parameters
I ←  $I_0$ ; //Initial global information

While (not stop condition) do
     $v$  ← select( $V, P, I$ );
    check( $v$ ); //verify if the property holds
    ( $V, I$ ) ← update( $V, v, P, I$ );
done

```

Fig. 1. The general randomized exploration scheme

a deadlock, exhaustion of the expected number of steps or simply reaching a target state. Some algorithms in the literature emphasize state storage and deletion strategies (FIFO, LFU, LRU, random ...), like the caching techniques [18] [17], so they focus in optimizing the *update* function. The *update* function modifies the sets V and I in order to optimize the consumed resources and make the evolution of the exploration effective. As mentioned in the introduction, our interest is mainly the exploration strategy itself, that is the *select* function. The *select* function chooses at each step the next node v , to be visited from the set of successors of V ; the already visited states still in memory. This choice can be guided by the information in I .

In this scheme, the random walk algorithm has as *stop condition* the reachability of a deadlock point or the reach of a target node according to the algorithm goal. The *select* function is a uniform random choice between the successors of the current node (the single stored in V), when the *update* function consists on simply replacing the current node by the one lastly chosen. In presence of a deadlock, the current node takes the value of the initial state and so on.

As we are interested in the exploration strategy, we propose a Uniform Random Search (URS) algorithm based on a new *select* function: see Figure 2. We have a set V of already visited states. V is of size N : that is, the algorithm ensures that there are never more than N states in V . Initially this set contains the initial state v_0 . At each step i , the URS algorithm picks uniformly one visited state u from V , and then uniformly chooses one successor v of u . Note that this does not imply a uniform choice from all the visited node successors. If v is not already visited then it is checked with respect to the safety property and added to the set of visited states. The algorithm stops, and eventually restarts, when the memory is full ($j = N$) or when the expected number of steps, n , is reached.

[14] presents an extended random-walk based algorithm called Deep Random Search (DRS). The *stop condition* of DRS does not consider the limited memory

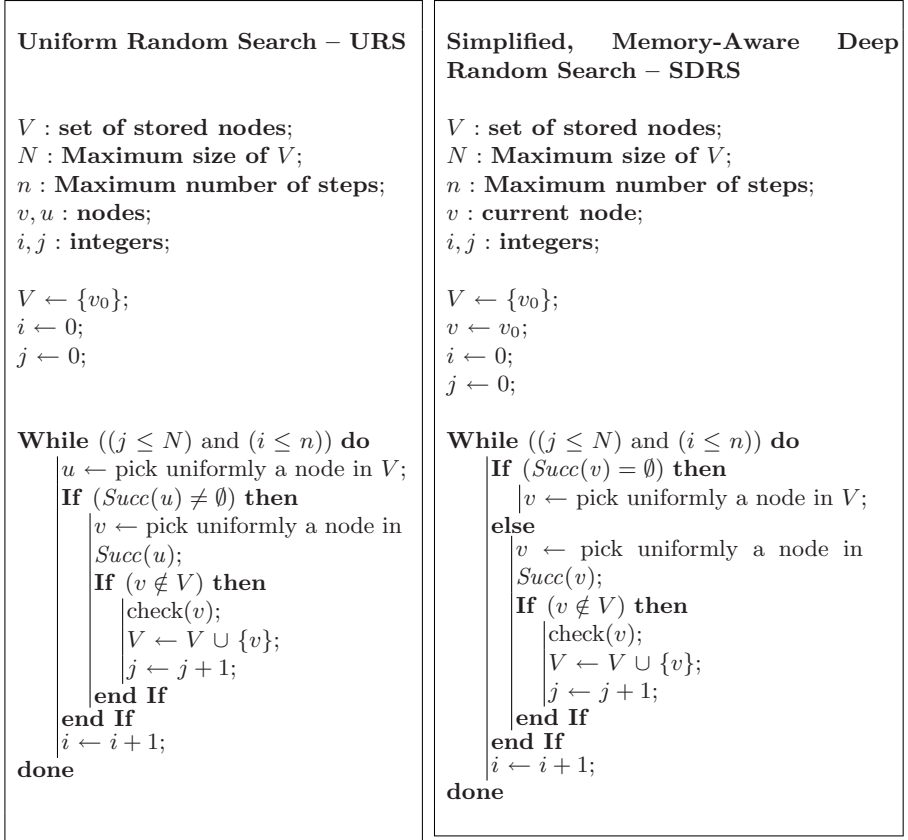


Fig. 2. The URS and SDRS algorithms

size and supposes that the set of “non-closed” nodes (i.e., that have at least one non-visited successor) fits in main memory. In this paper we use a simplified, but memory-aware, version of DRS, that we call SDRS. Like URS, SDRS can keep at most N states in memory at any given time. This puts the two algorithms in the same framework and allows comparisons. SDRS has the same stop condition as URS. The *select* and *update* functions of SDRS are the same as for simple random walk except that the current node is reset to a node chosen randomly in V and not to the initial node.

When the main memory is full, the algorithms are stopped, the memory is emptied and the algorithms are restarted. This can be repeated several times. The re-initialization can be done from the initial state or from another randomly chosen state from the set V of states visited during the last exploration. Note that the initialization from the initial state often does not result in a very high degree of redundancy because the number of states in each repetition is very large and can usually match the graph’s diameter. In the rest of the paper, we will

consider two situations in our analysis and experimental results. In one situation we suppose that the main memory is large enough to contain the entire state space of the graph under exploration. In this case, we will speak of the versions of the algorithms URS and SDRS where these do not have to be reinitialized. In the second, more realistic case for industrial-size examples, the main memory cannot store the entire state space, and the algorithms are run multiple times, after re-initialization as described above. In this case, we will denote the algorithms by RURS and RSDRS to emphasize the fact that they are re-initialized.

2.2 Evaluation Criteria

URS and SDRS are only two of the many possible memory-aware, randomized exploration algorithms one can think of. The question is, which algorithm is better, in which cases, and what exactly does “better” mean? To answer these questions, we need some criteria to evaluate performance of such algorithms. We define such criteria in two ways: stochastic and experimental.

One useful criterion is *mean cover time*. The cover time is the number of steps needed by a given algorithm which starts at the initial state to *cover* (i.e., visit) some percentage of the set of reachable states.¹ The mean cover time gives a good indication on the capacity of the algorithm to reach states and explore most of the graph. Cover time also reflects what can be termed *response time*, with an error ϵ . For example, if one needs a response about the system correctness with probability of error $\epsilon = 0.05$, the necessary time for giving this response can be defined as the cover time of 95% of the graph. Some exploration algorithms will provide this answer in less time than others.

When the set of reachable states is unknown, we compare the number of *covered* nodes (i.e., visited nodes). As the number of the visited nodes increases, the probability that a node already visited is revisited typically increases (redundancy). It results from this, that the number of newly visited nodes decreases according to the execution time T_e . From this fact, the coverage progression is, typically, a logarithmic curve according to T_e . This is confirmed by our theoretical and experimental results.

Another useful criterion is the *minimum reachability probability* over all reachable nodes. Reachability probability is the probability that a given node v of a graph G is visited by a given algorithm A , denoted $\mathbb{P}_{G,A}(v)$. Note that the model checking problem can be often seen as searching for a target (e.g., error) state. The reachability probability of a target state is thus meaningful. Due to the fact that the considered exploration algorithms are random, the list of visited nodes V is a random variable that depends on the algorithm and the particular graph structure. Thus, the membership of a given node v to V is a random variable of which the probability $\mathbb{P}_{G,A}(v)$ for a given graph G and a

¹ For the random walk, in the case of undirected graphs, the mean cover time of any graph is polynomial [26]. In the case of directed graphs it is in general exponential, except for some restricted classes [12]. These classes are so restricted that they are not very interesting for model checking.

given algorithm A differs from a node to another. The minimum reachability probability criterion is the minimum over all nodes of these probabilities:

$$\pi_{min}(G, A) = \min_v \mathbb{P}_{G,A}(v)$$

In general, re-iterating the randomized algorithm improves the probability of reaching states and finding errors.

Note that reachability probability depends on the resources that are available to an algorithm A , for instance, the available memory and time. In the case of URS and SDRS, for example, it depends on parameters N and n . Thus, another useful criterion is the *mean number of covered nodes*, for given resource parameters.

In practice, there are several types of graphs, and an algorithm performs differently depending on the form of the explored one. To compute precise analytic results, we have analyzed regular classes of graphs: trees and grids. Regular graphs are suitable to study analytically the behavior of exploration algorithms for several reasons:

- Although the model checking graphs are not regular, they contain frequently regular components [27].
- One can manipulate regular graphs to compute probabilistic measures analytically, which is practically impossible for graphs of irregular topology.
- By tuning the two parameters of a regular tree (depth and degree), we can get large or deep graphs and define a density factor suitable to our study.
- Trees and grids constitute two extreme cases of general graphs. In trees, there are no intersections between the successors, and in grids, there is intersection between all successors. Other graphs can be considered as an intermediate case between this two ones.

3 Theoretical Results

This section aims at a theoretical comparison of randomized exploration algorithms in terms of various statistics. More precisely, we investigate exact computations of the mean cover time, the mean number of covered nodes and other related criteria such as reachability probabilities, for URS and SDRS. We do this for two simple types of graphs: trees and grids. We first provide some general results that apply to any graph.

For URS, the ordered sequence $V_n = (v_1, \dots, v_n)$ of visited nodes in n steps can be represented as a sequence $w_1, \underbrace{\dots}_{\alpha_1}, w_2, \underbrace{\dots}_{\alpha_2}, w_3, \underbrace{\dots}_{\alpha_3}, \dots, w_{k-1}, \underbrace{\dots}_{\alpha_{k-1}}, w_k, \underbrace{\dots}_{\alpha_k}$

$$W_n = (w_1, \dots, w_k)$$

where each w_i corresponds to a novel visited node followed by α_i redundant visits, that is the considered sequence V_n is constituted by $n - k$ repeated nodes interlaced in an ordered set of k distinct nodes $\underline{w}_k = (w_1, \dots, w_k)$. Let $\underline{w}_{k-1} = (w_1, \dots, w_{k-1})$ and denote by $F(w_i)$ (resp. $C(w_i)$) the set of fathers (resp. children) of the node w_i , $i = 1, \dots, k$.

Lemma 1. *The probability $\mathbb{P}(\underline{w}_k, n)$ to cover node \underline{w}_k in n steps by URS is:*

$$\mathbb{P}(\underline{w}_k, n) = \alpha(\underline{w}_k)\mathbb{P}(\underline{w}_k, n - 1) + \beta(\underline{w}_k)\mathbb{P}(\underline{w}_{k-1}, n - 1)$$

$$\alpha(\underline{w}_k) = \frac{1}{k} \sum_{i=1}^k \frac{|C(w_i) \cap \underline{w}_k|}{|C(w_i)|}, \quad \beta(\underline{w}_k) = \frac{1}{k - 1} \sum_{v \in F(w_k) \cap \underline{w}_{k-1}} \frac{1}{|C(v)|}$$

Note that $\alpha(\underline{w}_k)$ is a redundancy factor, equal to the probability to revisit a node at step n (no node is newly covered), while $\beta(\underline{w}_k)$ is an innovation factor expressing the probability to cover at step n a new node, which must be w_k , since the set \underline{w}_k is stored in order of visits.

The elementary recursion for SDRS is a bit more complicated than for URS and one must distinguish closed and open points of exploration. The exploration is said to be in a closed point at step n , if it has reached a deadlock at step $n - 1$, it attempted, unsuccessfully, in step n to choose a successor from this deadlock and so it will be reinitialized in step $n + 1$ from a uniformly randomly chosen state of V_n . An open point is a point of the walk which is not a closed point.

Lemma 2. *Let $\mathbb{P}(\underline{w}_k, n, C)$ (resp. $\mathbb{P}(\underline{w}_k, n, O, v)$) be the probability to cover in n steps the set of nodes \underline{w}_k and to be, by step n , in a closed point (resp. in an open point at node v). Then:*

$$\mathbb{P}(\underline{w}_k, n, C) = \frac{|D(\underline{w}_k)|}{k} \mathbb{P}(\underline{w}_k, n - 1, C) + \sum_{v \in D(\underline{w}_k)} \mathbb{P}(\underline{w}_k, n - 1, O, v)$$

$$\mathbb{P}(\underline{w}_k, n, O, v) = \sum_{u \in F(v) \cap \underline{w}_k} \left[\frac{\mathbb{P}(\underline{w}_k, n - 1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_k, n - 1, C)}{k|C(u)|} \right. \\ \left. + 1_{w_k}(v) \left(\frac{\mathbb{P}(\underline{w}_{k-1}, n - 1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_{k-1}, n - 1, C)}{(k - 1)|C(u)|} \right) \right]$$

where $D(\underline{w}_k)$ is the set of deadlock nodes in \underline{w}_k and $1_{w_k}(v) = 1$ if $v = w_k$ and $1_{w_k}(v) = 0$ otherwise.

The algorithms URS and SDRS will be analyzed, and then compared, with respect to two criteria. The first is the redundancy of each algorithm due to its exploration scheme. To compute it, it is not necessary to consider the algorithms with re-initialization, we compare only the redundancy of the algorithms URS and SDRS applied without repetition. This redundancy analysis will be done in function of the time n , or the number of successive steps, needed to cover a given number k of nodes in the considered graph. The direct relation between redundancy and covering time is the following:

$$redundancy = \frac{n - k}{n}$$

In fact, an exploration algorithm, at each step of its run, can only visit a novel node or repeat an already visited one. In the first case, either the time n and the

number of covered nodes k are incremented by one, while in the second case the time is incremented but not the number of covered nodes, which increases the redundancy. The mean cover time will be exactly and efficiently computed meaning the recursions provided in the further section.

The second criterion of analysis is the mean number of covered nodes. This will be considered for the repeated versions of the algorithms, i.e. RURS and RSDRS. This corresponds to the more actual case, when the graph to be explored is too large with respect to the memory size. In this case our algorithm URS reinitialize itself each time the memory is full. Note that in [14], the re-initialization of the algorithm DRS is not considered and the case of memory shortage is not studied. Here we place the two algorithms in the same context where re-initialization is applied each time the number of covered nodes reaches a prefixed threshold, which is, in our case, the memory size.

In the context of large graphs, it is not easy to reach a coverage level up to 100%. Also, the graph sizes can be unknown, so, we consider the number of covered nodes rather than the coverage level. The algorithms RURS and RSDRS will be compared in terms of the mean number of covered nodes for a given time of exploration, which constitutes an equivalent criterion to the mean time for a given coverage that we applied for URS and SDRS. The mean number of covered nodes, function of time, will be exactly computed for RURS and RSDRS thanks to theorem 1.

Note that in our theoretical study we will consider hereafter graphs with medium to small sizes but which are more than 5 times greater than the considered memory size. The results obtained on these prototypes can then be scaled to greater graphs taking the same proportions of memory to graph size. The use of large size graphs is very heavy because the theoretical formula are recursive in the steps number and take much memory size to be computed.

3.1 Case of Trees

We consider an m -ary tree of depth h , that is, every non-leaf node has m successors, and every path from the root to a leaf has length h . Recall that n denotes the number of successive steps in a run of the algorithm.

The elementary recursion in lemma 1 (resp. in lemma 2) leads to a much more simplified one, depending only on the numbers of nodes of \underline{w}_k in each level of the tree and not on \underline{w}_k itself. Consider $\underline{K}_n = (K_n^1, \dots, K_n^h)$, the vector of random variables expressing the number of explored nodes at each level $j = 1, \dots, h$, at step n , and let $\mathbb{P}_{urs}(\underline{K}_n = \underline{k})$ the probability to cover the vector $\underline{k} = (k_1, \dots, k_h)$ in n steps by URS algorithm. For SDRS, we distinguish $\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, C)$ and $\mathbb{P}_{sdrs}(\underline{K}_n = \underline{k}, O)$ that denote the probabilities of covering \underline{k} in the closed and open cases respectively. For URS, for example, the aggregation (summation) of the elementary recursion in lemma 1 on the set of all sequences \underline{w}_k having k_j nodes in the level j , $j = 1, \dots, h$, gives the following simplified recursion :

$$\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) = \alpha(\underline{k}) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^h \beta_j(\underline{k}) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j)$$

where $\underline{k} - 1_j = (k_1, \dots, k_j - 1, \dots, k_h)$, $1 \leq j \leq h$. In the r.h.s. of this equation, as in the elementary one, two terms appear. The first one $\mathbb{P}_{urs}^{\mathcal{R}}(\underline{K}_n = \underline{k}) = \alpha(\underline{k}) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k})$ is a redundancy term, while the second $\mathbb{P}_{urs}^{\mathcal{J}}(\underline{K}_n = \underline{k}) = \sum_{j=1}^h \beta_j(\underline{k}) \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j)$ is the innovation term. The repetition factor $\alpha(\underline{k})$ is given by $\alpha(\underline{k}) = \frac{mk_h + k - 1}{mk}$. The innovation ones are $\beta_j(\underline{k}) = \frac{mk_{j-1} - k_j + 1}{m(k-1)}$.

The mean time $T_A(k)$ to cover k nodes by an algorithm A (URS or SDRS) can be expressed in function of the innovation probabilities as following:

$$T_A(k) = \sum_{|\underline{k}|=k} T_A(\underline{k}), \quad T_A(\underline{k}) = \sum_{n=k}^{\infty} n \mathbb{P}_A^{\mathcal{J}}(\underline{K}_n = \underline{k})$$

With some further investigations, the mean times $T_{urs}(\underline{k})$ and $T_{sdrs}(\underline{k})$ of covering \underline{k} by URS and SDRS, respectively, are given by recursive formula.

Applying the previous result, we obtain the mean cover time computed exactly for URS and SDRS and shown in Figure 3 (left) for three parameterized trees. The notation $T(h, m)$ means that the considered tree is of height h and degree m . Note that the mean cover time is plotted in function of the coverage level (percentage of reachable nodes that are covered) rather than in terms of number of covered nodes. Given the fact that our primary interest here is redundancy, the case of a set of covered nodes going beyond the memory size is not considered. It was, then, possible to make the comparison up to the full coverage where we obtained the more significant difference in term of mean cover time between the two algorithms.

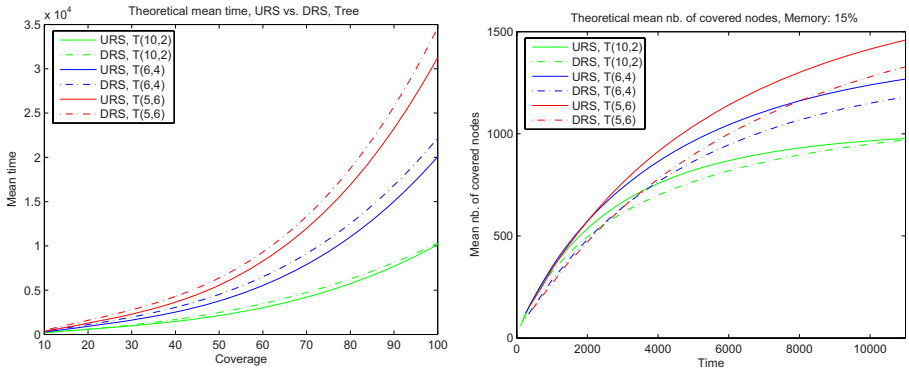


Fig. 3. Mean cover time (left) and mean number of covered nodes (right) for Trees

We can see in Figure 3 that the URS algorithm takes on average less time than SDRS to cover a given proportion of the graph. This is observed mainly for proportions more than 70% and for large trees. We define the *density factor* DF of an m -ary tree of depth h by the ratio $\frac{m}{h}$. In fact, the higher the density factor is, the larger the difference between the cover times of the algorithms is. In the case of a “thin” tree, which has small DF (typically < 0.05), SDRS can perform better than URS but this can be obtained only for extremely thin graphs.

In the following of this section we return to the more actual case, when the graph to explore is too large with respect to the memory size. We start by noting the relation in lemma 3, that holds for all algorithms A on all graphs G , between the probability $\mathbb{P}_A(K_n = k)$ to cover k nodes in n steps and the reachability probabilities $\mathbb{P}_A(v|K_n = k)$ to have, in n steps, reached a node v and covered exactly k nodes. Note that, in the case of trees, these last probabilities depend only on the node level i and not on the node v itself, because of symmetry. In the case of a grid, we must compute the probability to reach corner and non corner nodes at each level i .

Lemma 3

$$\mathbb{P}_A(K_n = k) = \frac{1}{k} \sum_{v \in G} \mathbb{P}_A(v|K_n = k)$$

As we said above, the criterion considered here is the mean number of covered nodes function of time. Thanks to lemma 3, this can be computed basing on reachability probabilities that we first compute by returning to the elementary recursions of the algorithms. In fact, as previously, by summing these recursions on the set of the sequences \underline{w}_k , containing the node i and having in each level $j = 1, \dots, h$, k_j nodes, one obtains recursive formula for the reachability probabilities $\mathbb{P}_{urs}(i|\underline{K}_n = \underline{k})$, $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, C)$, $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, O)$, and then $\mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}) = \mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, C) + \mathbb{P}_{sdrs}(i|\underline{K}_n = \underline{k}, O)$. These probabilities are defined exactly as previously except the fact that the node i is now considered to be covered. Note that these probabilities are associated with URS and SDRS without repetition and then computed for a number of covered nodes k less than the re-initialization threshold (the memory size) N . For example, for URS, one obtains, with $\gamma(\underline{k}) = \frac{1}{m(k-1)}$, :

$$\begin{aligned} \mathbb{P}_{urs}(i|\underline{K}_n = \underline{k}) &= \alpha(\underline{k}) \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^h \beta_j(\underline{k}) \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k} - 1_j) \\ &+ \gamma(\underline{k}) \left[\mathbb{P}_{urs}(i-1|\underline{K}_{n-1} = \underline{k} - 1_i) - \mathbb{P}_{urs}(i|\underline{K}_{n-1} = \underline{k} - 1_i) \right] \end{aligned}$$

Once these probabilities are calculated, one sets

$$\mathbb{P}_A(i, s) = \sum_{|\underline{k}| \leq N} \mathbb{P}_A(i|\underline{K}_s = \underline{k}), \quad \mathbb{P}_A^*(i, s) = \sum_{|\underline{k}|=N} \mathbb{P}_A(i|\underline{K}_s = \underline{k})$$

where N denotes the memory size and A denotes indifferently one of the algorithms URS or SDRS. Their repeated versions will be noted RA . Then, the mean number of covered nodes of RA in function of time n is given in the theorem 4:

Theorem 1. *If N is the memory size or a prefixed threshold of re-initialization, then the mean number of covered nodes by RA is given in function of time n as:*

$$\begin{aligned} Cov(n)_{n=} &= \sum_{n_1=M}^h m^i \mathbb{P}_{RA}(i, n), \quad \text{where} \\ \mathbb{P}_{RA}(i, n) &= \mathbb{P}_A(i, n) + \sum_{n_1=M}^h [\mathbb{P}_A^*(i, n_1) + (1 - \mathbb{P}_A^*(i, n_1))\mathbb{P}_{RA}(i, n - n_1)] \end{aligned}$$

Figure 3 (right) shows the evolution of the number of covered nodes in function of time. These curves, representing the behavior of the repeated algorithms RURS and RSDRS, are plotted for three trees. The repeated algorithms are experimented for a memory size (N) of 15% w.r.t. the size of the graph. We have considered other memory sizes (10% and 20%), but the results are similar: RURS algorithm performs, clearly, better than RSDRS, especially near to the total coverage rate. We observe also that the difference between RURS and RSDRS in the number of covered nodes is more important as more as the DF is greater.

Note that by using the reachability probabilities $\mathbb{P}_A(i, n)$ (resp. $\mathbb{P}_{RA}(i, n)$), one can compute the minimum reachability probabilities for URS and SDRS (resp. for RURS and RSDRS) in function of time. This criterion can be very interesting in practice if, in order to detect efficiently an eventual bug in the system, which corresponds to a defective node in the modeling graph, one can take account of the worst case where the bug is localized in a node of minimum reachability probability. Note that the number of such nodes can be great as in the case of tree like graphs.

3.2 Case of Grids

We place ourselves here in the context of multi-dimensional grid. As in the previous section, we are interested in efficient computations of statistics like the mean cover time and the mean number of covered nodes for URS and SDRS. We will analyse this matter basing on the fundamental recursion in lemma 1 and 2. We first note that all possible (macroscopic and then less difficult to compute) recursion for URS or SDRS should be a summation of the corresponding elementary one on some suitably chosen set S_k of sequences \underline{w}_k : the coefficients in the elementary recursion must be constant on S_k and the set of the \underline{w}_{k-1} 's, when $\underline{w}_k \in S_k$, must be easy to identify. For clarity sake, we analyse in details the equation in lemma 1 for our algorithm *URS*. The coefficients $\alpha(\underline{w}_k)$ and $\beta(\underline{w}_k)$ in this recursion must be constant on S_k and the set of the \underline{w}_{k-1} 's, when $\underline{w}_k \in S_k$, must be easily parameterizable. This seems to be very difficult to obtain, or impossible, even in the case of infinite, oriented, grid, but this problem will be overcome as explained below. In this case the output degree of the nodes is the same, say d , and one has:

$$\alpha(\underline{w}_k) = \frac{\sum_{i=1}^k |C(w_i) \cap \underline{w}_k|}{k \cdot d}, \quad \beta(\underline{w}_k) = \frac{|F(w_k) \cap \underline{w}_{k-1}|}{(k-1) \cdot d}$$

The difficulties to sum the elementary recursion satisfied by URS and SDRS, are due essentially to the great rate of communications (intersections) in the case of the grid. However, this is the same reason for which these recursions are useful in practice to calculate exact exploration statistics in this case, especially by meaning some managements. In fact due to intersections, the number of ordered sequences, with distinct nodes, generated by the algorithms is reasonable in many cases of study. Note also that the sizes of grids to be considered are in general little, as are grids in model-checking domain.

Figure 4 gives the results of comparisons of the mean covering time for three grids, where $G(L, d)$ means that the grid is of degree d and the length of each side is $L + 1$. It is clear that the URS algorithm outperform SDRS. Its superiority is even more clear than in the case of graphs without intersections (tree).

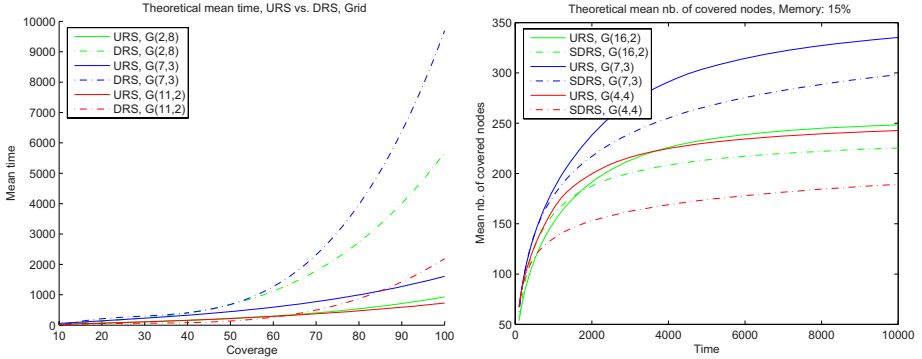


Fig. 4. Mean cover time (left) and mean number of covered nodes (right) for Grids

Moreover, for the repeated algorithms RURS and RSDRS, the mean number of covered nodes has been plotted in function of time for different grids. The reported result in Figure 4 corresponds to a memory size of 15% w.r.t. the size of the graph. As for trees, the algorithms RURS and RSDRS are experimented for three grid graphs and for three memory sizes (N) of 10%, 15% and 20% w.r.t. the size of the graphs. The results are similar for the three memory sizes: the performances RURS are clearly better than RSDRS. The superiority of RURS is more marked for high coverage and great values of the DF . This superiority is, again, more clear for grids than for trees.

4 Experimental Results

We complement our theoretical analysis with a set of experimental results. We implemented the two algorithms URS and SDRS on the model checker IF [28] and ran them on several examples. Several measures were computed for each algorithm. The examples have been chosen according to the experimental needs. First, to compute the mean cover time, we have chosen some examples of medium size, in order to be able to repeat the algorithms a sufficient number of times to achieve full coverage of the reachable state space. These examples have different *density factors*, which allows us to analyse their behavior according to this parameter. Second, in order to compare the randomized algorithms with the exhaustive BFS algorithm implemented in IF, we have used the same examples, with more processes and/or data, to get graphs of very large (unknown) sizes.

Our implementations of URS and SDRS use a hash table to keep visited nodes V . In this work, we have described the URS and SDRS algorithms, but

our implementation is more general, following the generic scheme, in particular in terms of the select function. Other variants of this scheme apart from URS and SDRS will be reported in future work. Our implementation allows the user to define the rate of leaves or internal nodes to be explored –which reflects depth- or breadth-oriented exploration– by tuning a *mixing parameter*. Choosing this parameter appropriately may require an a-priori knowledge of the graph structure (density and diameter), although, in some cases, this parameter may be computed and adapted *on the fly*.

4.1 Cover Time

Each algorithm was tested on different graph examples: the *Quicksort* algorithm, the *Token Ring Protocol*, *Fischer’s Mutual Exclusion Protocol* and a *Client/Server Protocol*. The computer architecture was a Intel Xeon quadri-processors, 1GHz, 4Mo cache and 8Go memory. Table 1 shows the size (i.e., number of states) and the diameter (i.e., length of the longest acyclic path) of each example. The table also shows the density factor of the graph of each example, defined as $DF = \frac{m}{h}$, where h is the graph diameter and m is the graph degree: m is computed approximately by reference to a regular tree of size $M \approx m^h$. Thus, for a graph of size M , we let $m = \sqrt[h]{M}$.

Table 1. Graphs description

Example	Quicksort	Token	Fischer	Server
Size (no. states)	6032	20953	34606	35182
Diameter	19	72	14	28
DF (density factor)	0.083	0.016	0.150	0.052

For each example, we repeated the experiment 100 times and we computed the mean cover time of 60%, 70%, 80%, 90% and 100% of the graph. The resulting times (in seconds) are reported in Table 2. We observe that the URS algorithm performs better except for the Token example. But even in this example URS performs better for 100% coverage.

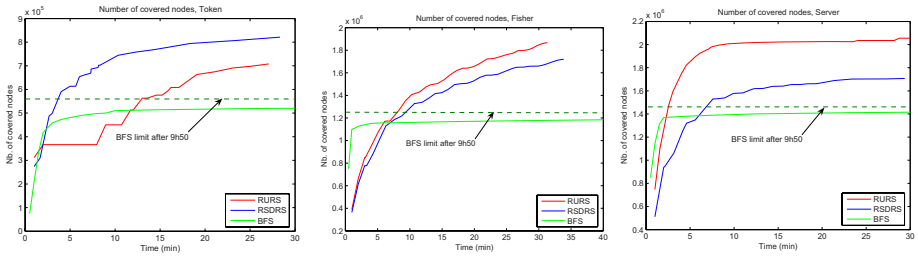
4.2 Resource-Aware vs. Exhaustive Verification

We have also experimented on very large graphs of unknown reachable size. These have been obtained by scaling-up the number of processes and/or data of the Token, Fischer and Server examples. Here we also compared URS and SDRS with an exhaustive BFS algorithm. Note that URS and SDRS re-initialize in these examples, since the state space does not fit in main memory: thus we denote them by RURS and RSDRS in the plots that follow. The number of explored states was collected over all runs and is plotted in Figure 5 as a function of time.

BFS stagnates as it approaches the limit of the the number of states that can fit in main memory. URS and SDRS go beyond this limit, and can explore

Table 2. Mean cover time (seconds)

Cov. level	Algo	Quicksort	Token	Fischer	Server
60%	URS	0.389	3.283	1.841	4.490
60%	SDRS	0.641	0.752	4.070	7.441
70%	URS	0.609	4.301	2.765	5.507
70%	SDRS	0.871	1.084	5.726	8.893
80%	URS	0.882	5.744	3.809	6.821
80%	SDRS	1.411	1.584	8.173	11.966
90%	URS	1.703	8.047	5.955	9.974
90%	SDRS	4.202	2.480	13.327	19.158
100%	URS	7.723	21.247	46.097	41.452
100%	SDRS	12.459	25.221	125.091	99.460

**Fig. 5.** The number of covered nodes evolution

up to 40% more nodes. Notice that the BFS limit occurs at a different number of nodes for each of the three case studies, even though they all use the same amount of main memory. This is because in each case study the amount of bytes needed to store a single state is different: it is higher in Token than in Server, and slightly higher in Server than in Fischer.

We observe that in the case of Fischer the randomized algorithms also stagnate after a certain amount of time. According to what we observed in our previous experiments on medium-size graphs, this happens when reaching close to 90% of the graph. In this case, exploring the “last” states becomes increasingly difficult because of redundancy.

5 Conclusions and Future Work

We have proposed resource-aware randomized state space exploration as a direction for research in scalable verification methods. In particular, we have proposed the URS algorithm that we believe to be the first memory-aware exploration scheme, that explicitly uses main memory resource limits to guide its behavior. Also, URS is not performing a typical random walk, in the sense that it may

choose to “branch” from different nodes along a random walk path. We have proposed comparison criteria such as mean cover time and used these to compare URS with a simplified version of the DRS algorithm proposed in [14]. We have also shown via experiments, that these two algorithms, when repeated several times, can explore a state space of more than 40% in addition to that explored by an exhaustive exploration based on breadth-first search.

As part of future work we would like to experiment with industrial case studies, for instance, from the hardware or software domains. We would also like to implement and test other resource-aware verification algorithms. For instance, instead of re-initializing when the memory is full, we could have a scheme where some states in the visited set V are removed and replaced by new states. Different policies to choose which states to remove could then be envisaged.

References

1. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Dezanı-Ciancaglini, M., Montanari, U. (eds.) *Programming 1982*. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
2. 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)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (1999)
4. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Evekıng, H. (eds.) *CHARME 1995*. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
5. Holzmann, G.J.: An analysis of bistate hashing. In: *PSTV. IFIP Conference Proceedings*, vol. 38, pp. 301–314. Chapman & Hall, Boca Raton (1995)
6. Nalumasu, R., Gopalakrishnan, G.: An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design* 20, 231–247 (2002)
7. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9, 77–104 (1996)
8. 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*, pp. 428–439. IEEE Computer Society, Los Alamitos (1990)
9. Rabin, M.O.: *Probabilistic Algorithms*, pp. 21–39. Academic Press, Inc., Orlando (1976)
10. West, C.H.: Protocol validation by random state exploration. In: *Protocol Specification, Testing and Verification*, pp. 233–242. North-Holland, Amsterdam (1986)
11. Owen, D., Menzies, T.: Lurch: a lightweight alternative to model checking. In: *SEKE*, pp. 158–165 (2003)
12. Haslum, P.: Model checking by random walk. In: *ECSEL Workshop* (1999)
13. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
14. Grosu, R., Huang, X., Smolka, S.A., Tan, W., Tripakis, S.: Deep random search for efficient model checking of timed automata. In: *Monterey Workshop*. LNCS, vol. 4888, pp. 111–124. Springer, Heidelberg (2006)

15. Sivaraj, H., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. *Electr. Notes Theor. Comput. Sci.* 89 (2003)
16. Kuehlmann, A., McMillan, K.L., Brayton, R.K.: Probabilistic state space search. In: ICCAD, pp. 574–579. IEEE, Los Alamitos (1999)
17. Geldenhuys, J.: State caching reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–38. Springer, Heidelberg (2004)
18. Godefroid, P., Holzmann, G.J., Pirrottin, D.: State-space caching revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
19. Tronci, E., Penna, G.D., Intrigila, B., Zilli, M.V.: A probabilistic approach to automatic verification of concurrent systems. In: APSEC, pp. 317–324. IEEE Computer Society, Los Alamitos (2001)
20. Lin, F.J., Chu, P.M., Liu, M.T.: Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *SIGCOMM Comput. Commun. Rev.* 17, 126–135 (1987)
21. Edelkamp, S., 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)
22. Groce, A., Visser, W.: Model checking java programs using structural heuristics. *SIGSOFT Softw. Eng. Notes* 27, 12–21 (2002)
23. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. *Int. J. Softw. Tools Technol. Transf.* 6, 117–127 (2004)
24. Sankaranarayanan, S., Chang, R., Jiang, G., Ivancic, F.: State space exploration using feedback constraint generation and Monte-Carlo sampling. In: ESEC-FSE 2007, pp. 321–330. ACM, New York (2007)
25. Chockler, H., Farchi, E., Godlin, B., Novikov, S.: Cross-entropy based testing. In: FMCAD 2007, pp. 101–108. IEEE, Los Alamitos (2007)
26. Feige, U.: A tight upper bound on the cover time for random walks on graphs. *Random Struct. Algorithms* 6, 51–54 (1995)
27. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: FMICS 2005, pp. 98–105. ACM, New York (2005)
28. Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J.P., Mounier, L.: If: A validation environment for timed asynchronous systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 543–547. Springer, Heidelberg (2000)

Incremental Hashing for SPIN

Viet Yen Nguyen¹ and Theo C. Ruys²

¹ RWTH Aachen University, Germany

<http://www-i2.cs.rwth-aachen.de/~nguyen/>

² University of Twente, The Netherlands

<http://www.cs.utwente.nl/~ruys/>

Abstract. This paper discusses a generalised incremental hashing scheme for explicit state model checkers. The hashing scheme has been implemented into the model checker SPIN. The incremental hashing scheme works for SPIN's exhaustive and both approximate verification modes: bitstate hashing and hash compaction. An implementation is provided for 32-bit and 64-bit architectures.

We performed extensive experiments on the BEEM benchmarks to compare the incremental hash functions against SPIN's traditional hash functions. In almost all cases, incremental hashing is faster than traditional hashing. The amount of performance gain depends on several factors, though.

We conclude that incremental hashing performs best for the (64-bits) SPIN's bitstate hashing mode, on models with large state vectors, and using a verifier, that is optimised by the C compiler.

1 Introduction

An explicit state model checker is a model checker where all states are explicitly represented in the state space. Explicit model checking is sometimes called stateful state space exploration, especially when checking reachability or safety properties (e.g. deadlocks, assertion violations).

Central to stateful state space exploration is the process of state matching: for every encountered state, it should be checked whether the state has already been visited or not. As the run-time of exploration is linear in the number of transitions (i.e. the amount of newly encountered states and re-visited ones), it is obvious that state matching should be as fast as possible. Typically, hash tables are used to store states. Upon exploration of each state, the hash table is consulted to check whether that state has already been explored or not.

The access to a hash table is through a *hash function*. Given a key k , a hash function h computes the hash code $h(k)$ for this key. This hash code $h(k)$ corresponds to the address in the hash table, where this key k should be stored. For model checking, this k is typically the (binary) representation of a state, called the *state vector*. Most *traditional hash functions* compute $h(k)$ by considering all elements of k . For example, if k is a string, a typical hash function h would compute $h(k)$ on the basis of all individual characters of k .

With respect to state space exploration, two observations can be made. Firstly, the size of a state vector is usually substantial. State vectors of several hundreds of bytes are not exceptions. This means that computing a traditional hash code for such states can become quite expensive. Secondly, when exploring the state space in a structured manner (e.g. depth first search), the transitions between two consecutive states is local: only a small part of the state changes with respect to the previous state.

This last observation is the idea behind so called *incremental hash functions*, which use the hash code of a previous key to compute the hash code for the new key. The application of incremental hashing within a model checker is not new. Mehler and Edelkamp [11] implemented an incremental hashing scheme in the model checker StEAM, a model checker for C++. However, their incremental hashing scheme is only practicable for hashing (large) stacks and queues incrementally. We have improved their hashing scheme by generalising it for hashing vector-based data structures (like state vectors) incrementally by using cyclic polynomials from [2].

This improved scheme was originally developed for MOONWALKER [14]¹ a software model checker for CIL bytecode programs, i.e. .NET applications. Unfortunately, after implementing our incremental hash function into MOONWALKER, initial tests showed no measurable performance gain. We studied this observation using a profiler and found out that the stake of hashing in MOONWALKER is extremely small [12]. Other tasks that have to be performed for each state (e.g. garbage collection, state compression, etc.) take much more time. Any performance gain in hashing would therefore not be visible in the total running time.

The model checker SPIN [6] is arguably one of the fastest explicit state model checkers available. The current version of SPIN uses two traditional hash functions: one composed by Bob Jenkins [20] and one composed by Paul Hsieh [19]. For SPIN verifiers – unlike for bytecode model checkers – hashing accounts for a substantial amount of the running time.

We have implemented our generalised incremental hashing scheme into SPIN 5.1.4. The incremental hashing scheme works for checking safety properties (-DSAFETY) for both 32-bit and 64-bit architectures. Furthermore, it works in exhaustive mode and both approximate modes: bitstate hashing and hash compaction. We performed numerous experiments on the BEEM benchmarks [16] to compare the incremental hash functions against SPIN’s traditional hash functions. From these experiments we learnt that incremental hashing is faster than SPIN’s traditional hash implementations without sacrificing too much accuracy.

The amount of performance gain depends on several factors:

- the verification mode: exhaustive or approximate,
- the architecture on which the verification is run: 32-bit or 64-bit,
- the size of the state vector, and

¹ MOONWALKER was previously known as MMC: the Mono Model Checker. Due to several name clashes, MMC has recently been renamed to MOONWALKER.

- the optimisation parameters used for the GCC compiler: the default setting (-00) or the most aggressive optimisation setting (-03), and
- the verifier arguments, e.g., hashtable sizes, maximal search depth.

Incremental hashing performs best for (64-bit) bitstate hashing mode, with larger state vectors and using an optimised verifier.

The rest of the paper is organised as follows. In section 2, we first discuss some related work in the context of (incremental) hashing. Section 3 presents the general incremental hashing scheme: both the intuition of the method and an implementation in C are discussed in detail. Section 4 explains how we have implemented the method in SPIN and discusses the experimental settings of the benchmark runs that we have conducted. In section 5 we present the results of the benchmark runs and we discuss the outcome of the experiments. Finally, in section 6 we summarise the results and give pointers for future work.

2 Related Work

The hash table is the cornerstone of stateful state space exploration. Accesses to a hash table are in amortised $O(1)$ time. Although this is a good worst-case time-complexity, the constant costs are high if a bad hash function is chosen.

A good hash function should fulfill the following requirements [10]:

- *Fast*. The computation of the hash function should be efficient and fast.
- *Accurate*. To avoid a large number of address collisions, the hash values should distribute evenly over the range of the hash function.

With respect to accuracy, the following rule of thumb is often used: “one bit change in the key should result in half of the bits flipped in its hash code”.

A well known hash function for hashing arrays is the rolling hash function [1]. Given a ring R , a radix $r \in R$ and a mapping function T that maps array elements to R , the rolling hash code for an array $a = a_0 \dots a_n$ is computed as follows:

$$H(a) = T(a_0) + rT(a_1) + \dots + r^n T(a_n) = \sum_{i=0}^n r^i T(a_i) \quad (1)$$

A possible suitable ring R is \mathbb{Z}/B , where B is a prime number and is also the amount of buckets in the hash table. This specialisation of the rolling hash function is called hashing by prime integer division [2].

It is not difficult to see that the rolling hash function is prone to overflow, especially due to the power operations with the radix. Remedying overflow is costly. A recursive formulation of the rolling hash code is less prone to overflow:

$$H(a_0) = T(a_0) \quad (2)$$

$$H(a_i) = rH(a_{i-1}) + T(a_i) \quad 1 \leq i \leq n \quad (3)$$

Note that the radices are reversely mapped to the array elements when compared to equation 1, and therefore hash codes derived from the recursive formulation should not be matched against hash codes derived from the non-recursive formulation.

2.1 Incremental Hashing

Karp and Rabin [8] describe an incremental recursive hash function for fast string pattern matching by using recursive hashing by prime integer division. Their idea is to reuse the hash code of the previous unmatched substring for the calculation of the shifted substring. In [2], this is generalised for matching of n -grams.

The rolling hash function is not only amenable for incremental recursive hashing, but also incremental linear hashing. The idea behind incremental linear hashing, is that the contribution of an array element is independent of the contributions of other array elements. In case of an array change, the influence of the old array element is known and thus can be removed, followed by adding the influence of the new array element [2]. In [11], this is expressed as follows. Consider an array $k = v_0 \dots v_i \dots v_n$ and its successor $k' = v_0 \dots v'_i \dots v_n$, then the hash code of k' can be computed as follows:

$$H(k') = H(k) - r^i T(k_i) + r^i T(k'_i) \quad (4)$$

Depending on the ring chosen, the power operation with a large index i can easily lead to overflow. Thus using this hashing scheme for arbitrary modification of large arrays is impractical. For stacks and queues however, [11] describes a rewritten version of that formula for push and pop operations with the power operation removed. They tested it in their StEAM model checker, and got at least a speedup factor by 10 compared to non-incremental hashing. Note that this speedup was achieved with fixed-sized stacks of eight megabyte. It is logical to assume that the speedup factor will be much lower with with arbitrary sized stacks.

2.2 SPIN

SPIN [216] is a state-of-the-art explicit state model checker, which is used as a reference for other model checkers. With respect to memory efficiency and verification time, SPIN is hard to beat. SPIN provides many ways to tune and optimize its verification runs. Most of these optimisation features can be enabled via compilation flags for the C compiler (e.g. GCC). See for details chapter 18 of [6].²

SPIN supports three verification modes. The most commonly used verification mode is the *exhaustive* mode, where all states are stored until the memory to store the states is exhausted. SPIN provides state compression techniques to fit more states in the same amount of memory.

For the cases where there is not enough memory to store all (compressed) states, SPIN supports two lossy, approximate verification modes, which are both heavily based on hashing functions: bitstate hashing and hash compaction. A good survey and extensive discussion on various approximate methods can be found in [10].

² As this paper is concerned with tuning and optimizing SPIN verification runs, we use several of these compilation parameters, usually prefixed with `-D`.

Bitstate hashing. Holzmann's bitstate (sometimes called supertrace) hashing [4,5] algorithm works as follows. Under the assumption that the number of slots in the hash table is high in comparison to the number of reachable states, then it is possible to identify a state by its address in the hash table. In this case, a single bit suffices to indicate whether a state has already been visited or not.

The coverage of bitstate hashing can be improved (on the expense of time) by using k different, independent hash functions for the same hash table. A state is considered visited before if the bits for all k hash functions are set. This variant of bitstate hashing is called k -fold bitstate hashing. Triple hashing [3] improves upon this scheme by using three hash values to generate k hashes. For bitstate hashing to be effective, the accuracy of the hash function(s) involved should be high.

Initially, SPIN used $k = 2$ by default, but since October 2004, SPIN's default is set to $k = 3$. Of course, the variable k can be set to larger values using a run-time option.

Bitstate hashing in SPIN is enabled using the `-DBITSTATE` parameter.

Hash compaction. Wolper and Leroy [15] introduced hash compaction, a indication variant of bitstate hashing. The idea of hash compaction is to store the addresses of the occupied bit positions in the bitstate table, rather than storing the whole array itself.

For hash compaction, the hash table is taken to be very large (e.g. 2^{64} bits), much too large to fit in memory. Now the address computed by the hash function (e.g. 64 bit-wide) is stored as being a normal state. Hash compaction is thus also viewed as a lossy form of state compression. Hash compaction is more accurate than k -fold bitstate hashing (for small k).

Hash compaction in SPIN is enabled using the `-DHC` parameter.

The current versions SPIN uses two traditional, linear hash functions:

- *Jenkins.* Since long, SPIN uses Jenkins hash function [20,17] for both its exhaustive and approximate verification runs. Jenkins' hash function is considered a fast but still quite accurate hash function.

For bitstate hashing, SPIN uses 96-bit and 192-bit versions of Jenkins' hash function. For exhaustive verification, a part of the 96-bit or 192-bit hash value is used.

- *Hsieh.* Since version 5.1.1 (Nov 2007), SPIN has adopted an alternative hash function by Hsieh [19]. Although perhaps not as accurate as Jenkins, Hsieh's hash function can in some cases be much faster.

Hsieh's hash function can be enabled with the parameter `-DSFH`, which stands for 'Super Fast Hash'. But Hsieh's hash function is also automatically selected for 32-bit safety runs (`-DSAFETY`). To speed up such verification runs even further, SPIN's default mask-compression of states is disabled for `-DSAFETY` runs as well. Hsieh's hash function is not only fast, but its implementation is also suitable for aggressive optimisation by the GCC compiler (i.e. using `-O2` or `-O3`).

Currently, there only exists a 32-bit version of Hsieh's hash function. Furthermore, in the current version of SPIN, Hsieh's hash function can *only* be enabled for checking safety properties.

3 Generalised Incremental Hashing Scheme

This section presents the intuition behind the incremental property, the time-complexity of the incremental hashing scheme and implementation variants. A few concepts, like polynomial rings, from field theory are used to express this. Readers unfamiliar with this may consult [2, Appendix A].

3.1 Incremental Property

Consider a Galois field (also known as a finite field) $R = GF(2)[x]/(x^w + 1)$, the ring consisting of polynomials in x whose coefficients are 0 or 1, reduced modulo the polynomial $x^w + 1$. Make sure that w matches the computer's word size, thus 32 for 32-bits words. The polynomials are represented by w -sized bitmasks by placing the coefficients of x^i at the i^{th} bit, creating an one-on-one correspondence between polynomials in R and the bitmasks.

As a radix, the polynomial $x^\delta \in R$ is chosen. By setting radix $r = x^\delta$, the following incremental hash function is derived from equation 4:

$$H(k') = H(k) + x^{\delta i}T(k_i) + x^{\delta i}T(k'_i) \quad (5)$$

The minus operation from equation 4 is replaced by an $+$, because addition and subtraction are the same in ring R . Now, consider an arbitrary member $q \in R$ with $q(x) = q_{w-1}x^{w-1} + q_{w-2}x^{w-2} + \dots + q_0$. The multiplication of the x and $q(x)$ is the following:

$$xq(x) = q_{w-1}x^w + q_{w-2}x^{w-1} + \dots + q_0x \quad (6)$$

$$= q_{w-2}x^{w-1} + q_{w-3}x^{w-2} + \dots + q_0x + q_{w-1} \quad (7)$$

Equation 7 is equation 6 reduced to modulo $x^w + 1$. The multiplication by polynomial x results to a left rotate of the coefficients in $q(x)$, hence the name cyclic polynomials. For most platforms, this is an efficient operation. At least all x86-architectures include native bit rotate instructions. Additions in equation 5 can be implemented using a exclusive-or operation, which is available on all processor platforms.

In order to reduce the amount of operations, equation 5 can be rewritten by applying the associativeness of the $+$ operation, as shown in the next equation:

$$H(k') = H(k) + x^{\delta i}(T(k_i) + T(k'_i)) \quad (8)$$

So far, only one variable is left unmentioned, namely δ . The choice of a δ for the radix x^δ was experimentally evaluated by [2]. No δ clearly stood out. For $\delta = 1$, the incremental hashing function worked well and they used is subsequently for their experiments. For this reason, we also take 1 for δ .

Furthermore, as described in [2], cyclic polynomials have one weakness. They have a cycle length of size w for which it computes the hashcode of zero. For example, if a key of size $2w$ starts with w elements followed by another identical sequence of w elements, then the hashcode for that key is zero. In practise, such keys are extremely rare in model checking, as their size must be exactly nw -sized, where $n \in \mathbb{N}$, and that its contents should be also w -cyclic as well.

3.2 Time-Complexity

The time-complexity of the incremental hashing scheme is differently defined compared to traditional hash functions. A fast traditional hash function has a time-complexity in $O(N)$, where N is the array length. The incremental hash function has a time-complexity of $O(1)$ for one change to the array. Theoretically, the incremental hash function is faster if the amount of changes between successive states is smaller than N . This is usually the case in model checking, where the amount of changes is usually 1 or 2 and almost never near N .

3.3 Variants

From the perspective of implementation, there are several variants of the incremental hashing scheme at one's disposal, namely by reordering the coefficients with respect to the bitmask and by using different mappings of function T in equation 4.

Reordering the Bitmask. The coefficients of polynomials in R were initially mapped to a bitmask whose position coincide with those in the bitmask. This mapping was chosen to allow efficient left-rotate operations on bitmasks as the equivalent to multiplication by x . Another ordering of coefficients that works equally well is by ordering the coefficients reversely: the coefficient of x^j is placed at the $64 - j$ bit position. Such a mapping results to right-rotate operations as the equivalent to multiplication by x .

Different Mappings. Our initial experiments with the incremental hashing scheme led to high collision rates. This was caused by the initial mapping of function T in equation 5, for which we originally chose the identity function. The source of the collisions lied in the entropy of changes between state vectors of successive states. Transitions are often of low entropy, like changing a variable from 0 to 1 or add 1 upon variable i . The incremental hash function recalculates the hash function upon such changes, but since the entropy is low, the resulting hash would not differ much as one desires for a good hash function. In order to increase entropy, we experimented with different functions of T .

Our approach is by using integer hash functions as a T . We initially used Wang's integer hash [22], but its constant time-complexity is relatively big compared to that of the incremental hashing scheme, and we observed in experiments that the slowdown made incremental hashing slower than traditional hashing functions.

The function T has therefore be very fast. An integer hash function for which we observed that it works out well is Knuth's multiplicative constant [9]. This hash function simply multiplies the input by a word-size dependent constant. For 32-bit words, the constant is 2654435769, and for 64-bits words, the constant is 11400714819323198485. The constant is calculated by multiplying one-bitmask (i.e., the largest number in the sized word) by the golden ratio $(\sqrt{5} - 1)/2 \approx 0.618034$. In [9], Knuth shows this integer hash function has a high likelihood of spreading the bits through the word, thereby increasing entropy.

Other constants are also applicable. We also experimented with the magic constants of the FNV hash function [18], which is 2166136261 for 32-bits words and 14695981039346656037 for 64-bits words. These FNV constants are ‘magic’, because their effectiveness was only evaluated by empiric evidence.

3.4 Implementation Examples

Here we present several C implementations of the incremental hashing scheme. The implementation below is one for SPIN:

