LNCS 4378

Irina Virbitskaite
Andrei Voronkov (Eds.)

Perspectives of Systems Informatics

6th International Andrei Ershov Memorial Conference, PSI 2006
Novosibirsk, Russia, June 2006
Revised Papers

Springer

Irina Virbitskaite   Andrei Voronkov (Eds.)

Perspectives of Systems Informatics

6th International Andrei Ershov Memorial Conference, PSI 2006
Novosibirsk, Russia, June 27-30, 2006
Revised Papers

Springer

Volume Editors

Irina Virbitskaite
A.P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentjev pr., 630090, Novosibirsk, Russia
E-mail: virb@iis.nsk.su

Andrei Voronkov
University of Manchester
Department of Computer Science
Oxford Road, Manchester, M13 9PL, UK
E-mail: voronkov@cs.man.ac.uk

Library of Congress Control Number: 2007920187

CR Subject Classification (1998): F.3, D.3, D.2, D.1

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

ISSN 0302-9743
ISBN-10 3-540-70880-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-70880-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12019588 06/3142 5 4 3 2 1 0

Preface

This volume contains the final proceedings of the Sixth International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI 2006), held in Akademgorodok (Novosibirsk, Russia), June 27-30, 2006.

The conference was held to honour the 75th anniversary of a member of the Russian Academy of Sciences Andrei Ershov (1931–1988) and his outstanding contributions towards advancing informatics. The role of Andrei Ershov in the establishment and development of the theory of programming and systems programming in our country cannot be overestimated. Andrei was one of the founders of the Siberian Computer Science School. He guided and took active part in the development of the programming system ALPHA and the multi-language system BETA, and authored some of the most remarkable results in the theory of programming. Andrei is justly considered one of the founders of the theory of mixed computation. In 1974 he was nominated as Distinguished Fellow of the British Computer Society. In 1981 he received the Silver Core Award for services rendered to IFIP. Andrei Ershov's brilliant speeches were always in the focus of public attention. Especially notable were his lectures "Aesthetic and Human Factor in Programming" and "Programming—The Second Literacy." He was not only an extremely gifted scientist, teacher and fighter for his ideas, but also a bright and many-sided personality. He wrote poetry, translated the works of R. Kipling and other English poets, and enjoyed playing guitar and singing. Everyone who had the pleasure of knowing Andrei Ershov and working with him will always remember his great vision, eminent achievements and generous friendship.

Another aim of the conference was to provide a forum for the presentation and in-depth discussion of advanced research directions in computer science. For a developing science, it is important to work out consolidating ideas, concepts and models. Movement in this direction was a further goal of the conference.

The previous five PSI conferences were held in 1991, 1996, 1999, 2001, and 2003, and proved to be significant international events. The sixth conference followed the traditions of the previous ones and included many of their subjects, such as theoretical computer science, programming methodology, and new information technologies, which were among the most important contributions of system informatics. Similarly to the previous PSI conferences, the programme includes invited papers in addition to contributed regular and short papers.

This time 108 papers were submitted to the conference by researchers from 28 countries. Each paper was reviewed by three experts, at least two of them from the same or closely related discipline as the authors. The reviewers generally provided high-quality assessment of the papers and often gave extensive comments to the authors for the possible improvement of the presentation. As a result, the Program Committee selected 30 high-quality papers for regular presentations and 10 papers for short presentations. A broad range of hot topics in system informatics was covered by five invited talks given by prominent computer scientists from different countries.

We are glad to express our gratitude to all the persons and organisations who contributed to the conference – to the authors of all the papers for their effort in producing the material included here, to the sponsors for their moral, financial and organizational support, to the members of the Steering Committee for the coordination of the conference, to the Programme Committee members and the reviewers who did their best to review and select the papers, and to the members of the Organizing Committee for their mutual contribution to the success of this event. Finally, we would like to mention the fruitful cooperation with Springer during the preparation of this volume.

November 2006

Irina Virbitskaite
Andrei Voronkov

Conference Organization

Programme Chairs

Irina Virbitskaite
Andrei Voronkov

Steering Committee

Dines Bjorner (Nomi, Ishikawa, Japan)
Manfred Broy (Munich, Germany)
Alexandre Zamulin (Novosibirsk, Russia)

Conference Secretary

Natalia Cheremnykh (Novosibirsk, Russia)

Programme Committee

Scott W. Ambler
Janis Barzdins
Frederic Benhamou
Stefan Brass
Ed Brinksma
Kim Bruce
Mikhail Bulyonkov
Albertas Caplinskas
Gabriel Ciobanu
Paul Clements
Miklos Csuros
Serge Demeyer
Alexandre Dikovsky
Javier Esparza
Jean-Claude Fernandez
Chris George
Ivan Golosov
Jan Friso Groote
Alan Hartman
Victor Kasyanov
Joost-Pieter Katoen
Alexander Kleschev
Nikolay Kolchanov

Gregory Kucherov
Johan Lilius
Dominique Mery
Bernhard Moeller
Hanspeter Moessenboeck
Torben Ægidius Mogensen
Ron Morrison
Peter Mosses
Peter Mueller
Fedor Murzin
Valery Nepomniaschy
Nikolaj Nikitchenko
José R. Paramá
Francesco Parisi-Presicce
Wojciech Penczek
Jaan Penjam
Peter Pepper
Alexander Petrenko
Jaroslav Pokorny
Wolfgang Reisig
Viktor Sabelfeld
Klaus-Dieter Schewe
David Schmidt
Sibylle Schupp
Timos Sellis
Alexander Semenov
Nikolay Shilov
Alexander Tomilin
Enn Tyugu
Alexander Wolf
Tatyana Yakhno
Wang Yi

Organizing Committee

Vladimir Philippov (Novosibirsk, Russia)
Gennady Alexeev (Novosibirsk, Russia)
Elena Bozhenkova (Novosibirsk, Russia)
Alexander Bystrov (Novosibirsk, Russia)
Tatyana Churina (Novosibirsk, Russia)
Olga Drobyshevich (Novosibirsk, Russia)
Pavel Emelianov (Novosibirsk, Russia)
Vera Ivanova (Novosibirsk, Russia)
Sergei Myl'nikov (Novosibirsk, Russia)
Tatyana Nesterenko (Novosibirsk, Russia)
Irina Kraineva (Novosibirsk, Russia)

Sponsors and Acknowledgments

This material is based on work supported by the European Research Office of the US Army under Contract No. N62558-06-P-0175. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the European Research Office of the US Army.

We wish to thank the following for their contribution to the success of this conference: European Office of Aerospace Research and Development, Air Force Office of Scientific Research and United States Air Force Research Laboratory.

We are grateful to the Russian Foundation for Basic Research, Microsoft Research, and Intel Corporation for the support of the conference.

We thank our Novosibirsk colleagues who contributed to PSI 2006.

External Reviewers

Lionel Morel
Martin Müller
Nachi Nagappan
Artur Niewiadomski
Thomas Noll
Nickolay Pakulin
Henrik Pilegaard
Agata Polrola
Ivan Porres
Konstantin Pyjov
Mathieu Raffinot
Dirk Reckmann
Arsenii Rudich
Joseph Ruskiewicz
Leo Ruest
Jelena Sanko
M. Satpathy
Frederic Saubion
Yael Shaham-Gafny
Dmitry Shkurko
Alex Sinyakov
Kim Solin
Alexander Stasenko
Dejvuth Suwimonteerabuth
Maciej Szreter
Alexei Tretiakov
Mars Valiev
Marc Voorhoeve
Sandro Wefel
Daniela Weinberg
Jan Martijn van der Werf
Wieger Wesselink
Jon Whittle
Tim Willemse
Christian Wimmer
Jozef Winkowski

Table of Contents

Separability in Conflict-Free Petri Nets

Eike Best[1], Javier Esparza[2], Harro Wimmel[1], and Karsten Wolf[3]

[1] Parallel Systems, Department of Computing Science
Carl von Ossietzky Universität Oldenburg, D-26111 Oldenburg, Germany
{eike.best,harro.wimmel}@informatik.uni-oldenburg.de
[2] Abteilung Sichere und Zuverlässige Softwaresysteme
Institut für Formale Methoden der Informatik, D-70569 Universität Stuttgart
esparza@informatik.uni-stuttgart.de
[3] Institut für Informatik, D-18051 Universität Rostock
karsten.wolf@uni-rostock.de

Abstract. We study whether transition sequences that transform markings with multiples of a number k on each place can be separated into k sequences, each transforming one k-th of the original marking. We prove that such a separation is possible for marked graph Petri nets, and present an inseparable sequence for a free-choice net.

1 Introduction

In concurrent systems verification, it is desirable to keep the portion of the state space that needs to be explored in order to check some property as small as possible. For example, if a system can be viewed as the composition of k independent but similar systems, it may be sufficient to check only one of them, instead of the whole set.

We are interested in Petri nets with *k-markings*, where by definition, a k-marking is a marking with a multiple of k tokens on each place (k being some positive natural number). We study under which conditions a Petri net with an initial k-marking M_0 can be separated, that is, viewed as k independent systems, each with initial marking $(1/k)\cdot M_0$. In such cases, some verification problems (for example, the reachability of a k-marking) can be solved in a system with greatly reduced state space.

The concept of separability has first been introduced and motivated in the context of workflow nets [6]. In that paper, a class of acyclic marked graphs [1,4] was proved to enjoy the separability property. In the present paper, we extend this result to all marked graphs. We also show by means of a counterexample that the separability property is not generally valid for free-choice nets [2].

The paper is organised as follows: Section 2 contains basic definitions and introduces the notion of separability formally. Section 3 contains the proof of the main result. In Section 4, we explore generalisations and limitations of this result. Section 5 contains concluding remarks.

2 Definitions

Definition 1 (Petri net). *A Petri net (S, T, F, M_0) consists of two finite and disjoint sets S (places) and T (transitions), a function $F\colon ((S\times T)\cup(T\times S)) \to \mathbb{N}$ (flow) and a marking M_0 (the initial marking). A marking is a mapping $M\colon S \to \mathbb{N}$. A Petri net is plain if the range of F is $\{0, 1\}$, i.e., F is a relation. A place s is a side-condition of a transition t if $F(s, t) \neq 0 \neq F(t, s)$.*

Definition 2 (Incidence matrix, Parikh vector). *For a transition t, let Δt be the vector with index set S defined by $\Delta t(s) = F(t, s)-F(s, t)$. The incidence matrix C is an $S \times T$ matrix of integers where the column corresponding to a transition t is, by definition, equal to the vector Δt. For a sequence σ of transitions, its Parikh vector Ψ_σ is a vector of natural numbers with index set T, where $\Psi_\sigma(t)$ is equal to the number of occurrences of t in σ.*

A transition t is enabled (or activated) in a marking M (denoted by $M[t\rangle$) if, for all places s, $M(s) \geq F(s, t)$. If t is enabled in M, then t can occur (or fire) in M, leading to the marking M' defined by $M' = M + \Delta t$ (notation: $M[t\rangle M'$). We apply definitions of enabledness and of the reachability relation to transition (or firing) sequences $\sigma \in T^*$, defined inductively: $M[\varepsilon\rangle M$, and $M[\sigma t\rangle M'$ iff there is some M'' with $M[\sigma\rangle M''$ and $M''[t\rangle M'$. A marking M is reachable (from M_0) if there exists a transition sequence σ such that $M_0[\sigma\rangle M$. We also generalise these notions to subsets $U \subseteq T$. A marking M enables U as a step (or concurrently) if for all places s, $M(s) \geq \sum_{t\in U} F(s, t)$. If U is enabled in M, all transitions of U can occur from M in some arbitrary order.

Two firing sequences σ and σ' are said to arise from each other by a single permutation if they are the same, except for the order of an adjacent pair of transitions which is concurrently enabled by the marking preceding them, thus:

$$\sigma \;=\; t_1 \dots t_k t t' \dots t_n \quad \text{and} \quad \sigma' \;=\; t_1 \dots t_k t' t \dots t_n,$$

such that the marking reached after $t_1 \dots t_k$ concurrently enables $\{t, t'\}$. Two sequences σ and σ' are said to be *permutations* of each other (written $\sigma \equiv \sigma'$) if they arise out of each other through a sequence of single permutations.