```
c_hash(int i, unsigned int old, unsigned int new) {
    const unsigned long knuth = 11400714819323198485UL;
    const unsigned long fnv = 14695981039346656037UL;
    unsigned long diff = ((new)*knuth) ^ ((old)*knuth);
    chash ^= ((diff << i) | (diff >> (64 - i)));
#ifdef BITSTATE || defined(HC)
    unsigned long diff2 = ((new)*fnv) ^ ((old)*fnv);
    chash2 ^= ((diff2 >> i) | (diff2 << (64 - i)));
    chash3 ^= ((diff >> i) | (diff << (64 - i)));
#endif
}
```

For exhaustive search, only Knuth’s multiplicative constant with left-rotatable bitmask are used. For hash compaction, a second hash value is maintained using FNV’s constant. For bitstate hashing, triple hashing is used by maintaining a third hash value using a right-rotatable bitmask in combination with Knuth’s multiplicative constant. We will refer to this implementation as CHASH (where the ‘C’ stands for cyclic).

Another triple incremental hashing approach is by viewing three words as one word, upon which a bit rotate is performed. This approach is less optimisable because no processor supports a native bit rotate operation for triple-word sized values. We experimented shortly with this approach but found out it always being outperformed by the above variant.

4 Experimental Method

We originally implemented CHASH in SPIN 4.3.0, and the results with it are described [12]. Since that thesis and this paper, version SPIN 5.1.4 came out and we ported CHASH to it. We subsequently used this newer implementation and benchmarked it extensively against SPIN’s default hash functions.

4.1 Implementation

The difficulty of implementing CHASH varies from language to language. In MOON-WALKER, which is written in C#, the implementation was extremely easy due to object encapsulation of the state vector, and therefore also all writes calls to it.

SPIN however is implemented in C and therefore lacks the expressive means for encapsulated state vector access. The state vector in SPIN is accessible via the global point `now` and is updated by writing to an offset from this pointer. In order to detect all these writes, which can happen throughout the generated verifier, we had to add a call to the incremental hashing function just before the state vector is updated at that point.

Besides this, we had to overcome several other issues due to specifics in the C language. For one, our implementation uses the memory address of the written variable as the index argument to function `c_hash`. This however did not work for the Promela datatypes `unsigned int` and `bit`. SPIN uses bitfields as the underlying C datatype, and bitfields have by definition no addresses. To solve this, we created a virtual memory mapping for unsigned ints and bits. When the verifier is generated, the address of the variable in the symboltable is used as the index instead. We could not use a virtual mapping for all variables because of arrays. Accesses to arrays in SPIN may have an expression as indexer and its value is only known at runtime, not when the verifier is generated.

Also, we could not just use the memory addresses, but we had to use memory offsets. Using offsets is important for approximative methods, because the approximation of the explored state space can slightly differ due to changed memory addresses. These are susceptible to operating system semantics. Offsets are relative and remain the same between runs.

Additionally, we optimised the time-complexity at a small cost of memory. When the DFS search backtracks, `CHASH` has to be called for a reverse operation in order to calculate the correct corresponding hash code. However instead, we store the hash values on the DFS stack, and write this value as the corresponding hashcode.

4.2 BEEM Benchmarks

We used the BEEM benchmark suite for evaluating the effectiveness of the incremental hashing scheme. This suite consists of 57 models, ranging from communication protocols, mutual exclusion algorithms, election algorithms, planning and scheduling solvers and puzzles [13,16]. The model are parameterised to yield different problem instances. The total amount of models is 298 and 231 of them are in Promela. We initially evaluated all the Promela models for our experiments. From this evaluation, we made a selection of the 40 largest models that did not run out of memory. These were subsequently used for comparing the different hashing configurations.

Due to space constraints, we are only able to present a selection of the results from these 40 models. We chose to present the ten best problem instances, the ten worst problem instances and averages of the forty selected BEEM benchmark suite, thereby giving a nuanced perspective of the results.

4.3 Setup

We ran the benchmarks on nine identical nodes, each equipped with Intel Xeon 2.33 GHz processors and 16 GB memory. When compiling the models, we always

enabled the `-DSAFETY` and `-DMEMLIM=15000`. As arguments to `pan`, we fixed the maximal search depth to $20 * 10^6$ and disabled stopping on errors and printing unreachable states. Furthermore, we conducted runs with the following compiler flags and `pan` arguments:

Compiler flags				Pan arguments
-m32	-DHASH32	-DSFH		-w26
-m32	-DHASH32	-DSPACE	-DNOCOMP	-w26
-m32	-DHASH32	-DCHASH	-DNOCOMP	-w26
-m32	-DHASH32	-DSPACE		-w26
-m32	-DHASH32	-DCHASH		-w26
-m32	-DHASH32	-DBITSTATE		-w32 -k3
-m32	-DHASH32	-DBITSTATE	-DCHASH	-w32 -k3
-m32	-DHASH32	-DHC		-w27
-m32	-DHASH32	-DHC	-DCHASH	-w27

These are 32-bits runs. Additionally, we also ran a series of 64-bits runs using the following compiler flags and `pan` arguments:

Compiler flags				Pan arguments
-m64	-DHASH64	-DSPACE		-w28
-m64	-DHASH64	-DCHASH		-w28
-m64	-DHASH64	-DBITSTATE		-w36 -k3
-m64	-DHASH64	-DBITSTATE	-DCHASH	-w36 -k3
-m64	-DHASH64	-DHC		-w29
-m64	-DHASH64	-DHC	-DCHASH	-w29

All configurations were run twice, namely without compiler optimisations (`-00`) and with (`-03`). We furthermore used the GNU profiler on all configurations. All these configurations come down to the total amount 2400 of verifications runs of which we captured their output, processed it and analysed it to present it in the next section.

5 Results and Discussion

Our benchmark runs generated extensive results which we cannot put all here. We therefore highlight the interesting observations and in case of interest, the full result set is downloadable from the incremental hashing webpage [\[17\]](#).

5.1 Exhaustive Verification

In 32-bits exhaustive verification we compared `CHASH` and Jenkins's against Hsieh's. The result is shown in table [\[1\]](#). The first column is the state vector size, followed by the state space size in 10^6 and transitions in 10^6 . Collision rates are indexed against the state space. They can be higher than 100 because Spin counts each chain hit in the collision chain as a collision. The verification times of Jenkins's and `CHASH` are indexed against Hsieh's.

The table shows that the average gain of `CHASH` over Hsieh's is three percent when the models are compiled with `-03` and 25 percent when the models are compiled with `-00`. Later on, we shall discuss the differences between `-00` and `-03`. Most of the ten worst performing models have higher collision rates when used with `CHASH` in comparison to Jenkins's. It is also noteworthy that Jenkins's

Table 1. 32-bits exhaustive search BEEM benchmark results of Hsieh versus Jenkins (Jen.) and CHASH

model	sv (bytes) states ($\cdot 10^6$)			transitions ($\cdot 10^6$)			Hsieh (%)			Jen. (%)			CHASH (%)		
	collrate	time -O3	time -O0	Hsieh $\geq 100\%$	Jen. $\geq 100\%$	CHASH $\geq 100\%$	Hsieh $\geq 100\%$	Jen. $\geq 100\%$	CHASH $\geq 100\%$	Hsieh $\geq 100\%$	Jen. $\geq 100\%$	CHASH $\geq 100\%$			
elevator_planning.2	48	11	93	939	51	434	47	81	79	54	87	66			
firewire_link.5	404	6	12	3	3	3	24	104	86	34	105	61			
adding.6	28	8	12	41	2	2	6	92	87	7	95	78			
telephony.4	52	12	64	26	20	28	28	99	89	37	106	71			
train-gate.3	136	20	57	16	16	15	49	101	90	73	102	67			
schedule_world.3	44	4	44	105	21	20	18	101	90	26	102	75			
phils.6	84	14	143	93	92	96	86	108	91	109	103	68			
peterson.6	44	9	33	13	12	15	16	99	92	21	101	77			
lann.3	128	5	24	8	8	11	24	98	94	33	104	77			
fischer.6	56	8	33	17	11	11	17	102	94	24	101	76			
...															
protocols.5	100	3	8	2	2	16	6	106	100	9	101	74			
elevator2.3	40	8	55	166	22	185	20	105	101	28	99	73			
driving_phils.4	84	11	30	6	6	106	15	103	102	22	104	72			
elevator.3	140	19	70	25	24	37	74	108	103	112	103	79			
lampport_nonatomic.4	180	16	60	19	19	82	69	98	105	96	101	73			
msmie.4	100	7	11	2	2	2	10	103	106	16	104	79			
bridge.2	60	9	27	12	12	12	21	103	107	36	102	89			
sorter.4	60	13	27	11	8	23	19	102	112	28	103	87			
reader_writer.3	160	1	4	1	1	1	16	99	126	29	101	94			
Average	98	12	47	52	17	47	30	102	97	43	102	75			

Table 2. 64-bits exhaustive search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model	sv (bytes) states ($\cdot 10^6$)			transitions ($\cdot 10^6$)			Jen. (%)			CHASH (%)		
	collrate	time -O3	time -O0	Jen. $\geq 100\%$	CHASH $\geq 100\%$	Jen. $\geq 100\%$	CHASH $\geq 100\%$	Jen. $\geq 100\%$	CHASH $\geq 100\%$			
brp.5	148	11	20	1	3	29	61	50	90			
brp.4	148	7	13	1	4	21	64	35	90			
driving_phils.4	88	11	30	2	9	22	88	46	83			
hanoi.3	116	14	43	5	7	39	91	91	86			
phils.6	140	14	143	23	23	108	92	243	86			
elevator2.3_prop4	52	8	55	5	9	26	92	57	88			
elevator2.3	52	8	55	5	9	26	92	58	86			
mcs.5	68	29	116	10	11	63	93	142	87			
train-gate.3	164	20	57	4	29	49	93	123	97			
telephony.7	64	22	114	9	15	55	93	130	87			
...												
schedule_world.3	52	4	44	5	12	23	96	52	87			
bakery.5	48	7	25	2	31	13	97	30	82			
lann.3	140	5	24	2	2	24	97	54	99			
msmie.4	180	7	11	0	1	14	97	31	92			
production_cell.4	304	10	42	5	6	50	97	138	97			
elevator.3	152	19	70	6	9	74	97	177	93			
protocols.5	112	3	8	1	7	8	98	17	96			
frogs.4	68	17	36	2	2	27	98	60	92			
train-gate.2	164	18	50	3	30	42	99	108	97			
reader_writer.3	276	1	4	0	1	19	99	39	100			
Average	120	12	47	4	20	34	94	79	91			

Table 3. 32-bits bitstate search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model		states $\frac{(-10^6)}{100\%}$		time Jen. (sec)		states/sec			
		Jen. (%)	CHASH (%)	Jen. (%)	CHASH (%)	Jen. (sec)	CHASH (sec)		
		coverage	-O3	rate -O3	-O0	rate -O0			
firewire_link.5	6	100	100	18	322	116	36	167	161
production_cell.4	10	100	100	42	237	109	92	109	153
phils.6	14	100	100	86	166	109	134	107	144
train_gate.3	20	100	100	43	465	109	78	253	139
train_gate.2	18	100	100	38	474	108	69	259	137
elevator2.3	8	100	47	25	302	107	36	211	122
fischer.6	8	100	100	20	426	107	30	280	128
elevator_planning.2	11	100	100	37	311	106	56	204	132
telephony.7	22	100	97	52	419	106	84	262	133
brp.5	11	100	100	18	603	105	32	337	135
...									
driving_phils.4	11	100	41	17	653	95	27	413	121
elevator.3	19	100	100	68	274	95	121	154	125
lann.4	13	100	99	55	231	92	105	120	123
bridge.2	9	100	100	24	392	92	40	230	106
msmie.4	7	100	100	12	605	92	20	359	123
frogs.4	17	100	100	26	662	90	42	412	111
protocols.5	3	100	63	8	413	89	12	265	111
sorter.4	13	100	89	21	625	89	34	383	117
reader_writer.3	1	100	91	16	47	76	30	25	102
hanoi.3	14	100	0	33	429	4	55	260	6
Average	12	100	91	31	421	98	51	265	123

has the lowest collision rates (as reflected in the average collision rate), followed by CHASH and Hsieh’s.

In 64-bits mode (see table 2) we see that CHASH is on average six percent faster than Jenkins’s for -O3 and nine percent faster for -O0. This gain is visible on all models for both -O0 and -O3, even though the collision rates of CHASH are either on par or worse.

5.2 Bitstate Hashing

For bitstate hashing, we denote the accuracy of the search by the coverage indexed against the full state space size, which we know from the exhaustive verifications. Furthermore, we indexed the verification times of CHASH against Jenkins’s here.

With 32-bits bitstate hashing (see table 3), we observed with -O3 an average performance decrease of two percent using CHASH. With -O0 there is a performance gain by 23 percent. With 64-bits bitstate hashing (see table 4, we see a performance gain of CHASH by 26 percent (with -O3) and 61 percent (with -O0).

The coverage rates of Jenkins’s shows that it is a good hash function in terms of accuracy. CHASH performs less in that respect, as few models like hanoi.3, protocols.5, sorter.4 and reader_writer.3 have relatively low coverage rates. We found out this is due to the low entropy input as discussed earlier. Though this is combated using integer hash multiplication, in rare cases as these it is yet insufficient. We found out that additional effective measures are reordering the declaration of variables and/or making the state vector more sparse by adding dummy

Table 4. 64-bits bitstate search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model		states $\{10^6\}$		time Jen. (sec)		states/sec			
		Jen. (%)	CHASH (%)	Jen. (%)	CHASH (%)	Jen. (states/sec)	CHASH (states/sec)	Jen. (%)	CHASH (%)
	coverage	-O3		rate -O3	-O0	rate -O0			
train-gate.2	18	100	100	76	234	163	138	129	207
production_cell.4	10	100	100	77	130	160	158	63	214
train-gate.3	20	100	100	79	250	149	155	128	208
production_cell.3	6	100	100	67	87	145	131	44	186
firewire_link.5	6	100	92	42	141	144	77	77	234
phils.6	14	100	100	208	69	137	305	47	196
bakery.7	28	100	100	83	332	136	130	211	161
fischer.6	8	100	100	40	206	135	67	125	170
brp.5	11	100	100	38	285	134	65	168	176
lamport_nonatomic.4	16	100	100	111	146	132	179	91	176
...									
schedule_world.3	4	100	100	38	113	120	60	71	149
bridge.2	9	100	100	41	229	120	70	134	133
peterson.6	9	100	100	42	206	120	59	146	141
lamport.7	5	100	100	25	186	119	35	134	140
adding.6	8	100	100	19	399	115	25	309	129
elevator_planning.2	11	100	100	78	147	114	117	98	156
at.4	7	100	100	32	203	113	49	136	150
sorter.4	13	100	88	39	335	112	64	207	151
reader_writer.3	1	100	91	26	29	98	46	16	114
protocols.5	3	100	37	19	165	58	26	119	78
Average	12	100	97	61	205	126	100	130	161

variables can be quite effective. Depending on the model, we gained nearly on par coverage. A patch against SPIN that generates such models can be downloaded from the incremental hashing webpage. Note that these more unconventional measures are not universally effective and we measured that in general they decrease coverage. For this reason, they are not enabled with CHASH by default.

5.3 Hash-Compaction

The results from hash-compaction are similar to those from bitstate hashing. Here too we measured a small decline in performance when CHASH is used in 32-bits mode and -O3 and a significant performance improvement of 26 percent when -O0 is used. See table 5.

With 64-bits hash-compaction (see table 6), we see that CHASH improves by ten percent over Jenkins’s with -O3 and 29 percent with -O0. For the same reasons as for bitstate hashing, we see that here too a lower coverage goes accompanied by lower performance.

5.4 Optimisation Flags

While we ran our benchmarks with and without compiler optimisations enabled, SPIN users usually do without them. XSpin does not enable them by default and users usually forget to enable them manually. Based our results, we see that enabling optimisations (i.e. -O3) makes a significant difference, as it reduces the

Table 5. 32-bits hash-compaction search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model		states ($\cdot 10^6$)		time Jen. (sec)		rate -O3		rate -O0	
		Jen. (%)	CHASH (%)	Jen. (%)	CHASH (%)	Jen. (states/sec)	CHASH (%)	Jen. (states/sec)	CHASH (%)
firewire_link.5	6	100	100	16	372	122	33	181	180
phils.6	14	100	100	63	229	116	109	132	149
train-gate.2	18	100	100	32	565	110	63	283	140
train-gate.3	20	100	100	36	554	110	71	278	139
production_cell.4	10	100	100	36	275	109	84	119	158
telephony.7	22	100	97	37	587	109	70	312	137
telephony.4	12	100	100	22	566	106	40	304	140
lampport_nonatomic.4	16	100	72	45	357	105	91	177	141
mcs.5	29	100	93	42	692	105	72	405	126
peterson.6	9	100	100	14	621	105	22	393	123
...									
lampport.7	5	100	100	7	661	97	11	423	113
driving_phils.4	11	100	41	12	923	95	23	489	127
protocols.5	3	100	63	6	561	94	10	327	119
lann.4	13	100	99	47	267	93	98	128	126
elevator_planning.2	11	100	100	29	398	91	48	237	121
bridge.2	9	100	100	19	497	90	35	264	110
msmie.4	7	100	100	9	826	89	16	433	123
sorter.4	13	100	89	16	840	83	29	455	111
reader_writer.3	1	100	91	15	50	75	29	26	99
hanoi.3	14	100	0	25	583	4	47	306	8
Average	12	100	91	23	582	98	44	322	126

Table 6. 64-bits hash-compaction search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model		states ($\cdot 10^6$)		time Jen. (sec)		rate -O3		rate -O0	
		Jen. (%)	CHASH (%)	Jen. (%)	CHASH (%)	Jen. (states/sec)	CHASH (%)	Jen. (states/sec)	CHASH (%)
phils.6	14	100	100	83	174	131	150	95	179
production_cell.4	10	100	100	39	256	128	91	110	150
train-gate.2	18	100	100	36	500	123	73	246	142
train-gate.3	20	100	100	39	507	122	84	237	149
firewire_link.5	6	100	92	19	312	120	38	157	156
mcs.5	29	100	100	54	535	118	94	308	146
lampport_nonatomic.4	16	100	100	49	328	117	98	165	136
elevator2.3	8	100	100	23	332	116	39	199	131
telephony.7	22	100	100	45	493	116	78	281	135
production_cell.3	6	100	100	34	168	116	78	74	137
...									
lampport.7	5	100	100	12	391	106	17	273	118
msmie.4	7	100	100	13	541	104	23	313	128
bridge.2	9	100	100	23	400	104	43	216	107
driving_phils.4	11	100	83	16	690	104	27	416	126
schedule_world.3	4	100	100	19	220	102	34	126	114
adding.6	8	100	100	9	836	101	12	634	111
sorter.4	13	100	88	21	628	99	34	388	111
elevator_planning.2	11	100	100	36	317	97	62	185	129
reader_writer.3	1	100	91	19	40	94	35	21	101
protocols.5	3	100	37	9	343	51	14	226	68
Average	12	100	97	29	429	110	53	250	129

verification time nearly by an half. This substantial improvement costs only a few seconds additional compilation time.

5.5 Memory Consumption

We also extracted memory utilisation statistics for runs³ with and without CHASH. For both -00 and -03, we measured an average memory overhead by CHASH of six percent compared to runs with Jenkins's. This is caused by our implementation, which maintains hash values on the DFS stack such that a reverse CHASH operation does not have to be computed.

5.6 Profiler Runs

We also wanted to find out how much CHASH improves and whether there is more room for improvement. We ran a profiled version on our selection of the BEEM benchmark suite. For pointing out the interesting points, it suffices to only present the combined profiler data from 32-bits and 64-bits exhaustive verification. See figures 1 and 2. In this figure, `d_hash` is Jenkins's hash function, `c_hash` is the CHASH implementation, `hstore` is the hashtable storage function, `new_state` is the DFS routine, `compress` is the mask-compression function, `do_transit` performs one transition from the current state and `misc` are all other functions.

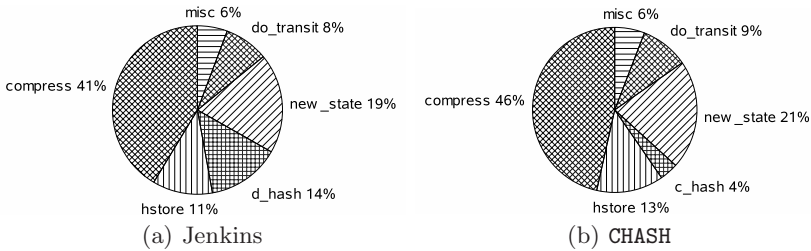


Fig. 1. Percental stakes of five most time-consuming functions for BEEM benchmarks compiled with -00

The stake of hashing with Jenkins's is for both -00 and -03 clearly visible in the total running time. When CHASH is enabled, it eliminates hashing as a visible stake in the total running time. For -03, the stake of CHASH is near zero and therefore not depicted in the figure.

Also noteworthy is that `compress` and `hstore` have a significant stake for both -00 and -03. The former is in SPIN 5.1.4 disabled by default in case of 32-bit exhaustive verification of safety properties. We measured the impact of this in our benchmarks, and detected that disabling mask-compression improves

³ We were not able to include runs with `compress` disabled for measuring memory overhead, as Spin does not output detailed memory statistics when `compress` is disabled.

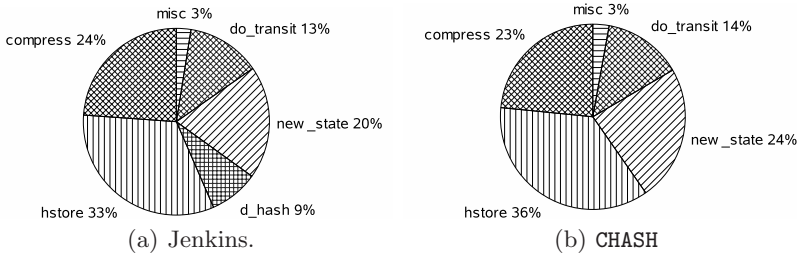


Fig. 2. Percentual stakes of five most time-consuming functions for BEEM benchmarks compiled with `-O3`

performance by ten percent in `-O3` and 36 percent in `-O0`. This however comes at the cost of increased memory consumption. Unfortunately, this is not measurable because runs with `compress` disabled do not output detailed memory statistics.

5.7 Extremely Long Runs

We also experimented with models that either run out of memory or have high verification times. We specifically reran the BEEM benchmark with 64-bits bit-state enabled, profiler enabled, compiler optimisations enabled, hashtable size of 2^{36} , maximal depth of 20 million and $k = 3$. The table below is a selection of five models with the longest verification times:

model	sv size (bytes)	type	states ($\cdot 10^9$)	transitions ($\cdot 10^9$)	depth	time (hours)	rate (states/sec)	gain (%)
firewire_link.6	420	Jenkins	19	64 1887952	54.9	94598	134	
		CHASH	19	67 1281175	42.1	126380		
peg_solitaire.6	60	Jenkins	2	25	36	12.4	53540	103
		CHASH	2	25	36	12.0	55238	
driving_phils.5	96	Jenkins	6	15	304	4.6	339291	133
		CHASH	5	14	304	3.3	450963	
lamport_nonatomic.5	224	Jenkins	2	8	max.	4.0	105885	121
		CHASH	1	7	max.	3.1	128062	
telephony.6	64	Jenkins	1	8	max.	2.1	186624	125
		CHASH	1	8	max.	1.7	233323	

The gain represents the verification rate index of `CHASH` when compared against Jenkins's. Runs for which the maximal depth was reached are indicated by `max.` in the depth column.

As can be seen, `CHASH` improves greatly over Jenkins with an improvement of up to 34 percent. As can be seen from the times, this improvement can save hours of verification. Also particularly interesting is that this selection of models have quite large state vectors. This suggests a correlation between the state vector

size and the performance gain by CHASH. The BEEM benchmark suite includes too little models with large state vectors and significant large state spaces in order to measure such a correlation.

6 Conclusions

There are still several ways to improve our incremental hashing scheme as implemented in Spin. First, we used integer hash functions to improve its uniformity, and though this works out well, there is room for improvement. We saw that for some models, the collision rates were relatively high and/or the coverage rates relatively low. By devising other methods for function T , the mapping of integers to the ring R , this may be improved. Our experiments with sparse state vectors and variable reordering for bitstate hashing also help, but more investigation is required to define an approach that is on average substantially better.

Also, currently untested is the use of incremental hashing with multi-core model checking (available since Spin 5) and the verification of liveness properties. This is likely to require additions to the CHASH implementation.

CHASH can be also used orthogonally upon traditional hash function, as a good second opinion second hash. This hash code can be stored along the state in the hash table, and used as an additional check before byte-for-byte comparison is done. This can improve the performance of `hstore` function, of which profiler results have shown that improvements in this function is likely to be reflected in the total running time.

The concept of incremental computation can also be extended to mask-compression and the state collapse. Having an incremental collapse also enables a nicer implementation of incremental hashing, as it will not be necessary anymore to add a `c_hash` at every update of the state vector.

Lastly, the BEEM benchmark suite served their purpose for the greater part of our experiments. It was only lacking on one point, and that is where we wanted to unfold a correlation between the state vector size and the performance gain. The problem lies in the lack of models that have both large state spaces and large state vectors. Adaptations of models in the current suite, or a series of new models that do have those properties would be welcoming for increasing the usefulness of the BEEM benchmark suite even further.

Conclusive, we described an incremental hashing that is applicable to any state vector datastructure, implemented it in SPIN and evaluated it using the BEEM benchmarks. From this evaluation, we observed that SPIN's default settings, namely with compiler optimisations disabled, that incremental hashing is superior to Hsieh's SFH and Jenkins's in all cases. With the most aggressive safe compiler optimisations enabled, namely `-O3`, SFH is generally better for 32-bits exhaustive search, Jenkins's for all other 32-bits verification modes and incremental hashing is better in all approximate modes and 64-bits in particular. The average reduction of applying compiler optimisation is nearly a half. We recommend it to always enable it, and in case when 64-bits machines are used, combine this with incremental hashing.

The full result set from the BEEM benchmarks and CHASH patch against SPIN 5.1.4 can be downloaded from the incremental hashing webpage [17].

References

1. Baase, S., van Gelder, A.: *Computer Algorithms*, 3rd edn. Addison-Wesley (2000)
2. Cohen, J.D.: Recursive Hashing Functions for N-grams. *Transaction On Information Systems* 15(3), 291–320 (1997)
3. Dillinger, P.C., Manolios, P.: Fast and Accurate Bitstate Verification for SPIN. In: Graf, S., Mounier, L. (eds.) *SPIN 2004*. LNCS, vol. 2989, pp. 57–75. Springer, Heidelberg (2004)
4. Holzmann, G.J.: On Limits and Possibilities of Automated Protocol Analysis. In: *Proc. 7th Int. Workshop on Protocol Specification, Testing and Verification (PSTV 1987)*, pp. 137–161. North-Holland, Amsterdam (1987)
5. Holzmann, G.J.: An Analysis of Bitstate Hashing. *Formal Methods in System Design* 13, 289–307 (1998)
6. Holzmann, G.J.: *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston (2004)
7. Jenkins, B.: Hash Functions. *Dr. Dobbs Journal* 22(9) (September 1997)
8. Karp, R.M., Rabin, M.O.: Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development* 31(2), 249–260 (1987)
9. Knuth, D.E.: *The Art of Computer Programming*, 2nd edn. Sorting and Searching, vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City (1998)
10. Kuntz, M., Lampka, K.: Probabilistic Methods in State Space Analysis. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) *Validation of Stochastic Systems*. LNCS, vol. 2925, pp. 339–383. Springer, Heidelberg (2004)
11. Mehler, T., Edelkamp, S.: Dynamic Incremental Hashing in Program Model Checking. *ENTCS* 149(2), 51–69 (2006); *Proc. of Third Workshop of Model Checking and Artificial Intelligence (MoChArt 2005)*
12. Nguyen, V.Y.: *Optimising Techniques for Model Checkers*. Master’s thesis, University of Twente, Enschede, The Netherlands (December 2007)
13. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) *SPIN 2007*. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
14. Ruys, T.C., de Brugh, N.H.M.A.: MMC: the Mono Model Checker. *ENTCS* 190(1), 149–160 (2007); *Proc. of Bytecode 2007*, Braga, Portugal
15. Wolper, P., Leroy, D.: Reliable Hashing without Collision Detection. In: Courcoubetis, C. (ed.) *CAV 1993*. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)
16. BEEM: BEENCHMARKs for EXPLICIT Model checkers,
<http://anna.fi.muni.cz/models/>
17. CHASH - Incremental Hashing for SPIN,
<http://www-i2.cs.rwth-aachen.de/~nguyen/incrementalHashing>
18. Fowler/Noll/Vo (FNV) Hash, <http://isthe.com/chongo/tech/comp/fnv/>
19. Hsieh, P.: Hash functions, <http://www.azillionmonkeys.com/qed/hash.html>
20. Jenkins, B.: A Hash Function for Hash Table Lookup,
<http://burtleburtle.net/bob/hash/doobs.html>
21. SPIN: on-the-fly, LTL model checking, <http://spinroot.com/>
22. Wang, T.: Integer Hash Function (2007),
<http://www.cris.com/~Ttwang/tech/inthash.htm>

Verifying Compiler Based Refinement of BluespecTM Specifications Using the SPIN Model Checker

Gaurav Singh and Sandeep K. Shukla

FERMAT Lab, Deptt of Electrical and Computer Engineering, Virginia Tech,
Blacksburg, VA 24061, USA
{gasingh,shukla}@vt.edu

Abstract. The underlying model of computation for PROMELA is based on interacting processes with asynchronous communication, and hence SPIN has been mainly used as a verification engine for concurrent software systems. On the other hand, hardware verification has mostly focused on clock synchronous register-transfer level (RTL) models. As a result, verification tools such as SMV which are based on synchronous state machine models have been used more frequently for hardware verification. However, as levels of abstractions are being raised in hardware design and as high-level synthesis is being promoted for synthesizing RTL, hardware design verification problems are changing in nature. In this paper, we consider a specific high-level hardware description language, namely, Bluespec System Verilog (BSV). The programming model of BSV is based on concurrent guarded actions, which we also call as Concurrent Action Oriented Specification (CAOS). High-level synthesis from BSV models has been shown to produce efficient RTL designs. Given the industry traction of BSV-based high-level synthesis and associated design flow, we consider the following formal verification problems: (i) Given a BSV specification \mathcal{S} of a hardware design, does it satisfy certain temporal properties? (ii) Given a BSV specification \mathcal{S} , and an implementation R synthesized from \mathcal{S} using a BSV-based synthesis tool, does R conform to the behaviors specified by \mathcal{S} ; that is, is R a refinement of \mathcal{S} ? (iii) Given a different implementation R' synthesized from \mathcal{S} using some other BSV-based synthesis tool, is R' a refinement of R as well? In this paper, we show how SPIN Model Checker can be used to solve these three problems related to the verification of BSV-based designs. Using a sample design, we illustrate how our approach can be used for verifying whether the designer intent in the BSV specification is accurately matched by its synthesized hardware implementation.

Keywords: Formal Verification, Hardware Designs, Bluespec System Verilog (BSV), SPIN Model Checker.

1 Introduction

The emphasis in PROMELA, which is the input specification language of SPIN, is on the modeling of process synchronization and coordination [1]. For this

reason, SPIN is mainly targeted for the verification of concurrent software systems (described in terms of interacting processes) [2], rather than the verification of hardware designs. On the other hand, design of concurrent hardware systems maps closely to clock-synchronous concurrency model as is primarily implemented by RTL HDLs (Hardware Description Languages) such as Verilog and VHDL. For describing such systems, synchronous state machines are better suited, making the use of tools like SMV [3] more prevalent for hardware verification.

However, for better handling of increasing design complexities and efficient hardware-software co-design, hardware design flows are changing in nature. Behavioral specifications of hardware designs are now being written in algorithmic style (similar to describing a software system) using C-like or other high-level HDLs at levels of abstraction above register-transfer level (RTL). Such high-level specifications can then be automatically converted into the corresponding RTL descriptions [4] using various high-level synthesis tools. Due to this changing trend, there is a need for new hardware verification techniques that are suitable for such high-level design flows. In this work, we show how SPIN, which is primarily a software verification tool, can be used for the verification of hardware designs generated from a particular type of high-level design specifications.

Recently, a new approach to high-level synthesis from Concurrent Action Oriented Specifications (CAOS) [5,6], as embodied in *Bluespec System Verilog (BSV)* [7,8], has been proposed. This is based on the idea that the functionality of any hardware design can be described in terms of high-level concurrent atomic actions. Note that since BSV semantics is based on guarded atomic actions, throughout this paper we use the term *action* instead of *rule* (as used in actual BSV syntax) in order to avoid any confusion between the two terms.

In BSV, each action consists of two parts - a guard and a body. Guard is a condition associated with an action which should evaluate to *True* for that action to be enabled. Body of an action consists of a group of operations all of which are executed atomically when the action is enabled. For example, a BSV specification of GCD (Greatest Common Divisor) design can be written in terms of two actions *Swap* and *Diff* (Figure 1). g_1 and g_2 are the guards of actions *Swap* and *Diff* respectively (x and y are the registers). The swap of the values in the body of action *Swap* occurs only when g_1 evaluates to *True*. The assignment $y \leftarrow y - x$ in the body of action *Diff* occurs only when g_2 evaluates to *True*.

Such a high-level BSV specification of a hardware design can be automatically converted into RTL code. High-level synthesis from BSV has been shown

$$\begin{aligned} \text{Action Swap : } g_1 &\equiv ((x > y) \ \&\& \ (y \neq 0)) \\ &x \leftarrow y; \quad y \leftarrow x; \\ \\ \text{Action Diff : } g_2 &\equiv ((x \leq y) \ \&\& \ (y \neq 0)) \\ &y \leftarrow y - x; \end{aligned}$$

Fig. 1. BSV Specification for GCD design

to produce hardware designs which are efficient in terms of area and latency [8]. Techniques for generating power efficient hardware designs from such specifications have also been proposed in the past [5,6].

However, not much work has been done in the area of automatic formal verification of hardware designs generated from BSV-based synthesis (which is the main focus of this paper). This can be attributed to the fact that BSV-based high-level synthesis has been recently proposed [7] and is relatively new. With regard to verification, it can be argued that various techniques proposed in the past to formally verify RTL implementations of hardware designs can also be used to verify hardware generated from BSV-based synthesis. However, as more and more functionality is being added to hardware designs, RTL is fast becoming too low level to efficiently handle the complexity of hardware designs, thus making it imperative to investigate techniques for the verification of high-level hardware descriptions earlier in the design cycle. High-level specifications do not contain details irrelevant to design's behavior, and hence formal verification of such specifications can aid in faster verification and architectural exploration leading to increase in the designer's productivity.

For a given BSV specification \mathcal{S} of a design, its RTL implementation schedules its actions for execution in different clock cycles. A simple hardware schedule randomly selects just one action for execution in each clock cycle. Such a *sequential execution semantics* contains all possible hardware behaviors corresponding to the specification \mathcal{S} but is undesirable from latency (total number of clock cycles elapsed until the execution halts) point of view. Thus, in RTL implementations generated using \mathcal{S} , multiple actions are scheduled for execution in a single clock cycle provided the set of behaviors shown by any such implementation is a subset of the set of behaviors of specification \mathcal{S} . In this sense, checking the schedule generated by such implementations against the behaviors of the specification \mathcal{S} as well as figuring out relationships among the behaviors shown by two different implementations of a design are important verification issues.

A hardware design can be represented using a corresponding automaton \mathcal{A} which encodes all the behaviors of the design in terms of its different states and transitions among those states. The language of automaton \mathcal{A} contains all such behaviors of the design and can be denoted as $\mathcal{L}(\mathcal{A})$. Also, essential properties of the behaviors of the design can be expressed as a set of LTL (Linear Temporal Logic) formulae EP written in terms of its states and transitions. Using these notations, important verification problems associated with BSV-based design flow can be defined as follows -

1. Given a BSV-based specification \mathcal{S} , which corresponds to an automaton $\mathcal{A}_{\mathcal{S}}$ based on the *sequential execution semantics*, does $\mathcal{A}_{\mathcal{S}}$ satisfy all the essential properties of EP ?
2. Given a BSV-based specification \mathcal{S} and its implementation R (synthesized using a BSV-based high-level synthesis tool), which corresponds to an automaton $\mathcal{A}_{\mathcal{R}}$, does $\mathcal{A}_{\mathcal{R}}$ conform to the behaviors of \mathcal{S} ; that is, does $\mathcal{L}(\mathcal{A}_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$ hold? In other words, is R a refinement of \mathcal{S} ?

3. Given automata \mathcal{A}_R and $\mathcal{A}_{R'}$ for two different implementations R and R' of specification \mathcal{S} (synthesized using two different BSV-based synthesis tools differing in their scheduling of actions) respectively, is R' a refinement of R ; that is, does $\mathcal{L}(\mathcal{A}_{R'}) \subseteq \mathcal{L}(\mathcal{A}_R)$ hold?

In order to solve these three problems efficiently, there is a need for performing automatic formal verification of BSV-based designs at a level of abstraction above RTL. [9] shows how a BSV-based design can be converted into corresponding PROMELA model for verification of its essential properties using SPIN Model Checker. However, [9] does not present a formal translation of the BSV description of a design into its corresponding PROMELA model. Also, the remaining two problems of refinement verification of a BSV-based design are not addressed in [9].

Thus, the two major contributions of this work are: (1) We extend the work in [9] by formally explaining various algorithms for converting a given action-based BSV specification of a hardware design and its implementations into corresponding process-based PROMELA models for verification of their essential properties using SPIN. (2) More importantly, we also present a technique which uses SPIN for proving strong language-containment results between two different models of a BSV-based design. Note that SPIN does not directly support proofs of language-containment between different but related PROMELA models. In this work, we present a technique which generates an LTL specification in the style of TLA (Temporal Logic of Actions) [10] for a given PROMELA model. Such an LTL specification can then be used for proving stronger language-containment results with respect to other related PROMELA models using SPIN's LTL Property Manager.

Thus, we provide a complete automation flow for solving all three verification problems related to BSV-based hardware design flow. As confirmed by experiments, our approach can be successfully used for quick and early verification of hardware designs generated using BSV-based synthesis. Due to space constraints, throughout the paper we use a particular small example of a *Vending Machine Controller* to illustrate the usefulness of our approach.

This paper is organized as follows. Section 2 presents a formal description of BSV-based high-level synthesis and various scheduling semantics for BSV designs. Correctness requirements for BSV designs are explained in Section 3. Algorithms for generating PROMELA models containing scheduling information corresponding to a BSV specification and its implementations are presented with sample models in Section 4. Section 5 presents algorithms for verification of essential properties of a BSV design and performing language-containment proofs between BSV models, and also discusses some sample experiments. Section 6 concludes the paper with a short discussion. *Finally, a short Appendix containing the references to various algorithms and listings of code is presented at the end of the paper. Due to space constraints, a more detailed Appendix is given in [11].*

2 Background - High-Level Synthesis from BSV

2.1 Hardware Description

Definition 1. A BSV *specification* \mathcal{S} of a hardware design consists of a set $S = \{s_1, s_2, \dots, s_k\}$ of k state elements of the design and a set $A = \{a_1, a_2, \dots, a_n\}$ of n actions of the design.

Definition 2. Each *state element* $s_i \in S$ is of the form (t_i, n_i, in_i) , where t_i , n_i and in_i represent the data-type, name and initial value of state element s_i respectively. The state elements of a BSV design can be in the form of registers, FIFOs or memories.

Definition 3. Each *action* $a_i \in A$ of specification \mathcal{S} consists of two parts - a guard and a body. For example, an action a_i can be of the form,

action $m_i()$ if $g_i(S_g^i)$ { $s_{i1} <= b_{i1}(S_b^i)$; $s_{i2} <= b_{i2}(S_b^i)$; $s_{i3} <= b_{i3}(S_b^i)$; } end;

Here, m_i is the name of action $a_i \in A$.

$S_g^i = \{s: \text{value of } s \in S \text{ is accessed in the guard of } a_i \in A\}$.

$S_b^i = \{s: \text{value of } s \in S \text{ is accessed in the body of } a_i \in A\}$.

Definition 4. g_i denotes the *guard* of action a_i . It is a condition associated with a_i which evaluates to either *True* or *False* depending on current values of elements of S_g^i . Action a_i is said to be **enabled** if g_i evaluates to *True*.

Definition 5. *Body* of action a_i consists of set of assignment statements of the form, $s_{ij} <= b_{ij}(S_b^i)$, computing next state values of the design and, in general, it can be expressed as,

$B_i = \{(s_{ij}, b_{ij}(S_b^i)) : b_{ij}(S_b^i) \text{ computes the next state value for } s_{ij} \in S_u^i \text{ using current values of the elements of } S_b^i\}$

where, $S_u^i = \{s: \text{value of } s \in S \text{ is updated in the body of } a_i \in A\}$.

Thus, B_i denotes the body of action $a_i \in A$. B_i consists of a group of operations all of which are executed atomically if a_i is enabled and selected for execution.

As an example, Listing 1.1 shows the BSV specification of a *Vending Machine Controller (VMC)*. Most of the description of Listing 1.1 is self-explanatory. For better understanding of BSV semantics and associated verification issues, we will explain our verification approach with respect to this design. The BSV specification of Listing 1.1 is composed of three parts-

1. **State elements**, *count* and *moneyBack* (Line 2), that correspond to the registers of the design.
2. **Atomic Actions**, *doDispenseMoney* (Line 4-10) and *doDispenseGum* (Line 11-14), that perform computations and update the state elements of the design when their guards (shown with *if* clause) evaluate to *True*. Action *doDispenseMoney* controls the dispensing of the money whereas *doDispenseGum* controls the dispensing of gum.

Listing 1.1. BSV Specification of Vending Machine Controller