For any string w and letter a, let $\#(a, w)$ denote the number of times a occurs in w. Two strings v_1 and v_2 are called *Parikh equivalent* if for all letters a, $\#(a, v_1) = \#(a, v_2)$. Note that if a firing sequence σ' is a permutation of a firing sequence σ, then $\#(t, \sigma) = \#(t, \sigma')$ for each transition t, and so σ and σ' are Parikh equivalent. However, two Parikh equivalent firing sequences are not necessarily permutations of each other.

For any string w define a pair of strings (v_1, v_2) to be a *border* of w if v_1 is a prefix of w, v_2 is a suffix of w, and v_1, v_2 are Parikh equivalent.[1] Every string w has the trivial borders $(\varepsilon, \varepsilon)$ and (w, w).

Definition 3 (k-marking, separation). *Let k be a positive natural number. A k-marking M is a marking where, for all places s, $M(s)$ is divisible by k. For a*

[1] Normally, one requires $v_1 = v_2$, but this is too strong for our purposes.

k-marking M and a transition sequence τ such that $M[\tau\rangle M'$, a separation (of τ, starting from M) is a list $\tau_1, \ldots, \tau_k$ of transition sequences and a list $M_1, \ldots, M_k$ of markings such that

$$\forall j, 1 \leq j \leq k\colon \frac{1}{k} M[\tau_j\rangle M_j \quad \textit{and} \quad \sum_{j=1}^{k} \Psi_{\tau_j} = \Psi_\tau .$$

Note that it depends on k whether or not some sequence can be separated. For instance, if $k = 1$, then every marking is a k-marking and every sequence σ can trivially be separated.

We will argue that separability is very much tied to the absence of arc weights greater than 1 and to the absence of conflicts. Intuitively speaking, a conflict situation is one in which some enabled transition can be disabled by the occurrence of some other transition.

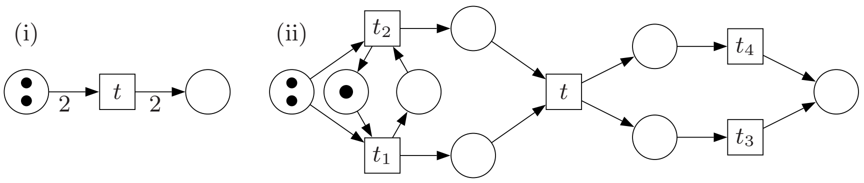


Fig. 1. A simple non-separable example (i) and an expansion (ii)

Consider the net in Figure 1(i), whose arcs have weight 2 and whose initial marking is a 2-marking. The firing sequence $\sigma = t$ (moving two tokens at the same time) cannot be separated (for $k = 2$), since no sequence can move only one token. Note that this net is (intuitively) free of conflicts.

Let us try to simulate such a net with arc weights ≤ 1. One possibility is shown in Figure 1(ii). This construction (using the regulatory circuit around t_1 and t_2) avoids both conflicts and deadlocks. The sequence $\sigma' = t_1 t_2 t t_3 t_4$ simulates the sequence $\sigma = t$ of Figure 1(i). However, σ' is separable (for $k = 1$); indeed, the initial marking is no longer a 2-marking. Putting one more token on the marked place of the regulatory circuit yields a 2-marking, but introduces a conflict between t_1 and t_2 after firing t_1 (and a deadlock as well, after $t_1 t_1$). The same is true if the circuit is omitted altogether. Thus, it appears impossible to simulate a conflict-free net with initial 2-marking and with arbitrary arc weights by a conflict-free net, also with initial 2-marking, which is plain.

In order to eliminate the first, rather trivial, source of non-separability, we will henceforth require all Petri nets to be plain, i.e., all arc weights to be 0 or 1. We then focus on studying what effects the absence or presence of conflicts has on separability.

In Petri net theory, the ‘absence of conflicts’ can be captured by defining various net classes which intuitively guarantee the absence of conflicts. In Section 3, we concentrate on marked graphs, which are a particularly simple and well-understood class of conflict-free nets. In Section 4, we recall other classes of conflict-free nets.

Definition 4 (Marked graphs [1,4]). *A net $N = (S, T, F, M_0)$ is called a marked graph, if for all places s, $\sum_{t \in T} F(s,t) \le 1$ and $\sum_{t \in T} F(t,s) \le 1$.*

3 Marked Graphs Are Separable

3.1 Theorem Statement, and Proof Outline

Theorem 1. *Let N be a marked graph and let M_0 be a k-marking ($k \ge 1$). Let $\tau \in T^*$ be a firing sequence starting from M_0. Then there is a separation of τ.*

Proof. By induction on $k \ge 1$.

Base: $k = 1$. Then the result is immediate: define $\tau_1 = \tau$.

Step: $k \ge 2$. Then $k = k' + 1$, with $k' \ge 1$. Consider the firing sequence $M_0[\tau\rangle$. As a consequence of Lemma 1, whose proof can be found below, there are firing sequences η and ξ with $(\frac{k'}{k}M_0)[\eta\rangle$, $(\frac{1}{k}M_0)[\xi\rangle$, and $\Psi_\eta + \Psi_\xi = \Psi_\tau$.

Since $\frac{k'}{k}M_0$ is a k'-marking and $k' < k$, we may apply the induction hypothesis, finding k' sequences $\tau_1, \ldots, \tau_{k'}$ with

$$(\frac{1}{k'}(\frac{k'}{k}M_0))[\tau_j\rangle \text{ for every } j, \quad \text{and} \quad \sum_{j=1}^{k'} \Psi_{\tau_j} = \Psi_\eta.$$

Putting $\tau_k = \xi$ yields a separation $\tau_1, \ldots, \tau_k$ of τ. ■1

The remainder of this section describes the proof of the following auxiliary result. Throughout this section, we assume N to be a marked graph, M_0 to be its initial marking, and M_0 to be a k-marking with $k = k' + 1$ and $k' \ge 1$.

Lemma 1. *Let τ be a firing sequence from M_0. Then there are firing sequences η and ξ such that*

$$(\frac{k'}{k}M_0)[\eta\rangle,\ (\frac{1}{k}M_0)[\xi\rangle, \quad \textit{and} \ \ \Psi_\eta + \Psi_\xi = \Psi_\tau.$$

The main idea is to use coloured firings in order to separate τ. Originally, all tokens are assumed to be 'black'. We may colour them, in some appropriate way, into red and green tokens. By *r-firing* (*g-firing*), we mean that a firing consumes and produces only red (green, respectively) tokens. We use indices $_{\mathrm{r}}$ and $_{\mathrm{g}}$, or $_{\mathrm{red}}$ and $_{\mathrm{green}}$, to indicate this. Any coloured set of tokens can be *decoloured* by turning red and green back into black. Separating a black firing sequence additively into two subsequences will be done by choosing a red/green-colouring and realising the black sequence by r-firings and g-firings. Once this is done, all r-firings can easily be collected into one subsequence and all g-firings into the

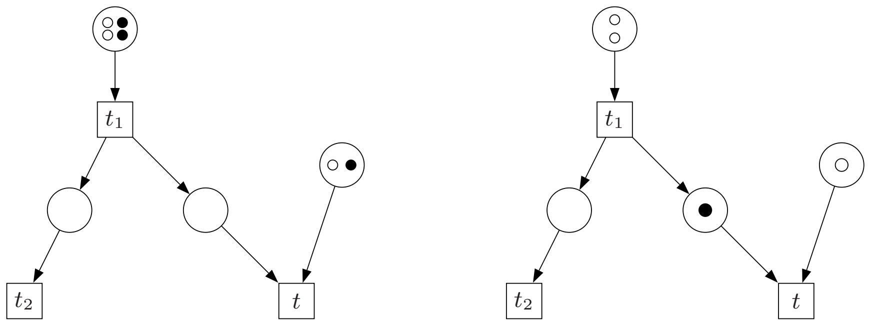


Fig. 2. A net with red and green tokens: initially (l.h.s.), and after $[t_1t_2t_1t_2t\rangle_{\mathrm{r}}$ (r.h.s.)

other subsequence. The main difficulty of the proof is to show that every black sequence can indeed be realised by r-firings and g-firings.

Figure 2 shows an example. Red tokens are represented as solid circles. The other (plain circle) tokens are supposed to be green. This colouring could correspond to a case in which $k' = 1$. Let M_0 be the initial marking of the net shown on the left-hand side, with black tokens, i.e., the decoloured version of the one actually shown in the figure. We discuss how the (black) firing sequence

$$M_0\,[\,t_1\,t_2\,t_1\,t_2\,t\,t\,\rangle$$

can be separated into two sequences, using the two colours. Suppose that we use red tokens as much as possible. Then we can fire as follows:

$$M_0\,[\underbrace{t_1\,t_2\,t_1\,t_2\,t}_{\text{red}}\rangle\;M\;\,[\,t\,\rangle_{\neg\mathrm{r}}^{\neg\mathrm{g}}$$

where M is the marking shown on the right-hand-side of Figure 2. At this point, t is (black-)enabled and needs to be fired next, but it is neither red- nor green-enabled. Obviously, a separation of (black) $t_1t_2t_1t_2tt$ cannot be found in this way. We need to use coloured firings more judiciously. For instance, if we choose to let the second subsequence t_1t_2 green-fire instead of red-fire, then we get

$$M_0\,[\underbrace{t_1\,t_2}_{\text{red}}\;\underbrace{t_1\,t_2}_{\text{green}}\;\underbrace{\;t\;}_{\text{red}}\rangle\;\widetilde{M}\;\,[\,t\,\rangle^{\mathrm{g}}_{\neg\mathrm{r}}.$$

In terms of black tokens, $\widetilde{M}$ is the same as M. However, red and green tokens are distributed differently, and t can green-fire at $\widetilde{M}$. In this way, we get a separation of (black) $\tau = t_1t_2t_1t_2tt$ into (red) $\tau_1 = t_1t_2t$ and (green) $\tau_2 = t_1t_2t$. Indeed, $\Psi_\tau = \Psi_{\tau_1} + \Psi_{\tau_2}$, as required.

By a *recolouring* of a firing sequence, we mean a sequence in which some firings have been coloured differently. A recolouring does not itself have to be a firing sequence, but we will be careful to apply recolouring when it is certain that the recoloured sequence is actually firable. A recoloured sequence is Parikh equivalent to the original, since the transition count is not affected.

By a *rearrangement* of a firing sequence, we mean a firing sequence that arises from the original one by permutations and/or recolourings. The Parikh vector of a rearranged sequence is still identical to that of the original sequence. In particular, if two sequences are rearrangements of each other, and if they are started from the same (black) marking, then they reach the same marking.

We return to the proof of Lemma 1. An appropriate r/g-colouring will be defined as follows. Since M_0 is a k-marking, there is, for every $s \in S$, a number $d_s \geq 0$ with $M_0(s) = d_s{\cdot}k$. Recall that $k = k' + 1$. For every $s \in S$, we define

$$M_{0,\text{red}}(s) = d_s{\cdot}k' \quad \text{and} \quad M_{0,\text{green}}(s) = d_s. \tag{1}$$

Then on each marked place, the ratio of red tokens to green tokens is k', and decolouring gives back the original M_0. If we can realise a black firing sequence with red-/green-firings starting from such a colouring, the red part can be seen to start from $(k'/k)M_0$ and the green part from $(1/k)M_0$, as required in the lemma. It follows that Lemma 1 is proved, provided we can prove the following:

Lemma 2. *With a red/green-colouring as in* (1)*, whenever t is a transition and*

$$M_0\ [\,\sigma\,\rangle\ M\ [\,t\,\rangle^{\neg\,g}_{\neg\,r}$$

(that is, σ leads from M_0 to M, M enables t, and M neither r-enables nor g-enables t), then there is a rearrangement σ' of σ such that $M_0[\sigma'\rangle\widetilde{M}$ and $\widetilde{M}$ r-enables t or $\widetilde{M}$ g-enables t.

Proof. This proof is divided into three steps as follows. Section 3.2 contains a lemma about borders in marked graphs, Section 3.3 describes the special case that σ has only r-firings, and the general case is dealt with in Section 3.4. ■2

3.2 Borders of Firing Sequences in Marked Graphs

The next lemma holds for arbitrary (uncoloured) marked graphs. Its purpose is to identify subsequences of firing sequences, such as t_1t_2 in the example, which need to be coloured in different colours.