```

1. module mkVending()
2.   Reg(Int(7)) count = 0; Reg(Bool) moneyBack = False;
3.   Wire dispenseMoney, gumControl;
4.   // action that controls dispensing of money
5.   action doDispenseMoney() if(moneyBack)
6.     if(count == 0) moneyBack <= False;
7.     else { let newCount = count - 10; count <= newCount;
8.           dispenseMoney.send();
9.           if(newCount == 0) moneyBack <= False; }
10.  end;
11.  // action that controls dispensing of gum
12.  action doDispenseGum() if(!moneyBack && count >= 50)
13.    count <= count - 50; gumControl.send();
14.  end;
15.  // input-handling methods
16.  method tenCentIn() if(!moneyBack) count <= count + 10; end;
17.  method fiftyCentIn() if(!moneyBack) count <= count + 50; end;
18.  method moneyBackButton() if(!moneyBack) moneyBack <= True; end;
19.  // wires connecting money and gum outputs
20.  method dispenseTenCents() return dispenseMoney; end;
21.  method dispenseGum() return gumControl; end;
22. endmodule;

```

3. **Interface methods**, *tenCentIn*, *fiftyCentIn*, *moneyBackButton*, *dispenseTenCents* and *dispenseGum* (Lines 15-21), that interact with an external module such as a testbench or other hardware design. Interface methods also behave like atomic actions but their execution is controlled by the external module.

Listing 1.1 also contains some combinational wires (Line 3) which are used for controlling the outputs of VMC. Let $A_m \subseteq A$ be the set of actions corresponding to the interface methods of the BSV design. Thus, for the VMC specification, we have,

$$S = \{count, moneyBack\}.$$

$$A = \{doDispenseMoney, doDispenseGum, tenCentIn, fiftyCentIn, moneyBackButton, dispenseTenCents, dispenseGum\}.$$

$$A_m = \{tenCentIn, fiftyCentIn, moneyBackButton, dispenseTenCents, dispenseGum\}.$$

2.2 Synthesis

A given BSV specification \mathcal{S} of a hardware design can be synthesized to generate efficient RTL code, as exemplified by *Bluespec Compiler (BSC)*, which is a commercial high-level synthesis tool based on BSV semantics [78]. Figure 2 shows the translation from such BSV-based description of a design to hardware. As shown in Figure 2, hardware synthesis from atomic actions can be achieved by implementing each *guard* and *body* as a combinational logic and synthesizing a control circuitry for appropriate scheduling and data-selection. BSV-based

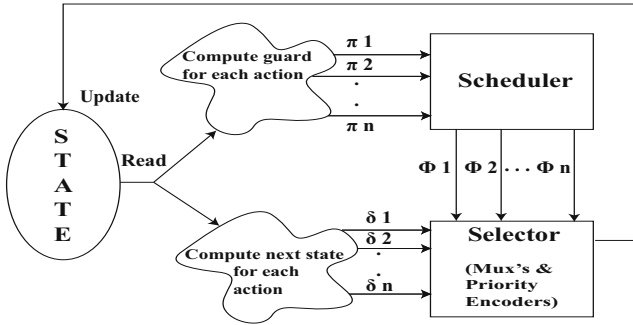


Fig. 2. Synthesis from BSV

synthesis is behaviorally higher in abstraction than RTL-synthesis because it supports automatic handling of concurrency and synchronization issues (such as due to updates on the shared state elements) via atomic actions.

2.3 Scheduling of Actions

Let $d(s_i)$ denote the domain of state element $s_i \in S$ of the design. For hardware designs, usually $d(s_i)$ is the Boolean domain but for BSV specification, which is at a higher level of abstraction, domains such as 32-bit integer, n-bit bit-vector, etc. can be used. For example, if s_i is an 8-bit register, then $d(s_i) = \{0,1\}^8$, which is a set of all possible 8-bit strings of 0s and 1s.

Lets consider a vector $\hat{s} = \langle s_1, s_2, \dots, s_k \rangle$ of the state elements of the design. Note that \hat{s} contains same elements as in set S . In this section, instead of S , we will use \hat{s} in order to appropriately denote the state of the design for defining its behaviors. Let $\sigma_c(\hat{s}) = \langle d_1, d_2, \dots, d_k \rangle \in \prod_{i=1}^k d(s_i)$ denote the state of the design at the end of clock cycle c . Thus, σ is the function which maps the state elements of the design to their respective values at some point during the design execution. A behavior of the design can be defined as a sequence of states (possibly infinite) given as, $\beta = (\sigma_0(\hat{s}), \sigma_1(\hat{s}), \sigma_2(\hat{s}), \dots, \sigma_c(\hat{s}), \dots)$ such that $\sigma_{c+1}(\hat{s})$ results from executing actions in $A_c \subseteq A$ in clock cycle $(c + 1)$ when the design is in state $\sigma_c(\hat{s})$.

Refinement of Behaviors. Consider two behaviors β and β' of the design. Let C_β and $C_{\beta'}$ denote the set of clock cycles of β and β' respectively such that $C_\beta \subset \mathcal{N}$ and $C_{\beta'} \subset \mathcal{N}$, where \mathcal{N} denotes the set of natural numbers. β' is said to be a refinement of β if $\exists r : C_\beta \rightarrow C_{\beta'}$, where r is an injective and monotonic function such that $\forall c \in C_\beta, \sigma_c^\beta(\hat{s}) = \sigma_{r(c)}^{\beta'}(\hat{s})$.

2.3.1 AOA Semantics

During the execution of the design generated using BSV-based synthesis, multiple actions can get enabled in a clock cycle c . In a simple hardware schedule, one such enabled action can be randomly chosen for execution in c , thus proceeding

the execution of the design in a sequential manner. The execution of the design halts when none of its actions are enabled in some clock cycle. We call such a sequential execution semantics where only one action is randomly chosen for execution in each clock cycle as *Any One Action (AOA) Semantics*.

Behavior in AOA Semantics. Behavior of the design in *AOA Semantics* can be given as, $\beta^{AOA} = (\sigma_0(\hat{s}), \sigma_1(\hat{s}), \sigma_2(\hat{s}), \dots, \sigma_c(\hat{s}), \dots)$ such that $\sigma_{c+1}(\hat{s})$ results from executing actions in $A_c \subseteq A$, $|A_c| = 1$, in clock cycle $(c + 1)$ when the design is in state $\sigma_c(\hat{s})$.

2.3.2 Concurrent Semantics

In spite of being behaviorally correct, the sequential execution of just one action in each clock cycle as per *AOA Semantics* is undesirable from latency point of view, especially for designs containing large number of actions. Thus, in a hardware implementation R , synthesized from specification \mathcal{S} , multiple enabled set of actions $A_c \subseteq A$, $|A_c| \geq 1$, can be allowed to execute concurrently in a clock cycle c provided the atomicity of actions in A_c is maintained. This means that, in implementation R , behavior of the design resulting from concurrent execution of actions in A_c should be equivalent to at least one sequential behavior of actions in A_c based on *AOA Semantics*. We call such a scheduling semantics where multiple actions are allowed to execute concurrently in a single hardware clock cycle as *Concurrent Semantics*.

Conflicting Actions. In hardware generated from BSV-based synthesis, maintaining such atomicity among various actions belonging to A_c may lead to complicated combinational circuit. To avoid this, a notion of **conflict** is introduced. An example of a conflict is two actions updating the same hardware register; that is, two actions $a_i, a_j \in A$ can be said to be conflicting with each other if $S_u^i \cap S_u^j \neq \phi$. Other kinds of conflicts can also exist within two actions of the design, thus forbidding the concurrent execution of those actions. In general, two actions are considered to be conflicting with each other if executing their operations in the same clock cycle is undesirable for pragmatic reasons (like long critical paths, write-write conflicts, complicated hardware analysis, etc.). In the synthesized circuit, such restrictions are enforced using small overhead logic.

For example, for VMC, actions *doDispenseGum*, *tenCentIn* and *fiftyCentIn* conflict with each other since they all update register *count*. In case two or more conflicting actions are enabled in the same clock cycle, a notion of **priority** is used to decide which of those actions should be executed in that cycle. A higher priority action is always chosen for execution over all the other lower priority conflicting actions. Let $C(i, j)$ represent a function which returns *True* if two actions $a_i, a_j \in A$ conflict with each other, and *False* otherwise. Then, $C_i = \{ a_j : C(i, j) = True; a_j \in A \text{ has higher priority than } a_i \in A \}$ denotes the set of actions conflicting with a_i which are preferred for execution over a_i .

Sequential Ordering. In order to generate appropriate scheduling and control logic that maintains the atomicity of various actions of a design executing within

the same clock cycle, BSV-based synthesis involves constructing (at compile time) a single sequential ordering S_{order} of all actions belonging to A of specification \mathcal{S} . Lets define a relation $<_s$ among any two actions $a_i, a_j \in A$, $C(i, j) = False$, such that $a_i <_s a_j$ holds if concurrent execution of a_i and a_j in a single clock cycle is equivalent to executing a_i followed by a_j in two consecutive clock cycles each executing just one action. For VMC, actions *doDispenseGum* and *moneyBackButton* can be executed concurrently since their concurrent execution is equivalent to the following sequential ordering- *doDispenseGum*, *moneyBackButton*. This can be denoted as: $doDispenseGum <_s moneyBackButton$.

To construct S_{order} , transitivity property of relation $<_s$ is used and cycles are broken appropriately during the synthesis process. For VMC, one such possible ordering S_{order} of actions is given as - *tenCentIn*, *fiftyCentIn*, *doDispenseMoney*, *doDispenseGum*, *moneyBackButton*. Note that for simplification, we ignore actions *dispenseTenCents* and *dispenseGum* in this ordering since these actions neither perform any computations nor change the state of the design.

Behavior in Concurrent Semantics. For an implementation R , which is synthesized from specification \mathcal{S} based on *Concurrent Semantics*, a behavior of the design can be defined as, $\beta^R = (\sigma_0(\hat{s}), \sigma_1(\hat{s}), \sigma_2(\hat{s}), \dots, \sigma_c(\hat{s}), \dots)$, such that -

- (1) $\sigma_{c+1}(\hat{s})$ results from executing actions belonging to $A_c \subseteq A$ in clock cycle $(c + 1)$ when the design is in state $\sigma_c(\hat{s})$.
- (2) If $|A_c| > 1$, then $C(i, j) = False \forall a_i, a_j \in A_c, i \neq j$; that is, A_c denotes a set of non-conflicting actions.
- (3) β^R corresponds to an equivalent behavior β_{seq}^R generated using the sequential ordering S_{order} of R , with just one action being executed in each clock cycle in β_{seq}^R .

3 Correctness Requirements for BSV Designs

3.1 AOA Semantics

Depending on what actions are selected for execution in different clock cycles, specification \mathcal{S} of the design can consist of multiple behaviors of the form β^{AOA} based on *AOA Semantics*. Let $\mathcal{A}_{\mathcal{S}}$ be the automaton encoding all such possible behaviors of specification \mathcal{S} . The language of automaton $\mathcal{A}_{\mathcal{S}}$ is denoted by $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$, and is said to contain all behaviors of specification \mathcal{S} . Let EP represent the set of all essential properties of the design expressed as LTL (Linear Temporal Logic) formulae.

Correctness Requirement 1 (CR-1). The correctness constraint mandates that for \mathcal{S} to be a valid specification of the hardware design, each behavior β^{AOA} of \mathcal{S} should satisfy all properties in EP ; that is, $\forall \beta^{AOA} \in \mathcal{L}(\mathcal{A}_{\mathcal{S}}), \forall p \in EP, \beta^{AOA}$ should satisfy p .

3.2 Concurrent Semantics

For an implementation R generated from specification \mathcal{S} based on *Concurrent Semantics*, any behavior of the form β^R shown by R corresponds to an equivalent behavior β_{seq}^R generated using S_{order} . Let A_R be an automaton encoding all possible behaviors of the form β_{seq}^R shown by R .

Correctness Requirement 2 (CR-2). The correctness constraint mandates that for R to be a valid implementation of \mathcal{S} , R should be a refinement of \mathcal{S} . In other words, language-containment relation $\mathcal{L}(A_R) \subseteq \mathcal{L}(A_S)$ should hold.

3.2.1 Maximal Concurrent Schedule (MCS)

For latency minimization, maximal set of actions of the design can be chosen for execution in each hardware clock cycle. Such a schedule of a design can be termed as a *Maximal Concurrent Schedule (MCS)* and an implementation R_{MCS} of the design generated based on this schedule contains multiple different behaviors of the form β^R such that $A_c = A_c^M$, where $A_c^M \subseteq A$ is a maximal set of non-conflicting actions scheduled for execution in clock cycle c . As mentioned earlier, each behavior β^R corresponds to an equivalent behavior β_{seq}^R generated using the sequential ordering S_{order} of R_{MCS} .

Let A_R^{MCS} be the automaton encoding all such possible behaviors $\mathcal{L}(A_R^{MCS})$ of a hardware design under the maximal concurrent schedule. The correctness constraint requires that the language-containment relation $\mathcal{L}(A_R^{MCS}) \subseteq \mathcal{L}(A_S)$ should hold; that is, R_{MCS} should be a refinement of specification \mathcal{S} of the design. BSC performs automatic concurrent scheduling of hardware designs, and generates RTL code adhering to one such maximal concurrent refinement which satisfies $\mathcal{L}(A_R^{MCS}) \subseteq \mathcal{L}(A_S)$. (BSV is the CAOS-style input specification language of BSC.)

3.2.2 Alternative Concurrent Schedule (ACS)

As mentioned earlier, BSC schedules maximal set of actions in each clock cycle for latency minimization. However, concurrent execution of large number of actions for improving the latency of a hardware design is usually associated with a corresponding degradation of other attributes of the design, such as its area, peak-power, etc. This might not be desirable for a design having conflicting constraints on its latency and other attributes.

In such cases, instead of executing maximal set of actions A_c^M in clock cycle c , an alternative implementation R_{ACS} of the design needs to be derived which selects only a set of actions $A_c \subseteq A$ for execution in c such that all the constraints of the design are satisfied. Such a schedule of a design can be termed as a *Alternative Concurrent Schedule (ACS)*. Let A_R^{ACS} be the automaton encoding all possible behaviors $\mathcal{L}(A_R^{ACS})$ of a hardware design corresponding to R_{ACS} . Again, the correctness constraint mandates that $\mathcal{L}(A_R^{ACS}) \subseteq \mathcal{L}(A_S)$; that is, any alternative implementation R_{ACS} of a design based on such a schedule is required to be a refinement of specification \mathcal{S} of the design.

3.3 Comparing Two Implementations

Two different implementations of a BSV-based design differ in their scheduling of the actions of the design. In general, an implementation R of a BSV specification \mathcal{S} may either enhance or restrict its set of behaviors as compared to some other implementation R' . However, as mentioned earlier, all behaviors of R and R' should conform to the set of behaviors $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$ of the specification \mathcal{S} , thus satisfying $\mathcal{L}(\mathcal{A}_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$ and $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$.

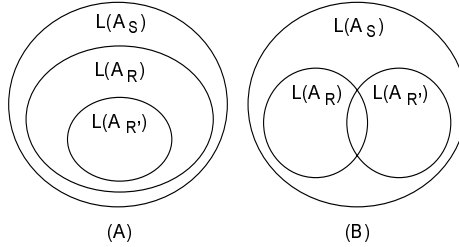


Fig. 3. Language-Containment Relationships

Furthermore, depending on the design requirements, in some cases it might also be desirable to show that R' is a refinement of R ; that is, $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}})$ holds as shown in Figure 3(A). For other cases, relation shown in Figure 3(B) may hold.

Correctness Requirement 3 (CR-3). For two different valid implementations R and R' of specification \mathcal{S} , R' is a refinement of R iff $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}})$.

4 Converting BSV Model to PROMELA Model

Atomicity as well as priority of operations are important concepts in BSV which are also well supported in PROMELA. Moreover, as demonstrated by our approach, different hardware scheduling semantics can also be efficiently modeled in PROMELA using various constructs of the language [1]. Hence, PROMELA can be used for modeling the desired semantics of BSV designs, thus providing a path for verification of such designs using SPIN at a level of abstraction above RTL.

4.1 Generating PROMELA Variables and Processes

PROMELA model \mathcal{M} of a system consists of a set V of variables and a set P of processes (P includes the ‘init’ process) used to describe the system. Given a BSV specification \mathcal{S} of a hardware design, we present Algorithm *GenPROMELA* (Figure 4) to generate PROMELA model \mathcal{M} corresponding to \mathcal{S} . Algorithms *GenVARS*, *GenPROCS* and *GenProcCycle* used by Algorithm *GenPROMELA* are explained in short below. The parameters of various algorithms are described in Appendix with detailed algorithms presented in [11] due to space constraints.

ALGORITHM: *GenPROMELA*. **INPUT:** BSV Specification \mathcal{S} .
OUTPUT: PROMELA model \mathcal{M} .

1. Initialize $V = \phi$, $P = \phi$. (**Note:** V and P are sets of variables and processes of PROMELA model \mathcal{M} respectively.)
 2. Using Algorithm *GenVARS* (Appendix - Figure 7), generate the set of variables V for \mathcal{M} using \mathcal{S} . Algorithm *GenVARS* generates variables corresponding to the state elements of the BSV design as well as other variables needed for modeling the concurrent hardware behavior of the design.
 3. Using Algorithm *GenPROCS* (Appendix - Figure 8), generate the set of processes P for \mathcal{M} using \mathcal{S} and V . For each action of \mathcal{S} , Algorithm *GenPROCS* generates a corresponding process in \mathcal{M} modeling the behavior of the action.
 4. In order to model the hardware behavior in PROMELA, use Algorithm *GenProcCycle* (Appendix - Figure 9) to generate a process pr using \mathcal{S} , V and P . Add pr to P . The execution of this process is used to denote the start of a hardware cycle, thus modeling the synchronous execution of a hardware design.
 5. Using V and P , generate PROMELA ‘init’ process which initializes all variables in V and instantiates all processes in P . Add ‘init’ to P
 6. Output PROMELA model \mathcal{M} whose sets of variables and processes are denoted by V and P respectively.
-

Fig. 4. Algorithm for generating PROMELA model from BSV specification

4.2 Adding Scheduling Information to PROMELA Model

Algorithm *GenPROMELA* (Figure 4) generates sets of variables and processes for PROMELA model \mathcal{M} which corresponds to specification \mathcal{S} . For modeling a particular hardware execution semantics in PROMELA, a new model \mathcal{M}_f needs to be generated by enhancing model \mathcal{M} with the corresponding hardware scheduling information.

AOA Semantics. In order to model a schedule based on *AOA Semantics* in PROMELA, we present Algorithm *AddSeqSched* (Appendix - Figure 10). The algorithm enhances PROMELA model \mathcal{M} generated by Algorithm *GenPROMELA* such that during the execution of the model, ‘start_of_cycle’ process (whose execution denotes the start of a new hardware clock cycle) is executed after every execution of any other process of the model. The generated model \mathcal{M}_f accurately models the behaviors $\mathcal{L}(\mathcal{A}_S)$ of specification \mathcal{S} as per *AOA Semantics*.

Concurrent Semantics. During the BSV-based synthesis, a sequential ordering S_{order} of the actions of the design is generated to which any concurrent execution of actions will correspond. Given such an ordering S_{order} for an implementation R , we present an Algorithm *AddConcSched* (Appendix - Figure 11), which generates model \mathcal{M}_f by enhancing PROMELA model \mathcal{M} (generated by Algorithm *GenPROMELA* shown in Figure 4) with the scheduling information of implementation R . Note that behaviors of model \mathcal{M}_f correspond to all possible behaviors $\mathcal{L}(\mathcal{A}_R)$ of R .

Algorithm *AddConcSched* is generic in the sense that it can model any particular schedule (MCS as well as ACS) of a design based on *Concurrent Semantics*. For this, it takes ordering S_{order} corresponding to R , and maximum number of actions allowed to execute concurrently in R as inputs. In order to model the hardware behavior, after every execution of ‘start_of_cycle’ process, Algorithm *AddConcSched* checks each process for execution based on S_{order} until maximum number of processes have executed.

4.3 Sample PROMELA Models

Detailed Appendix of [11] shows Listings 1.2 and 1.3 which are generated PROMELA models corresponding to the implementations of VMC specification of Listing 1.1. Listings 1.2 and 1.3 show two PROMELA models - one corresponding to the implementation R_{MCS} which executes maximal set of actions in a single clock cycle, and another corresponding to implementation R_{ACS} based on an alternative schedule which executes only one action as per a sequential ordering.

These models are generated using Algorithm *GenPROMELA* (Figure 4) and Algorithm *AddConcSched* (Appendix - Figure 11). Both the PROMELA models are shown in Listings 1.2 and 1.3 (Appendix of [11]) using appropriate markings for implementation-specific lines of code. These models consist of multiple processes including the PROMELA ‘init’ process and five other processes corresponding to different actions of the VMC specification. Not all processes could fit in Listing 1.2 so the remaining ones are shown in Listing 1.3. The main characteristics of such a conversion process that translates a given BSV model into corresponding PROMELA model are -

1. Each action of a BSV design with its corresponding guard and operations is modeled as a process in PROMELA. Moreover, as shown in Listing 1.2 (Appendix of [11]), set of variables V (corresponding to the state elements of VMC) are declared in the beginning of the PROMELA code. Variables *count_old*, *action_fired* and *one_action_fired* are used for verification purposes. In order to model the atomicity of operations of an action, PROMELA construct ‘atomic’ is used [1]. This avoids the interleaving of operations of various processes, which is consistent with BSV semantics and aids in faster verification. Also, ‘do’, which is a repetition construct in *PROMELA* [1], is used for forwarding the execution of processes as in the real hardware (cycle by cycle).
2. In order to model the hardware behavior, an extra process named ‘start_of_cycle’ is generated as shown in Listing 1.3 (Appendix of [11]). This process denotes the start of a hardware clock cycle and reads inputs, if any, from environment external to the design (like a testbench or another hardware design). For VMC, such external inputs are read in variables *tenCentIn*, *fiftyCentIn* and *moneyBackButton*. These variables are used to signal if processes *IFC_tenCentIn*, *IFC_fiftyCentIn* and *IFC_moneyBackButton* which correspond to interface methods of the VMC specification are executed or not.

3. For processes which do not correspond to interface methods, the execution is dependent on a condition which contains logic related to the guard of the corresponding action of the design, as well as logic based on the conflicts with other higher priority actions. If this condition is *True*, then corresponding process is executed, otherwise next process in the ordering is considered for execution (as controlled by variable *action* and PROMELA's 'unless' construct in Listings 1.2 and 1.3).
4. For implementations based on *Concurrent Semantics*, variable named *action* is used to enforce a particular sequential ordering S_{order} of execution among the processes of the generated PROMELA model such that its behavior maps exactly to the concurrent hardware behavior. This is implemented using Algorithm *AddConcSched* (Appendix - Figure 10).

Maximal Concurrent Schedule (MCS) - As shown in Listings 1.2 and 1.3 (Appendix of 11), in order to model an implementation based on MCS, each execution of a process in PROMELA code assigns a new value to variable *action*. The new value is assigned such that in the next step, next process in S_{order} is checked for execution. Such assignments are shown with lines of code marked as "FOR MAXIMAL CONCURRENT SCHEDULE" in Listing 1.2. In such a model, all the processes of the model are checked for execution in every clock cycle.

Alternative Concurrent Schedule (ACS) - For VMC, an implementation corresponding to MCS will execute actions *doDispenseGum* and *moneyBackButton* concurrently whenever *count* becomes greater than 50 cents, *moneyBack* is *False* and external environment signals the execution of action *moneyBackButton*. However, if the peak-power constraint of the design allows only one action to execute in a single clock cycle, then an alternative implementation of the VMC specification adhering to the peak-power constraint needs to be generated. Listings 1.2 and 1.3 (Appendix of 11) also show generated PROMELA model corresponding to such an implementation of VMC. For that implementation, appropriate assignments to variable *action* are shown with lines of code marked as "FOR ALTERNATIVE CONC SCHEDULE". The value of variable *action* is updated to six in each process, thus signifying the end of a hardware clock cycle after one process executes.

Note. Enforcing a particular sequential ordering (as done in Algorithm *AddConcSched*) suppresses non-determinism in the behavior of the PROMELA model but is needed to model the deterministic hardware behavior. However, note that no such sequential ordering is enforced in the PROMELA model generated by Algorithm *AddSeqSched* (Appendix - Figure 10). In that model, execution of a single process marks the end of a hardware clock cycle, and in the next cycle, a new process is non-deterministically (and not based on a sequential ordering) executed. Thus, such a model will contain all possible behaviors $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$ of a BSV specification \mathcal{S} .

5 Formal Verification Using SPIN

5.1 Verifying Correctness Requirement 1 (CR-1)

Proposition 1. *Given a set EP of essential properties, a BSV specification \mathcal{S} satisfies property $p \in EP$ iff its corresponding PROMELA model \mathcal{M}_f satisfies property p_m , where p_m is equivalent to p and is expressed with respect to \mathcal{M}_f .*

Based on Proposition 1, a BSV specification \mathcal{S} can be verified for *Correctness Requirement 1 (CR-1)* mentioned in Section 3 using Algorithm *VerfCR1* (Figure 5).

ALGORITHM: *VerfCR1*.

INPUT: 1. BSV specification \mathcal{S} , 2. Set EP of Essential Properties.

OUTPUT: Verify if \mathcal{S} meets *Correctness Requirement 1 (CR-1)*?

1. Using Algorithm *GenPROMELA* (Figure 4) and Algorithm *AddSeqSched* (Appendix - Figure 10), generate a PROMELA model \mathcal{M}_f based on *AOA Semantics* using \mathcal{S} .
 2. Initialize $EP_m = \phi$.
 3. For each property $p \in EP$
 - (a) Convert p into p_m such that p_m is an LTL formula expressed with respect to \mathcal{M}_f (using variable set V of \mathcal{M}_f).
 - (b) Add p_m to EP_m .
 4. $\forall p_m \in EP_m$, perform verification of \mathcal{M}_f against p_m using SPIN.
 5. If verification is successful $\forall p_m \in EP_m$, then \mathcal{S} meets *Correctness Requirement 1*.
-

Fig. 5. Algorithm for Verifying Correctness Requirement 1

5.2 Verifying Correctness Requirement 2 (CR-2)

Proposition 2. *Given a BSV specification \mathcal{S} and its implementation R , R is a refinement of \mathcal{S} iff \mathcal{M}_f^R is a refinement of \mathcal{M}_f^S , where \mathcal{M}_f^R and \mathcal{M}_f^S are corresponding PROMELA models of R and \mathcal{S} respectively.*

Based on Proposition 2, an implementation R of a BSV specification \mathcal{S} can be verified for *Correctness Requirement 2* (Section 3) using Algorithm *VerfLangCont* (Figure 6). Algorithm *VerfLangCont* generates PROMELA model \mathcal{M}_f^R for implementation R . It also generates an LTL specification LTL_S encoding all the behaviors $\mathcal{L}(\mathcal{A}_S)$ of specification \mathcal{S} (based on *AOA Semantics*) with respect to the state in \mathcal{M}_f^R . Such LTL specifications are generated in the style of TLA 10 (as an example, see Listing 1.4 which is shown in Appendix of 11 and is explained later in this paper.) because BSV specifications are written in terms of actions and not explicitly in terms of the state of the design. However, note that Algorithm *VerfLangCont* only encodes the safety properties and not the liveness assumptions in the generated LTL specification LTL_S . This is because we are interested in showing that safety properties encoded in LTL_S also hold in implementation R . In other words, R is a refinement of \mathcal{S} .

ALGORITHM: *VerfLangCont*.

INPUT: 1. BSV specification \mathcal{S} , 2. Implementation R of \mathcal{S} .

OUTPUT: Verify if $\mathcal{L}(\mathcal{A}_R) \subseteq \mathcal{L}(\mathcal{A}_S)$ holds or not?

1. Using Algorithm *GenPROMELA* (Figure 4), generate PROMELA model \mathcal{M}^R for implementation R . Let V and P denote the set of variables and processes corresponding to \mathcal{M}^R respectively.
 2. Initialize $LTL_S = \text{True}$.
 3. For each process $pr \in P$, such that $pr \equiv (m, B, g)$ corresponds to an action $a \in A$,
 - (a) Generate a set NSV of all next state values in \mathcal{M}^R that are possible according to the behaviors of \mathcal{S} (*AOA Semantics*) when g becomes *True* in \mathcal{M}^R . (**Note:** Only next state values in \mathcal{M}^R corresponding to the execution of an action in \mathcal{S} needs to be captured. For this, in order to signify the execution of a process in \mathcal{M}^R , value of variable *action_fired* in V can be checked to be *True*.)
 - (b) Generate an LTL property expression LTL_p of the form $LTL_p \equiv ([] (g -> X(\| NSV)))$, where $(\| NSV)$ is *True* only when at least one of the elements of NSV is *True*, and *False* otherwise.
 - (c) $LTL_S = LTL_S \ \&\& \ LTL_p$.
 4. Optimize LTL_S to reduce the number of different LTL expressions while retaining all its behaviors. (**Note:** At this stage, LTL_S contains all possible behaviors of \mathcal{S} with respect to the state in \mathcal{M}^R .)
 5. Using Algorithm *AddConcSched* (Appendix - Figure 11), generate PROMELA model \mathcal{M}_f^R using \mathcal{S} , \mathcal{M}^R and R .
 6. Using SPIN's LTL Property Manager, perform verification of \mathcal{M}_f^R against LTL_S . If verification is successful, then $\mathcal{L}(\mathcal{A}_R) \subseteq \mathcal{L}(\mathcal{A}_S)$ holds, otherwise not.
-

Fig. 6. Algorithm for Proof of Language Containment

Using SPIN's LTL Property Manager, model \mathcal{M}_f^R is then verified against LTL_S for ensuring the language-containment relation $\mathcal{L}(\mathcal{A}_R) \subseteq \mathcal{L}(\mathcal{A}_S)$. This allows using SPIN for proving strong language-containment relationships for BSV designs.

5.3 Verifying Correctness Requirement 3 (CR-3)

Proposition 3. *Given two different implementations R and R' of a BSV specification \mathcal{S} , R' is a refinement of R iff $\mathcal{M}_f^{R'}$ is a refinement of \mathcal{M}_f^R , where $\mathcal{M}_f^{R'}$ and \mathcal{M}_f^R are corresponding PROMELA models of R' and R respectively.*

Based on Proposition 3, Algorithm *VerfLangCont* (Figure 6) can also be used to verify *Correctness Requirement 3* (Section 3). For this, all the steps of the Algorithm *VerfLangCont* remain same except that the inputs to the algorithm in this case will be implementations R and R' (instead of specification \mathcal{S} and R as shown in Figure 6). Consequently, the algorithm will verify PROMELA model of implementation R' against the LTL specification encoding all the behaviors of implementation R with respect to state in R' .

5.4 Sample Experiments

I. We used Algorithm *VerfLangCont* (Figure 6) to successfully verify the language-containment relations $\mathcal{L}(\mathcal{A}_R^{MCS}) \subseteq \mathcal{L}(\mathcal{A}_S)$ and $\mathcal{L}(\mathcal{A}_R^{ACS}) \subseteq \mathcal{L}(\mathcal{A}_S)$ among VMC specification of Listing 1.1 and its implementations R_{MCS} and R_{ACS} , shown in Listings 1.2 and 1.3 (Appendix of [11]). Listing 1.4 (Appendix of [11]) shows the LTL specification LTL_S generated by Algorithm *VerfLangCont*, which corresponds to all behaviors $\mathcal{L}(\mathcal{A}_S)$ of the VMC specification. In Listing 1.4, variables *count_old*, *action_fired* and *one_action_fired* are used for expressing LTL_S with respect to the state in the PROMELA models of Listings 1.2 and 1.3 as follows -

1. *count_old* stores the old value of *count* at the start of every process and is used in LTL_S to compare any updates on the value of *count* (during the execution of the process) with the old value.
2. *action_fired* and *one_action_fired* are used in LTL_S to check the state of the PROMELA model at points (just after a process has executed its atomic block) which map to the state changes during the execution of the BSV design.

II. We also used Algorithm *VerfLangCont* to verify if $\mathcal{L}(\mathcal{A}_R^{MCS}) \subseteq \mathcal{L}(\mathcal{A}_R^{ACS})$ holds for VMC.

Result. Verification done by SPIN proved that $\mathcal{L}(\mathcal{A}_R^{MCS}) \subseteq \mathcal{L}(\mathcal{A}_R^{ACS})$ does not hold for VMC, and pointed out a behavior shown by R_{MCS} which is not shown by R_{ACS} . This behavior corresponds to the case when *count* ≥ 100 holds, *moneyBack* is *False* and external environment signals the execution of action *moneyBackButton* (assuming actions *tenCentIn* and *fiftyCentIn* are not signalled to execute). In such a state, the behaviors of the two implementations R_{MCS} and R_{ACS} differ as follows-

1. R_{MCS} (to which the generated PROMELA model corresponds) executes actions *doDispenseGum* as well as *moneyBackButton* in the first clock cycle. This is followed by the execution of *doDispenseMoney* in the next clock cycle.
2. R_{ACS} (to which the generated LTL specification corresponds) only executes *doDispenseGum* in the first clock cycle. This is because R_{ACS} allows only one action to execute in a single clock cycle, and based on the sequential ordering it just executes action *doDispenseGum*, thus dispensing gum and reducing *count* by 50 cents. In the next clock cycle, *count* ≥ 50 holds and *doDispenseGum* will be again selected for execution (assuming actions *tenCentIn* and *fiftyCentIn* are not signalled to execute). Action *moneyBackButton* is not executed in such a behavior.

Thus, as successfully highlighted by SPIN, $\mathcal{L}(\mathcal{A}_R^{MCS}) \subseteq \mathcal{L}(\mathcal{A}_R^{ACS})$ does not hold for VMC.