Lemma 3. *Let t be a transition and let σ be a firing sequence starting from M_0. Suppose that $\#(t,\sigma) \geq 1$. Then σ can be permuted into a sequence $\widetilde{\sigma}$ such that $\widetilde{\sigma}$ has a border (β_1, β_2) with the following property:*

$$\forall x \in T\colon\ \ \#(x,\beta_1) \ = \ \#(x,\beta_2) \ = \ \begin{cases} 1 & \textit{if } \#(x,\sigma) > \#(t,\sigma) \\ 0 & \textit{if } \#(x,\sigma) \leq \#(t,\sigma). \end{cases}$$

In particular, the border that is claimed to exist by this lemma does not contain t, because t does not occur more than $\#(t,\sigma)$ times in σ.

Proof. We will proceed by induction, primarily on the number of transitions that occur more often than t in σ, and secondarily on the length of σ. That is, we consider a constant number of ts. Let the length of σ be n.

Base case: All transitions occur at most $\#(t, \sigma)$ times in σ. Then we immediately have a border of the desired kind, namely $(\varepsilon, \varepsilon)$.

Induction step: Assume that there is at least one transition x with $\#(x, \sigma) > \#(t, \sigma)$.

The first thing we will do is to permute σ in such a way that the first transition is one which occurs most often in σ. More precisely, let $m = \max\{\#(x, \sigma) \mid x \in T\}$ and let $U = \{u \in T \mid \#(u, \sigma) = m\}$. By the above assumption, we have $t \notin U$. We will make sure that the new first transition is one from the set U.

To this end, we start with σ, pick any $u \in U$ and consider the *first* (leftmost) occurrence of u in σ. We try to exchange this occurrence of u successively with its immediate left neighbouring transition. Suppose that this is not possible, for some left neighbour u'. Then we have the following situation:

$$M_0 \, [\, \ldots \, \rangle \, M' \, [\, u' \, \rangle \, \widehat{M} \, [u \, \rangle \, M'' \, [\qquad \underbrace{\rho}_{u \ occurs \ \#(u,\sigma)-1 \ times} \qquad \rangle \, M_n$$

such that there is some place $s \in u'^{\bullet} \cap {}^{\bullet}u$ with $M'(s) = 0$. Transition u' cannot occur less often in σ than u, because otherwise, s could not get sufficiently many tokens during the tail ρ for u to occur as often as it does there (namely $\#(u, \sigma)-1$ times, since we chose the first occurrence).

Hence, u' is another transition in U. We abandon u and continue in the same way with u', choosing *its* first occurrence in the part leading up to M''. Moving this occurrence of u' to the left cannot encounter u again, since all occurrences of u are to the right. Continuing in this way, eventually, we end up with a sequence of the form

$$M_0 \, [\, \underbrace{t_1 \, \overbrace{t_2 \, \ldots \, t \, \ldots \, t_n}^{\sigma_0}}_{\sigma'} \, \rangle \, M$$

in which $t_1 \in U$, and in particular, $\#(t_1, \sigma) > \#(t, \sigma)$. Call this sequence σ' and note that $t_1 \neq t$ and that, by construction, $\sigma' \equiv \sigma$.

We chop the first element, viz. t_1, off σ', denoting by σ_0 the shorter sequence, viz. $t_2 \ldots t_n$. This sequence starts with the marking M_1 reached after t_1, i.e. by firing $M_0[t_1\rangle M_1$. Note that we might have $t_2 = t$. Note also that σ_0 still contains every transition except t_1 (and in particular, t) as often as σ does.

Now it is possible to apply the inductive hypothesis to σ_0. The induction hypothesis implies that there is some permutation $\widetilde{\sigma_0}$ of σ_0 such that $\widetilde{\sigma_0}$ has a border, say (γ_1, γ_2), which contains exactly once every transition that occurs more often than t in σ_0 (and that is also the set of transitions the desired border of the longer sequence should contain, except possibly for t_1).

Let us now consider the following sequence σ'':

$$M_0 \, [\, \underbrace{t_1 \, \overbrace{\gamma_1 \, \ldots \, t \, \ldots \, \gamma_2}^{\widetilde{\sigma_0}}}_{\sigma''} \, \rangle \, M_n,$$

which is a permutation of σ' since $\widetilde{\sigma_0}$ is a permutation of σ_0. σ'' has a prefix of the form $t_1 \, \gamma_1$ and a suffix of the form γ_2. Note that γ_1 and γ_2 do not overlap, since there is some t in between, and neither of them contains t. If we could find some occurrence of t_1 between γ_1 and γ_2 which could be moved in front of the suffix γ_2, then we would have constructed a border of the longer sequence and we would be done.

First case: γ_1 (and hence also γ_2) does not contain t_1. Then, there is some occurrence of t_1 between them, because otherwise, there would be only one occurrence of t_1 in the whole of σ'' (namely, its first element), contradicting the fact that t_1 occurs at least twice (at least once more often than t, which occurs at least once) in σ''.

Choose the last occurrence of t_1 (which is also the last one between the two γ s). This can be moved to the right in front of the γ_2, for the following reason:

Suppose that there is some occurrence of some transition t' with which this last occurrence of t_1 cannot be permuted. We show that this occurrence of t' must be in the first element of the suffix-border γ_2, and hence that we have moved t_1 far enough already. More precisely, we consider the following situation:

$$M_0 \left[\, t_1 \, \gamma_1 \; \ldots \right\rangle M' \left[\, t_1 \right\rangle \widehat{M} \left[t' \right\rangle M'' \left[\ldots \; \gamma_2 \right\rangle M_n$$

such that there is some place $s \in {t_1}^{\bullet} \cap {}^{\bullet}t'$ with $M'(s) = 0$. Because $M'(s) = 0$, there are at least as many instances of t' in the sequence leading from M_0 to M' as there are instances of t_1 in it. But because we are moving the last t_1 and since $t_1 \in U$, we also have $t' \in U$, and moreover, the t' after $\widehat{M}$ is the last of its kind. By $t_1 \neq t'$, $t' \in U$, and the inductive hypothesis, there must be some occurrence of t' in γ_2. But because the t' after $\widehat{M}$ is the last of its kind, it is the first element of the border.

Hence, in this case, we find a permutation $\widetilde{\sigma}$ of σ'' (and hence of σ) of the form

$$M_0 \left[\underbrace{\overbrace{t_1 \, \gamma_1}^{\beta_1} \; \ldots \; t \; \ldots \; \overbrace{t_1 \, \gamma_2}^{\beta_2}}_{\widetilde{\sigma}} \right\rangle M_n$$

with border $(\beta_1, \beta_2) = (t_1\gamma_1, t_1\gamma_2)$.