III. Furthermore, we used SPIN to prove that $\mathcal{L}(\mathcal{A}_R^{ACS}) \subseteq \mathcal{L}(\mathcal{A}_R^{MCS})$ also does not hold for VMC. This implies that both implementations R_{MCS} and R_{ACS} of VMC conform to its specification but one is not a refinement of other. For some BSV designs, such relationships are acceptable because *Correctness Requirement*

3 (*CR-3*) (Section 3) needs to be satisfied only if required based on the design requirements. For VMC, R_{ACS} and R_{ACS} contain behaviors conforming to the specification and both implementations are acceptable.

These experiments demonstrate that the language-containment based verification approach of Algorithm *VerfLangCont* (Figure 6) can be successfully used to compare behaviors of different implementations of BSV designs.

6 Summary

Verification of hardware designs at a level of abstraction above RTL aids in faster and efficient verification early in the design cycle. BSV-based high-level synthesis converts a BSV specification of a hardware design into its corresponding RTL code. However, verification of BSV designs at levels of abstraction above RTL has not been looked at until now. In this paper, we present a verification approach that provides such a verification path for BSV designs. We propose the conversion of BSV-based hardware designs into corresponding PROMELA models containing implementation-specific scheduling information. Such PROMELA models can then be verified using SPIN for their essential properties. Moreover, for stronger language-containment proofs, we propose a technique that uses SPIN to verify if a particular implementation of the BSV design is a refinement of its specification or some other implementation. We successfully used our verification techniques to check different BSV designs for correctness and language-containment based proofs. Note that for BSV designs consisting of large number of actions, the proposed SPIN-based verification techniques in this paper might not scale well. However, a targeted model checker based on the presented verification techniques will scale better. In this paper, our intent is to conceptually show how a model checker like SPIN can be used to verify BSV-based hardware designs early in the design cycle.

References

1. Holzmann, G.J.: The SPIN Model Checker. Addison Wesley, Reading (2004)
2. Holzmann, G.J.: The model checker SPIN. *Software Engineering* 23(5), 279–295 (1997)
3. SMV, <http://www-cad.eecs.berkeley.edu/~kenmcml/>
4. Raghunathan, A., Jha, N.K., Dey, S.: High-Level Power Analysis And Optimization. Kluwer Academic Publishers, Dordrecht (1998)
5. Singh, G., Shukla, S.K.: Low-Power Hardware Synthesis from TRS-based Specifications. In: International Conference on Formal Methods and Models for Codesign (MEMOCODE 2006) (2006)
6. Singh, G., Schwartz, J.B., Ahuja, S., Shukla, S.K.: Techniques for Power-aware Hardware Synthesis from Concurrent Action Oriented Specifications. *Journal of Low Power Electronics (JOLPE)* 3(2), 156–166 (2007)
7. Hoe, J.C.: Arvind: Hardware Synthesis from Term Rewriting Systems. In: *Proceeding of VLSI 1999*, Lisbon, Portugal (December 1999)

8. Arvind, N.R., Rosenband, D., Dave, N.: High-level synthesis: An Essential Ingredient for Designing Complex ASICs. In: Proceedings of the International Conference on Computer Aided Design (ICCAD 2004), November 2004, pp. 775–782 (2004)
9. Singh, G., Shukla, S.K.: Model Checking Bluespec Specified Hardware Designs. In: Microprocessor Test and Verification (MTV 2007) (2007)
10. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
11. Singh, G., Shukla, S.K.: Verifying Compiler Based Refinement of Bluespec Specifications using the SPIN Model Checker. Technical report 2008-03, Virginia Tech, FERMAT Lab, Blacksburg, VA (April 2008), <http://fermat.ece.vt.edu/Publications/pubs/techrep/techrep0803.pdf>

Appendix - Algorithms and Code Listings

Listings 1.2, 1.3 and 1.4 are available in Appendix of [11].

ALGORITHM: *GenVARS*. **IN:** BSV Specification \mathcal{S} . **OUT:** Set of variables V .

For details of algorithm, see Figure 7 in Appendix of [11].

Fig. 7. Algorithm for generating set of variables of PROMELA Model

ALGORITHM: *GenPROCS*. **INPUT:** 1.BSV Specification \mathcal{S} , 2.Set of Variables V . **OUTPUT:** Set of processes P .

For details of algorithm, see Figure 8 in Appendix of [11].

Fig. 8. Algorithm for generating set of processes of PROMELA Model

ALGORITHM: *GenProcCycle*.
INPUT: 1. BSV Specification \mathcal{S} , 2. Set of Variables V , 3. Set of processes P .
OUTPUT: PROMELA process *start_of_cycle*.

For details of algorithm, see Figure 9 in Appendix of [11].

Fig. 9. Algorithm for generating process denoting start of hardware cycle in PROMELA Model

ALGORITHM: *AddSeqSched*. **INPUT:** 1. BSV Specification \mathcal{S} , 2. PROMELA Model \mathcal{M} without scheduling information.

OUTPUT: PROMELA Model \mathcal{M}_f executing processes based on *AOA Semantics*.

For details of algorithm, see Figure 10 in Appendix of [11].

Fig. 10. Algorithm for modeling AOA Execution Semantics in PROMELA Model

ALGORITHM: *AddConcSched*.

INPUT: 1. BSV Specification \mathcal{S} , 2. PROMELA Model \mathcal{M} without scheduling information, 3. Sequential Ordering S_{order} of an implementation R , 4. Maximum number of actions max allowed to execute concurrently in R .

OUTPUT: PROMELA Model \mathcal{M}_f based on scheduling information of R .

For details of algorithm, see Figure 11 in Appendix of [11].

Fig. 11. Algorithm for modeling Concurrent Execution Semantics in PROMELA Model

Symbolic Context-Bounded Analysis of Multithreaded Java Programs^{*}

Dejvuth Suwimonterabuth, Javier Esparza, and Stefan Schwoon

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany

Abstract. The reachability problem is undecidable for programs with both recursive procedures and multiple threads with shared memory. Approaches to this problem have been the focus of much recent research. One of these is to use context-bounded reachability, i.e. to consider only those runs in which the active thread changes at most k times, where k is fixed. However, to the best of our knowledge, context-bounded reachability has not been implemented in any tool so far, primarily because its worst-case runtime is prohibitively high, i.e. $O(n^k)$, where n is the size of the shared memory. Moreover, existing algorithms for context-bounded reachability do not admit a meaningful symbolic implementation (e.g., using BDDs) to reduce the run-time in practice. In this paper, we propose an improvement that overcomes this problem. We have implemented our approach in the tool jMoped and report on experiments.

1 Introduction

The analysis of procedural multithreaded programs has been intensively studied in the last years. If both recursion and multithreading are allowed, checking assertions such as reachability of program points is undecidable, even for programs whose variables are all of boolean type (see for instance [1]).

In order to cope with this undecidability result, three approaches have been proposed. First, the reachability of program points (and other, more general problems) has been studied and shown to be decidable for several special cases, like no communication between threads (but unrestricted thread creation) [2]; communication through nested locks [3,4]; communication following a transactional policy [5]; or communication following a task-based policy [6]. In a second approach, techniques have been developed that compute an overapproximation of the set of reachable states. In [7,8] it is shown that so called commutative abstractions of the set of reachable states can be effectively computed, while [9,10] describes a CEGAR loop for another class of abstractions; this loop has been implemented in the MAGIC and Spade systems.

This paper lines with the third approach to the problem, namely the computation of underapproximations of the set of reachable states. In [11], Qadeer and Rehof proposed the first nontrivial technique, working for possibly recursive

^{*} Partially supported by the DFG project *Algorithms for Software Model Checking*.

multithreaded programs communicating through global variables. They introduce the notion of *context switch* (transfer of control from one thread to another) and show how to compute, for a fixed k , the set of states reachable by computations with at most k context switches. The algorithm of [11] was extended in [12] to a more general programming model allowing for both global and local variables. In [13] it was also adapted to the analysis of concurrent queue systems.

Given a computation with k context switches, let us define its *trace* as the sequence of valuations of the global variables at which the switches take place. A shortcoming of the algorithms of [11][12] is that they require to explicitly examine each trace one by one. If the global variables have n possible valuations, the number of traces is $O(n^k)$, which seriously limits the applicability of the approach. Recently, Lal et al. have proposed a new algorithm which avoids this problem at the expense of, loosely speaking, computing the reachability relation for a thread instead of only the set of reachable states [14]. To the best of our knowledge, none of the techniques for computing underapproximations presented in [11][12][14] has been implemented yet.

We present an improvement of context-bounded reachability algorithms that no longer requires to consider each trace individually. Our algorithm admits a symbolic implementation, using BDDs, which makes it usable in practice. We have implemented this approach as an extension of the jMoped tool [15]. The implementation can deal with Java code “as is”, without replacing non-native libraries by stubs or manually translating the code into the modelling language of the checker. More specifically, the contributions of the paper are as follows:

- A new algorithm for the computation of the states reachable after a bounded number of context switches, based on *lazy splitting*. The algorithm deals symbolically with sets of traces that do not need to be distinguished, and splits them only when necessary. It addresses the same problem as [14], but with a different approach. The techniques of [11][12] and [14] are compared in some more detail in the conclusions.
- Implementations of the algorithm of [11][12] and the new algorithm in the jMoped tool.
- Optimizations of the algorithm for dealing with Java programs.
- Experimental evaluation on different versions of the Bluetooth driver considered in [9][10], and on the `java.util.Vector` class.

We proceed as follows: Section 2 discusses preliminaries, recalls the details of the context-bounded reachability, and gives an overview of previous work (i.e., [11][12]) and ours. Section 3 presents the novel elements of our algorithm. Section 4 discusses details of our implementation, and Section 5 provides experimental data. We conclude in Section 6.

2 Context-Bounded Reachability

In this section we define the problem we are working on and briefly discuss previous solutions.

2.1 Pushdown Networks

We will consider systems with n parallel processes, where n is a positive integer. Let $[n] = \{1, \dots, n\}$. A *pushdown network* is a triple $\mathcal{N} = (G, \Gamma, (\Delta_i)_{[n]})$, where:

- G is a finite set of *globals*;
- Γ is a finite *stack alphabet*;
- Δ_i , for each $i \in [n]$, is the finite set of *transition rules* for the i -th process (see below for its precise definition).

A *local configuration* of \mathcal{N} is a pair $(g, \alpha) \in G \times \Gamma^*$, i.e. a global and a word over the stack alphabet. A *global configuration* of \mathcal{N} is a tuple $(g, \alpha_1, \dots, \alpha_n)$, where g is a global and α_1 to α_n are words over the stack alphabet. For better distinction, we will denote local configurations by lowercase letters (e.g., c) and global configurations by uppercase letters (e.g., C). Intuitively, the system consists of n processes, each of which have some local storage (i.e., the local storage of the i -th process is the word α_i), and the processes can communicate by reading and manipulating the global storage represented by g . A *pushdown system* is a pushdown network where $n = 1$.

For each $i \in [n]$, Δ_i contains rules of the form $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$, where g, g' are globals, $\gamma \in \Gamma$, and $\alpha \in \Gamma^*$. We define the local transition relation of the i -th process, written \rightarrow_i , as follows: $(g, \gamma\beta) \rightarrow_i (g', \alpha\beta)$ iff $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$ in Δ_i and $\beta \in \Gamma^*$. In other words, each process by itself is a pushdown system; however, its control location is the global store shared by all processes. The transition relation of \mathcal{N} , denoted $\rightarrow_{\mathcal{N}}$ or just \rightarrow for short, is defined as follows: $(g, \alpha_1, \dots, \alpha_i, \dots, \alpha_n) \rightarrow_{\mathcal{N}} (g', \alpha_1, \dots, \alpha'_i, \dots, \alpha_n)$ iff $(g, \alpha_i) \rightarrow_i (g', \alpha'_i)$. By \rightarrow_i^* , $\rightarrow_{\mathcal{N}}^*$, \rightarrow^* , we denote the reflexive and transitive closures of these relations.

2.2 Extensions

The computational model introduced in Section 2.1 is equivalent to the *concurrent pushdown systems* (CPS) originally used by Qadeer and Rehof [11]. In [12] an extension was studied, called APN. There, every thread has an additional local state, and transitions can either be “global” (depending on the global state, the local state of the thread and its stack) or “local” (depending only on the latter two). APN have the same expressive power as CPS, but allow for more refined complexity analysis. Since this aspect plays only a minor role in this paper, we chose to omit local states to simplify the presentation. However, it will be easy to see that our techniques also work for APN, with minor modifications.

Both [11] and [12] also studied extensions of the model with “dynamic” rules, i.e. the ability to fork new threads. We do not present this aspect in any detail here because the context-bounded reachability problem for systems with thread creation can be reduced to the context-bounded reachability problem for systems without (see [11] for details). Our implementation, discussed in Sections 4 and 5, does handle thread creation, and in these cases we denote a “fork” rule by $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha' \rangle \triangleright \alpha''$. In this case, the global changes from g to g' , the active process replaces its top-of-stack symbol γ by α' , and a new thread with stack contents α'' is generated.

2.3 The Reachability Problem for Pushdown Networks

Let $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i \in [n]})$ be pushdown network. We define the following reachability problems:

- Given $i \in [n]$ and an initial local configuration $c_0 = (g_0, \alpha_0)$, the *local (forward) reachability problem* for the i -th process is to compute the set $post_i^*(c_0) = \{c \mid c_0 \rightarrow_i^* c\}$, i.e. the set of local configurations reachable by moves of the i -th process alone.
- Given an initial global configuration $C_0 = (g_0, \alpha_1, \dots, \alpha_n)$, the *(forward) reachability problem* for \mathcal{N} is to compute the set $post^*(C_0) = \{C \mid C_0 \rightarrow^* C\}$ of globally reachable configurations.

Both problems can be extended to sets of configurations in the usual manner. It is well-known that *local reachability* is closed under regularity, i.e. $post_i^*(c_0)$ is a regular set, and the result still holds when c_0 is replaced by a regular set of configurations. Moreover, the local reachability problem can be solved efficiently, in time proportional to $|G|^2 \cdot |\Delta_i|$ [16].

In contrast, the (global) reachability problem is undecidable; more precisely, it is in general undecidable whether $C \in post^*(C_0)$, for a given pair C, C_0 [1]. For this reason, one tries to approximate $post^*(C_0)$. One such approximation, introduced in [11], uses the notion of *context-bounded* computations:

A *context* of \mathcal{N} is a sequence of transitions where all moves are made by a single process. In other words, let us define a (global) reachability relation \rightsquigarrow as follows: $(g, \alpha_1, \dots, \alpha_i, \dots, \alpha_n) \rightsquigarrow (g', \alpha_1, \dots, \alpha'_i, \dots, \alpha_n)$ iff $(g, \alpha_i) \rightarrow_i^* (g', \alpha'_i)$ for some $i \in [n]$. Then \rightsquigarrow is a relation between global configurations reachable from each other in a single context. Correspondingly, we define $\overline{post}_i^*(C_0) = \{C \mid C_0 \rightarrow_i^* C\}$, i.e. $\overline{post}_i^*(C_0)$ is the set of global configurations reachable from C_0 by moves of the i -th process. Moreover, we denote by \rightsquigarrow_j , where $j \geq 0$, the reachability relation within j contexts: \rightsquigarrow_0 is the identity relation on global configurations, and $\rightsquigarrow_{i+1} = \rightsquigarrow_i \circ \rightsquigarrow$. We can now define our central problem:

- Given $k \geq 1$ and an initial global configuration C_0 , the (forward) *context-bounded reachability problem* is to compute the set of configurations reachable in at most k contexts, i.e. the set $post_{\leq k}^*(C_0) = \{C \mid \exists j \leq k: C_0 \rightsquigarrow_j C\}$.

The context-bounded reachability problem is decidable, and its solution can be computed in a time that is essentially proportional to $(n \cdot |G|)^k$ [11][12].

2.4 View Tuples

Let us fix a pushdown network $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i \in [n]})$, a global configuration C_0 , and a context bound $k \geq 1$ for the rest of the section.

The principal problem that one faces when solving the context-bounded reachability problem is to find a data structure for representing the set $post_{\leq k}^*(C_0)$. Note that while the global storage can assume only finitely many values, the number of possible stack contents is infinite, thus finding a suitable data structure for representing sets of global configurations is not straightforward. Here,

we define a data structure that will be helpful to discuss the algorithms in previous work [11,12] and in this paper. The main idea is to represent $\text{post}_{\leq k}^*(C_0)$ by so-called *view tuples*, which represent subsets of $\text{post}_{\leq k}^*(C_0)$.

Definition 1. Let $c = (g, \alpha_1, \dots, \alpha_n)$ be a global configuration. For $i \in [n]$, we call the local configuration (g, α_i) the i -view of c . A view tuple $T = (V_1, \dots, V_n)$ is a collection where V_i is a regular set of local configurations, i.e. a set of i -views for each $i \in [n]$, represented by a finite automaton. T is associated with the following set of configurations:

$$\llbracket T \rrbracket = \{ (g, \alpha_1, \dots, \alpha_n) \mid (g, \alpha_i) \in V_i \text{ for all } i \in [n] \}$$

Not every set of global configurations can be represented as a view tuple. As a running example, let us consider a system with two processes, globals g, g', g'' and stack alphabet a, b . Consider the set $C_0 = \{(g, a, a), (g', b, a), (g'', a, a), (g'', b, b)\}$. Suppose that there is a view tuple $T = (V_1, V_2)$ such that $\llbracket T \rrbracket = C_0$. Then V_1 necessarily contains the pair (g'', a) and V_2 the pair (g'', b) . But then, $\llbracket T \rrbracket$ also contains (g'', a, b) , which is not in C_0 .

More importantly, the sets arising in the context-bounded reachability problem are not representable as view tuples. Continuing the example, suppose that $\Delta_1 = \{(g, a) \leftrightarrow (g', b)\}$ and $\Delta_2 = \{(g, a) \leftrightarrow (g'', a), (g', a) \leftrightarrow (g'', b)\}$. Then $\text{post}_{\leq 2}^*((g, a, a))$ is exactly the set C_0 from above.

In general, therefore, the result of a context-bounded reachability query is only representable as a *union* of view tuples. For instance, C_0 can be partitioned into the sets $C_1 := \{(g, a, a), (g'', a, a)\}$ and $C_2 := \{(g', b, a), (g'', b, b)\}$, which are both representable as view tuples. As we shall see, our work differs from [11,12] in the way we choose the view tuples contained in this union; more to the point, our representation requires (in general) fewer tuples.

2.5 A Meta-Algorithm for Context-Bounded Reachability

In this section we discuss a meta-algorithm to solve the context-bounded reachability problem that unifies the solutions in [11,12] and in this paper. It can be characterised as a worklist algorithm that computes the effect of one context at a time. While the algorithms from [11,12] differ in some details, they can – for the purposes of this paper – be summarised by Algorithm 1.

The entries of the worklist are triples (j, i, T) , where T is a view tuple reachable within j contexts such that i was the process that made the last move ($i = 0$ iff $j = 0$). Initially, the worklist contains just one view tuple representing the initial configuration C_0 . In each iteration, the algorithm picks a view tuple from the worklist and computes the configurations that can be reached through a single additional context. Notice that since we are dealing with regular sets of configurations, this can be done by solving the local reachability problem, see, e.g., [16] or [17] for details. The previously active process, i , is excluded from consideration because it would not add any new information.

The result of the local reachability algorithm is denoted by P , and the principal problem is that P may no longer be representable as a single view tuple.

Input: $\mathcal{N} = (G, \Gamma, (\Delta_i)_{i \in [n]})$, context bound k , initial configuration
 $C_0 = (g_0, \alpha_0^1, \dots, \alpha_0^n)$

Output: the set of reachable global configurations.

```

1 result := ∅;
2 worklist := { (0, 0, ({(g_0, α_0^1)}, ..., {(g_0, α_0^n)})) };
3 while worklist ≠ ∅ do
4   remove (j, i, T) from worklist;
5   add [T] to result;
6   if j < k then
7     forall i' ∈ [n] \ {i} do
8       P = post_{i'}([T]);
9       forall T' ∈ split(P) do
10        add (j + 1, i', T') to worklist;
11 return result;

```

Algorithm 1. Worklist algorithm for context-bounded reachability

The task of the split function in line 9 is to generate a set of view tuples such that $\bigcup_{T' \in \text{split}(P)} [T'] = P$. Our work differs from previous solutions in the way this function is implemented. In [11][12], split works as follows:

$$\text{split}(P) = \{ T_g \mid g \in G \}, \text{ where}$$

$$T_g = P \cap \{ (g, \alpha_1, \dots, \alpha_n) \mid \alpha_i \in \Gamma^*, i \in [n] \}$$

It can be shown that the resulting sets are always view tuples. However, after each context, every worklist entry is split $|G|$ different ways. In the following, we call this approach *eager splitting*. Loosely speaking, eager splitting processes $n^k \cdot |G|^k$ worklist entries. Moreover, after each split, the algorithm will consider every element of G individually, which does not lend itself to a meaningful symbolic implementation (e.g., using efficient set representations such as BDDs). These reasons have been the major obstacle for a practical adoption of these algorithms.

In Section 3, we identify a coarser partition of P that leads to fewer splits, in the hope of making the algorithm faster in practice. We call this approach *lazy splitting*. We also describe how the partition can be computed using BDDs, which gives rise to a symbolic implementation of our algorithm.

3 Lazy Splitting

As discussed in Section 2.5, the algorithm for context-bounded reachability is parametrised by a function that splits the result of a local reachability query into view tuples. In this section, we present the key ingredient for our *lazy splitting* approach, i.e. we show how to (symbolically) compute a coarse partitioning.

To simplify the presentation we consider the case of two processes and assume w.l.o.g. that the second process is active, i.e. given a view tuple $T = (V_1, V_2)$,

the task is to (i) compute the set $\overline{post}_2^*(\llbracket T \rrbracket)$ and (ii) split this set into new view tuples. Recall that a global configuration of a pushdown network with two processes is a tuple $c = (g, \alpha_1, \alpha_2)$, where g is a global and α_i is a local configuration of the i -th process.

Throughout this section we identify a set $X \subseteq X_1 \times \dots \times X_n$ and the predicate $X(x_1, \dots, x_n)$ such that $X(a_1, \dots, a_n)$ holds iff $(a_1, \dots, a_n) \in X$. We liberally mix set and logical notations, and write for instance $A(x) = \exists y: B(x, y)$ to mean $A = \{x \mid \exists y: B(x, y)\}$. Abusing notation, we shall sometimes denote the set $\llbracket T \rrbracket$, where T is a view tuple, simply by T .

We proceed as follows: We first identify a property between globals (called *confluence*) that prevents certain configurations from being included in the same partition. We then show how the confluence relation can be computed symbolically, using BDDs, and finally how partitions can be computed from this relation.

3.1 Confluence and Safe Partitions

Let $R_2(g, \alpha, g', \alpha')$ be the reachability predicate for the second thread, i.e., $R_2(g, \alpha, g', \alpha')$ holds iff $(g, \alpha) \rightarrow_2^* (g', \alpha')$. (As usual, we use unprimed variables for the initial configuration and primed ones for the final configuration.) Using standard logical manipulations we obtain

$$\overline{post}_2^*(T)(g', \alpha_1, \alpha'_2) = \exists g : \left(V_1(g, \alpha_1) \wedge \underbrace{\exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha'_2)}_{=: U_2(g, g', \alpha'_2)} \right).$$

Since g is existentially quantified, $\overline{post}_2^*(T)$ is not always a view tuple. We present a generic approach for representing it as a union of view tuples. The approach is parameterized by a partition of G . We need the following definition.

Definition 2. *Two distinct global values $g_a, g_b \in G$ are confluent if there exist $g', \alpha'_{2a}, \alpha'_{2b}$ such that $U_2(g_a, g', \alpha'_{2a})$ and $U_2(g_b, g', \alpha'_{2b})$ hold. A partition of G is safe if none of its sets contains two confluent values.*

Intuitively, two values in the same set of a safe partition cannot be transformed by the second thread into the same value. For instance, let us return to the example from Section 2.4. If we choose T_0 such that $\llbracket T_0 \rrbracket = \{(g, a, a), (g', b, a)\}$, then $\overline{post}_2^*(T_0) = \mathcal{C}_0$ because $(g, a, a) \rightarrow_2 (g'', a, a)$ and $(g', b, a) \rightarrow_2 (g'', b, b)$. In other words we have $U_2 = \{(g, g'', a), (g', g'', b)\}$. Therefore, g and g' are confluent, and any safe partition must keep these two values apart.

Notice that safe partitions always exist, because the partition that splits G into singletons is always safe. However, finding a coarser safe partition is not necessarily straightforward because U_2 may contain infinitely many tuples, and we will show how to deal with this problem later. For the time being, it suffices to point out that *any* safe partition can be used to represent $\overline{post}_2^*(T)$ as a union of view tuples. Let G_1, \dots, G_m be a safe partition of G . We define sets V'_{11}, \dots, V'_{1m} of 1-views and sets V'_{21}, \dots, V'_{2m} of 2-views as follows:

$$V'_{1j}(g', \alpha_1) = \exists g : V_1(g, \alpha_1) \wedge G_j(g) \wedge \exists \alpha'_2 : U_2(g, g', \alpha'_2) \quad (1)$$

$$V'_{2j}(g', \alpha'_2) = \exists g : U_2(g, g', \alpha'_2) \wedge G_j(g) \quad (2)$$

Intuitively, V'_{1j} contains the local configurations of the first thread for which the second thread can reach the local configuration α'_2 while leaving the global variable in state g' . Therefore, if the first thread initially has (g, α_1) as 1-view, it ends with (g', α_1) : the local configuration α_1 has not changed, but the value of the global variable has. The intuition behind V'_{2j} is similar.

In the example above, we could choose $G_1 = \{g, g''\}$ and $G_2 = \{g'\}$ as a safe partition. Under this assumption the view tuples (V'_{11}, V'_{21}) and (V'_{12}, V'_{22}) as defined above would represent the sets \mathcal{C}_1 and \mathcal{C}_2 from Section 2.4, whose union is indeed equal to \mathcal{C}_0 . The following theorem, whose proof is given in the appendix, states that this works for *every* safe partition.

Theorem 1. *Let $\{V'_{1j}\}_{j \in [m]}$ and $\{V'_{2j}\}_{j \in [m]}$ be defined as in (1) and (2). Then*

$$\overline{post}_2^*(T)(g', \alpha_1, \alpha'_2) = \bigvee_{j=1}^m \left(V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) \right).$$

Proof. (\Rightarrow):

$$\begin{aligned} & \overline{post}_2^*(T)(g', \alpha_1, \alpha'_2) \\ &= \exists g : V_1(g, \alpha_1) \wedge U_2(g, g', \alpha'_2) && \text{(def. of } \overline{post}_2^*(T)) \\ &= \exists g : V_1(g, \alpha_1) \wedge U_2(g, g', \alpha'_2) \wedge \exists j \in [m] : G_j(g) \\ &= \exists j \in [m] : V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) && \text{(logic, def. of } V'_{1j}, V'_{2j}) \\ &\Rightarrow \bigvee_{i=1}^m \left(V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) \right) \end{aligned}$$

(\Leftarrow): Let $(g', \alpha_1, \alpha'_2)$ be a triple satisfying $V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2)$ for some $j \in [m]$. By the definition of V'_{1j} and V'_{2j} there exist g_a, g_b , and α''_2 such that $V_1(g_a, \alpha_1)$, $G_j(g_a)$, $U_2(g_a, g', \alpha''_2)$, $U_2(g_b, g', \alpha'_2)$, and $G_j(g_b)$ hold. So, in particular, g_a and g_b belong to the same set of the partition of G , namely G_j . Furthermore, since $U_2(g_a, g', \alpha''_2)$, $U_2(g_b, g', \alpha'_2)$, it follows from Definition 2 that g_a and g_b are either confluent or equal. Since the partition used to construct $\{V'_{1j}\}_{j \in [m]}$ and $\{V'_{2j}\}_{j \in [m]}$ is safe, we get $g_a = g_b$. So, in particular, $U_2(g_a, g', \alpha'_2)$ holds, which together with $V_1(g_a, \alpha_1)$ implies $\overline{post}_2^*(T)(g', \alpha_1, \alpha'_2)$.

3.2 Computing the Confluence Relation

In this part, we show how to compute the relation $C(x, y)$ of confluent pairs x, y symbolically, using BDDs. By Definition 2, we have

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g', \alpha'_{2a}, \alpha'_{2b} : U_2(g_a, g', \alpha'_{2a}) \wedge U_2(g_b, g', \alpha'_{2b})$$

Notice that the relation U_2 contains stack words and cannot be directly represented by a BDD. However, we first show that U_2 can be represented as a *symbolic* finite automaton and then use the automaton to compute C .

Let us recall the definitions of $U_2(g, g', \alpha'_2)$ and $\overline{post}_2^*(V_2)(g', \alpha'_2)$:

$$\begin{aligned} U_2(g, g', \alpha'_2) &= \exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha'_2) \\ \overline{post}_2^*(V_2)(g', \alpha'_2) &= \exists g, \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha'_2) \end{aligned}$$

We now reduce the computation of U_2 to a local reachability problem w.r.t. a modified pushdown system $(G \times G, \Gamma, \Delta'_2)$. In other words, we change the system by duplicating the globals. Moreover, we have $\langle (\bar{g}, g), \gamma \rangle \leftrightarrow \langle (\bar{g}, g'), \alpha \rangle$ in Δ'_2 iff $\langle g, \gamma \rangle \leftrightarrow \langle g', \alpha \rangle$ in Δ_2 , i.e. the value of the first copy is never changed by any transition rule. The reachability relation for the second thread of the modified system is given by $\bar{R}_2((\bar{g}, g), \alpha_2, (\bar{g}', g'), \alpha'_2) = R_2(g, \alpha_2, g', \alpha'_2) \wedge \bar{g} = \bar{g}'$. Define $\bar{V}_2((\bar{g}, g), \alpha_2) = V_2(g, \alpha_2) \wedge \bar{g} = g$. We have:

$$\begin{aligned} U_2(g, g', \alpha'_2) &= \exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha'_2) \\ &= \exists \alpha_2 : \bar{V}_2((g, g), \alpha_2) \wedge \bar{R}_2((g, g), \alpha_2, (g, g'), \alpha'_2) \\ &= \exists \bar{g}, \bar{g}, \alpha_2 : \bar{V}_2((\bar{g}, \bar{g}), \alpha_2) \wedge \bar{R}_2((\bar{g}, \bar{g}), \alpha_2, (g, g'), \alpha'_2) \\ &= \text{post}_2^*(\bar{V}_2)((g, g'), \alpha'_2) \end{aligned}$$

Recall that if $T = (V_1, V_2)$ is a view tuple, then V_1 and V_2 (and \bar{V}_2) are regular sets, representable by *symbolic* finite automata [17]. Moreover, [17] shows how to transform a symbolic automaton for \bar{V}_2 into one for $U_2 = \text{post}_2^*(\bar{V}_2)$.

We turn to the question how to compute C from the automaton representing U_2 . For this, let us define $U'_2(g, g') := \exists \alpha : U_2(g, g', \alpha)$. Then we have:

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g' : U'_2(g_a, g') \wedge U'_2(g_b, g')$$

The modified pushdown system defined above has $G \times G$ as its set of globals. Thus, the symbolic automaton for U_2 uses $G \times G$ as initial states, and a configuration $((g, g'), \alpha)$ is accepted if, starting at state (g, g') , the automaton can read the input α and end up in a final state. Thus, $U'_2(g, g')$ holds if some input is accepted from the state (g, g') . Since the transitions of a symbolic automaton are represented by BDDs, this can be easily implemented with standard BDD operations.

3.3 Computing a Safe Partition

Given the confluence relation C , our final goal now is to compute a safe partition. Notice that a partition is safe if and only if its sets are cliques of $\neg C$, the complement of C . Since finding a minimal partition into cliques of a given graph is NP-complete, we restrict ourselves to finding a reasonably coarse safe partition in a symbolic manner. The resulting performance of the reachability algorithm is evaluated in Section 5.

Algorithm 2 shows the computation of the partition. Its inputs are the confluence relation C and an arbitrary total order relation L on globals. The algorithm repeatedly computes sets of the partition. The inner loop makes sure that F is a clique of S when exiting the loop. D contains the confluent pairs (x_1, x_2) of $F \times F$ such that x_1 is smaller than x_2 with respect to the order L . If $D = \emptyset$ then F is a clique. Otherwise, for each $(x_1, x_2) \in D$ we remove x_1 . The rôle of L is to guarantee that D is antisymmetric, and so that if x_1 and x_2 are confluent we remove exactly one of them from F .

Notice that the algorithm only uses boolean operations and existential quantification, and can therefore be easily implemented in a BDD library, given BDD representations of L and C . The computation of C was presented in Section 3.2.

Input: Confluence relation $C(x, y)$, total order $L(x, y)$

Output: A safe partition G_1, \dots, G_m of G

```

1  $S(x, y) := \neg C(x, y); j := 0;$ 
2 while  $S \neq \emptyset$  do
3   pick  $(x_0, y_0)$  from  $S$ ;
4    $F(x) := S(x, y_0);$ 
5   while true do
6      $D(x, y) := L(x, y) \wedge F(x) \wedge F(y) \wedge \neg S(x, y);$ 
7     exit if  $D = \emptyset;$ 
8      $F(x) := F(x) \wedge \neg(\exists y : D(x, y))$ 
9    $j := j + 1;$ 
10   $G_j(x) := F(x);$ 
11   $S(x, y) := S(x, y) \wedge \neg F(x) \wedge \neg F(y);$ 

```

Algorithm 2. An algorithm for computing equivalence classes

and a BDD representation for $L \subseteq G \times G$ is trivial to generate, because by assumption the set G is finite, and any total order (e.g. some lexicographical ordering based on the BDD variables) will do.

Finally, equations (1) and (2) only use G_j, V_1, U_2 , which are all representable as BDDs or as symbolic automata, connected by boolean operations. Thus, the new view tuples can be obtained by standard operations on BDDs and automata.

4 Implementation

We implemented the algorithm presented here in jMoped [18,15], an Eclipse plug-in for testing Java programs by means of model-checking techniques. To test a method, users specify the number of *bits* of the program variables and the *heap* size (no knowledge of model-checking techniques is required). jMoped computes the reachable states of the program for all values of the method's arguments within the given range. During the analysis, jMoped displays progress by labelling lines of code with diverse markers, e.g. red markers for assertion violations, green and black markers for reachable and unreachable lines, respectively. Like Java virtual machines we use heaps to simulate Java objects. In particular, the heap size determines the number of objects that can be generated.

4.1 The Model

Internally, jMoped operates on pushdown networks that can also contain rules for thread creation (cf. Section 2.2). jMoped uses a symbolic representation of pushdown networks like in [17], where the stack symbols are tuples (l, γ) such that l is a valuation of local variables and γ a label, i.e. a possible value of the program counter. A network is stored as a list of symbolic rules. For instance, given

labels $\gamma, \gamma', \gamma''$, the set of all rules of the form $\langle g, (l, \gamma) \rangle \mapsto \langle g', (l', \gamma')(l'', \gamma'') \rangle$ is represented by one single rule annotated with a relation R :

$$\gamma \mapsto \gamma' \gamma'' \quad R(g, l, g', l', l'')$$

R specifies which tuples correspond to a rule and is stored as a BDD.

4.2 The Translator

jMoped analyzes which classes are statically reachable from the starting method, and then translates their bytecodes into a pushdown network. The translation process is relatively simple: in most cases a bytecode instruction is mapped into one symbolic rule. However, the BDD for the symbolic rule is not computed beforehand; we only store the information needed to construct it on-the-fly if needed. Constructing BDDs only on demand saves considerable resources.

We maintain four types of variables when analyzing Java bytecodes. Static variables and local variables are modelled by globals and locals, respectively. Heaps are essentially arrays of globals. When an object is created, it occupies some parts of the array where it keeps relevant information such as fields, object type, and lock information. The object itself can be seen as a *pointer* to the array. Objects are never garbage collected in the current implementation.

The Java virtual machine uses an *operand stack* for each method call. This stack can be loaded with constants or values from local variables or fields. Many instructions pop operands from the stack, operate on them, and push the result back. Operand stacks are also used to prepare parameters for method calls and to receive method results. The maximum depth of the operand stack for a given method is determined at compile time and stored in the corresponding class file. jMoped models operand stacks by arrays of locals plus an extra top-of-stack pointer. The array lengths are equal to the maximum depths of the stacks.

We give a flavour of how jMoped works. Figure 1 shows a small Java program, its bytecodes, and a simplified version of the translation into a pushdown network. Bytecode instructions are translated one-to-one into transition rules. The operand stack is simulated by the array s and the top-of-stack pointer sp . Similarly, we use $heap$ and ptr for the heap and the heap pointer. The top-of-stack and heap pointers are initialized to 0 and 1, respectively. The heap at index zero is reserved for null objects. Local variables have identifiers of the form lv_i .

At the beginning of m , the global variable x is initialized to zero in two steps. The constant 0 is pushed onto the operand stack, retrieved, and stored in x . Then, a new object of type `Thread` is created, and a reference to the object is pushed onto the operand stack. jMoped simulates this behaviour by pushing the current value of the heap pointer and updating it to a next (empty) heap element. The heap at ptr is also set to the object type of `Thread`, which is 1 in this example. We update the heap pointer based on sizes of objects. Every object needs one heap element for each field plus an extra heap element for storing its type. The object for `Thread` has size two (see later), and so the pointer gets updated by two. The instruction `dup` duplicates the top element of the operand

<code>class C {</code>	<code>0: iconst_0</code>
<code> static int x;</code>	<code>1: putstatic C.x</code>
<code> static void m() {</code>	<code>4: new Thread</code>
<code>x = 0;</code>	<code>7: dup</code>
<code> new Thread(new Runnable() {</code>	<code>8: new C\$1</code>
<code> public void run() {</code>	<code>11: dup</code>
<code> // New thread works</code>	<code>12: invokespecial C\$1.<init></code>
<code> }).start();</code>	<code>15: invokespecial Thread.<init></code>
<code> // Main thread works</code>	<code>18: invokevirtual Thread.start</code>
<code>}</code>	<code>21: ...</code>
<code>}</code>	<code>e: return</code>

$m_0 \hookrightarrow m_1$	$(s[sp]' = 0 \wedge sp' = sp + 1)$
$m_1 \hookrightarrow m_4$	$(x' = s[sp - 1] \wedge sp' = sp - 1)$
$m_4 \hookrightarrow m_7$	$(s[sp]' = ptr \wedge sp' = sp + 1 \wedge heap[ptr]' = 1 \wedge ptr' = ptr + 2)$
$m_7 \hookrightarrow m_8$	$(s[sp]' = s[sp - 1] \wedge sp' = sp + 1)$
$m_8 \hookrightarrow m_{11}$	$(s[sp]' = ptr \wedge sp' = sp + 1 \wedge heap[ptr]' = 2 \wedge ptr' = ptr + 1)$
$m_{11} \hookrightarrow m_{12}$	$(s[sp]' = s[sp - 1] \wedge sp' = sp + 1)$
$m_{12} \hookrightarrow c_0 \ m_{15}$	$(lv'_0 = s[sp - 1] \wedge sp'' = sp - 1)$
$m_{15} \hookrightarrow t_0 \ m_{18}$	$(lv'_0 = s[sp - 2] \wedge lv'_1 = s[sp - 1] \wedge sp'' = sp - 2)$
$m_{18} \hookrightarrow m_{21} \triangleright r_0$	$(heap[heap[s[sp - 1] + 1]] = 2 \wedge lv''_0 = s[sp - 1] \wedge sp' = sp - 1)$
	\dots
$m_e \hookrightarrow \epsilon$	\dots

Fig. 1. A small Java programs, its bytecodes, and a corresponding pushdown network

stack. At offset 8, an object of type `C$1` is allocated. Class `C$1` is an inner class of `C` which implements the interface `Runnable`. `C$1` specifies the method `run` which will be executed when the thread starts. Note that `C$1` has type 2 and size 1.

Two initialization methods are called at offsets 12 and 15 for `C$1` and `Thread`, respectively. The corresponding translation also shows how arguments are passed. A reference to `C$1` (resp. to `Thread`) is passed to lv_0 when initializing `C$1` (resp. `Thread`). However, for `Thread` a reference to `C$1` is also passed as the second argument, and a reference to `C$1` is stored as its only field (not shown). Recall that `Thread` has size 2 for the purpose of storing an object reference which implements `Runnable` interface. This information is used later on at offset 18. There, we fork a new thread r_0 if the only field of the thread specified by the top element of the operand stack has type 2. Also, a reference to `C$1` is passed to the new thread. Note that we need information about object types to start the right thread. The same technique is used in the case of virtual method calls.

jMoped translates *all* Java bytecode instructions. Calls to Java libraries are not replaced by stubs, since the bytecodes of the library are available. Notice however that some classes contain native code, and for those stubs are necessary.

5 Experiments

All experiments were performed on an AMD 3 GHz machine with 64 GB memory.

5.1 java.util.Vector Class

In this experiment we consider class `java.util.Vector` from the Java library. The `Vector` class implements a growable array of objects. In [19], a race condition in a constructor of `Vector` was reported. The following test method illustrates the situation where the race condition can occur.

```
static void test(Integer x) {
    final Vector<Integer> v1 = new Vector<Integer>();
    v1.add(x);
    new Thread(new Runnable() {
        public void run() { v1.removeAllElements(); }
    }).start();
    Vector<Integer> v2 = new Vector<Integer>(v1);
    assert(v2.isEmpty() || v2.elementAt(0) == x);
}
```

The method creates two vectors. First an empty vector `v1` is created, and then an integer `x` is added to it as its first element. After that, a new thread is forked, which removes all elements from `v1` (only `x` in this case). In parallel, the first thread creates a copy `v2` of `v1`. Intuitively, only two cases are possible: if the elements of `v1` are removed before `v2` is created, then `v2` is empty; if `v2` is created before the elements of `v1` are removed, then the first element of `v2` is equal to `x`. The last line of code asserts this property.

However, in Java 5.0, the constructor of `v2` is not atomic, and as a result the assertion can be violated. `jMoped` detects this bug. The first half of Table 1 shows the time until the bug is found, the numbers of BDD nodes required, and the numbers of view tuples inspected in several experiments. In all experiments the bit size of all variables except `x` is set to 8, the heap size to 50 blocks, and the context bound to 3. The experiments differ on the size of `x` (1 to 8 bits), and on the splitting mode (eager or lazy).

In the current version of Java (version 6.0), the bug has been fixed. We reran all experiments with Java 6.0 and verified that, within the given bounds, the assertion is not violated. The second half of Table 1 presents the results.

The behaviour of the program is independent of the value of `x`. The lazy approach benefits from this, and does not split at all when switching contexts. Therefore, the running time remains essentially constant when the number of bits of `x` increases. On the other hand, the time for eager splitting increases exponentially. However, the eager approach is faster and requires fewer BDD nodes when `x` is small. One of the reasons is that the lazy approach requires an extra copy of globals for keeping relations between current values of globals and initial values when the thread is awakened, which results in bigger BDDs.

One could argue that, since the Java 5.0 bug is already detected when `x` has 1 bit, the lazy approach does not give any advantage in this case. For Java 6.0, however, the analysis of larger ranges provides more confidence in the correctness of the code, and here the lazy approach clearly outperforms eager splitting.

Table 1. Experimental results: `java.util.Vector` class

Sizes of x (bits)			1	2	3	4	5	6	7	8
Java 5.0	Eager	Time (s)	9.3	10.8	16.9	31.1	67.9	117.8	225.7	457.9
		Nodes ($\times 10^6$)	0.4	0.5	0.8	1.4	2.5	5.2	9.0	18.1
		View tuples	48	87	167	327	648	1348	2567	5126
	Lazy	Time (s)	19.7	17.7	19.6	17.5	17.2	18.9	16.7	18.8
		Nodes ($\times 10^6$)	1.2	1.2	1.2	1.3	1.2	1.3	1.3	1.3
		View tuples	3	3	3	3	3	3	3	3
Java 6.0	Eager	Time (s)	15.1	18.6	37.5	64.3	147.7	301.7	642.0	1732.0
		Nodes ($\times 10^6$)	0.4	0.7	1.1	2.0	3.7	7.1	13.9	27.9
		View tuples	105	209	417	833	1655	3329	6657	13313
	Lazy	Time (s)	20.9	20.8	19.4	22.3	20.8	18.8	23.4	23.2
		Nodes ($\times 10^6$)	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
		View tuples	3	3	3	3	3	3	3	3

Finally, we remark that the example is not as small as it seems. While the test method has only a few lines of code, the class `Vector` actually involves around 130 classes which together translate into a pushdown network of 30,000 rules. We are able to automatically translate all classes without any manipulations except `java.lang.System`, where the method `arraycopy` is implemented in native code. We need to manually create a stub in this case.

5.2 Windows NT Bluetooth Driver

In this experiment, we consider three versions of a Windows NT Bluetooth driver [20,9]. Figure 2 shows a Java implementation of the second version. All three versions follow the same idea and differ only in some implementation details. All versions use the following class `Device`, which contains four fields:

```
int pendingIo; boolean stopFlag, stopEvent, stopped;
Device(){ pendingIo = 1; stopFlag = stopEvent = stopped = false; }
```

- `pendingIo` counts the number of threads that are currently executing in the driver. It is initialized to one in the constructor, increased by one when a new thread enters the driver, and decreased by one when a thread leaves.
- `stopFlag` becomes true when a thread tries to stop the driver.
- `stopEvent` models a stopping event, fired when `pendingIo` becomes zero. The field is initialized to false and set to true when the event happens.
- `stopped` is introduced only to check a safety property. Initially false, it is set to true when the driver is successfully stopped.

The drivers has two types of threads, *stoppers* and *adders*. A stopper calls `stop` to halt the driver. It first sets `stopFlag` to true before decrementing `pendingIo` via a call to `dec`. The method `dec` fires the stopping event when `pendingIo` is zero. An adder calls the method `add` to perform I/O in the driver. It calls the method `inc` to increment `pendingIo`; `inc` returns a successful status if `stopFlag`

```

static void add(Device d) {
    int status = inc(d);
    if (status > 0) {
        assert(!d.stopped);
        // Performs I/O
    }
    dec(d);
}
static void stop(Device d) {
    d.stopFlag = true;
    dec(d);
    while (!d.stopEvent) {}
    d.stopped = true;
}
static int inc(Device d) {
    int status;
    synchronized(d) {
        d.pendingIo++;
    }
    if (d.stopFlag) {
        dec(d);
        status = -1;
    } else status = 1;
    return status;
}
static void dec(Device d) {
    int pio;
    synchronized (d) {
        d.pendingIo--;
        pio = d.pendingIo;
    }
    if (pio == 0)
        d.stopEvent = true;
}

```

Fig. 2. Version 2 of Bluetooth driver

is not yet set. It then asserts that `stopped` is false before start performing I/O in the driver. The adder decrements `pendingIo` before exiting.

In the first version of the driver, the method `inc` was implemented differently:

```

private int inc(Device d) {
    if (d.stopFlag) return -1;
    synchronized (d) { d.pendingIo++; }
    return 0;
}

```

Moreover, the if-statement in `add` reads `if (status == 0)`. [20] reports a race condition for this version, which occurs when the adder first runs until it checks the value of `stopFlag`. Then, the stopper thread runs until the end, where it successfully stops the driver. When the context switches back to the adder, it returns from `inc` with status zero and finds out that the assertion is violated.

In [9] a bug in the second version of the driver was reported. The bug only occurs in the presence of at least two adders, and four context switches are required to unveil it: (i) The first adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The stopper sets `stopFlag` to true, decreases `pendingIo` back to 1, and waits for the stopping event. (iii) The second adder increases `pendingIo` to 2. However, since `stopFlag` is already set it decreases `pendingIo` back to 1 again. It returns from `inc` with status `-1`, which makes `pendingIo` become 0 and fires the stopping event. (iv) The stopper acknowledges the stopping event and sets `stopped` to true. (v) The first adder violates the assertion. Note that the bug can also be found in a slightly different manner where the second adder starts before the stopper.

Table 2. Experimental results: Bluetooth drivers (left) and binary search trees (right)

	Version 1		Version 2		Version 3	
	Eager	Lazy	Eager	Lazy	Eager	Lazy
Time (s)	1.1	1.3	51.7	36.0	11.9	6.0
Nodes ($\times 10^3$)	46	88	720	1851	195	518
View tuples	21	4	1460	154	234	16
Contexts	3		5		4	

Threads, Contexts	Time(s)
1 + 1, 3	3.8
1 + 1, 4	8.3
2 + 1, 4	127.1
2 + 1, 5	712.3
2 + 1, 6	5528.2
2 + 2, 5	6488.0
2 + 2, 6	timeout

The third version moves `dec(d)` inside the if-block in the method `add`. This eliminates the bug for the case with two adders and one stopper. However, jMoped found another assertion violation for one adder and two stoppers. We believe that this has not been previously reported, although it is less subtle than the previous bugs, requiring three context switches: (i) The adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The first stopper decreases `pendingIo` to 1. (iii) The second stopper decreases `pendingIo` to 0 and sets `stopped` to true. (iv) The adder violates the assertion.

Table 2 reports experimental results on these three versions. Notice that the lazy approach always involves fewer view tuples. This becomes more obvious when the number of contexts grows. We argue that by splitting lazily we can palliate explosions in the context-bounded reachability problem.

5.3 Binary Search Trees

We briefly give an intuition on the scalability of our approach by considering a binary search tree implementation [21] that supports concurrent manipulations on trees. Unlike the previous two experiments, this algorithm is recursive. There are two types of threads, *inserter* and *searcher*. An inserter puts a node into the tree while a searcher looks for a node with a given value. We consider the situation where inserters insert non-deterministic values into the tree and searchers search for the same values. We run jMoped with different numbers of inserters and searchers, and generate all reachable configurations within given contexts.

Table 2 gives the running times. The numbers of threads are in the form $x + y$, where x and y are the numbers of inserters and searchers, respectively. The analysis took more than three hours in the case of $2 + 2, 6$.

6 Conclusions

We have reported on (to the best of our knowledge) the first implementation of the context-bounded technique of Qadeer and Rehof [11]. The implementation extends the jMoped tool, and allows to deal directly with Java code, mostly without having to manually manipulate it or replace Java libraries by stubs.

The algorithm for context-bounded reachability as presented in [11] explicitly deals with each possible trace of the system within the context bound. Therefore, if the number of traces is exponential, then the algorithm *necessarily* takes exponential time. We have presented a symbolic technique, lazy splitting, to palliate this problem. Loosely speaking, the technique tries to symbolically examine all traces in the same computation, and splits the set of traces only when necessary.

We have tested our implementation on a number of examples. Our best result so far is the fact that we can find the bug in the `Vector` Java class reported in [19] without any need for manual manipulation or unsound steps. The program is compiled including all Java libraries, and the Java bytecode is automatically translated into our formal model, without manual supervision.

Lal et al. have proposed a new algorithm which does not require to explicitly examine all possible traces of the system [14]. The main idea is to compute for each thread a regular transducer accepting the reachability relation of the thread. Even though this leads to an attractive fully symbolic solution to the problem, its performance in practice still needs investigation. Even for finite-state systems, experiments show that the symbolic computation of the reachability relation by means of iterative squaring is substantially more expensive than the computation of the set of reachable states (see for instance [22]). While in the case of multithreaded programs the advantage of a fully symbolic procedure may compensate for the overhead of computing the reachability relation, this remains to be seen. We have not followed this path because the algorithm for the computation of the reachable states is the core of jMoped, and the result of many optimizations, and so naturally we wished to reuse it.

On a more abstract level, the idea of [14] is that the effect of running a context on a particular thread can be expressed by a summary. The idea of reusing summaries could also be useful in our setting: as a side effect, the pushdown reachability algorithm implementing the $post_i^*$ function computes (partial) procedure summaries. Summaries are largely independent of the context for which they were computed and could potentially be re-used many times during other calls to $post_i^*$. We did not yet use this trick in our implementation.

Acknowledgements. The authors thank Tomáš Brázdil for his helpful comments and Tayssir Touili and Mihaela Sighireanu for pointing us to some examples.

References

1. Ramalingam, G.: Context-sensitive synchronisation-sensitive analysis is undecidable. *ACM Trans. Programming Languages and Systems* 22, 416–430 (2000)
2. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) *CONCUR 2005*. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
3. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
4. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: *Proc. POPL*, pp. 303–314. ACM, New York (2007)

5. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Proc. POPL, pp. 245–255. ACM, New York (2004)
6. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
7. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: Proc. POPL, pp. 62–73. ACM Press, New York (2003)
8. Bouajjani, A., Esparza, J., Touili, T.: Reachability analysis of synchronized PA-systems. In: Proc. Infinity (2004)
9. Chaki, S., Clarke, E.M., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
10. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of multithreaded dynamic and recursive programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 254–257. Springer, Heidelberg (2007)
11. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
12. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
13. La Torre, S., Madhudson, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Proc. TACAS. LNCS, vol. 4963, pp. 299–314 (2008)
14. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Proc. TACAS. LNCS, vol. 4963, pp. 282–298 (2008)
15. jMoped: The tool’s website, <http://www7.in.tum.de/tools/jmoped/>
16. 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)
17. 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)
18. Suwimonteerabuth, D., Berger, F., Schwoon, S., Esparza, J.: jMoped: A test environment for Java programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 164–167. Springer, Heidelberg (2007)
19. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. 32(2), 93–110 (2006)
20. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI, pp. 14–24 (2004)
21. Kung, H.T., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Trans. Database Syst. 5(3), 354–382 (1980)
22. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE TCAD 13(4), 401–424 (1994)

Efficient Stateful Dynamic Partial Order Reduction^{*}

Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby

School of Computing, University of Utah
Salt Lake City, UT 84112, USA

Abstract. In applying stateless model checking methods to realistic multithreaded programs, we find that stateless search methods are ineffective in practice, even with dynamic partial order reduction (DPOR) enabled. To solve the inefficiency of stateless runtime model checking, this paper makes two related contributions. The first contribution is a novel and conservative light-weight method for storing abstract states at runtime to help avoid redundant searches. The second contribution is a stateful dynamic partial order reduction algorithm (SDPOR) that avoids a potential unsoundness when DPOR is naively applied in the context of stateful search. Our stateful runtime model checking approach combines light-weight state recording with SDPOR, and strikes a good balance between state recording overheads, on one hand, and the elimination of redundant searches, on the other hand. Our experiments confirm the effectiveness of our approach on several multithreaded benchmarks in C, including some practical programs.

1 Introduction

Despite all the advances in developing new concurrency abstractions, explicit thread programming using thread libraries remains one of the most practical ways of realizing concurrent programs that take advantage of multiple cores. Many high level concurrency abstractions (*e.g.*, software transaction memories) also require the use of threads for their implementation. Unfortunately, it is not easy to write bug-free thread programs [1]. In this paper, we focus on the efficient checking of a given multithreaded program for safety violations over *all possible interleavings* on specific inputs.

Runtime model checking [2,3] is a promising method for bug detection. As model building, extraction, and model maintenance are expensive to carry out for thread programs written in practice, we believe in the importance of developing efficient runtime checking methods, as pioneered in [2]. However, even when running under specific inputs, the number of interleavings of a concurrent program can grow astronomically due to their internal concurrency.

Much of the interleaving explosion that occurs in practice during stateless runtime model checking can be attributed to redundant searches from already

^{*} Supported in part by NSF award CNS00509379, Microsoft HPC Institute Program, and SRC Contract 2005-TJ-1318.

```

const int N = 64;
int d = 0;

    thread1:                                thread2:
local int i = 0;                            local int j = 0;
L0: while (i < N){                          M0:  while (j < N){
L1:  atomic{                                M1:  atomic{
      d = d + i;                             M2:    d = d - j;
      assert(d % 5 != 4)                    M3:    assert(d % 5 != 4);
    }                                       M4:  }
L2:  i = i + 5;                            M5:  }
L3: }

```

Fig. 1. A simple example for illustrating the idea

visited states. The example in Figure 1 illustrates this problem. This program has two threads that, in their own `atomic` blocks that are nested within loops, write to a shared variable `d`. Stateless search methods cannot handle this example even with the help of dynamic partial order reduction (DPOR) [4]. This is because: (i) the number of interleavings grows exponentially with respect to the number of loop iterations; (ii) working under stateless DPOR, at any reached state where both threads are enabled, there exists no non-trivial persistent set.

Consequently, with stateless search, many states of this program are re-visited multiple times via different interleavings. For example, given `thread1` at L1 and `thread2` at M1, whether `thread1` executing L1, L2 followed by `thread2` executing M1, M2, or vice versa, the program reaches the same state. Figure 2 illustrates this, where T1 represents `thread1` and T2 represents `thread2` (Note: The dotted states are the intermediate states attained after executing the visible operation of a transition; this detail illustrates a convention introduced in Section 2.) Failing to detect visited states makes the stateless search methods repeatedly explore visited state spaces, which results in very low efficiency.

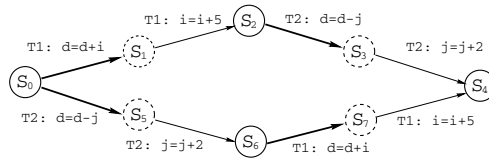


Fig. 2. Two different executions of the program in Figure 1 lead to the same state

While one straightforward solution that avoids redundant searches involves the use of visited states maintained in a hash table, this method, however, is complicated owing to the difficulty of capturing the states of realistic multi-threaded program at runtime. This is especially true for programs written in program languages such as C/C++. Although there have been model checkers such as CMC [5] and Java Pathfinder [6] that have attempted such program

state capture, these approaches are quite heavy-weight. For example, if we take CMC’s approach, we need to capture the state of the kernel space plus the user space. Alternately, if we follow Java PathFinder’s approach, we will have to build a virtual machine for C/C++ programs. Is there a light-weight approach to recording the states of concurrent programs at runtime? If we have such an approach, how do we combine it with partial order reduction techniques soundly? In this paper, we solve these problems in the context of terminating multithreaded programs. We make the following contributions:

- We propose a novel light-weight scheme for capturing the *local* states of threads. We observe that while capturing the entire state of a realistic program at runtime is difficult and expensive, capturing the *changes* between two successive local states of a thread can be easy and inexpensive. Based on this observation, we abstract local states of threads with IDs, and try to discover the same local state of a thread among different executions by tracking the changes (i.e., “deltas”) between successive local states of threads. While an actual total system state of a thread program with N threads would be a tuple $(g, (l_1, \dots, l_N), (p_1, \dots, p_N))$ where g is the global state, l_k are the actual thread local states and p_k are the actual thread PCs, an *abstract* state would be $(g, (i_1, \dots, i_N), (p_1, \dots, p_N))$ where i_k are *IDs* we assign for thread local states. These IDs are computed in a conservative way based on the sequence of deltas that each thread undergoes, as explained in Section 3.
- We present a stateful dynamic partial order reduction (SDPOR) algorithm, which combines our light-weight runtime state capturing approach with dynamic partial order reduction. By introducing states in dynamic partial order reduction, an obvious soundness problem is not updating the backtrack set along a new path that revisits a state. To solve this problem efficiently, we dynamically construct a *visible operation dependency graph* while performing the search. When a visited state is encountered, we compute the summary of the visited sub-state-space using the visible operation dependency graph. With the summary, we conservatively update the backtrack sets of states and guarantee the soundness of our approach.
- We have implemented SDPOR within our runtime model checker **Inspect** [7], and evaluated our approach on a set of multithreaded C benchmarks. The experiments show that SDPOR is much more effective than the stateless DPOR.

The rest of the paper is organized as follows. We introduce the background definitions in Section 2. In Section 3, we describe how local states can be captured in a light-weight and conservative manner. Section 4 presents how the DPOR algorithm can be adapted, with the stateful search. Sections 5 and 6 then present the implementation details and the experimental results. At the end, we discuss related work and conclude the paper.

2 Background Definitions

In this section, we define the notations we employ in the rest of the paper, following the style of [4]. We consider a terminating multithreaded program

with a fixed number of sequential threads as a state transition system. We use $Tid = \{1, \dots, n\}$ to denote the set of thread identities. Threads communicate with each other via *global* objects which are visible to all threads. The operations on global objects are called *visible* operations, while thread local variable updates and PC updates are *invisible* operations. The total system state (S), the program counters of the threads (PCs), and the local states of threads ($Locals$) are now defined:

$$\begin{aligned} S &\subseteq Global \times Locals \times PCs \\ PCs &= Tid \rightarrow PC \\ Locals &= Tid \rightarrow Local \end{aligned}$$

Here, *Global* is the state of global objects, *Local* the local state of any thread, and *PC* the program counter of any thread. For $s \in S$, we use $g(s) \in Global$ to denote the state of global objects in s , $l(s) \in Locals$ to denote the local state component, and $l_\tau(s) \in Locals(\tau)$ to denote the local state of thread τ in s . For $ls \in Locals$, we write $ls[h := l]$ to denote the map that is identical to ls except that it maps the thread h to the local state l .

A transition $t : S \rightarrow S$ advances the program from one state to a subsequent state. More specifically, it starts with one visible operation, followed by a finite sequence of zero or more invisible operations of the same thread, and ends just before the next visible operation of the same thread. It is possible for a transition t to appear in the execute trace of a multithreaded program multiple times.

We can view a transition t as a composition of the *global* transition t_g and the *local* transition t_l . That is, $t = t_l \circ t_g$ where $t_l, t_g \in S \rightarrow S$. Here, t_g corresponds to the visible operation that the transition t starts with. It updates the state of global objects and the program counter of the thread. t_l corresponds to the finite sequence of invisible operations that follows t_g . It can only affect the local state and the program counter of the thread. For instance, in Figure 2, $T1:d=d+i; i=i+5;$ is a transition. It starts with a visible operation $T1: d=d+i$, which is an atomic operation on updating the global object d . $T1:i=i+5$ is the invisible operation that follows $T1: d=d+i$.

Let \mathcal{T} denote the set of all transitions of a multithreaded program. A transition $t \in \mathcal{T}$ is enabled in a state s if $t(s)$ is defined. Two transitions t and t' are *independent* iff they can neither disable nor enable each other, and swapping their order of execution does not change the combined effect 4. t and t' are *dependent* if they are not independent with each other. We say a transition t *may be co-enabled* with a transition t' if there may exist some state s such that both t and t' are enabled in s .

If a transition t is enabled in a state s and $t(s) = s'$, we use $s \xrightarrow{t} s'$ to mean that s' is the successor of s by executing transition t . We use $tid(t)$ to denote the identity of the thread that executes t . Obviously we have $tid(t) \in Tid$.

The behavior of a multithreaded program P is given by a transition system $M = (S, s_0, \Gamma)$, where s_0 is the initial state, and $\Gamma \subseteq S \times S$ is the transition relation. $(s, s') \in \Gamma$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$.

3 Capturing Local States of Threads

Although the local states of threads are not easy to capture precisely at runtime, we observe that in many cases, the changes δ between successive local states are easy to capture. For example, due to the correlations among global objects [8], it is commonly the case that there exist sequences of transitions in which each transition has only the visible operation component, with the invisible operation component being absent. In this case, the local states of threads do not change. It is also common that the changes of local states only involve several local objects and are easy to capture. As an example, in the program of Figure 1, the local state change of `thread1` between two successive executions of the atomic statement labeled L1 only involves the local object `i`. Likewise, the local state change of `thread2` between two successive executions of the atomic block at M1 only involves the local object `j`. This motivates us to capture the local states of threads by tracking the changes among local states.

We now detail our algorithm for capturing local states of threads at runtime, in the context of a depth first search of the state space of the threads. The key idea of the algorithm is to represent each local state of a thread with an abstract ID, and to *link* these IDs by tracking changes between successive local states of threads. This scheme helps conservatively determine whether local states of threads are repeating across different executions.

Let *LocalId* denote the set of local state IDs (natural numbers). We define the *abstract state* of a multithreaded program formally as follows:

$$\begin{aligned} S_a &\subseteq Global \times Locals_a \times PCs \\ Locals_a &= Tid \rightarrow LocalId \\ LocalId &\subseteq \mathbb{N} \end{aligned}$$

With the local state IDs, a multithreaded program can be represented as a transition system $M_a = (S_a, s_{0_a}, \Gamma_a)$, where s_{0_a} is the initial state of the program, and $\Gamma_a \subseteq S_a \times S_a$ is the transition relation. Note that because of our conservative state maintenance scheme which we present later in this section, there could be more than one abstract state associated with a real state.

Let s_a be an abstract state in S_a . When the context is clear, we still use $g(s_a) \in Global$ to denote the global state of s_a , and use $ls(s_a) \in Locals_a$ to denote the local states identities. We use $lid_\tau(s_a) \in LocalId$ to denote the assigned local state identity of thread τ . For $ls_a \in Locals_a$, we write $ls_a[\tau := x]$ to denote that the map that is identical to ls_a except that it maps the thread τ to the local state identity x .

As the state of global objects are in general easy to capture, we do not abstract the states of global objects. Let $s_a \in S_a$ be an abstract state and s be its corresponding state in S . We have $g(s_a) = g(s)$. Similarly, we also have $s_a.PCs = s.PCs$.

Let $s, s' \in S$ be two states, and t be a transition such that $s \xrightarrow{t} s'$. Let $\tau = tid(t)$. We define the changes of the local state of thread τ between s and s' as $\delta_\tau = l_\tau(s') \setminus l_\tau(s)$. We use δ_ε to represent that the local state does not change for a thread. That is, for the thread τ in the above, $l_\tau(s') = l_\tau(s)$. Also, we write

δ_{\perp} to denote that the local state changes are unknown. δ_{\perp} is used when it is hard to capture the local state changes, e.g. when the transition t_i involves calls to library routines, etc. We use Δ to denote the set of all possible local state changes (δ s) for all threads in the program.

In order to detect that the same local state of thread is appearing in different executions of the multithreaded program, we maintain a *local state hash table* for each thread of the program. The local state hash table records the IDs of the local states that have been visited, as well as the changes between two successive local states. In more detail, for each thread τ , we have a local state hash table L_{τ} to store the IDs of the visited local states of τ . $L_{\tau} : LocalId \times \Delta \rightarrow LocalId$ is a mapping from a local state IDs plus the change to a local state, to a potentially new local state ID. However, if L_{τ} already contains the domain point, then the local state ID already in the hash table is returned. We use L to denote the set of local state hash tables for all threads.

Our basic search algorithm with abstract state recording is presented in Figures 3 and 4. The final SDPOR algorithm in Section 4 will build on this algorithm. Figure 3 shows DFS, a recursive procedure for depth-first search of the state space. DFS calls NEXTLOCAL of Figure 4 to compute the local state IDs of a thread. The main data structures used are:

- A hash table H to store all program states $s \in S_a$ that have already been visited during the search.
- For each thread τ , we have a local state hash table L_{τ} to store the identities of the visited local states of τ .

DFS of Figure 3 has four parameters: the abstract state hash table H , the local state hash tables L , the current state s , and finally s_a , which is the abstract state of s . Starting from the initial state, DFS recursively explores the successor states of all states encountered during the search, provided that the correspondent abstract state is not in the hash table. For each visited state, DFS stores the correspondent abstract state in the hash table H . Each time we reach a state s' by executing a transition t which is enabled in a state s , we will compute the abstract state of s' (line 7-9 of Figure 3), and recursively call DFS to explore the next level of the state space.

Figure 4 shows the algorithm for computing the local state identity of a thread. In the procedure NEXTLOCAL, we consider four possible cases:

- If the local state change is difficult to capture precisely, we simply return a new local state ID x .
- If the local state does not change (i.e., $\delta_{\tau} = \delta_{\epsilon}$), the same ID is returned.
- If the hash table L_{τ} already has an entry for $(i, \delta_{\tau}) \rightarrow y$, then we return y as the ID.
- Otherwise, we return a new local state ID x , and at the same time add an entry $\langle (i, \delta_{\tau}) \rightarrow x \rangle$ to L_{τ} .

```

1: Initially:  $H$  is empty;  $\forall L_\tau \in L : L_\tau$  is empty;  $\text{DFS}(H, L, s_0, s_{0_a})$ ;
2:  $\text{DFS}(H, L, s, s_a)$  {
3:   if  $(s_a \in H)$  return;
4:   enter  $s_a$  in  $H$ ;
5:   for each transition  $t$  that is enabled in  $s$  {
6:     let  $s' \in S$  such that  $s \xrightarrow{t} s'$ ;
7:     let  $\tau = \text{tid}(t)$ ,  $\delta_\tau = l_\tau(s') \setminus l_\tau(s)$ ;
8:     let  $x = \text{NEXTLOCAL}(L_\tau, \text{lid}_\tau(s_a), \delta_\tau)$ ;
9:     let  $s'_a \in S_a$  s.t.  $g(s'_a) = g(s') \wedge \text{ls}(s'_a) = \text{ls}(s_a)[\tau := x] \wedge s'_a.PCs = s'.PCs$ ;
10:     $\text{DFS}(H, L, s', s'_a)$ ;
11:  }
12: }

```

Fig. 3. Depth-first search with a light-weight state capturing scheme

```

1:  $\text{NEXTLOCAL}(L_\tau, i, \delta_\tau)$  {
2:   let  $x \in \text{LocalId}$  be a unique new local state identity;
3:   if  $(\delta_\tau = \delta_\perp)$  return  $x$ ;
4:   if  $(\delta_\tau = \delta_\varepsilon)$  return  $i$ ;
5:   if  $(\exists y : \langle (i, \delta_\tau) \rightarrow y \rangle \in L_\tau)$  return  $y$ ;
6:   add  $\langle (i, \delta_\tau) \rightarrow x \rangle$  to  $L_\tau$ ;
7:   return  $x$ ;
8: }

```

Fig. 4. Computing the local state identity of a thread

Now with DFS and NEXTLOCAL, we have the following theorem:

Theorem 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. In a depth first search on S following the algorithm of Figure 3, let $s, s' \in S$ be states that can be reached from s_0 , and let $s_a, s'_a \in S_a$ be the abstract states corresponding to s and s' . Then $\forall \tau \in \text{Tid} : \text{lid}_\tau(s_a) = \text{lid}_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$. \square*

The detailed proof is in the appendix. Theorem 1 states that to detect the visited states at runtime, instead of capturing the local states of threads in detail, we can conservatively infer the equality of local states using the local state changes δ . In practice, capturing δ is usually much easier than capturing the whole local state. Therefore, the task of explicitly capturing states at runtime can be greatly simplified. In the next section, we show how to combine our approach of state capturing with dynamic partial order reduction.

4 Stateful Dynamic Partial Order Reduction

4.1 Background

Dynamic partial order reduction [4] has been shown as an effective reduction technique in stateless search. In DPOR, given a state s , the persistent set [9]

of s is not computed immediately after reaching s . Instead, DPOR explores the states that can be reached from s using depth-first search, and adds backtrack information into the backtrack set of s while exploring the sub-space that is reachable from s .

In more detail, let t_i be a transition that is enabled at state s . Suppose the model checker first selects t_i to execute at s . Let t_j be a transition which can be enabled with a depth first search (in one or more steps) from s by executing t_i . Then before t_j is executed, DPOR will check whether t_j and t_i are dependent and can be enabled concurrently, i.e. *co-enabled*. If so, $tid(t_j)$ or the id of the thread which t_j is dependent on will be added to the backtrack set of s if a transition of $tid(t_j)$ is enabled at s . Later, in the process of backtracking, if the state s is found with non-empty backtrack set, DPOR will select one transition t which is enabled at s and $tid(t)$ is in the backtrack set of s , and explore a new branch of the state space by executing t from s ; at the same time, $tid(t)$ will be removed from the backtrack set of s .

For convenience, we use the following notations to represent the notions used in DPOR:

- $s.enabled$ denotes the set of transitions that are enabled at s . We say a thread τ is enabled at s if $\exists t \in s.enabled : tid(t) = \tau$.
- $s.backtrack$ refers to the backtrack set of state s , i.e. the set of threads whose transitions are enabled at s but have not been executed, $s.backtrack \subseteq Tid$.
- $s.done$ denotes the set of threads whose transitions are enabled at s and have been executed from s , $s.done \subseteq Tid$.

As DPOR is a *stateless* depth first search, it also suffers from the redundant exploration of the state space as described in Section 11. In the rest of this section, we show how to adapt dynamic partial order reduction in the context of stateful search, and how to combine the state capturing scheme of Section 3 with the stateful dynamic partial order reduction.

4.2 Stateful DPOR

The Problem: In a depth first search with DPOR, it seems that if visited states can be recognized, DPOR can simply stop the search at the visited states and start backtracking. However, it is not that simple because the transitions to be executed after the visited states may update the backtrack sets of the states in the search stack. Simple backtracking may result in unsoundness. For example, suppose we have two different executions

$$S_1 = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{u-1}} s_u \dots$$

$$S_2 = s_0 \xrightarrow{t'_0} s'_1 \xrightarrow{t'_1} \dots \xrightarrow{t'_{v-1}} s'_v \dots$$

of a program starting from the same state s_0 , and s'_v is a visited state, $s'_v = s_u$, $u, v \geq 0$ (a fact also noted in [10]). Also, suppose that S_2 is explored after S_1 in the depth first search with DPOR. Now, for every transition t which is executed

after s'_v , the backtrack sets of states s_0, s'_1, \dots, s'_v of S_2 may have to be updated. As a result, if we simply stop exploring the state space after s'_v , we may miss exploring a subset of the state space, i.e., this naïve approach is not sound.

Initial Solution: A straightforward way to fix this problem is that when a visited state is encountered, for each state s in the search stack, we update the backtrack set as follows – for all $t \in s.enabled$ where $tid(t) \notin s.done$, add $tid(t)$ into $s.backtrack$. This solves the problem of missing potential backtrack sets. However, it may also have the side effect of introducing too many unnecessary backtrack points which would not be introduced in the stateless DPOR. [\[1\]](#) This side effect may put significant overhead on the stateful approach and make the stateful DPOR run *slower* than the stateless one. Our initial experiments confirmed this conjecture.

Visible Operation Dependency Graph: To avoid unsoundness we employ an efficient mechanism called *visible operation dependency graph*. Let s_v be a visited state encountered in the stateful DPOR. As only the visible operations of transitions determine whether two transitions are dependent or not, our approach is to compute a summary for the state space the element of which can be reachable from s_v . This summary captures all the visible operations that might be executed from s_v in one or more steps. We can use this summary to update the backtrack sets of the states that are in the search stack.

Obviously, computing such a summary for every state is very heavy-weight. However, we observe that with multithreading, the programs are usually designed in such a way that each thread is assigned some specific tasks to get the most benefit out of parallelism. The number of resources that require mutual exclusive accesses, and the number of conditions that threads need to be synchronized are limited, and usually small in number. This implies that *while the number of states of a multithreaded program can be large, the number of visible operations that each thread may execute is limited*.

For instance, for the program of [Figure 1](#), although the number of states can be large, the only visible operation that `thread1` and `thread2` may execute is updating the global object `d`.

Based on this observation, instead of trying to maintain a summary for each state and keep the summaries updated, *we compute the summary dynamically only when a visited state is encountered by looking up the visible operation dependency graph which is constructed dynamically during the search*.

In more detail, let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let \mathcal{T} be the transition set of M . A visible operation dependency graph $G = \langle V, E \rangle$ for M is a directed graph which captures the happen-before relation of visible operations for the traversed state space. Every node $v \in V$ of G is a visible operation. That is, $\forall v \in V : \exists t \in \mathcal{T} : t_g = v$. For each transition sequence $s_1 \xrightarrow{t} s_2 \xrightarrow{t'} s_3$ we encounter during the search, we add a directed edge (t_g, t'_g) into the graph.

¹ The solution in [\[10\]](#) was this, but the method was experimented only in the context of a custom-built model checker on very small MPI program examples.

In a depth-first search, when a visited state s is encountered, all the states that are reachable from s must have been visited because of the depth first search. Hence, all the visible operations that may be executed in states reachable from s must have been executed. Therefore, we can traverse the visible operation dependency graph to find out all the visible operations that may be executed from some transition in $s.enabled$, and use this as a summary to update the backtrack sets of the states which are in the search stack. As the size of the graph is proportional to the number of visible operations that a multithreaded program may execute, this is a light-weight method for computing summaries of the visited states.

SDPOR: Figure 5 presents our stateful dynamic partial order reduction algorithm (SDPOR). The procedure SDPOR takes three parameters: the search stack S , the state hash table H , and the visible operation dependency graph G . Similar to DPOR, given a multithreaded program P , SDPOR first explores an arbitrary interleaving of the program, and thereafter, continues explore alternative interleavings until all relevant interleavings are explored, i.e., when no backtrack points are in the search stack. The differences between SDPOR and DPOR are:

- SDPOR uses a hash table H to record the visited states (line 9 of Figure 5). When a visited state is encountered, SDPOR conservatively updates the backtrack sets for states in the search stack S , and start backtracking (line 5-7 of Figure 5).
- SDPOR uses a visible operation dependency graph G to dynamically learn the happen-before relation of visible operations during the depth-first search (line 19 of Figure 5). G is used to compute the state summary \mathcal{U} when a visited state is encountered (line 5 of Figure 5).

SDPOR uses UPDATEBACKTRACKSETS of Figure 6 to update the backtrack sets for states in the search stack. UPDATEBACKTRACKSETS is the same as that in the stateless DPOR. We present it here for completeness.

Theorem 2. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. For every execution of a transition $s \xrightarrow{t} s'$ of M , if it is explored by the stateless DPOR, it must be explored by SDPOR. \square*

The soundness of SDPOR is guaranteed by Theorem 2. The detailed proof is given in the appendix. This theorem shows that given a multithreaded program, the set of states visited by SDPOR is a superset of the states visited by DPOR. This means that SDPOR is a conservative approach.

Note that DPOR may re-explore the same state space many times, while SDPOR will, whenever abstract states are found in the hash-table, avoid all those re-visits. Therefore, the bag of DPOR visited states usually has size far higher than the bag of states that SDPOR visits. This is the reason that SDPOR can be more efficient than DPOR in checking multithreaded programs. The experiments to be shown in Section 6 confirm that comparing with DPOR, SDPOR is more efficient in checking realistic multithreaded programs.