Second case: t_1 occurs in γ_1 and in γ_2. Consider the (unique) occurrence of t_1 in the prefix γ_1, which is the second overall occurrence of t_1 in σ''. Because every other transition in γ_1 occurs there once only, this occurrence of t_1 can be right-moved to the end of the prefix γ_1. Thereafter, we may just forget about it, i.e., exclude it from the border-to-be-constructed.

More precisely, we now have a permutation $\widetilde{\sigma}$ of σ'' of the following form:

$$M_0 \left[\underbrace{t_1 \; \overbrace{\gamma_1' \, t_1}^{\equiv \gamma_1} \; \ldots \; t \; \ldots \; \gamma_2}_{\widetilde{\sigma}} \right\rangle M_n$$

where γ_1' does not contain t_1, but is otherwise is the same as γ_1. We can now combine the very first t_1 with γ_1' to form a border $(\beta_1, \beta_2) = (t_1\gamma_1', \gamma_2)$ of $\widetilde{\sigma}$:

$$M_0 \; [\underbrace{\overbrace{t_1 \; \gamma_1'}^{\beta_1} \; t_1 \; \ldots \; t \; \ldots \; \overbrace{\gamma_2}^{\beta_2}}_{\widetilde{\sigma}} \rangle \; M_n.$$

The two cases are exhaustive.[2] This proves the claim. ■3

3.3 A Special Case of Lemma 2

Note first that the colouring defined in (1) satisfies the following properties:

$$\begin{array}{ll} \text{(A):} & \forall s \in S\colon M_{0,\text{red}}(s) \geq M_{0,\text{green}}(s) \\ \text{(B):} & \forall s \in S\colon M_{0,\text{red}}(s) > 0 \Rightarrow M_{0,\text{green}}(s) > 0. \end{array}$$

Lemma 4. *Let t be a transition and let $M_0 \; [\sigma\rangle_{red} \; M[t\rangle^{\neg g}_{\neg r}$.*

Then there is a firable (at M_0) rearrangement of σ, leading to $\widetilde{M}$, such that $\widetilde{M}$ g-enables t.

Note that the conclusion is slightly stronger than required to prove Lemma 2. This facilitates the inductive proof of the general case, as will be seen later.

Proof. Suppose that

$$\sigma \;\; = \;\; t_1 \ldots t_n$$

with $n \geq 0$. We construct a rearrangement of σ such that t is eventually g-enabled.

First, note that M_0 does not enable t, for the following reason. Since t is not g-enabled at M, and since no green tokens are moved during σ, t is not g-enabled at M_0 either. By (B), t is not enabled at all at M_0. In particular, $n \geq 1$.

If t does not occur in σ, then $M[t\rangle^{\neg g}_{\neg r}$ is impossible. To see this, consider any input place s of t with $M_{green}(s) > 0$ and $M_{red}(s) = 0$. Such a place must exist because otherwise, t is not enabled or M r-enables t. At M_0, we have $M_{0,green}(s) > 0$ as well, since green tokens have not been moved. Hence $M_{0,red}(s) > 0$, by (A). But since the net is a marked graph and since t does not occur in σ, these red tokens on s cannot have been moved in $M_0[\sigma\rangle M$, contradicting $M_{red}(s) = 0$.

Hence, t occurs at least once in σ, and then we may decompose σ as follows:

$$M_0 \;\; [\;\; \underbrace{t_1 \ldots t_i}_{\text{no } t \text{ occurs here}} \;\; \rangle_{\text{red}} \;\; M_i \;\; [\underbrace{\;t\;}_{t = t_{i+1}} \rangle_{\text{red}} \;\; [\, t_{i+2} \ldots t_n \, \rangle_{\text{red}} \;\; M[t\rangle^{\neg \text{g}}_{\neg \text{r}} \;\; . \qquad (2)$$

Moreover, we have $i \geq 1$ since M_0 does not enable t.

We define structurally a set of transitions that must occur within $\{t_1, \ldots, t_i\}$, as follows. Let $^\circ(t, M_0)$ be the set of transitions, excluding t, that lie on a directed,

[2] In the second case, it may be impossible to move the t_1 that occurs in γ_2 to the front of γ_2. Hence the border satisfies $\beta_1 \equiv \beta_2$, but possibly, $\beta_1 \neq \beta_2$.

red-token-empty (at M_0), path leading into t. The set $^\circ(t, M_0)$ is nonempty, since t has at least one input place without any red tokens at M_0, and this place must have some input transition since otherwise, t would be dead at M_0. Moreover, $^\circ(t, M_0)$ is free of cycles since otherwise t would again be dead.

Intuitively, $^\circ(t, M_0)$ is the set of transitions that must r-fire at least once, prior to the first r-enabling of t. We now claim that

every transition in $^\circ(t, M_0)$ occurs more often in σ than t.

First, note that the transitions in $^\circ(t, M_0)$ occur at least as often as t even *before* the last occurrence of t in σ. This is so, since otherwise there is some place on some path from some transition in $^\circ(t, M_0)$ to t which has negative red-token balance,which is impossible.

Second, to see that every transition in $^\circ(t, M_0)$ occurs at least once more in σ, assume that, on the contrary, $t' \in {}^\circ(t, M_0)$ occurs exactly as often in σ as t. Let s be the (unique) input place of t that lies on a directed path from t' to t. Since $M_{0,red}(s) = 0$, we also have $\widehat{M}_{red}(s) = 0$, where $\widehat{M}$ is the marking reached after the last t in σ, and hence also $M_{red}(s) = 0$, since no t' occurs later. By the fact that M enables t, we have $M_{green}(s) > 0$, and hence also $M_{0,green}(s) > 0$, and then, by (A), $M_{0,red}(s) > 0$. However, this contradicts the fact that, by the definition of the set $^\circ(t, M_0)$, $M_{0,red}(s) = 0$.