```

1: Initially:  $S.push(s_0)$ ;  $H$  is empty;  $G$  is empty;

2: SDPOR( $S, H, G$ ) {
3:    $s \leftarrow S.top$ ;
4:   if ( $s \in H$ ) {
5:     let  $\mathcal{U} = \{v \mid \exists t \in s.enabled, v \text{ is reachable in } G \text{ from the node } t_g\}$ ;
6:     for each  $t \in \mathcal{U}$ , UPDATEBACKTRACKSETS( $S, t$ );
7:     return;
8:   }
9:   add  $s$  into  $H$ ;
10:  for each  $t \in s.enabled$ , UPDATEBACKTRACKSETS( $S, t$ );
11:  if ( $\exists$  thread  $\tau, \exists t \in s.enabled, tid(t) = \tau$ ) {
12:     $s.backtrack \leftarrow \{\tau\}$ ;
13:     $s.done \leftarrow \emptyset$ ;
14:    while ( $\exists h \in s.backtrack \setminus s.done$ ) {
15:       $s.backtrack \leftarrow s.backtrack \setminus \{h\}$ ;
16:       $s.done \leftarrow s.done \cup \{h\}$ ;
17:      let  $t \in s.enabled, tid(t) = h$ , and let  $s' = next(s, t)$ ;
18:       $S.push(s')$ ;
19:      if  $\exists s_x \in S$  s.t.  $s_x \xrightarrow{t_x} s \xrightarrow{t} s'$ , add a directed edge  $(t_{x_g}, t_g)$  to  $G$ 
20:      SDPOR( $S, H, G$ );
21:       $S.pop()$ ;
22:    }
23:  }
24: }

```

Fig. 5. Stateful dynamic partial order reduction (SDPOR)

```

1: UPDATEBACKTRACKSETS( $S, t$ ) {
2:   let  $T$  be the sequence of transitions that are executed from the initial state of
   the program, following the sequence of states in  $S$ ;
3:   let  $t_d$  be the latest transition in  $T$  that is dependent and may be co-enabled
   with  $t$ ;
4:   if ( $t_d \neq \text{null}$ ) {
5:     let  $s_d$  be the state in  $S$  from which  $t_d$  is executed;
6:     let  $E$  be  $\{q \in s_d.enabled \mid tid(q) = tid(t), \text{ or } q \text{ in } T, q \text{ was after } t_d, \text{ and there is}$ 
       a happens-before relation for  $(q, t); \}$ 
7:     if ( $E \neq \emptyset$ )
8:       choose any  $q$  in  $E$ , add  $tid(q)$  to  $s_d.backtrack$ ;
9:     else
10:       $s_d.backtrack \leftarrow s_d.backtrack \cup \{tid(q) \mid q \in s_d.enabled\}$ ;
11:   }
12: }

```

Fig. 6. Updating the backtrack sets for states in the search stack

4.3 Efficient SDPOR

The algorithm of Figure 5 assumes that the model checker is capable of capturing the program states precisely. Here we present the practical algorithm which combines SDPOR of Figure 5 with the light-weight state capturing scheme that is presented in Section 3. Figure 7 shows the algorithm. Here the procedure SDPOR takes four parameters – although the parameter S is still the search stack, the element of the stack is a pair (s, s_a) such that $s \in S$, $s_a \in S_a$, and s_a is the abstract state of s . The parameter L is the set of local state hash tables. The parameter H , G are the same as in Figure 5.

```

1: Initially:  $S.push(s_0)$ ;  $H$  is empty;  $\forall L_\tau \in L : L_\tau$  is empty;  $G$  is empty;

2: SDPOR( $S, H, L, G$ ) {
3:    $\langle s, s_a \rangle \leftarrow S.top$ ;
4:   if ( $s_a \in H$ ) {
5:     let  $\mathcal{U} = \{v \mid \exists t \in s.enabled, v \text{ is reachable in } G \text{ from the node } t_g\}$ ;
6:     for each  $t \in \mathcal{U}$ , UPDATEBACKTRACKSETS( $S, t$ );
7:     return;
8:   }
9:   add  $s_a$  into  $H$ ;
10:  for each  $t \in s.enabled$ , UPDATEBACKTRACKSETS( $S, t$ );
11:  if ( $\exists$  thread  $\tau$ ,  $\exists t \in s.enabled, tid(t) = \tau$ ) {
12:     $s.backtrack \leftarrow \{\tau\}$ ;
13:     $s.done \leftarrow \emptyset$ ;
14:    while ( $\exists h \in s.backtrack \setminus s.done$ ) {
15:       $s.backtrack \leftarrow s.backtrack \setminus \{h\}$ ;
16:       $s.done \leftarrow s.done \cup \{h\}$ ;
17:      let  $t \in s.enabled, tid(t) = h$ , and let  $s' = next(s, t)$ ;
18:      let  $\delta_h = l_h(s') \setminus l_h(s)$ , and  $x = NEXTLOCAL(L_h, lid_h(s_a), \delta_h)$ ;
19:      let  $s'_a \in S_a$  s.t.  $g(s'_a) = g(s') \wedge ls(s'_a) = ls(s_a)[\tau := x] \wedge s'_a.PCs = s'.PCs$ ;
20:       $S.push(\langle s', s'_a \rangle)$ ;
21:      if  $\exists s_x \in S$  s.t.  $s_x \xrightarrow{t_x} s \xrightarrow{t} s'$ , add a directed edge  $(t_{x_g}, t_g)$  to  $G$ 
22:      SDPOR( $S, H, L, G$ );
23:       $S.pop()$ ;
24:    }
25:  }
26: }
```

Fig. 7. The combination of SDPOR shown in Figure 5 with the light-weight state capturing scheme which is presented in Section 3

Comparing with SDPOR in Figure 5, in this combined algorithm, line 18-19 are the new statements for computing the abstract states, line 3 and line 20 are modified to adapt the changes of the search stack, and line 22 is changed to adapt the local state hash tables. The rest of the algorithm is the same as in Figure 5.

5 Implementation

We implemented the algorithm of Figure 7 in the infrastructure of the runtime model checker `Inspect` [7]. `Inspect` can instrument a multithreaded C program with code to intercept the visible operations, compile the instrumented program along with a stub library into an executable, and uses a centralized monitor to systematically explore interleavings of the program by concretely executing the program.

`Inspect` uses escape analysis [12] to reveal potential visible operations in a multithreaded program. Building upon this approach, we implemented an intra-procedural forward data-flow analysis to determine the local state changes between successive visible operations. For any transition t , we treat δ_t as δ_\perp when: (i) t_g of t is the first visible operation in the procedure, or (ii) there are function calls or updates of pointers between the previous visible operation and t_g . Otherwise, we compute δ_t by capturing the changes of the local variables.

In [11], we described how automated instrumentation is done for stateless runtime checking. To capture the local state changes of threads, we instrument extra code into the program under test to inform the scheduler the local state changes.

6 Experimental Results

We performed experiments on a set of multithreaded benchmarks: `example1` is the program shown in Figure 1, `sharedArray` is a benchmark from [13]. It has two threads that iteratively write to different elements of a shared array. `bbuf` is an implementation of a bounded buffer with concurrent producers and consumers. `bzip2smp` [14] and `pfscan` [15] are two real multithreaded applications. `bzip2smp` is a multithreaded compression program that uses multiple threads to speed up the compression of a file. `pfscan` is a multithreaded file scanner that uses multiple threads to search in parallel through directories. `bzip2smp` contains 6.4k lines of C code, and `pfscan` has 1k lines of C code. We used a 1MB text file as the input to `bzip2smp`, and a directory with two files as the input to `pfscan`.

Table 1 shows the experimental results using stateless DPOR and our stateful approach. All the experiments were performed on a PC with an Intel quad-core CPU of 2.4GHz and 2GB of memory. We use “-” to denote that the program cannot be completely checked within 24 hours (86400 seconds).

We compared SDPOR with DPOR on the number of executions (or runs) they require to check a program, the number of transitions explored, and the checking time. Note that for SDPOR, the number of transitions being explored minus the number of “re-visited” states is the number of states encountered in the search. From the experimental results, it is clear that our stateful DPOR approach is more effective than the stateless DPOR, in reducing both the number of transitions to be explored and the checking time.

Table 1. Experimental results on the comparison between DPOR and SDPOR

Benchmarks	Threads	DPOR			SDPOR			
		runs	transitions	time(s)	runs	transitions	re-visited	time(s)
example1	2	-	-	-	35	2,084	12	1.41
sharedArray	2	-	-	-	98	18,557	33	5.70
bbuf	4	47,096	1,058,962	938.27	16,246	349,717	669	344.88
bzip2smp	4	-	-	-	4,598	26,442	4460	1311.15
bzip2smp	5	-	-	-	18,709	92,276	18,278	9456.34
bzip2smp	6	-	-	-	51,400	236,863	50,401	25659.38
pfscan	3	84	1,157	0.527	71	967	2	0.485
pfscan	4	13,617	189,218	240.74	3,168	40,395	334	57.43
pfscan	5	-	-	-	272,873	3,402,486	39,008	5328.84

7 Related Work

There has been substantial work on stateful model checking. Model checkers such as SPIN [16] and Bogor [17] have been very successful in revealing bugs and proving the correctness of systems. However, it is difficult for classic model checkers to check realistic multithreaded programs, which often heavily use library routines and have sophisticated memory manipulation operations. The advantage of our approach is able to directly examine the programs and avoid the modeling overhead (and potential consistency issues).

Musuvathi et al. [5] developed CMC, which is a runtime model checker that can precisely capture the states of a concurrent program by snapshotting the kernel space plus the user space of the program. In our work, we do not capture the whole state of a multithreaded program. Instead, we abstract the local states of threads as identities, and try to recognize the same states in different executions by tracking the local state changes. Compared with CMC, our approach is more light-weight in capturing states at runtime.

Gueta et al. [13] proposed Cartesian partial order reduction, which reduces the search space by delaying unnecessary context switches using Cartesian vectors. Cartesian partial order reduction performs stateful search, and can deal with cyclic state space. However, their approach assumed that the model checker is capable of capturing the states precisely, and did not address the problem of practical state capturing at runtime. We present a light-weight method for capturing the states of concurrent programs at runtime, and show how to adapt the stateful search into dynamic partial order reduction.

Yi et al. [18] proposed another stateful dynamic partial order reduction method based on the summary of interleavings. [18] also assumed that the model checker is able to precisely capture the states, and did not address the problem of state capturing at runtime. Their definition of *summary* for interleavings is a set of happen-before transition mappings. In their method, each state is associated with a summary of interleaving information, which could be very expensive to store and to keep updated. When a visited state is encountered, our SDPOR computes a summary for the states that can be reached from the visited state in one or more

steps. Different from their work, we use a visible operation dependency graph to dynamically compute the summary when a visited state is encountered. As a result, in our approach, the state summary computation is more light-weight.

8 Conclusion

To overcome the problem of capturing local states of multithreaded C programs at runtime, we propose a novel light-weight state abstraction scheme to conservatively capture local states. We also propose a stateful dynamic partial order reduction algorithm, and show how to combine it with our light-weight state capturing scheme. Compared with the traditional stateless DPOR approach, our approach is able to detect commutativity of transitions in different executions of multithreaded programs at runtime, and avoid exploring redundant interleavings. The experiments show that our approach is more efficient than stateless DPOR in checking realistic programs.

References

1. Lee, E.A.: The problem with threads, vol. 39, pp. 33–42. IEEE Computer Society Press, Los Alamitos (2006)
2. Godefroid, P.: Model Checking for Programming Languages using Verisort. In: POPL, pp. 174–186 (1997)
3. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 446–455. ACM, New York (2007)
4. Flanagan, C., Godefroid, P.: Dynamic Partial-order Reduction for Model Checking Software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 110–121. ACM, New York (2005)
5. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code. In: OSDI (2002)
6. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: ASE, pp. 3–12 (2000)
7. <http://www.cs.utah.edu/~yuyang/inspect>
8. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: PLDI, pp. 320–331. ACM Press, New York (2006)
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Heidelberg (1996)
10. Palmer, R.L.: Formal Analysis for MPI-based High Performance Computing Software, Ph.D. Dissertation, University of Utah (2007)
11. Yang, Y., Chen, X., Gopalakrishnan, G.: UUCS-08-004:Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical report (2008)
12. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPOPP, pp. 12–23. ACM Press, New York (2001)
13. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
14. <http://bzip2smp.sourceforge.net/>

15. <http://freshmeat.net/projects/pfscan>
16. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
17. Robby, Dwyer, M.B., Hatchiff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC / SIGSOFT FSE, pp. 267–276 (2003)
18. Yi, X., Wang, J., Yang, X.: Stateful Dynamic Partial-Order Reduction. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 149–167. Springer, Heidelberg (2006)

Appendix

Theorem 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. In a depth first search on S following the algorithm of Figure 3, let $s, s' \in S$ be states that can be reached from s_0 , and let $s_a, s'_a \in S_a$ be the abstract states of s and s' . Then $\forall \tau \in Tid : lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$.*

Proof. In a depth first search on S following the algorithm of Figure 3, let n be the number of times that NEXTLOCAL returns to DFS from line 4 or line 5 of Figure 4. That is, n is the number of times that NEXTLOCAL is invoked with either $\delta_\tau = \delta_\varepsilon$ or $\exists y. \langle (i, \delta_\tau) \rightarrow y \rangle \in L_\tau$. We now prove the theorem by induction on n .

- (Base case) $n = 0$: Following NEXTLOCAL, if $n = 0$, all calls to NEXTLOCAL must return from either line 3 or line 7. That is, NEXTLOCAL has never been invoked with $\delta_\tau = \delta_\varepsilon$ or $\exists y. \langle (i, \delta_\tau) \rightarrow y \rangle \in L_\tau$. Hence, following the algorithm of Figure 4, every local state that has been visited must be assigned a unique id. Therefore, $lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$ holds in this situation.
- (Induction hypothesis) Let $k \geq 0$. For all $n, n \leq k$, Theorem 1 holds.
- (Induction step) Let $n = k + 1$. Let $s' \in S$ be the state and $\tau \in Tid$ be the thread such that by invoking NEXTLOCAL($L_\tau, lid_\tau(s'), \delta_\tau$) at line 8, n changes from k to $k + 1$. Consider the situation that DFS has finished executing line 8 of Figure 3, but has not started executing line 9. Let $s \in S$ be a state and t be a transition such that $s \xrightarrow{t} s'$ and $\tau = tid(t)$. There are two cases with respect to s' :
 - If $\delta_\tau = \delta_\varepsilon$, according to line 7 of DFS, we have $l_\tau(s) = l_\tau(s')$, and $lid_\tau(s_a) = lid_\tau(s'_a)$. Obviously $lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$ holds. Hence the theorem holds.
 - If $\exists y. \langle (lid_\tau(s_a), \delta_\tau) \rightarrow y \rangle \in L_\tau$: Let $s_1, s_2 \in S$ be the two states that have been visited and t_1 be the transition such that $\langle (lid_\tau(s_a), \delta_\tau) \rightarrow y \rangle$ was added to L_τ when DFS explored $s_1 \xrightarrow{t_1} s_2$. Let s_{1_a} and s_{2_a} respectively be the abstract state of s_1 and s_2 . Obviously we have $lid_\tau(s_{1_a}) = lid_\tau(s_a)$, $lid_\tau(s_{2_a}) = y$, and $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$. According to the induction hypothesis, $l_\tau(s_1) = l_\tau(s)$ must hold. As $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$, we have $l_\tau(s_2) = l_\tau(s')$. Hence, the theorem holds. Otherwise, it contradicts the induction hypothesis.

□

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let s be a state in S . We use R_s to denote the set of states that are reachable from s by executing one or more transitions. Obviously we have $R_s \subseteq S$.


```

1: Initially:  $c = 0$ ;  $S.push(s_0)$ ;  $H$  is empty;

2: SDPOR $_k(S, H)$  {
3:    $s \leftarrow S.top$ ;
4:   if ( $s \in H$ ) {
5:      $c \leftarrow c + 1$ ;
6:     if ( $c \leq k$ ) {
7:       let  $T_p = \{t_g \mid t \text{ can be executed from states which are reachable from } s \}$ ;
8:       for each  $t \in T_p$ , UPDATEBACKTRACKSETS( $S, t$ );
9:       return;
10:    }
11:  }
12:  add  $s$  into  $H$ ;
13:  for each  $t \in s.enabled$ , UPDATEBACKTRACKSETS( $S, t$ );
14:  if ( $\exists$  thread  $\tau$ ,  $\exists t \in s.enabled, tid(t) = \tau$ ) {
15:     $s.backtrack \leftarrow \{\tau\}$ ;
16:     $s.done \leftarrow \emptyset$ ;
17:    while ( $\exists h \in s.backtrack \setminus s.done$ ) {
18:       $s.backtrack \leftarrow s.backtrack \setminus \{h\}$ ,  $s.done \leftarrow s.done \cup \{h\}$ ;
19:      let  $t \in s.enabled, tid(t) = h$ , and let  $s' = next(s, t)$ ;
20:       $S.push(s')$ ;
21:      SDPOR $_k(S, H)$ ;
22:       $S.pop()$ ;
23:    }
24:  }
25: }

```

Fig. 8. SDPOR $_k$ only stops depth-first search and backtrack immediately at the first k revisited states

Let SDPOR $_k$ be the algorithm of Figure 8. Comparing with SDPOR, the only difference between SDPOR $_k$ and SDPOR is that SDPOR $_k$ takes one more parameter k , which bounds SDPOR $_k$ to return only at the first k visited states. In more detail, SDPOR $_k$ uses a global counter c to record the number of visited states that it has encountered during the depth-first search (line 5 of Figure 8). When a visited state s_v is encountered, if $c \leq k$, then SDPOR $_k$ updates the backtrack sets of states in the search stack (line 7-8 of Figure 8) and returns immediately; otherwise, SDPOR $_k$ continues exploring R_{s_v} .

We have a class of algorithms $\{\text{SDPOR}_0, \text{SDPOR}_1, \dots\}$ by assigning k specific values. Let \mathcal{A} denote an algorithm that explores the state space of M . Let $S_{\mathcal{A}} \subseteq S$ refer to the set of states that is explored using \mathcal{A} by starting from s_0 . Obviously, we have $S_{\text{SDPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_{\infty}}$.

To prove the correctness of Theorem 2, we first prove Lemma 1, which characterize the relationship between SDPOR $_k$ and SDPOR $_{k+1}$.

Lemma 1. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let $s, s' \in S$ and t be a transition of M such that $s \xrightarrow{t} s'$. Let $k \geq 0$. If $s \xrightarrow{t} s'$ is explored by SDPOR $_k$, it must be explored by SDPOR $_{k+1}$.*

Proof. Let r be the value of the global variable c when we finish checking the state space of M using SDPOR_k . There are two cases with respect to r :

- If $r \leq k$, obviously that the state spaces traversed by SDPOR_k and SDPOR_{k+1} are identical. The lemma holds.
- If $r > k$, let v_i be the i -th visited state SDPOR_k and SDPOR_{k+1} encounter while exploring the state space of M . Let $\Gamma_i^x \subseteq \Gamma$ be the transition relation that has been explored by SDPOR_x before reaching the i -th visited states. It is obvious that $\forall i, i \leq k+1 : \Gamma_i^k = \Gamma_i^{k+1}$ holds.

Now we consider the exploration of the search space by SDPOR_k and SDPOR_{k+1} after they encounter v_{k+1} . Following the algorithm of Figure 8, while encountering v_{k+1} , SDPOR_k is equivalent to a stateless search that does not record the search history, and explores $R_{v_{k+1}}$. However, SDPOR_{k+1} does not explore $R_{v_{k+1}}$. Before encountering v_{k+1} , the state spaces explored by SDPOR_k and SDPOR_{k+1} are identical. Hence, the search stacks of SDPOR_k and SDPOR_{k+1} are identical at the point of encountering v_{k+1} . Let s^k and s^{k+1} denote the correspondent states that are respectively in the search stacks of SDPOR_k and SDPOR_{k+1} . To prove the lemma, all that we need to prove is that, when SDPOR_k and SDPOR_{k+1} backtrack from $s_{v_{k+1}}$, for all pairs of states $\langle s^k, s^{k+1} \rangle$, we have $s^k.\text{backtrack} \subseteq s^{k+1}.\text{backtrack}$.

This can be proved by contradiction. Suppose while backtracking from v_{k+1} , there exists $\langle s^k, s^{k+1} \rangle$ such that $s^k.\text{backtrack} \supset s^{k+1}.\text{backtrack}$. This implies that $\exists h \in \text{ Tid} : h \in s^k.\text{backtrack} \wedge h \notin s^{k+1}.\text{backtrack}$. As the only difference between SDPOR_k and SDPOR_{k+1} is that SDPOR_k explores $R_{v_{k+1}}$ while SDPOR_{k+1} does not, this can happen if and only if $\exists s_1, s_2 \in R_{v_{k+1}}, \exists t \in s^{k+1}.\text{enabled} : s_1 \xrightarrow{t_1} s_2$ and t_1 is dependent with t . However, following the algorithm of Figure 8, if the execution of t_1 happens before backtracking v_{k+1} in SDPOR_k , $t_1 \in T_p$. Hence, if $h \in s^k.\text{backtrack}$, h must be in $s^{k+1}.\text{backtrack}$. This contradicts $h \notin s^{k+1}.\text{backtrack}$. \square

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let G be the transition dependency graph of M . G is dynamically constructed following the algorithm of Figure 5. Let \mathcal{U} be the set of visible operations that is computed at line 5 of Figure 5. Let T_p be the set of visible operations that is computed as line 7 of Figure 8. We have the following lemma:

Lemma 2. $T_p \subseteq \mathcal{U}$.

Proof. Following the algorithm of Figure 5, it is clear that while backtracking from a state s , all transition dependency edges that can appear in R_s must have been added to G . Therefore, we have $\forall t \in T_p : t \in \mathcal{U}$. \square

Theorem 2. *Let $M = (S, s_0, \Gamma)$ be a multithreaded program. For every execution of a transition $s \xrightarrow{t} s'$ of M , if it is explored by the stateless DPOR, it must be explored by SDPOR.*

Proof. With Lemma 1, Lemma 2, $S_{\text{DPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_\infty}$, it is clear that the theorem holds. \square

Symbolic String Verification: An Automata-Based Approach^{*}

Fang Yu, Tefvik Bultan, Marco Cova, and Oscar H. Ibarra

Department of Computer Science
University of California, Santa Barbara
{yuf,bultan,marco,ibarra}@cs.ucsb.edu

Abstract. We present an automata-based approach for the verification of string operations in PHP programs based on symbolic string analysis. String analysis is a static analysis technique that determines the values that a string expression can take during program execution at a given program point. This information can be used to verify that string values are sanitized properly and to detect programming errors and security vulnerabilities. In our string analysis approach, we encode the set of string values that string variables can take as automata. We implement all string functions using a symbolic automata representation (MBDD representation from the MONA automata package) and leverage efficient manipulations on MBDDs, e.g., determinization and minimization. Particularly, we propose a novel algorithm for language-based replacement. Our replacement function takes three DFAs as arguments and outputs a DFA. Finally, we apply a widening operator defined on automata to approximate fixpoint computations. If this conservative approximation does not include any *bad* patterns (specified as regular expressions), we conclude that the program does not contain any errors or vulnerabilities. Our experimental results demonstrate that our approach works quite well in checking the correctness of sanitization operations in real-world PHP applications.

1 Introduction

Unsanitized string variables are a common cause of security vulnerabilities in Web applications. In typical interactive Web applications, user-provided input strings are often used to query back-end databases. If the user input is not properly checked and filtered (i.e., sanitized), the input strings that contain hidden destructive commands can be sent to back-end databases and cause damage. Using the string analysis techniques proposed in this paper, it is possible to automatically verify that a string variable is properly sanitized at a program point, showing that such attacks are not possible.

We present a string analysis technique that computes an over approximation of possible values that a string expression can take at a given program point. We use a deterministic finite automaton (DFA) to represent the set of values string

^{*} This work is supported by NSF grants CCF-0614002 and CCF-0716095.

expressions can take. At each program point, each string variable is associated with a DFA. The language accepted by the DFA corresponds to the values that the corresponding string variable can take at that program point.

The string analysis technique we present is a forward reachability computation that uses DFA as a symbolic representation. We use the symbolic DFA representation provided by the MONA DFA library [4], in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs). We iteratively compute an over approximation of the least fixpoint that corresponds to the reachable values of the string expressions. In each iteration, given the current state DFAs for all the variables, we compute the next state DFAs. We present algorithms for next state computation for string operations such as concatenation and language-based replacement. Particularly, we present an algorithm for the language-based replacement operation that computes the DFA for $\text{REPLACE}(M_1, M_2, M_3)$ where M_1 , M_2 , and M_3 are DFAs that accept the set of original strings, the set of match strings, and the set of replacement strings, respectively.

Our language-based replacement operation is essential to model various built-in functions of PHP language that can be used to perform input validation. These functions provide a general mechanism to scan a string for matches to a given pattern, expressed as a regular expression, and to replace the matched text with a replacement string. As an example of modeling these functions, consider the following statement:

```
$username = ereg_replace("<script *>", "", $_GET["username"]);
```

The expression `$_GET["username"]` returns the string entered by the user, the `ereg_replace` call replaces all matches of the search pattern with the empty string, and the result is assigned to the variable `username`. This statement can be modeled by our language-based replacement operation, where M_1 accepts arbitrary strings, M_2 accepts the set of strings that start with `<script` followed by zero or more spaces and terminated by the character `>`, and M_3 accepts the empty string.

We believe that we are the first to extend the MONA automata package to analyze these complex string operations on real programs. In addition to computing the language-based replacement operation, another difficulty is implementing these string operations without using the standard constructions based on the ϵ -transitions, since the MBDD-based automata representation used by MONA does not allow ϵ -transitions. We model non-determinism by extending the alphabet with extra bits and then project them away using the on-the-fly subset construction algorithm provided by MONA. We apply the projection one bit at a time, and after projecting each bit away, we use the MBDD-based automata minimization to reduce the size of the resulting automaton.

Since DFAs can represent infinite sets of strings, the fixpoint computations are not guaranteed to converge. To alleviate this problem, we use the automata widening technique proposed by Bartzis and Bultan [3] to compute an over-approximation of the least fixpoint. Briefly, we merge those states belonging to the same equivalence class identified by certain conditions. This widening operator was originally proposed for automata representation of arithmetic constraints

but the intuition behind it is applicable to any symbolic fixpoint computation that uses automata.

We implemented the proposed string analysis technique for PHP programs. PHP is a scripting language which is widely used in implementing interactive Web applications. Our experiments show that the proposed symbolic analysis technique works quite well and can be used to prove the correctness of sanitization in real-world PHP applications.

An Example: Consider the PHP program fragment below which demonstrates a vulnerability from a guestbook application called PBLguestbook-1.32:

```

1:  foreach ($_POST as $name => $value) {
2:      if ($name != 'process' && $name != 'password2') {
3:          $count++;
4:          $result .= "$name' = '$value'";
5:          if ($count <= $numofparts)
6:              $result .= ", ";
7:      }
8:  }
9:  $query = "UPDATE 'pblguestbook_config' SET $result";
10: mysql_query($query);

```

This program fragment traverses the input strings entered by the user (which are stored in the `_POST` array) in a loop (lines 1-8) and constructs a query string by accumulating them (by concatenating them to the `result` variable). This query is then sent to the back-end database (line 10).

This program shows an example of a SQL injection vulnerability. Input strings are concatenated in the loop at lines 1-8 to form the string used to query the application's database. Since no sanitization is performed, an attacker can modify the query, for example, by injecting a parameter with value `' ; DROP DATABASE #`. In this case, the SQL string sent to the database will be `UPDATE 'pblguestbook_config' SET 'name' = '' ; DROP DATABASE #`. Note that the `' ;` character separates distinct queries and the `#` character starts a comment. Therefore, if the database allows the execution of multiple queries, it will execute the legitimate query intended by the developer and the injected query that drops the entire database. The vulnerability can be fixed by adding a sanitization step on the input parameters before the query string is formed.

A properly sanitized version of this program fragment would be:

```

1:  foreach ($_POST as $name => $value) {
1.1:  $name = preg_replace("/[^\a-zA-Z0-9]/", "", $name);
1.2:  $value = preg_replace("/'/", "", $value);
2:      if ($name != 'process' && $name != 'password2') {
3:          $count++;
4:          $result .= "$name' = '$value'";
5:          if ($count <= $numofparts)
6:              $result .= ", ";
7:      }
8:  }

```

```

9:   $query = "UPDATE 'pblguestbook_config' SET $result";
10:  mysql_query($query);

```

The sanitization is achieved in lines 1.1 and 1.2 by deleting potentially problematic characters in the variables `$name` and `$value`, hence preventing the presented SQL command injection attack. We analyzed both the vulnerable and the sanitized versions of this program fragment using our string analysis tool. Our string analysis tool constructed a DFA that gives an over-approximation of the string values that the variable `query` can take at line 10. We wrote a regular expression characterizing strings that can be used for SQL command injection and converted it to a DFA. (Note that these types of attack DFAs can be constructed once and stored in a library. They do not have to be specified separately for each program that is being analyzed). Then, we checked if the intersection of the language recognized by the DFA for the `query` variable at line 10, and the DFA characterizing the SQL command injection attack is empty. When we applied our analysis to the vulnerable program fragment shown above, our string analysis tool reported that the intersection is not empty, i.e., the program fragment might be vulnerable. However, when we applied our analysis to the sanitized version, our tool reported that the intersection is empty, proving that the variables are properly sanitized.

It is worthwhile to note some of the challenges in analyzing the example given above. First, in order to prove that the variables are properly sanitized, we need to statically interpret the replacement function `preg_replace` with reasonable precision. Second, our fixpoint computation has to converge even though the above program fragment contains a loop. We are able to handle both of these challenges by 1) proposing and implementing a novel language-based replacement operation and 2) using an automata widening operator. Note that, for the sanitized program fragment, the fixpoint computation without widening will not converge. Moreover, a naive over-approximation, that sets the values of the variables that are updated in a loop to all possible strings, will not be a tight enough approximation to verify the sanitized program fragment.

The rest of the paper is organized as follows. In Section 2, we describe our symbolic string analysis algorithm. In Section 3, we describe the implementation of the closure, concatenation and replacement operations. In Section 4, we discuss the widening operation. In Section 5, we summarize our experiments. In Section 6, we discuss the related work, and, in Section 7, we conclude the paper.

2 Automata-Based String Analysis

Most of the string manipulation operations performed in real-world applications can be reduced to the following four operations:

- *assignment*: assigns the current string value of a variable to another variable (the assignment operator in PHP is “=”);
- *concatenation*: concatenates two string variables and/or constants (the concatenation operation in PHP is “.”);

- *replacement*: replaces the parts of a string that match the given pattern with the given replacement string (there are several string replacement functions in PHP such as `htmlspecialchars`, `tolower`, `toupper`, `str_replace`, `trim`, `preg_replace` and `ereg_replace`, and they can all be converted to this form).
- *restriction*: restricts the value of a string variable based on a branch condition.

The first step of string analysis is to construct a control flow graph (CFG) that only contains string variables and operations on string variables. We define a CFG as a tuple (V, S, E) where V is the set of string variables, S is the set of statements and $E \subseteq S \times S$ is the set of control flow edges. Each statement $s \in S$ could be one of the following operations: *null*, *assign*, *concat*, *replace*, *restrict*, *input*. The *null* operation represents the statements that do not influence the string variables and, hence, have been removed. We use *assign* to denote the assignment of a string constant or a variable to a string variable. We use *concat* to denote the assignment operations that assign the concatenation of two string constants and/or variables to a string variable. We use *replace* to denote the assignment of a string value computed by a replacement operation to a string variable. We use *restrict* to denote the restriction of a string value in order to model branch conditions. For instance, considering a branch condition $v = e$, where e is a regular expression, we add *restrict*(v, e) at the beginning of the truth branch and *restrict*(v, \bar{e}) at the beginning of the false branch where \bar{e} indicates to restrict the string values of v to the complement set of e . A similar idea has been discussed in [15]. Finally, we use *input* to denote a read operation, where a string variable is assigned a value provided by a user.

Automata Operations: In order to implement the automata-based string analysis, we implement the following operations:

- **CONSTRUCT**(regexp e): Returns a DFA M , $L(M) = \{w \mid w \in L(e)\}$.
- **CLOSURE**(DFA M_1): Returns a DFA M , $L(M) = \{w_1w_2 \dots w_k \mid k > 0, \forall i, 1 \leq i \leq k, w_i \in L(M_1)\}$.
- **CONCAT**(DFA M_1 , DFA M_2): Returns a DFA M , $L(M) = \{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.
- **REPLACE**(DFA M_1 , DFA M_2 , DFA M_3): Returns a DFA M , $L(M) = \{w_1c_1w_2c_2 \dots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \dots w_kx_kw_{k+1} \in L(M_1), \forall i, x_i \in L(M_2), w_i \text{ does not contain any substring accepted by } M_2, c_i \in L(M_3)\}$.
- **UNION**(DFA M_1 , DFA M_2): Returns a DFA M , $L(M) = L(M_1) \cup L(M_2)$.
- **INTERSECT**(DFA M_1 , DFA M_2): Returns a DFA M , $L(M) = L(M_1) \cap L(M_2)$.
- **WIDENING**(DFA M_1 , DFA M_2): Returns a DFA M , $L(M) \supseteq L(M_1) \cup L(M_2)$.
- **EQUCHECK**(DFA M_1 , DFA M_2): Checks whether $L(M_1) = L(M_2)$.
- **EMPCHECK**(DFA M): Checks whether $L(M) = \emptyset$.
- **EMPTY**(ϵ): Returns a DFA which does not accept any string.
- **UNIVERSAL**(ϵ): Returns a DFA which accepts all the strings.

String Analysis Algorithm: The string analysis algorithm, takes a CFG, a program point, a string variable and an attack pattern as input. It computes $|V| \times |S|$ DFAs, where the DFA (v, s) accepts the language that corresponds to all

```

Input:  $(V, S, E)$ , attackpattern, statement, variable
DFA attack := CONSTRUCT(attackpattern)
DFA old[1...|V|][1...|S|], new[1...|V|][1...|S|], temp[1...|V|]
for each  $v \in V$ ,  $s \in S$ , old[ $v$ ][ $s$ ] := EMPTY(), new[ $v$ ][ $s$ ] := EMPTY()
repeat
  for each  $v \in V$ ,  $s \in S$ , old[ $v$ ][ $s$ ] := new[ $v$ ][ $s$ ]
  for each  $s \in S$ 
    for each  $v \in V$ , temp[ $v$ ] := EMPTY()
    for each  $(s', s) \in E$ , temp[ $v$ ] := UNION(temp[ $v$ ], old[ $v$ ][ $s'$ ])
    for each  $v \in V$ , new[ $v$ ][ $s$ ] := temp[ $v$ ]
  switch s.type
    case null skip
    case read //  $v := \text{get input}$ 
      new[ $v$ ][ $s$ ] := UNIVERSAL()
    case assign //  $v := v_1$ 
      new[ $v$ ][ $s$ ] := temp[ $v_1$ ]
    case concat //  $v := \text{concat}(v_1, v_2)$ 
      new[ $v$ ][ $s$ ] := CONCAT(temp[ $v_1$ ], temp[ $v_2$ ])
    case replace //  $v := \text{replace}(v_1, e, c)$ 
      where  $e$  is a regular expression and  $c$  is a string.
      DFA  $t_1$  := CONSTRUCT( $e$ ), DFA  $t_2$  := CONSTRUCT( $c$ )
      new[ $v$ ][ $s$ ] := REPLACE(temp[ $v_1$ ],  $t_1$ ,  $t_2$ )
    case restrict // restrict( $v, e$ )
      DFA  $t_1$  := CONSTRUCT( $e$ )
      new[ $v$ ][ $s$ ] := INTERSECT(old[ $v$ ][ $s$ ],  $t_1$ )
  for each  $v \in V$ ,  $s \in S$ , old[ $v$ ][ $s$ ] := WIDENING(old[ $v$ ][ $s$ ], new[ $v$ ][ $s$ ])
until (for all  $v, s$ , EQUCHECK(old[ $v$ ][ $s$ ], new[ $v$ ][ $s$ ]))
if (EMPCHECK(INTERSECT(new[variable][statement], attack))) then VER else ERR

```

Fig. 1. String analysis algorithm

the string values that the variable v can take at the program point s during any program execution. We compute these DFA using a least fixpoint computation as shown in Figure 1. Since the lattice is infinite, it might not be possible to reach the least fixpoint using an iterative algorithm. To tackle this problem, we apply the automata widening operator in [3] to our analysis. Following Bartzis and Bultan's results, we characterize a set of languages that this widening operator can result in the precise fixed point. Our string analysis algorithm returns VER if it is not possible for the input variable to have a string value that matches the attack pattern at the given program point; however, it may yield a false alarm while it returns ERR.

Symbolic Automata Representation: We use the DFA library of MONA [4] to implement the string operations listed above. In MONA, transition relations of DFA are symbolically represented using Multi-terminal Binary Decision Diagrams (MBDDs). A MBDD is a BDD with multiple roots and multiple leaves. In MONA's DFA representation, each state of the DFA is a root and points to

a BDD node, and each leaf value is a state of the DFA. Given the current state and a symbol $a \in B^k$, where B^k is alphabet of bit vectors of length k , one can find the next state by following the BDD nodes according to the bit vector of a from the BDD node pointed by the current state. We use a 7-bit vector, i.e., B^7 , as our alphabet representing the binary value of ASCII symbols, e.g., for the ASCII symbol ‘a’, the ASCII code is 97 which is represented as ‘1100001’ in our encoding.

The MONA DFA library provides efficient implementations of standard automata operations. These operations include product, project and determinize, and minimize [4]. The product operation takes the Cartesian product of the states of the two input automata. We use the product operation to implement the intersection and union operations. The project and determinize operation, denoted as $\text{PROJECT}(M, i)$, where $1 \leq i \leq k$, converts a DFA M recognizing a language L over the alphabet B^k , to a DFA M' recognizing a language L' over the alphabet B^{k-1} , where L' is the language that results from applying the tuple projection on the i^{th} bit to each symbol of the alphabet. The process consists of removing the i^{th} track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction.

3 String Operations on Automata

In this section, we describe how to implement the closure, concatenate and replace operations. Since we use MBDD representation for DFA, we are not able to introduce ϵ -transitions. Instead, to avoid the non-determinism introduced by these operations, we extend the alphabet by adding extra bits, and then use projection to map the resulting DFA to the original alphabet.

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where Q is a finite set of states, q_0 is the initial state, $\Sigma \subseteq B^k$ is the alphabet, where each symbol is encoded as a k -bit string. $F : Q \rightarrow \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \rightarrow Q$ is the transition relation. A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α) , $\delta(q, \alpha)$ goes to a *sink* state. In the constructions below, we also ignore the transitions that lead to a sink state.

Given $\alpha \in B^k$, we use $\alpha 0$ or $\alpha 1 \in B^{k+1}$ to denote the bit string that is α appended with ‘0’ or ‘1’. For instance, if α is ‘110011’ then $\alpha 0$ is ‘1100110’.

Closure: The DFA M is a closure-DFA of the DFA M_1 , if $L(M) = \{ w_1 w_2 \dots w_k \mid \exists k > 0, \forall 1 \leq i \leq k, w_i \in L(M_1) \}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, its closure M can be constructed by first constructing an intermediate DFA $M' = \langle Q_1, q_{10}, \Sigma', \delta', F_1 \rangle$ as:

- $\Sigma' = \{ \alpha 0 \mid \alpha \in \Sigma \} \cup \{ \alpha 1 \mid \alpha \in \Sigma \}$
- $\forall q, q' \in Q_1, \delta'(q, \alpha 0) = q', \text{ if } \delta_1(q, \alpha) = q'.$
- $\forall q \in Q_1, \delta'(q, \alpha 1) = q', \text{ if } F_1(q) = + \text{ and } \delta_1(q_{10}, \alpha) = q'.$

Then, $M = \text{PROJECT}(M', k + 1)$ is the closure of M_1 .

Since M_1 is a DFA, the project operation requires the subset construction only when there exists $q \in Q_1, F_1(q) = +$, and $\exists \alpha, q', q'', \alpha \in \Sigma, q', q'' \in Q_1, q' \neq q'', \delta_1(q, \alpha) = q', \delta_1(q_{10}, \alpha) = q''$.

Concatenation: The DFA M is a concatenation-DFA of the DFA M_1 and M_2 , if $L(M) = \{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$ and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, the concatenation-DFA M can be constructed as follows. Without loss of generality, we assume that $Q_1 \cap Q_2$ is empty. We first construct an intermediate DFA $M' = \langle Q', q_{10}, \Sigma', \delta', F' \rangle$, where

- $Q' = Q_1 \cup Q_2$
- $\Sigma' = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta'(q, \alpha 0) = q', \text{ if } \delta_1(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta'(q, \alpha 0) = q', \text{ if } \delta_2(q, \alpha) = q'$
- $\forall q \in Q_1, \delta'(q, \alpha 1) = q', \text{ if } F_1(q) = + \text{ and } \exists q' \in Q_2, \delta_2(q_{20}, \alpha) = q'$
- $\forall q \in Q_1, F'(q) = +, \text{ if } F_1(q) = + \text{ and } F_2(q_{20}) = +; F'(q) = -, \text{ o.w.}$
- $\forall q \in Q_2, F'(q) = F_2(q)$.

Then, $M = \text{PROJECT}(M', k + 1)$. Again, since both M_1 and M_2 are DFA, the subset construction happens only when there exists $q \in Q_1, F_1(q) = +$ such that $\exists \alpha, q', q'', \alpha \in \Sigma, q' \in Q_1, q'' \in Q_2, \delta_1(q, \alpha) = q', \delta_2(q_{20}, \alpha) = q''$.

Replacement: A DFA M is a replaced-DFA of a DFA tuple (M_1, M_2, M_3) , if and only if $L(M) = \{w \mid k > 0, w_1x_1w_2 \dots w_kx_kw_{k+1} \in L(M_1), w = w_1c_1w_2 \dots w_kc_kw_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k + 1, w_i \notin \{w'_1x'_1w'_2 \mid x' \in L(M_2), w'_1, w'_2 \in \Sigma^*\}\}$.

This definition requires that all occurrences of matching sub-strings in a word are replaced. The intuition of the implementation of this language-based replacement is that we first insert marks into automata, then identify matching sub-strings by intersection of automata, and finally construct the final automaton by replacing these matching sub-strings.

We consider a new alphabet $\bar{\Sigma} = \{\bar{\alpha} \mid \alpha \in \Sigma\}$, and let \bar{x} denote a new string in which we add bar to each character in x . Assume that M_1, M_2, M_3 have the same alphabet Σ , where $\#_1, \#_2 \notin \Sigma$, and $\forall \alpha \in \Sigma, \bar{\alpha} \notin \Sigma$. We define M'_1, M'_2 and M as follows, and claim that M accepts the same language as the replaced-DFA of the tuple (M_1, M_2, M_3) .

- M'_1 , where $L(M'_1) = \{w' \mid k > 0, w = w_1x_1w_2 \dots w_kx_kw_{k+1} \in L(M_1), w' = w_1\#_1\bar{x}_1\#_2w_2 \dots w_k\#_1\bar{x}_k\#_2w_{k+1}\}$.
- M'_2 , where $L(M'_2) = \{w' \mid k > 0, w' = w_1\#_1\bar{x}_1\#_2w_2 \dots w_k\#_1\bar{x}_k\#_2w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), \forall 1 \leq i \leq k + 1, w_i \in L(M_h)\}$, where $L(M_h)$ is the set of strings which do not contain any substring in $L(M_2)$. The language $L(M_h)$ is defined as the complement set of $\{w_1xw_2 \mid x \in L(M_2), w_1, w_2 \in \Sigma^*\}$.
- M , where $L(M) = \{w \mid k > 0, w_1\#_1\bar{x}_1\#_2w_2 \dots w_k\#_1\bar{x}_k\#_2w_{k+1} \in L(M'_1) \cap L(M'_2), w = w_1c_1w_2 \dots w_kc_kw_{k+1}, \forall 1 \leq i \leq k, c_i \in L(M_3)\}$.

To distinguish the original and bar alphabets, we append an extra bit to α so that α is $\alpha 0$ and $\bar{\alpha}$ is $\alpha 1$. Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, and $M_3 = \langle Q_3, q_{30}, \Sigma, \delta_3, F_3 \rangle$, the process to construct a replaced-DFA M can be decoupled into the following steps:

1. Construct M_1' from M_1 ,
2. Construct M_2' from M_2 ,
3. Generate M' as the intersection of M_1' and M_2' ,
4. Construct M'' from M' where the strings that appear between $\#_1$ and $\#_2$ are replaced by words in $L(M_3)$, and
5. Generate M from M'' by projection.

We formally describe the implementation of these steps below. As a running example, we use $L(M_1) = \{baab\}$, $L(M_2) = a^+$ (M_2 accepts the language $\{a, aa, aaa, \dots\}$) and $L(M_3) = \{c\}$ or $L(M_3) = \{\epsilon\}$. Let $|M|$ denote the number of states of M . An upper bound for each intermediate automaton before projection and minimization is also described.

Step 1: $M_1' = \langle Q_1', q_{10}, \Sigma', \delta_1', F_1' \rangle$ is constructed from M_1 , where

- $Q_1' = Q_1 \cup Q_{1'}$, $Q_{1'}$ is the duplicate of Q_1 . For all $q \in Q_1$, there is a one to one mapping $q' \in Q_{1'}$.
- $\Sigma' = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\} \cup \{\#_1, \#_2\}$
- $\delta_1'(q_1, \alpha 0) = q_2$ and $\delta_1'(q_{1'}, \alpha 1) = q_{2'}$, if $\delta_1(q_1, \alpha) = q_2$
- $\forall q_1 \in Q_1, \delta_1'(q_1, \#_1) = q_{1'}$ and $\delta_1'(q_{1'}, \#_2) = q_1$
- $\forall q \in Q_1, F_1'(q) = F_1(q)$ and $\forall q \in Q_{1'}, F_1'(q) = 0$.

An example for constructing M_1' from M_1 , where $L(M_1) = \{baab\}$, is given in Fig 2. $|M_1'|$ is bounded by $2|M_1|$.

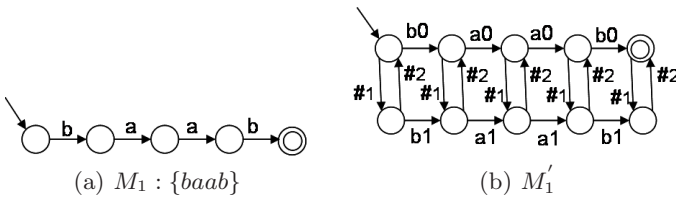


Fig. 2. Constructing M_1' from M_1

Step 2: To construct M_2' , we first construct M_h which accepts the complement set of $\{w_1 x w_2 \mid w_1, w_2 \in \Sigma^*, x \in L(M_2)\}$. For instance, as shown in Fig 3(b), for $L(M_2) = a^+$, M_h is the DFA that accepts $(\Sigma \setminus \{a\})^*$. Let M_* be the DFA accepting Σ^* . M_h can be constructed by $\text{NEGATE}(\text{CONCAT}(\text{CONCAT}(M_*, M_2), M_*))$. We obtain the DFA in Fig 3(b) by applying this construction with minimization.

Assume $M_h = \langle Q_h, q_{h0}, \Sigma, \delta_h, F_h \rangle$, and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$. $M_2' = \langle Q_2', q_{h0}, \Sigma', \delta_2', F_2' \rangle$ can then be constructed as:

- $Q'_2 = Q_h \cup Q_2$
- $\Sigma' = \{\alpha 0 \mid \forall \alpha \in \Sigma\} \cup \{\alpha 1 \mid \forall \alpha \in \Sigma\} \cup \{\#1, \#2\}$
- $\forall q, q' \in Q_h, \delta'_2(q, \alpha 0) = q', \text{ if } \delta_h(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta'_2(q, \alpha 1) = q', \text{ if } \delta_2(q, \alpha) = q'$
- $\forall q \in Q_h, \delta'_2(q, \#1) = q_{20} \text{ if } F_h(q) = +$
- $\forall q \in Q_2, \delta'_2(q, \#2) = q_{h0} \text{ if } F_2(q) = +$
- $\forall q \in Q_h, F'_2(q) = F_h(q) \text{ and } \forall q \in Q_2, F'_2(q) = -.$

The corresponding M'_2 for our example is shown in Fig 3(c). $|M'_2|$ is bounded by $|M_h| + |M_2|$, where $|M_h|$ is bounded by $|M_2| + 2$.

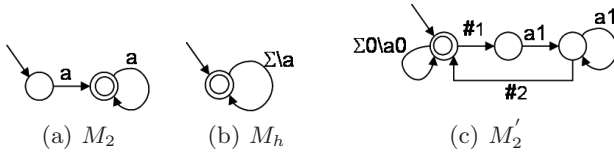


Fig. 3. Constructing M'_2 from M_2 and M_h

Step 3: $M' = \langle Q', q'_0, \Sigma', \delta', F' \rangle$ is generated as the intersection of M'_1 and M'_2 based on production. The example M' is shown in Fig 4 (a). $|M'|$ is bounded by $|M'_1| \times |M'_2|$.

Step 4: Before we construct M'' from M' , we first introduce a function $reach : Q' \rightarrow 2^{Q'}$, which maps a state to all its $\#$ -reachable states in M' . We say q' is $\#$ -reachable from q if there exists a sequence q, q_1, \dots, q_n, q' so that (1) $n \geq 1$, (2) $\delta'(q, \#1) = q_1$, (3) $\delta'(q_n, \#2) = q'$, and (4) $\forall 0 < i < n, \delta'(q_i, x) = q_{i+1}$, where $x \in \{\alpha 1 \mid \forall \alpha \in \Sigma\}$. For instance, in Fig 4 (a), one can find that $reach(i) = \{j, k\}$ and $reach(j) = \{k\}$. Intuitively, one can think that each pair (q, q') , where $q' \in reach(q)$, identifies a word in $L(M_2)$.

Our goal is, for each $q' \in reach(q)$, inserting paths between q and q' that recognize all words in $L(M_3)$. If there exist $q', q'' \in reach(q)$ and $q' \neq q''$, this insertion will cause nondeterminism. To tackle this problem, as we did in the construction of closure and concatenation, we add extra bits to the alphabet and later project them away. Assume n is the maximum size of $reach(q)$ for all $q \in Q'$. We need at most $\lceil \log(n + 1) \rceil$ bits to be added to the alphabet so that the construction can result in a DFA. Let $P = \{q \mid q \in Q', reach(q) > 0\}$. Let $m = \lceil \log(n + 1) \rceil$, where n is the maximum size of $reach(q)$ for all $q \in P$. Let m_q be an m -bit string. For $\alpha \in B^k$, $\alpha m_q \in B^{k+m}$ is a string in which m_q is appended to α . Let m_0 be an m -bit string of 0s. We assume $\forall q, m_q \neq m_0$, and for any $q \in P, m'_q \neq m''_q$ if $q', q'' \in reach(q)$.

The construction of M'' depends on $L(M_3)$. We consider the following three cases: (1) M_3 only accepts single characters, i.e., $L(M_3) \subseteq \Sigma$, (2) M_3 only accepts words with more than one character, i.e., $L(M_3) \subseteq \Sigma^+ \setminus \Sigma$, (3) M_3 only accepts the empty string, i.e., $L(M_3) = \{\epsilon\}$.

Case 1: $\forall w \in L(M_3), |w| = 1$. $M'' = \langle Q', q'_0, \Sigma'', \delta'', F' \rangle$ is constructed as:

- $\Sigma'' \subseteq B^{k+m}$
- $\forall q \in Q', \delta''(q, \alpha m_0) = q'$, if $\delta'(q, \alpha 0) = q'$
- $\forall q \in P, \forall q' \in reach(p), \forall \alpha \in L(M_3), \delta''(q, \alpha m_{q'}) = q'$.

In Fig 4(a), $P = \{i, j\}$, $reach(i) = \{j, k\}$ and $reach(j) = k$. Let $L(M_3) = \{c\}$. M'' of our example is shown in Fig 4(b). Each symbol is appended with two extra bits, e.g., $\delta(i, c01) = j$ and $\delta(i, c10) = k$. $|M''|$ is bounded by $|M'|$.

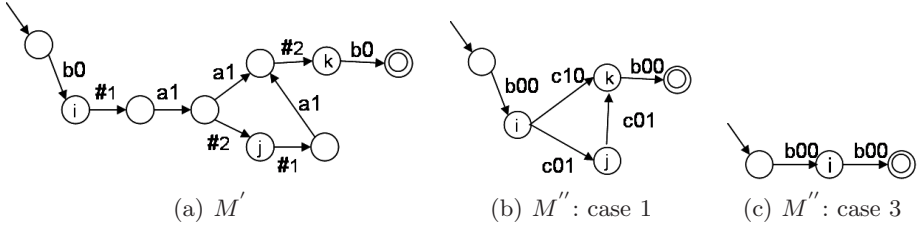


Fig. 4. Constructing M'' from M' . M' is the intersection of M_1' and M_2'

Case 2: $\forall w \in L(M_3), |w| \geq 2$. For each $p \in P$, we construct a copy of M_3 as $M_p = \langle Q_p, q_{p0}, \Sigma, \delta_p, F_p \rangle$. M'' is constructed by inserting M_p between p and $reach(p)$.

$M'' = \langle Q'', q''_0, \Sigma'', \delta'', F'' \rangle$, where

- $Q'' = Q' \cup_{p \in P} Q_p$
- $\Sigma'' \subseteq B^{k+m}$
- $\forall q \in Q', \delta''(q, \alpha m_0) = q'$, if $\delta'(q, \alpha 0) = q'$
- $\forall p \in P, \forall q \in Q_p, \delta''(q, \alpha m_0) = q'$, if $\delta_p(q, \alpha) = q'$.
- $\forall p \in P, \delta''(p, \alpha m_q) = q$, if $\delta_p(q_{p0}, \alpha) = q$.
- $\forall p \in P, \forall q \in reach(p), \delta''(q', \alpha m_0) = q$, if $\delta_p(q', \alpha) = q''$ and $F_p(q'') = +$.
- $\forall q \in Q', F''(q) = F'(q)$
- $\forall p \in P, q \in Q_p, F''(q) = -$.

In this case, $|M''|$ is bounded by $|M'| + |M'| \times |M'| \times |M_3|$.

Case 3: $\forall w \in L(M_3), |w| = 0$. We consider this case as *deletion*. Before we start the construction, it is worth to know that for deletion, one may change the argument M_2 to N , where $L(N) = L(M_2)^+$ (Kleene plus closure), and get the same result. We specify this property as follows.

Property 1. Let $M = REPLACE(M_1, M_2, M_3)$, and $M' = REPLACE(M_1, N, M_3)$, where $L(N) = L(M_2)^+$. $L(M) = L(M')$ if $L(M_3) = \{\epsilon\}$.

The correctness comes from the fact that, by construction, if there exists $w \in L(N)$, then there exists $k > 0$, $w = w_1 w_2 \dots w_k$, where $\forall 1 \leq i \leq k, w_i \in L(M_2)$. Since w or any w_i will be deleted after the replacement, using N instead of M_2 yields the same result.

Note that the \sharp -reachable states of M' using N is actually the set of reachable closure of the \sharp -reachable states of M' using M_2 . This facilitates our construction by taking all deleted pairs into account in one step. In the following construction, without loss of the generality, we assume that the matching strings are accepted by N . N can be constructed from the original M_2 by our closure operation.

M'' can then be constructed as $\langle Q', q'_0, \Sigma'', \delta'', F'' \rangle$, where

- $\Sigma'' \subseteq B^{k+m}$
- $\forall q \in Q', \delta''(q, \alpha m_0) = q'$, if $\delta'(q, \alpha 0) = q'$
- $\forall p \in P, \forall q \in reach(p), \delta''(p, \alpha m_{q'}) = q'$, if $\delta'(q, \alpha 0) = q'$.
- $\forall p \in P, F''(p) = +$, if $\exists q \in reach(p), F'(q) = +$.
- $F''(q) = F'(q)$, o.w.

Let $L(M_3) = \{\epsilon\}$. The result of M'' is shown in Fig 4(c). Note that if $M_2 = \{a\}$, we would get the same result. $|M''|$ is bounded by $|M'|$.

Finally, consider M_3 as a general DFA. REPLACE(M_1, M_2, M_3) can be constructed as the union of the results of the following three operations:

- REPLACE(M_1, M_2, M_{3_1}), where $L(M_{3_1}) = L(M_3) \cap \Sigma$
- REPLACE(M_1, M_2, M_{3_2}), where $L(M_{3_2}) = L(M_3) \cap \Sigma^+ \setminus \Sigma$
- REPLACE(M_1, M_2, M_{3_3}), where $L(M_{3_3}) = L(M_3) \cap \{\epsilon\}$

Our replacement operation is defined in a general case in terms of M_3 . For all replacement statements in PHP programs, such as `str_replace`, `preg_replace`, and `ereg_replace`, $L(M_3)$ is a constant string. In our implementation, we determine which type of construction to apply based on the length of this string.

Step 5: Finally, we get M over Σ by iteratively projecting away the extra bits. The subset construction is only applied when needed.

The final DFA $M = \text{REPLACE}(M_1, M_2, M_3)$, where $L(M_1) = \{baab\}$, $L(M_2) = a^+$, and $L(M_3) = \{c\}$, is shown in Fig 5. M accepts $\{bcb, bccb\}$.

In PHP programs, replacement operations such as `ereg_replace` can use different replacement semantics such as *longest match* or *first match*. Our replacement operation provides an over approximation of such more restricted replace semantics. For the example above, in the longest match semantics, M only accepts bcb , in which the longest match aa is replaced by c . In the first match semantics, M only accepts $bccb$, in which two matches a and a are replaced with c . Both of these are included in the result obtained by our replacement operation. This over approximation works well for our benchmarks, and does not raise false alarms. Indeed, we have observed that most statements we encountered yield the same result in the first and longest match semantics, e.g., `ereg_replace("<script *>", "", $_GET["username"]);`, and are precisely modelled by our language-based replacement operation.

4 Widening Automata

In this section, we describe the widening operator we use, which was originally proposed for arithmetic automata by Bartzis and Bultan [3].

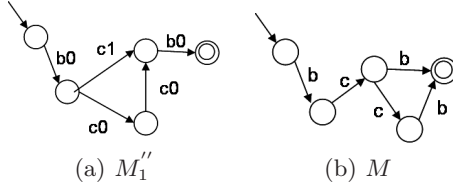


Fig. 5. M_1'' is PROJECT(M'' , $k + 2$), M is PROJECT(M_1'' , $k + 1$)

Given two finite automata $M = \langle Q, q_0, \Sigma, \delta, F \rangle$ and $M' = \langle Q', q'_0, \Sigma, \delta', F' \rangle$, we first define the binary relation \equiv_{∇} on $Q \cup Q'$ as follows. Given $q \in Q$ and $q' \in Q'$, we say that $q \equiv_{\nabla} q'$ and $q' \equiv_{\nabla} q$ if and only if

$$\forall w \in \Sigma^*. F(\delta^*(q, w)) = + \Leftrightarrow F(\delta'^*(q', w)) = +. \tag{1}$$

$$\text{or } q, q' \neq \text{sink} \wedge \exists w \in \Sigma^*. \delta^*(q_0, w) = q \wedge \delta'^*(q'_0, w) = q', \tag{2}$$

where $\delta^*(q, w)$ is defined as the state that M reaches after consuming w starting from state q . In other words, condition [1](#) states that $q \equiv_{\nabla} q'$ if $\forall w \in \Sigma^*$, w is accepted by M from q then w is accepted by M' from q' , and vice versa. Condition [2](#) states that $q \equiv_{\nabla} q'$ if $\exists w \in \Sigma$, M reaches state q and M' reaches state q' after consuming w from its initial state. For $q_1 \in Q$ and $q_2 \in Q$ we say that $q_1 \equiv_{\nabla} q_2$ if and only if

$$\exists q' \in Q'. q_1 \equiv_{\nabla} q' \wedge q_2 \equiv_{\nabla} q' \quad \vee \quad \exists q \in Q. q_1 \equiv_{\nabla} q \wedge q_2 \equiv_{\nabla} q \tag{3}$$

Similarly we can define $q'_1 \equiv_{\nabla} q'_2$ for $q'_1 \in Q'$ and $q'_2 \in Q'$.

It can be seen that \equiv_{∇} is an equivalence relation. Let C be the set of equivalence classes of \equiv_{∇} . We define $M \nabla M' = \langle Q'', q''_0, \Sigma, \delta'', F'' \rangle$ by:

$$\begin{aligned} Q'' &= C \\ q''_0 &= c \text{ s.t. } q_0 \in c \wedge q'_0 \in c \\ \delta''(c_i, \sigma) &= c_j \text{ s.t. } (\forall q \in c_i \cap Q. \delta(q, \sigma) \in c_j \vee \delta(q, \sigma) = \text{sink}) \wedge \\ &\quad (\forall q' \in c_i \cap Q'. \delta'(q', \sigma) \in c_j \vee \delta'(q', \sigma) = \text{sink}) \\ F''(c) &= + \text{ s.t. } \exists q \in F \cup F'. q \in c. \quad F''(c) = - \text{ o.w.} \end{aligned}$$

In other words, the set of states of $M \nabla M'$ is the set C of equivalence classes of \equiv_{∇} . Transitions are defined from the transitions of M and M' . The initial state is the class containing the initial states q_0 and q'_0 . The set of final states is the set of classes that contain some of the final states in F and F' . It can be shown that, given two automata M and M' , $L(M) \cup L(M') \subseteq L(M \nabla M')$ [3](#).

In Fig [6](#), we give an example for the widening operation. $L(M) = \{\epsilon, ab\}$ and $L(M') = \{\epsilon, ab, abab\}$. The set of equivalence classes is $C = \{q''_0, q''_1\}$, where $q''_0 = \{q_0, q'_0, q_2, q'_2, q_4\}$ and $q''_1 = \{q_1, q'_1, q_3\}$. $L(M \nabla M') = (ab)^*$.

As shown in Fig [1](#), we use this widening operator iteratively to compute an over-approximation of the least fixpoint that corresponds to the reachable values of string expressions. To simplify the discussion, let us assume a program with

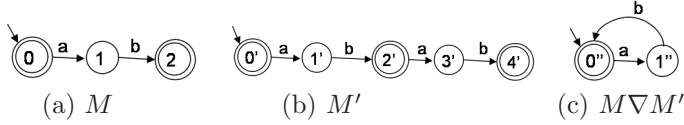


Fig. 6. Widening automata

a single string variable represented with one automaton M . Let M_i represent the automaton computed at the i^{th} iteration and let I denote the initial value of the string variable. The fixpoint computation will compute a sequence $M_0, M_1, \dots, M_i, \dots$, where $M_0 = I$ and $M_i = M_{i-1} \cup post(M_{i-1})$ where the post-condition for different statements is computed as described in Fig 4. We reach the least fixpoint M_j if at some iteration, $M_j = M_{j-1}$. Since we are dealing with an infinite state system, the computation may not converge. In the following, we use M_∞ to denote the least fixpoint.

Given the widening operator, we actually compute an sequence $M'_0, M'_1, \dots, M'_i, \dots$, that over-approximates the fixpoint computation where M'_i is defined as: $M'_0 = M_0$, and for $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}))$. Let M'_∞ denote the least fixpoint of this approximate sequence. Then we have the following result [3]:

Definition 1. $M_1 = \langle Q_1, q_{01}, \Sigma, \delta_1, F_1 \rangle$ is simulated by $M_2 = \langle Q_2, q_{02}, \Sigma, \delta_2, F_2 \rangle$ iff there exists a total function $f : Q_1 \setminus \{sink\} \rightarrow Q_2$ such that $\delta_1(q, \sigma) = sink$ or $f(\delta_1(q, \sigma)) = \delta_2(f(q), \sigma)$ for all $q \in Q_1 \setminus \{sink\}$ and $\sigma \in \Sigma$. Furthermore, $f(q_{01}) = q_{02}$ and for all $q \in F_1$, $f(q) \in F_2$.

Definition 2. $M = \langle Q, q_0, \Sigma, \delta, F \rangle$ is state-disjoint iff there is no state $q \in Q$ such that there exist $\alpha \in \Sigma$ and $q', q'' \in Q$, $q' \neq q''$, and $\delta(q', \alpha) = q$ and $\delta(q'', \alpha) = q$.

Theorem 1. If (1) M_∞ exists, (2) M_∞ is a state-disjoint automaton, and (3) M_0 is simulated by M_∞ , then (1) M'_∞ exists and (2) $M'_\infty = M_\infty$.

Consider a simple example where we start from an empty string and simply concatenate a substring ab at each iteration. The exact sequence $M_0, M_1, \dots, M_i, \dots$ will never converge to the least fixpoint, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$. However, M_∞ exists and $L(M_\infty) = (ab)^*$. In addition, M_∞ is a state-disjoint automaton, and M_0 is simulated by M_∞ . Based on Theorem 1, these conditions imply that once the computation of the approximate sequence reaches the fixpoint, the fixpoint is equal to M_∞ and the analysis is precise. Computation of the approximate sequence is shown in Fig 7. $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}, R))$, where $post(M)$ returns an automaton that accepts $\{wab \mid w \in L(M)\}$. In this case, we reach the fixpoint at the 3rd iteration and $M'_\infty = M_\infty = M'_3$.

A more general case that we commonly encounter in real programs is that we start from a set of initial strings (accepted by M_{init}), and concatenate an arbitrary but fixed set of strings (accepted by M_{tail}) at each iteration. Based on

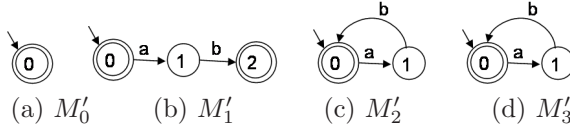


Fig. 7. An approximate sequence

Theorem 1 one can conclude that if the DFA M that accepts $L(M_{init})L(M_{tail})^*$ is state-disjoint, then our analysis via widening will reach the precise least fix-point when it terminates.

5 Experiments

We experimented with our string analysis tool on a number of test cases extracted from a set of real-world, open source applications: `MyEasyMarket-4.1` (a shopping cart program), `PBLguestbook-1.32` (a guestbook application), `Aphpkb-0.71` (a knowledge base management system), `BloggIT-1.0` (a blog engine), and `proManager-0.72` (a project management system). We believe that these programs are representative of how web applications use regular expression based replacement functions to modify their input (in particular, in a security context, to perform input sanitization), and, thus, are good test cases for our technique. These vulnerable functions were identified and sanitized by Balzarotti et al. in [2, 11].

Table 5 shows the results of applying our string analysis tool to these programs. The first column of Table 5 identifies the application, the function that was analyzed and the line number for the vulnerable operation. For each test case we analyzed the original version of the program (that contained the vulnerability) and a modified version which was modified with the intention of fixing the vulnerability. Our analysis is quite efficient and takes a couple of seconds. Since our string analysis tool is sound, it identifies the existing vulnerabilities correctly in each case. However, since our conservative approximations can lead to false positives, the fact that our tool identifies a possible vulnerability does not mean that it is guaranteed to be a vulnerability.

The impressive part of our results is that for all the modified program segments our approach is able to prove that the sanitization is correct. This indicates that the approximations we use work quite well in real-world applications.

We also experimented with Saner [1] to check these benchmarks. We discuss this tool in related work. The results are shown in table 5. Our tool performs slightly better than Saner in terms of time. It is interesting to note that there are some conflicts on the verification results. Saner performs bounded verification and approximates the value of out of bound computation as arbitrary strings. This rough approximation raises a false alarm while checking the sanitized version of `PBLguestbook-1.32(1210)`. While checking `BloggIT-1.0`, Saner, in the default configuration, assumes that data from the database are sanitized; while we assume that these data may be tainted and model them the same as data from

Table 1. Experimental results. Application: name of the application and the checked program point. Version: o-original, m-modified. Res.: y-the intersection of attack strings is not empty (vulnerable), n-the intersection of attack strings is empty (secure). Final DFA is the minimized DFA at the checked program point, and Peak DFA is the largest DFA observed during the fixpoint iteration. state: number of states. bdd: number of bdd nodes. n: number of warnings raised by Saner. type:(1) xss - cross site scripting vulnerability, (2) sql - SQL injection vulnerability, (3) reg - regular expression error.

Application File(line)	Ver.	Res.	Final DFA state(bdd)	Peak DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Saner n(type)	Saner Time(sec)
MyEasyMarket-4.1 trans.php(218)	o	y	17(133)	17(148)	0.010+0.002	444	1(xss)	1.173
	m	n	17(132)	17(147)	0.009+0.001	451	0	1.139
PBLguestbook-1.32 pblguestbook.php(1210)	o	y	42(329)	42(376)	0.019+0.001	490	1(sql)	1.264
	m	n	49(329)	42(376)	0.016+0.002	626	1(sql)	1.665
PBLguestbook-1.32 pblguestbook.php(182)	o	y	842(6749)	842(7589)	2.57+0.061	13310	1(reg)	4.618
	m	n	774(6192)	740(6674)	1.221+0.007	8184	1(reg)	4.331
Aphpkb-0.71 saa.php(87)	o	y	27(219)	289(2637)	0.045+0.003	2436	1(xss)	1.220
	m	n	18(157)	1324(15435)	0.177+0.009	11388	0	1.622
BloggIT 1.0 admin.php(23,25,27)	o	y	79(633)	79(710)	0.499+0.002	3569	0	0.558
	o	y	126(999)	126(1123)				
	o	y	138(1095)	138(1231)				
	m	n	79(637)	93(1026)	0.391+0.006	5820	0	0.559
proManager-0.72 message.php(91)	o	y	387(3166)	2697(29907)	1.771+0.042	13900	1(xss)	6.980
	m	n	423(3470)	2697(29907)	2.091+0.051	19353	0	7.201

users. Saner raises an error for the sanitization routine in PBLguestbook-1.32(182) since it does not support the syntax of the replace operator used in that routine.

6 Related Work

Due to its importance in security, string analysis has been widely studied. Christensen, Møller and Schwartzbach [7] proposed a grammar-based string analysis (implemented in a tool called JSA) to statically determine the values of string expressions in Java programs. They convert the flow graph into a context free grammar where each string variable corresponds to a nonterminal, and each string operation corresponds to a production rule. Then, they convert this grammar to a regular language by computing an over-approximation. Gould et al. [11] use this grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications [7]. Christodorescu et al. [8] present an implementation of the grammar-based string analysis technique for executable programs for the x86 architecture. Minamide [13] supports string-based replacement operations by escaping replace operations to finite-state transducers, and describes a string analysis similar to JSA to statically detect cross-site scripting vulnerabilities and to validate pages generated by web applications written in the PHP language. Wassermann et al. [18] proposed a static analysis to detect SQL injections following Minamide [13]. There are some other tools for string analysis [19,6,15,9]. Shannon et al. [15] propose forward bounded symbolic execution to perform string

analysis on Java programs. Similar to our approach, automata are used to trace path constraints and encode the values of string variables. They support trim and substring operations. Xie and Aiken [19] support string assignment and validation operations. Fu et al. [9] and Choi et al. [6] support string-based replacement (as opposed to language-based replacement). None of the tools mentioned above addresses language-based replacement operations. This defect causes the approximations computed by these tools to be too coarse for some input sanitization routines.

Language-based replacement has been discussed in computational linguistics [12,14,10,17]. These algorithms are based on the composition of finite state transducers. By composing specific transducers, constraints like longest match and first match can be precisely modeled. However, each composition may result in a quadratic size of non-deterministic automaton, and is more likely to blow-up compared to our construction. The transducer-based replacement function [14] has been implemented in Finite State Automata utilities (FSA) [16], where automata are stored and manipulated using an explicit representation. We use a symbolic DFA representation based on MBDDs. This symbolic encoding enables us to perform complex automata operations, such as closure, concatenation, replace, and widening, efficiently using the MBDDs.

Balzarotti et al. [1] combine both dynamic and static techniques to verify PHP programs. They support language-based replacement by incorporating FSA [16], but they only support bounded computation for loops and approximate variables updated in a loop as arbitrary strings once the computation does not converge within a fixed bound. We incorporate the widening operator in [3] to tackle this problem and obtain a tighter approximation that enables us to verify a larger set of programs.

Choi et al. [6] also investigates a widening method to analyze strings. The widening operator is defined on strings and the widening of a set of strings is achieved by applying the widening operator pairwise to each string pair. The widening operator we use is defined on automata, and was originally proposed for arithmetic constraints [3]. The intuition behind this widening operator is applicable to any symbolic fixpoint computation that uses automata. In [3] it is proved that for a restricted class of systems the widening operator computes the precise fixpoint and we extend this result to our analysis. Moreover, in our experiments, the over-approximation computed by this widening operator works well to prove the properties we were interested in.

Finally, the use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [5]). In this paper we focus on verification of string manipulation operations in PHP programs.

7 Conclusion

We proposed a symbolic approach for string verification on PHP programs. Our approach computes a conservative approximation of the set of values that a string variable can take at a given program point. We use a symbolic automata

representation based on MBDDs and implement the string operations such as concatenation and replacement on this symbolic representation. Our experiments demonstrate that the proposed string analysis technique is capable of verifying the correctness of string sanitization operations in real-world PHP programs.

References

1. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Proc. Symposium on Security and Privacy (2008)
2. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: Multi-module vulnerability analysis of web-based applications. In: Proc. 14th ACM conference on Computer and communications security, pp. 25–35. ACM, New York (2007)
3. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Proc. 16th International Conference on Computer Aided Verification, pp. 321–333 (2004)
4. Biehl, M., Klarlund, N., Rauhe, T.: Algorithms for guided tree automata. In: Raymond, D.R., Yu, S., Wood, D. (eds.) WIA 1996. LNCS, vol. 1260. Springer, Heidelberg (1997)
5. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proc. 12th International Conference on Computer Aided Verification, pp. 403–418 (2000)
6. Choi, T.-H., Lee, O., Kim, H., Doh, K.-G.: A practical string analyzer by the widening approach. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 374–388. Springer, Heidelberg (2006)
7. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
8. Christodorescu, M., Kidd, N., Goh, W.-H.: String analysis for x86 binaries. In: Proc. 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005), September 2005, ACM Press, New York (2005)
9. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting sql injection vulnerabilities. In: Proc. 31st Annual International Computer Software and Applications Conference. COMPSAC 2007, Washington, DC, USA, vol. 1, pp. 87–96. IEEE Computer Society, Los Alamitos (2007)
10. Gerdemann, D., van Noord, G.: Transducers from rewrite rules with backreferences. In: Proc. 9th Conference of the European Chapter of the Association for Computational Linguistics, pp. 126–133 (1999)
11. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: Proc. 26th International Conference on Software Engineering, pp. 645–654 (2004)
12. Karttunen, L.: The replace operator. In: Proc. 33rd annual meeting on Association for Computational Linguistics, pp. 16–23 (1995)
13. Minamide, Y.: Static approximation of dynamically generated web pages. In: Proc. 14th International World Wide Web Conference, pp. 432–441 (2005)
14. Mohri, M., Sproat, R.: An efficient compiler for weighted rewrite rules. In: Proc. 34th annual meeting on Association for Computational Linguistics, pp. 231–238. Association for Computational Linguistics (1996)

15. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Proc. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, Washington, DC, USA, pp. 13–22. IEEE Computer Society, Los Alamitos (2007)
16. van Noord, G.: FSA utilities toolbox, <http://odur.let.rug.nl/~vannoord/Fsa/>
17. van Noord, G., Gerdemann, D.: An extendible regular expression compiler for finite-state approaches in natural language processing. In: Proc. of the 4th International Workshop on Implementing Automata (WIA), July 1999, pp. 122–139. Springer, Heidelberg (1999)
18. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 32–41 (2007)
19. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proc. 15th conference on USENIX Security Symposium, Berkeley, CA, USA, p. 13. USENIX Association (2006)

Verifying Multi-threaded C Programs with SPIN

Anna Zaks and Rajeev Joshi*

¹ New York University

² Lab for Reliable Software, Jet Propulsion Laboratory

Abstract. A key challenge in model checking software is the difficulty of verifying properties of implementation code, as opposed to checking an abstract algorithmic description. We describe a tool for verifying multi-threaded C programs that uses the SPIN model checker. Our tool works by compiling a multi-threaded C program into a typed bytecode format, and then using a virtual machine that interprets the bytecode and computes new program states under the direction of SPIN. Our virtual machine is compatible with most of SPIN's search options and optimization flags, such as bitstate hashing and multi-core checking. It provides support for dynamic memory allocation (the `malloc` and `free` family of functions), and for the `pthread` library, which provides primitives often used by multi-threaded C programs. A feature of our approach is that it can check code *after* compiler optimizations, which can sometimes introduce race conditions. We describe how our tool addresses the state space explosion problem by allowing users to define data abstraction functions and to constrain the number of allowed context switches. We also describe a reduction method that reduces context switches using dynamic knowledge computed on-the-fly, while being sound for both safety and liveness properties. Finally, we present initial experimental results with our tool on some small examples.

1 Introduction

A key challenge in applying model checking to software is the difficulty of verifying properties of implementation code, as opposed to checking abstract algorithmic descriptions. Even well understood protocols such as Peterson's protocol for mutual exclusion, whose algorithmic description takes only half a page, have published implementations that are erroneous. This problem is especially acute in our domain of interest – small, real-time embedded systems in use on robotic spacecraft – where limits on memory and processor speeds require implementation ingenuity, but where the risks of failure are high.

* The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was also provided by the NASA ESAS 6G project on Reliable Software Engineering.

Our work extends previous work on *model-driven verification*, in which model checking was applied to the verification of *sequential C* programs [HJ04], [GJ08]. Model-driven verification is a form of software model checking for C programs that works by executing C code embedded in a PROMELA model. The SPIN model checker [Hol03] translates a PROMELA model (along with an LTL property to be checked) into a C program `pan.c` that encodes a model checker that checks the property in question (in a sense, therefore, SPIN is really a *model checker generator*). Because SPIN compiles models into C code, recent versions (since SPIN 4.0) allow fragments of C programs to be embedded within PROMELA models. Each such fragment is treated as a deterministic atomic transition (in SPIN parlance, the equivalent of a “dstep”). This allows SPIN to be used to check C programs against LTL specifications using most¹ of SPIN’s search options, including bitstate hashing, and multi-core execution [HB07].

A significant limitation of model-driven verification is that each fragment of embedded C code is executed as an atomic transition by SPIN. This in turn means that it is hard to (a) check properties (such as assertions and invariants) at control points *within* the embedded C code, (b) interrupt the control flow within a C function (to simulate, say, a device interrupt or an asynchronous reset), and (c) explore interleavings of more than one fragment of C code, which is needed in order to check multi-threaded C programs. A discussion of how to address the first two limitations appears elsewhere [GJ08]; in this paper, we address the third limitation of checking multi-threaded C programs. We describe a tool (named “pancam”) which implements a virtual machine for executing programs in the LLVM bytecode language [LA04]. Since the state space of even a small C program is typically much larger than that of most PROMELA models, we consider various approaches to combat the space explosion problem. In particular, we describe a technique called *superstep reduction* that can increase the granularity of atomic steps on-the-fly during a model checking run. We also discuss how context-bounding [MQ07] can easily be integrated with our tool, with only a small modification to our virtual machine, and how pancam allows users to define data abstractions to reduce state space still further. Finally, we present initial experimental results with using our tool on some small multi-threaded C programs.

2 Model Checking C Programs with Pancam

Our approach to checking a concurrent C program with SPIN is to first translate the program into bytecode for the Low Level Virtual Machine (LLVM) compiler infrastructure [LA04]. This bytecode is then checked by executing it within the context of an explicit-state model checker by using a virtual machine interpreter. In a sense, this approach is similar to Java Pathfinder (JPF) [VHB⁺03]. However,

¹ Two key options not supported for embedded code by SPIN are breadth-first search, which would require too much additional overhead, and partial-order reduction, which is difficult for C programs because computing a nontrivial independency relation is hard.