Hence, every transition in $^\circ(t, M_0)$ occurs more often than t in σ. By an appeal to Lemma 3 (using the red tokens only), we find a permutation $\widetilde{\sigma}$ of σ which has a border (β_1, β_2) such that all transitions in $^\circ(t, M_0)$, but no t, occur in β_1 and in β_2; i.e., $\widetilde{\sigma} = \beta_1 \, \kappa \, \beta_2$ with

$$M_0 \left[\underbrace{\overbrace{\beta_1}^{\text{all of } {}^\circ(t,M_0) \text{ occur here}} \quad \overbrace{\kappa}^{\text{every } t \text{ occurs here}} \quad \overbrace{\beta_2}^{\text{all of } {}^\circ(t,M_0) \text{ occur here}}}_{\widetilde{\sigma} \text{ (r-firing)}} \right\rangle M [t\rangle^{\neg\,\mathrm{g}}_{\neg\,\mathrm{r}} \,.$$

Moreover, by Lemma 3, no transition occurs more than once in β_1.

Up till now, no recolouring has taken place; all firings in $\widetilde{\sigma}$ are still red. However, we will now change the suffix β_2 into β_1 and let it green-fire instead of red-fire:

$$M_0 \, \underbrace{\left[\overbrace{\beta_1}^{\text{r-firing}} \right\rangle \widetilde{M'} \left[\overbrace{\kappa}^{\text{r-firing}} \right\rangle \widetilde{M''} \left[\overbrace{\beta_1}^{\text{g-firing}} \right\rangle}_{\sigma'} \widetilde{M} \, [t\rangle^{\mathrm{g}}_{\neg\,\mathrm{r}} \,. \tag{3}$$

Then we have:

(i) β_1 can indeed be g-fired at $\widetilde{M''}$, since β_1 could be r-fired to start with, property (B) holds, and the green tokens have not been moved during the first β_1 and κ. Since every transition occurs at most once in β_1, there are sufficiently many green tokens in M_0 to fire β_1 (even though M_0 may contain less green than red tokens).

(ii) The sequence $\sigma' = \beta_{1,\mathrm{red}} \, \kappa_{\mathrm{red}} \, \beta_{1,\mathrm{green}}$ in (3) is indeed a rearrangement of σ, since β_1 is Parikh equivalent with β_2, and σ' is therefore a rearrangement of

$\widetilde{\sigma}$, which is a permutation of σ. Thus, the marking $\widetilde{M}$ reached in (3) is the same as the marking M reached in (2) in terms of black tokens; however, the red and green tokens are differently distributed.

(iii) Finally, $\widetilde{M}$ g-enables t. To see this, note that since green tokens have not been moved between M_0 and $\widetilde{M''}$, $^\circ(t, M_0)$ equals $^\circ(t, \widetilde{M''})$, if the latter set is calculated using only the green tokens. Previously, in terms of the red tokens, by firing every transition of $^\circ(t, M_0)$ at least once, but not t, it was possible to r-enable t (and keep it r-enabled until it occurs) from M_0. But the suffix β_1 contains every transition in $^\circ(t, M_0)$ at least once, and no t. Hence g-firing β_1 from $\widetilde{M''}$ g-enables t. ■4

In the example discussed on Figure 2, the border is $(\beta_1, \beta_2) = (t_1t_2, t_1t_2)$.

3.4 The General Case of Lemma 2

Lemma 5. *Let t be a transition and let $M_0\ [\sigma_1\rangle_{red}\ [\sigma_2\rangle_{green}\ M\ [t\rangle^{\neg g}_{\neg r}$.*

Then there is a firable (at M_0) rearrangement of $\sigma_1\sigma_2$, leading to $\widetilde{M}$, such that $\widetilde{M}$ g-enables t.

Note that $M_0[\sigma_1\rangle_{\mathrm{red}}[\sigma_2\rangle_{\mathrm{green}}M$ describes the general case, since r-firings and g-firings can be arbitrarily permuted.

Proof. Suppose that

$$M_0\ \underbrace{[\sigma_1\rangle}_{\text{r-firing}}\ M'\ \underbrace{[\sigma_2\rangle}_{\text{g-firing}}\ M\,[t\rangle \quad \text{with}\ \ \sigma_1 = t_1 \ldots t_n\ \ \text{and}\ \ \sigma_2 = x_1 \ldots x_m.$$

We wish to show that $\sigma_1\sigma_2$ can be rearranged in such a way that eventually, t is g-enabled, and we prove this by induction on the number of pairs $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, m\}$ with $t_i = x_j$.

Base case: There are no such pairs.

Then we have $\{t_1, \ldots, t_n\} \cap \{x_1, \ldots, x_m\} = \emptyset$ and

$$M_0\ [\ t_1 \ldots t_n\ \rangle_{\mathrm{red}}\ M'\ [\ x_1 \ldots x_m\ \rangle_{\mathrm{green}}\ M\ [t\rangle.$$

We show that

$$\forall s \in {}^\bullet\{x_1, \ldots, x_m\} \colon M'_{\mathrm{red}}(s) \geq M'_{\mathrm{green}}(s).$$

This is so because the same property already holds at M_0 by (A), since none of the x_j occurs amongst the $\{t_1, \ldots, t_n\}$ and because as a result (and by the marked graph property), none of the t_i can move any red token away from any input place of any of the x_j.

Hence, under the conditions of the base case, the sequence $x_1 \ldots x_m$ is not only g-firable, but also r-firable from M':

$$M_0\ [\ t_1 \ldots t_n\ \rangle_{\mathrm{red}}\ M'\ [\ x_1 \ldots x_m\ \rangle_{\mathrm{red}}\ \widetilde{M'}\ [t\rangle,$$

where $\widetilde{M}'$ is the same as M, except for the distribution of red and green tokens. The claim thus reduces to Lemma 4.

Inductive step: There is at least one such pair.

Then we choose some pair $(i, j) \in \{1, \dots, n\} \times \{1, \dots, m\}$ with minimal sum $i{+}j$, that is, one of the 'first' such pairs. Let $u = t_i = x_j$. We then have:

$$M_0 \,[\, t_1 \dots t_{i-1} \,\rangle \,[\, u \,\rangle \,[\, t_{i+1} \dots t_n \,\rangle \; M' \,[\, x_1 \dots x_{j-1} \,\rangle \,[\, u \,\rangle \,[\, x_{j+1} \dots x_m \,\rangle \; M \;[t\rangle,$$

such that $\{t_1, \dots, t_{i-1}\} \cap \{x_1, \dots, x_{j-1}\} = \emptyset$ and $u \notin \{t_1, \dots, t_{i-1}\}$ and $u \notin \{x_1, \dots, x_{j-1}\}$, by the minimality of the pair (i, j).