```

#include <pthread.h>
struct pa_desc {
    volatile int *f0, *f1 ;
    int last ;
} ;
...
volatile int pa_f0, pa_f1, pa_last ;
...
void pa_desc_lock(struct
pa_desc *d) {
    for (*d->f0=1, pa_last=d->last;
        *d->f1==1 && pa_last==d->last;
        ) ;
}
...
int count = 0 ;
void threadx_critical(void) {
    count++ ;
    ... // critical section
    count-- ;
}

void * thread1_main(void *args) {
    struct pa_desc d ;
    pa_desc_init(&d, 1) ;
    for (;;) {
        pa_desc_lock(&d) ;
        threadx_critical() ;
        pa_desc_unlock(&d) ;
    }
    return NULL ; /* NOT REACHED */
}

/* pancam helpers */

void init(void) {
    pa_f0 = pa_f1 = pa_last = 0 ;
}
Bool check_exclusion(void) {
    return (count <= 1) ;
}

```

Fig. 1. Excerpt of C implementation of Peterson’s Algorithm using pthreads, from the Wikipedia. The two highlighted occurrences of `volatile` were missing, causing a potential race condition.

unlike JPF, we do not integrate the model checker with the bytecode interpreter. Instead, our virtual machine executes bytecode as directed by SPIN, by providing a method `pan_step(i)` that is called by SPIN to execute the next transition of thread `i`. In a sense, therefore, SPIN *orchestrates* the search by deciding which thread to execute next, by storing visited states in its hash table, and by restoring a previous state during a backtracking step. This division of labor allows us to freely benefit from SPIN’s unique abilities, notably its scalability, search heuristics, and, lately, the capability to deploy it on multi-core CPUs [HB07]. The C language does not have any built-in primitives for concurrency, so our framework provides support for the constructs from the standard pthreads library such as mutexes and condition variables. Even though the dynamic thread creation is not yet fully implemented in pancam, the extension can be organically incorporated into the framework. The only limitation would be on the total number of threads, which should not exceed 255 (the bound imposed by SPIN).

To illustrate how our tool works, Fig. 1 shows the C program that appears in the Wikipedia entry² for Peterson’s mutual exclusion protocol [wik]. The property to be checked is mutual exclusion, which is defined by the boolean valued function `check_exclusion`.

² To simplify the statement of the mutual exclusion property, we have added an additional variable `count` as shown.

Our tool first compiles this program into LLVM bytecode, using the `llvm-gcc` compiler (an extension of the GNU `gcc` compiler that generates LLVM bytecode). LLVM bytecode is like typed assembly language; a sample appears in Fig. 2, which shows the bytecode corresponding to the `pa_desc_lock` function shown in Fig. 1.

```

define void @pa_desc_lock(%struct.pa_desc* %d) {
entry:
    %tmp1 = getelementptr %struct.pa_desc* %d, i32 0, i32 0
    %tmp2 = load i32** %tmp1
    volatile store i32 1, i32* %tmp2
    %tmp4 = getelementptr %struct.pa_desc* %d, i32 0, i32 2
    %tmp5 = volatile load i32* %tmp4
    volatile store i32 %tmp5, i32* @pa_last
    %tmp8 = getelementptr %struct.pa_desc* %d, i32 0, i32 1
    %tmp9 = load i32** %tmp8
    br label %bb6

bb6:
    %tmp10 = volatile load i32* %tmp9
    %tmp11 = icmp eq i32 %tmp10, 1
    br i1 %tmp11, label %cond_next, label %return

cond_next:
    %tmp15 = volatile load i32* %tmp4
    %tmp16 = volatile load i32* @pa_last
    %tmp17 = icmp eq i32 %tmp15, %tmp16
    br i1 %tmp17, label %bb6, label %return

return:
    ret void
}

```

Fig. 2. LLVM bytecode for function `pa_desc_lock`

To check the C code for Peterson’s algorithm with our tool, we use a PROMELA model to make appropriate calls to schedule the threads via our virtual machine. Fig. 3 shows a SPIN model for checking the program in Fig. 1. The `c_decl` primitive is used to declare external C types and data objects that are used in the embedded C code. For simplicity, we assume the declarations needed by our model are in the header file `pancam_peterson.h`. Next, the `c_track` declarations are *tracking* statements, which are discussed below. The PROMELA process `init` defines the initialization steps for the SPIN model: as shown, they consist of initializing the interpreter (by calling `pan_setup()`), registering an invariant (defined by the C function `check_exclusion`) with the interpreter, performing one-time initialization of the C program (`pan_run_function()`), creating and starting the threads, and, finally, starting one PROMELA process for each thread. As shown, each PROMELA process then consists of repeatedly executing a single step of the associated thread (by calling `pan_step()`) provided that the thread is enabled.

The `c_track` declarations provide the essential ingredient that allows us to use the SPIN model checking engine in conjunction with our interpreter. During

```

c_decl {
  \#include "pancam_peterson.h"
}
c_track "csbuf" "CS_SIZE" "Matched";
init() {
  c_code {
    pan_setup() ;
    pan_invariant("check_exclusion") ;
    pan_run_function("init") ;
    pan_start_thread(0,
      "thread0_main", NULL) ;
    pan_start_thread(1,
      "thread1_main", NULL) ;
  } ;
  run thread0() ;
  run thread1()
}
}

proctype thread0() {
  do
    :: c_expr{pan_enabled(0)}
      -> c_code{pan_step(0);}
  od
}
proctype thread1() {
  do
    :: c_expr{pan_enabled(1)}
      -> c_code{pan_step(1);}
  od
}

```

Fig. 3. Spin driver for executing pancam on Peterson’s Algorithm

its depth first search³, whenever SPIN reaches a state with no new successors, it backtracks to the most recent state that has not been fully explored. For PROMELA variables, restoration of earlier values when backtracking is automatic, since they are stored in the state vector maintained by SPIN. However, the state of the pancam virtual machine is not part of the PROMELA model. Thus the model checker needs explicit knowledge of the region of memory where this state is stored, so that it can copy and restore this memory during its backtracking search. This knowledge is provided through the `c_track` declarations. In our framework, the bytecode interpreter maintains its state in a single contiguous region of memory starting at address `csbuf` and occupying `CS_SIZE` bytes; this corresponds to the `c_track` declaration shown in the figure.

In using our tool to verify the Wikipedia C implementation of Peterson’s protocol, we discovered a bug in the implementation. The bug is interesting because it manifests itself when the code is compiled with optimization enabled. The problem arose from the fact that certain global variables were not originally marked as `volatile` (as indicated by the shaded keywords in Fig. 1). As a result, the optimized bytecode reused stale values read earlier. For example, in the procedure `pa_desc_lock` from Fig. 2, all the instructions that occur after the second store were removed, leading to scenarios where mutual exclusion was violated. We have since fixed the Wikipedia entry.

3 Addressing State Space Explosion

Not surprisingly, the biggest challenge in using a tool such as ours is the problem of state space explosion. Even though our main interest is in checking small

³ SPIN currently supports execution of embedded C code only when using depth first search mode.

embedded C programs, the typical state vectors we encounter are much larger (of the order of hundreds or even thousands of bytes) as compared to typical PROMELA models (whose state vectors are smaller by one or two orders of magnitude). In addition, because a single line of C may translate into many steps of bytecode, a naive exploration of all interleavings of a set of threads would quickly make even the smallest of problems intractable. To address these issues, pancam uses three techniques: (a) it allows users to provide data abstraction functions, (b) it provides the ability for the user to enforce context-switch bounding (see below), and (c) it employs an algorithm that performs a kind of partial order reduction on-the-fly to reduce the number of context switches without losing soundness of checking. We describe the first two of these techniques in the rest of this section; our reduction method is described in Section 4.

3.1 Abstraction

The ability of our tool to support abstractions is derived from the distinction between *tracked* and *matched* objects in SPIN. As discussed in Section 2, a tracked data object is stored on the stack used by SPIN's depth first search (DFS), so that an earlier state of that object can be restored on each backtracking step during the DFS. In almost all cases⁴, any data that changes during an execution should be tracked. A matched object, on the other hand, is one that is part of the *state descriptor* that SPIN uses to determine if a state has been seen before. By declaring an object to be tracked but not matched, we can therefore exclude it from the state descriptor. Support for this is provided by the "Matched" and "UnMatched" keywords in SPIN. (These keywords were introduced in SPIN version 4.1.)

The ability to separate tracked and matched data allows us to use data abstraction to reduce the size of the state space [HJ04]. A simple but effective scheme is to define a new auxiliary variable `abs` for storing the abstract state, and provide a function `update_abs()` which updates the value of `abs` based on the (current) values of the concrete program variables. Then, to make SPIN search the abstract state space, we declare all concrete program variables as tracked but "UnMatched", and declare the abstraction variable `abs` as tracked and "Matched", and we ensure that the function `update_abs()` is called after every transition that changes concrete state.

Our tool supports this scheme for data abstraction by providing a buffer `abs`. The user provides the function `update_abs`, which computes the data abstraction and writes it to the buffer. Our tool ensures that this function is invoked if any of the variables that appear in the body of this function changes during a transition.

3.2 Context-Bounded Checking

The idea in context-bounded model checking [QR05, MQ07, MQ08] is to avoid state space explosion in multi-threaded programs by enforcing an upper bound on

⁴ There are valid reasons for not tracking certain data even though it changes during an execution [GJ08]; see Section 4.2.

```

c_decl {
  int last_proc = -1 ;
  int nswitch = 0 ;
  int MAX_SWITCH = -1 ;

  Bool pan_enabled_cb(int p) {
    int i ;

    if (!pan_enabled(p)) /* thread p is disabled */
      return FALSE ;

    if (last_proc == p) /* no context switch */
      return TRUE ;

    /* Check if bound not specified, or not reached */
    if ((MAX_SWITCH < 0) || (nswitch < MAX_SWITCH))
      return TRUE ;

    /* Check if any other thread is enabled */
    for (i=0 ; i<ThreadCount ; i++)
      if ((i != p) && pan_enabled(i))
        return FALSE ;

    /* all other threads are disabled, so don't preempt */
    return TRUE ;
  }

  c_track "&nswitch"      "sizeof(int)"  "UnMatched";
  c_track "&last_proc"    "sizeof(int)"  "UnMatched" ;
}

```

Fig. 4. Code for implementing context bounding with pancam

the number of allowed preemptive context switches. A context switch from process p to process q is called preemptive if process p is enabled (and could therefore continue execution if the context switch did not occur). Experience with context-bounded model checking suggests that, in most cases, errors in multi-threaded software typically have shortest counterexample traces that require only a small number of context switches [MQ07]. Thus exhaustive exploration of runs with a small budget of allowed context switches has a good chance of finding errors.

To extend our tool with support for context-bounded search, we change the top-level SPIN model that orchestrates the run by replacing calls to `pan_enabled(p)` (which check if thread p is enabled) by calls to the function `pan_enabled_cb(p)` (which additionally checks the condition for context-bounding). Fig. 4 shows the C code for the function `pan_enabled_cb`. As shown, we add two additional integers `last_proc` and `nswitch` to the state space (but note that these variables are only tracked, and not matched). It is not hard to show that by using it to replace the original `pan_enabled` function, (and by appropriately updating `last_proc` and `nswitch` whenever a thread is executed) we achieve the desired effect of limiting the number of preemptive context switches to the user-provided bound of `MAX_SWITCH`.

4 On-the-fly Superstep Reduction

As described in Section 2, our tool uses a SPIN model to orchestrate the state space search by choosing, at each step, a thread to execute, and executing its next

transition by invoking the virtual machine. An exhaustive search along these lines would require exploring all possible interleavings of the threads in the program, which is intractable for all but the smallest of programs. A common technique used to deal with the problem is partial order reduction [Pel93, CGP00]. Intuitively, partial order reduction works by reducing the number of context switches, exploiting the fact that transitions in different threads are often independent (in the sense that the order in which they occur does not affect visible program behavior).

Most partial order methods described in the literature are *static* in the sense that they determine independence of transitions by analyzing program text. Such analyses are, however, not terribly effective with C programs, and typically allow only very simple and conservative independence relations to be computed. For C programs, therefore, it is more instructive to look at *dynamic* partial order reduction methods [FG05, GFYS07], in which independence relationships are computed dynamically, during a model checking run. For example, one of the simplest approaches to dynamic partial order reduction is to only allow a context switch after an update or an access to a global memory location.

In the context of our tool, however, there is one additional complication caused by the fact that the model checking engine (SPIN) treats the model as having a single transition (denoted by the function `pan_step`). In particular, this means that support for partial order reduction therefore requires either exposing additional pancam state (which would require modification of SPIN, which we hope to avoid), or for the reduction to be implemented entirely within pancam. We adopt the latter strategy. Pancam performs partial order reduction on the state space by allowing a thread i to execute a sequence of more than one instruction as part of a single SPIN transition from a state s . We refer to such a sequence of instructions as a “superstep” and denote it by the notation A_i^s . Since the model checker only sees the first and last states of a superstep, the intermediate states are hidden from the model checker, which in turn reduces the number of interleavings to be explored (and therefore the number of states and transitions).

Of course, as with traditional partial order reduction, there are certain conditions that must be satisfied by such supersteps in order to preserve soundness of model checking. In the next subsection, we describe a set of conditions under which we can preserve the soundness of next-time free LTL properties.

4.1 Correctness of Superstep Reduction

For convenience, we consider programs with k deterministic threads (or processes), where the only source of nondeterminism comes from thread scheduling. We also assume that each instruction can access at most one global memory location. This assumption is safe to make about the LLVM bytecode, which uses designated instructions *store* and *load* to access memory.

We say that two transitions α and β are *independent* if neither enables nor disables the other, and for any state, execution of α followed by execution of β results in the same state as execution of β followed by α . We say that two transitions *conflict* if both access a common memory location and at least one

of them is a write. Under the assumption that one thread may enable or disable another only by means of mutexes, which are a type of a shared object, the absence of a conflict between transitions implies independence as long as the transitions do not belong to the same thread.

Claim 1. *The soundness and completeness of next-time free LTL model checking is preserved as long as for every thread i enabled in state s , superstep sequence A_i^s satisfies the following three requirements:*

1. **Superstep Size.** A_i^s must be finite and contain at least one transition. The check for finiteness can be implemented conservatively by setting an upper bound on the number of transitions in a superstep sequence or the number of loop heads within the sequence.
2. **Independenc.** Only the very last transition of the path A_i^s conflicts with any of the transitions in A_k^s for any thread $k \neq i$.
3. **Visibility.** At most one transition which changes the value of any of the atomic propositions is allowed in A_i^s . If exists, it must be the very last transition of the superstep sequence.

The formal proof of the claim is similar to the one presented in [CGP00] and is beyond the scope of this paper. Here we only present some intuition about the correctness of the superstep reduction.

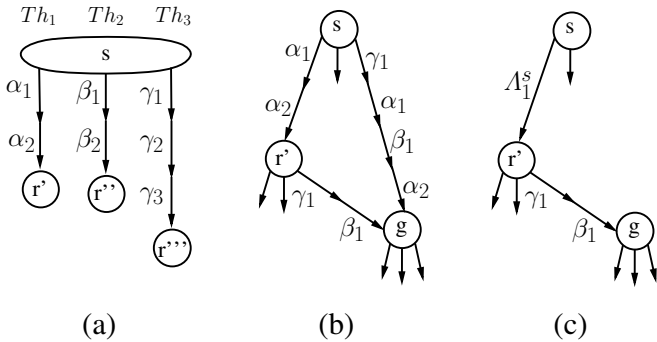


Fig. 5. Superstep POR

All the paths outgoing from a state s can be partitioned into sets, where each set is covered by one of the superstep sequences as following. If, on the path θ^s , all the transitions of the superstep sequence A_i^s precede the last transitions of the superstep sequences that correspond to all the other threads, θ^s is said to be covered by A_i^s . Consider the example in Fig. 5(a) that depicts the superstep sequences of the three program threads (Th_1 , Th_2 , and Th_3) from the program state s . In Fig. 5(b), the transitions $\gamma_1, \alpha_1, \beta_1, \alpha_2$ form the prefix of a number of the program paths outgoing from state s . All these paths correspond to the

program runs in which, from the state s , the threads are scheduled in the following order. First, one transition of Th_3 is scheduled, followed by a transition from Th_1 , a transition from Th_2 , and another transition from Th_1 . We say that all these paths are covered by the superstep sequence of Th_1 since α_2 occurs before β_2 and γ_3 on each of the paths. At each state s , the superstep reduction prunes away all the program paths which do not have a superstep sequence as a prefix and substitutes the superstep sequence with just one summary transition as shown in Fig. 5(c).

Let Θ^s be the minimal prefix of θ^s such that it contains all the transitions of Λ_i^s . Then, all the states reachable after following path Θ^s can also be reached after following Λ_i^s due to the fact that all the transitions of Θ^s which are not in Λ_i^s do not conflict with the transitions in Λ_i^s and can be commuted out. Going back to our example, since transitions β_1 and γ_1 do not conflict with α_2 , the state g , reachable by following transitions $\gamma_1, \alpha_1, \beta_1, \alpha_2$, can also be reached by following the superstep sequence α_1, α_2 .

It only remains to show that the intermediate states of the paths Θ^s and Λ_i^s do not have to be exposed to a model checking algorithm. The paths are finite; and, by definition of Θ^s , its last transition is equal to the last transition of Λ_i^s . Following the visibility requirement, only the very last transition of Λ_i^s and, consequently, only the very last transition of Θ^s may change the values of the predicates participating in the LTL property being checked. Thus, all the states on the paths except for the very last ones are indistinguishable from the state s . Moreover, since the transitions of Θ^s that do not occur in Λ_i^s do not modify the values of the predicates, the new values of the predicates in the last state of Θ^s are the same as in the last state of Λ_i^s . Thus, if a transition changes one of the predicates, it will always be visible to the model checker. In the example on Fig. 5 only the transition α_2 can be visible. All the states can be partitioned into two groups depending on the values of the predicates: the states undistinguishable from the state s and the states undistinguishable from the state g .

Notice that the listed requirements are general enough to allow for different choices of superstep sequences. However, as long as they are satisfied, the soundness and completeness of LTL model checking is preserved.

4.2 Implementation of Superstep Reduction in Pancam

Next, we describe how superstep reduction is implemented as part of our tool, which piggybacks the nested depth-first search algorithm used by SPIN. One of the attractions of using SPIN's nested depth-first search is that, unlike the case with breadth-first search [GFYS07], our implementation is fully compatible with checking of liveness properties. (And, although, we do not describe it here, our method can be straightforwardly extended to cooperate with breadth-first search, if desired.)

During its state exploration, SPIN issues calls to `pan_step(i)`, which, given the current state s , computes the state s' obtained by executing one or more instructions of thread i . The executed instructions form the superstep sequence Λ_i^s . The superstep size requirement guarantees that at least one instruction would

```

ConflictType = { CONTINUE, POST_STOP, PRE_STOP }

pan_step(ThreadID i) {
    superstep_length = 0;

    if (not backtracking) {
        init_independence_tester();
    }

    while (true) {
        tri = get_next_instruction(i);
        ConflictType error = test_for_independence(i, tri);
        if (error == PRE_STOP) break;
        execute_instruction(tri);
        superstep_length++;
        if (superstep_length ≥ MAX_SUPERSTEP_LENGTH) break;
        if (error == POST_STOP) break;
        if (is_proposition_modifying(tri)) break;
    } }

```

Fig. 6. Pseudocode of `pan_step` with superstep reduction

be executed; consequently, unless there is a loop in the state space, $s \neq s'$. Due to the nature of depth-first search, `pan_step` will be called multiple times on the same state s . In particular, after exploring the state space in which thread i is executed from state s , SPIN backtracks and attempts to execute the thread $i+1$ from the same state s in response to which `pancam` computes A_{i+1}^s .

The pseudocode of `pan_step` is presented in Fig. 6. If the state s is visited by the depth-first search for the very first time, `pan_step` executes initialization routines. Further, each time SPIN calls `pan_step(i)`, we compute the superstep sequence for thread i by interpreting the enabled instructions of thread i one by one. On each iteration, we check that addition of the corresponding instruction to the sequence does not violate any of the requirements stated above (in practice, the checks are only required for the instructions that access a global program location).

The most non-trivial check is the verification of the independence condition for which one could use various static and dynamic methods. Fig. 7 presents the dynamic independence check employed by `pancam`. Due to the nature of the independence requirement, the superstep of one thread depends on the transitions that constitute the supersteps of the other threads. An eager approach to this problem is to compute the supersteps for every thread the very first time the state s is visited (with the request to take step on thread one) and use the pre-computed supersteps on all the subsequent visits to the same state (when SPIN backtracks to take step on the other threads). However, this solution leads to inefficiencies since computing the supersteps effectively entails computation of the successors of the state s . Storing the successor states along with the current state leads to a large space overhead. Recomputing the successor states, on the other hand, would impair the running time.


```

init_independence_tester() {
  for (every enabled thread k) {
    AccessTableks.add( get_access_pair( get_next_instruction(k) ) );
  }
}
test_for_independence(ThreadID i, Instruction tri) {
  ai = get_access_pair(tri);
  for ( all threads k : k ≠ i ) {
    for (all ak ∈ AccessTableks) {
      if (conflict(ai, ak)) {
        if (ak ≠ last_of(AccessTableks)) {
          return PRE_STOP;
        } else {
          if (tri ≠ first_of(Λis)) AccessTableis.add(ai);
          return POST_STOP;
        }
      }
    }
  }
  if (tri ≠ first_of(Λis)) AccessTableis.add(ai);
  return CONTINUE;
}

```

Fig. 7. Pseudocode of the independence condition tester

The solution we present computes the supersteps lazily-whenever `pan_step(i)` is called, it only computes the superstep for thread i . To convey the information about the supersteps which have already been computed, we store additional information along with the program state on the depth-first search stack. For each state s and each thread i , we store `AccessTableis` - the list of location and access type pairs. Each instruction of Λ_i^s that accesses a global is represented by a pair (l, ty) ; it records the global location l that is accessed and the flag ty stating whether the transition is a read or a write. `AccessTable` is not stored as the part of the state tracked by SPIN but maintained externally within `pancam` VM since the data it stores is updated each time the state is visited.

The very first time state s is visited, `init_independence_tester()` initializes the `AccessTableis` of each enabled thread i with the access information derived from the very first instruction to be executed on the thread i . Further, before adding an instruction tr_i to the superstep sequence of thread i , `pan_step` consults with `test_for_independence(i, tri)` to ensure that the independence condition is met. `test_for_independence` may return three different values. `CONTINUE` means that the instruction can be added to the superstep Λ_i^s since it does not conflict with any instructions in Λ_k^s for all threads $k \neq i$. `POST_STOP` means that tr_i introduces a conflict with some other thread, but adding it to Λ_i^s does not violate the independence requirement as long as it is the very last transition of Λ_i^s . Finally, `PRE_STOP` means that adding tr_i to Λ_i^s leads to a violation since the transition with which it conflicts is not the very last transition of thread k for some $k \neq i$; thus, tr_i must not be executed. Due to the initialization of `AccessTableis`, it is not possible to have a `PRE_STOP` on the very first transition of any of the threads; thus, the superstep

size

requirement is met - `pan_step` always executes at least one transition. Finally, `test_for_independence(i, tri)` updates the `AccessTablesi` with the access pair derived from `tri` if the instruction is to be added to A_i^s and if it is not the very first instruction of A_i^s . Recall that the `AccessTable` is updated with the access pairs corresponding to the very first instructions of each thread as part of the initialization routine.

The above technique requires no space overhead when used as part of breadth-first search state exploration. However, when used with depth-first search, the `AccessTable` must be stored on the search stack. In cases when the sets are quite large, one could use approximations. For instance, one idea is to use a *coloring* abstraction, in which the memory is partitioned into regions with distinct colors, and each transition is associated with the set of colors it reads and writes.

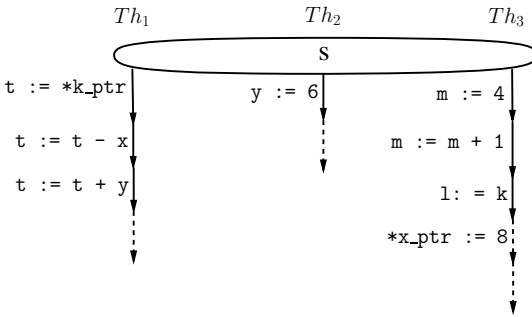


Fig. 8. The example demonstrating the application of the Superstep POR algorithm. The solid arrows represent the instructions that form the supersteps from the state s .

Example 1. Let us demonstrate the algorithm on an artificial example from Fig. 8 that depicts the instructions that the three threads can execute from the state s . We assume that the variables k , x , y , and m are global variables; t , l , x_ptr , k_ptr are local; x_ptr and k_ptr are the pointers to x and k , respectively.

When the state s is visited for the very first time, `init_independence_tester` initializes the `AccessTable` with the information derived from the very first instructions of each thread as following:

$$\begin{aligned} \text{AccessTable}_1^s &= ((k_ptr, read)) \\ \text{AccessTable}_2^s &= ((ad(y), write)) \\ \text{AccessTable}_3^s &= ((ad(m), write)) \end{aligned}$$

Here $ad(x)$ stands for the address in memory where the variable x is stored (`AccessTable` stores the actual addresses of the accessed variables). After the initialization, `pan_step` issues calls to `test_for_independence` passing the instructions of Th_1 one by one. The function returns `CONTINUE` when passed $t := *k_ptr$ and $t := t - x$. However, since $t := t + y$ conflicts with the very first instruction of Th_2 , `POST_STOP` is returned as the result of the third call. The table is updated accordingly:

$$\text{AccessTable}_1^s = ((k_ptr, read); (ad(x), read); (ad(y), read))$$

When the depth-first search backtracks to schedule Th_2 , `pan_step` calls `test_for_independence` with $y := 6$ as the argument. Due to the conflict with the last instruction of Th_1 , the function returns `POST_STOP`, making $y := 6$ to be the only instruction forming A_2^s . The `AccessTable`₂^s does not need to be updated.

Finally, when `pan_step(3)` is called, the check for independence on the first three instructions of Th_3 returns `CONTINUE`. Even though both the third instruction of Th_3 and the first instruction of Th_1 read from the same memory location: it can be determined at run time that k_ptr equals $ad(k)$, no conflict is reported. However, the fourth instruction, `*x_ptr := 8`, conflicts with the second entry in `AccessTable`₁^s raising the `PRE_STOP` return code. Since the conflicting transition is not the last transition of A_1^s , `*x_ptr := 8` should not be included in A_3^s .

5 Experimental Results

We have gathered some initial experimental results with our prototype on a few small multi-threaded C programs. Fig. 9 shows results from checking two versions of the implementation of Peterson’s algorithm in C, described in Section 2. Fig. 9(a) shows the number of states explored against varying context bounds for the version of the program with the missing `volatile` keyword bug, while Fig. 9(b) shows similar results for the version of the program without the bug. The graphs also compare a heuristic that runs a thread until it makes an access to any global state (labeled “global access” in the figure) versus our superstep reduction method (labeled “superstep”). As the graphs indicate, the bug is found fairly easily in all versions, though increasing the context bound beyond a certain point makes it harder to find the bug. (This is likely a consequence of the fact that SPIN uses depth-first search.) The graphs also show the benefit of an abstraction function we used which tracks only the algorithmic state of the protocol (the value of the abstract “flag” and “turn” variables).

Fig. 10 shows results from the “robot” benchmark example [GFYS07]. This example consists of two threads that move across a shared board of size $N \times N$

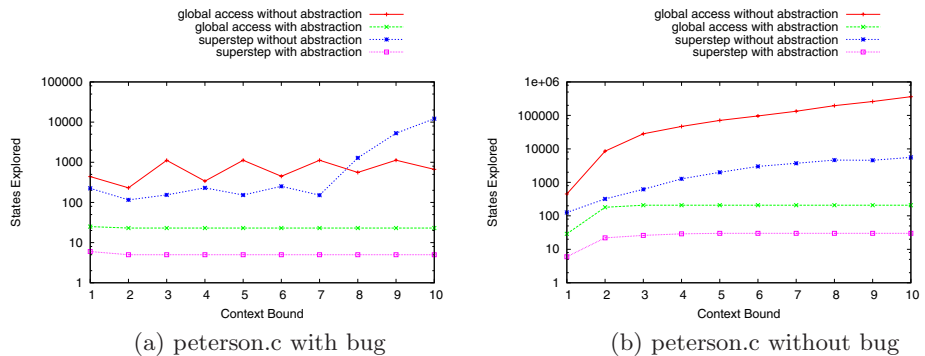


Fig. 9. Growth of state space with increasing context switch bound

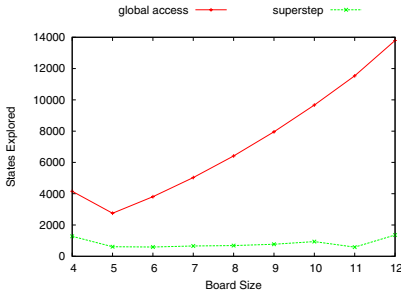


Fig. 10. Robot Example

Benchmark	#states	#states
	global access	superstep
Phil n=2	59	37
Phil n=3	534	380
Phil n=4	4762	3130
Phil n=5	42386	25021
IPC m=1	156863	234
IPC m=2	625359	316
IPC m=3	1529342	479
IPC m=4	!	654
IPC m=15	!	22629

Fig. 11. Other Examples

in slightly different patterns; the program checks that the robots meet only in expected locations. As the graph shows, our superstep method provides a noticeable reduction in the number of states over the global access method, as the size of the board grows.

Fig. 11 compares the improvement of superstep reduction with respect to the global access heuristic on two more examples. The first is a C implementation of the classic dining philosophers algorithm, with varying number of philosophers (denoted by parameter n). The second example is an inter-process communication module for an upcoming mission. The module consists of around 2800 lines of (non-commented) C source code (including some support modules that it relies on). It implements a communication system that supports prioritized messages and provides thread-safe primitives for sending and receiving messages. To give meaningful results, we restricted the model to a single producer-consumer pair, and forced a bound of 4 context switches, while varying queue depth (denoted by m). Even with the small configuration parameters, the default global access heuristic exhausts memory resources for $m = 4$ (as denoted by the symbol !) on a machine with 32 GBytes of RAM, whereas the superstep method can handle much larger configurations (well over $m = 15$).

6 Related Work

There has been considerable interest in applying model checking directly to implementation code. The Bandera checker [CDH⁺00] translates Java programs to the input language for a model checker, while Java Pathfinder (JPF) [VHB⁺03] uses an approach more similar to ours, in that it interprets bytecode. However, JPF tightly integrates model checking with the virtual machine, whereas our tool uses SPIN to orchestrate the search, using our virtual machine to execute transitions. This allows us to inherit (for free) the various optimizations and features of SPIN (both those that exist, and those yet to be invented). In spite of this loose integration, our approach is flexible; for instance, as shown in Section 2, adding support for bounding context-switches was done fairly easily in our tool. Using Modex [HS99] - a tool which extracts PROMELA models from C implementations

provides similar benefits. However, the model extractor is guided by user-defined abstractions, construction of which requires a considerable manual effort.

For verification of multi-threaded C programs, the CMC tool [MPC⁺02] uses explicit-state model checking. One limitation of CMC, however, is that it requires a manual step by the user to convert an existing C program into a form that can be used by CMC. In contrast, by working directly on bytecode, our tool design is simpler (interpreting typed LLVM bytecode is much easier than interpreting C). In addition, we are able to detect errors introduced during compiler optimization (like the Wikipedia error in Peterson’s algorithm, described in Section 2).

The changes introduced by compiler optimizations are also addressed by work in connection with the CodeSurfer tool [BRMT05], in which program analysis is applied to a model constructed from an executable. Advantages of this approach are that, since it deals directly with object code, it is not tied to a specific compiler and it also catches errors introduced by the compiler back-end. However, the constructed model is not precise since it has to recover information about variables and types, which is especially difficult for aggregate types (such as structures and arrays).

Another tool for verifying C programs is VeriSoft [God97], which uses *stateless* model checking. VeriSoft uses static partial order reduction, which typically results in little reduction when applied to C programs, since the independence relation is hard to compute. More recent work on dynamic partial-order reduction [FG05] addresses this problem, and the modifications have been implemented in the Inspect tool [YCGK07]. However, a limitation of these stateless approaches is that it requires the search depth to be bounded, which poses a challenge for programs whose state graphs have cycles.

More directly related to the superstep reduction presented in Section 4 is the work on “cartesian partial order reduction” [GFYS07], which is a method that dynamically computes independence relationships, and tries to avoid context switching whenever possible. The ideas behind cartesian partial order reduction and superstep reduction are closely related, though there are significant implementation differences. In particular, our reduction is done in the context of SPIN’s depth-first search. While this complicates the design somewhat, and incurs some additional memory overhead, it can be applied even when checking liveness properties. (In contrast, the cartesian reduction method was applied only in the context of checking assertion violations and deadlocks.)

Our approach to enforcing context-bounding is directly inspired by the work on the CHES model checker for concurrent C code [MQ08, MQ07]. One point of departure is that, even with fair scheduling, CHES only checks livelocks; in contrast, our approach is able to handle general liveness properties.

7 Conclusion

We have described a tool that can be used in conjunction with the SPIN model checker to check multithreaded C programs. Our tool works by generating typed bytecode generated for the Low-Level Virtual Machine (LLVM), which is then

interpreted by a virtual machine (named “pancam”). The virtual machine is designed to be used with SPIN, and the resulting tool therefore supports almost all SPIN features such as bitstate verification and multi-core operation. We have also shown we address the state explosion problem by allowing users to specify abstraction functions, context-switching bounds, and by using an on-the-fly algorithm for reducing unnecessary context switches. We are currently working on extending our tool to support checking liveness properties in the context of SPIN nested depth-first search.

References

- [BRMT05] Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: Wysinwyx: What you see is not what you execute. In: Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE) (October 2005)
- [CDH⁺00] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., Zheng, H.: Bandera: Extracting finite-state models from java source code. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE) (June 2000)
- [CGP00] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
- [FG05] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM Symposium on Programming Languages (POPL), pp. 110–121 (2005)
- [GFYS07] Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: SPIN Workshop on Model Checking of Software, pp. 95–112 (2007)
- [GJ08] Groce, A., Joshi, R.: Extending model checking with dynamic analysis. In: Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation (2008)
- [God97] Godefroid, P.: Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL) (1997)
- [HB07] Holzmann, G.J., Bosnacki, D.: The design of a multi-core extension of the spin model checker. In: IEEE Transactions on Software Engineering, October 2007, vol. 33, pp. 659–674 (2007)
- [HJ04] Holzmann, G.J., Joshi, R.: Model-driven software verification. In: SPIN Workshop on Model Checking of Software, pp. 76–91 (2004)
- [Hol03] Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
- [HS99] Holzmann, G., Smith, M.: A practical method for verifying event-driven software. In: International Conference on Software Engineering, pp. 597–607 (1999)
- [LA04] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California (March 2004)

- [MPC⁺02] Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.: CMC: A pragmatic approach to model checking real code. In: Symposium on Operating System Design and Implementation (2002)
- [MQ07] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 34th ACM Symposium on Programming Languages (POPL), pp. 446–455 (2007)
- [MQ08] Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI) (2008)
- [Pel93] Peled, D.: All from one, one for all: on model checking using representatives. In: Proceedings of the 5th Conference on Computer Aided Verification, pp. 409–423. Springer, Heidelberg (1993)
- [QR05] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Tools and Algorithms for the Construction and Analysis of Systems, April 2005, pp. 93–107 (2005)
- [VHB⁺03] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. *Automated Software Engineering* 10(2), 203–232 (2003)
- [wik] Peterson's algorithm,
http://en.wikipedia.org/wiki/Peterson's_algorithm
- [YCGK07] Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: Proceedings of the 14th International SPIN Workshop (July 2007)

Author Index

- Abed, Nazha 214
Andel, Todd R. 26
- Baier, Christel 60
Bao, Tonglaga 42
Bjørner, Nikolaj 9
Bultan, Tevfik 306
- Chen, Xiaofang 288
Choi, Yunja 144
Ciesinski, Frank 60
Cova, Marco 306
- de Halleux, Jonathan 9
Dwyer, Matthew B. 1
- Esparza, Javier 270
Evangelista, Sami 77
- Fecher, Harald 95
- Ganai, Malay K. 114
Gopalakrishnan, Ganesh 288
Groce, Alex 134
Größer, Marcus 60
Gupta, Aarti 114
- Hansen, Eric A. 160
Holzmann, Gerard J. 134
- Ibarra, Oscar H. 306
- Jones, Mike 42
Joshi, Rajeev 134, 325
- Kim, Hotae 144
Kim, Moonzoo 144
Kim, Yunho 144
Kirby, Robert M. 288
- Lamborn, Peter 160
Leue, Stefan 176
- Mateescu, Radu 196
- Nguyen, Viet Yen 232
- Oudot, Emilie 196
- Parker, David 60
Purandare, Rahul 1
- Qadeer, Shaz 3
- Ruys, Theo C. 232
- Schulte, Wolfram 9
Schwoon, Stefan 270
Shoham, Sharon 95
Shukla, Sandeep K. 250
Singh, Gaurav 250
Smaragdakis, Yannis 7
Ștefănescu, Alin 176
Suwimonteerabuth, Dejevuth 270
- Tillmann, Nikolai 9
Tripakis, Stavros 214
- Vanoverberghe, Dries 9
Vincent, Jean-Marc 214
- Wei, Wei 176
- Yang, Yu 288
Yasinsac, Alec 26
Yu, Fang 306
- Zaks, Anna 325