We claim that u is already enabled at M_0. For suppose it isn't. Then there is some input place $s \in {}^\bullet u$ such that $M_0(s) = 0$. The token count on s turns from $= 0$ at M_0 to > 0 both after $t_1 \dots t_{i-1}$ and after $x_1 \dots x_{j-1}$. Hence the (unique, by the marked graph property) input transition of s occurs both in $t_1 \dots t_{i-1}$ and in $x_1 \dots x_{j-1}$, contradicting $\{t_1, \dots, t_{i-1}\} \cap \{x_1, \dots, x_{j-1}\} = \emptyset$.

Hence u is already enabled, and thus both r-enabled and g-enabled, at M_0. In fact, by (1), every input transition of u has at least k' red tokens and at least one green token at M_0. Therefore, u can be left-moved once in the green sequence and at least once, and up to k' times, in the red sequence (provided that it contains that many u-transitions). Let ℓ be the number of occurrences of u in $t_1 \dots t_n$.

Case 1: $\ell \geq k'$.

Then we can permute as follows:

$$M_0 \,[\, \underbrace{u \dots u}_{k' \text{ times}} \;\; t'_1 \dots t'_q \,\rangle_{\text{red}} \; M' \,[\, u \, x_1 \dots x_{j-1} \, x_{j+1} \dots x_m \,\rangle_{\text{green}} \; M \;[t\rangle,$$

and the first green u can be permuted further to the front to occur just after the first k' red us. Now we can red-fire u k' times and green-fire u once, in order to obtain the following:

$$M_0 \,[\underbrace{u \,\dots\, u}_{k' \text{ times}}\rangle_{\text{red}} \,[\, u \,\rangle_{\text{green}} \; \widetilde{M_0} \,[\, t'_1 \dots t'_q \,\rangle_{\text{red}} \,[\, x_1 \dots x_{j-1} \, x_{j+1} \dots x_m \,\rangle_{\text{green}} \; M \;[t\rangle,$$

The marking $\widetilde{M_0}$ satisfies the same properties (in particular, a distribution of colours as in (1)) as the marking M_0, and the remaining r/g-sequence

$$t'_1 \dots t'_q x_1 \dots x_{j-1} x_{j+1} \dots x_m$$

has at least one pair of common transitions less than the original one. The claim follows from the induction hypothesis.

Case 2: $\ell < k'$.

Then there are, in M_0, $k' - \ell > 0$ excess red tokens on each input place of u, which are not used during the red sequence, nor, of course, during the green sequence. Hence we can insert $k' - \ell$ additional red-firings of u at the beginning of the sequence, thus:

$$M_0\,[\underbrace{u\,\ldots\,u}_{(k'-\ell)+\ell}\,\rangle_{\mathrm{red}}\,[\,u\,\rangle_{\mathrm{green}}\,\widetilde{M_0}'\,[\,t_1''\ldots t_p''\,\rangle_{\mathrm{red}}\,[\,x_1\ldots x_{j-1}\,x_{j+1}\ldots x_m\,\rangle_{\mathrm{green}}\,\widehat{M}\,[t\rangle.$$

The induction hypothesis can be applied to the sequence from $\widetilde{M_0}'$ to $\widehat{M}$ and yields a rearrangement $\widehat{\sigma}$ of $t_1''\ldots t_p''x_1\ldots x_{j-1}x_{j+1}\ldots x_m$ such that

$$M_0\,[\underbrace{u\,\ldots\,u}_{(k'-\ell)+\ell}\,\rangle_{\mathrm{red}}\,[\,u\,\rangle_{\mathrm{green}}\,\widetilde{M_0}'\,[\,\widehat{\sigma}\,\rangle\,\widehat{M}'\,[t\rangle_{\mathrm{green\text{-}enabled}}\,.$$

Since the tokens produced by the excess $k'-\ell$ initial red u transitions are nowhere needed to fire subsequent transitions, the last $k'-\ell$ red u transitions can be taken out of the sequence $[\,\underbrace{u\ldots u}_{k'\text{ times red}}\,\rangle\,[u\rangle_{\mathrm{green}}\,[\widehat{\sigma}\rangle$, settling Case 2 as well. ■5

4 Generalisations

The proof in section 3 uses the marked graph property several times. The marked graph property prevents the appearance of conflict situations by imposing a strong restriction on the structure of the net. It is natural to ask whether separability is caused by the absence of conflicts alone, or by the structural property. In this section we define a hierarchy of notions of conflict-freeness, and we show that for live and bounded nets, and for a part of this hierarchy, the two properties (absence of conflicts and the marked graph property) coincide. We also show that separability is not generally valid in free-choice nets, another net class extending marked graphs.

Definition 5 (Liveness, boundedness). *A Petri net is live if, for all reachable markings M and transitions t, there is a transition sequence σ that can occur from M and contains t. A Petri net is bounded if the set of markings reachable from its initial marking is finite.*

Definition 6 (Some Petri net classes). *A net $N = (S, T, F, M_0)$ is*

– *output-nonbranching* (on) *if all places s satisfy $|s^\bullet| \le 1$;*
– *conflict-free* (cf) *(*see e.g. [9]*) if all places s satisfy $|s^\bullet| > 1 \Rightarrow s^\bullet \subseteq {}^\bullet s$;*
– *behaviourally conflict-free* (bcf) *if, whenever a reachable marking M enables two transitions t, t' with $t \ne t'$, then ${}^\bullet t \cap {}^\bullet t' = \emptyset$;*
– *persistent* [7]*, if for all $U \subseteq T$ and all reachable markings M, if M enables every $t \in U$, then M enables U as a step;*
– *and free-choice* [2] *if transitions sharing pre-places share all their pre-places, i.e., if for all transitions t and t' and for all places s, $F(s,t) \ne 0 \ne F(s,t')$ implies $F(s,t) = F(s,t')$.*

The first four notions can be viewed as different formalisations of the intuitive notion of 'freeness of conflicts'. We have the following hierarchies:

$$\begin{array}{rl} & \text{marked graph} \Rightarrow \text{on} \Rightarrow \text{cf} \Rightarrow \text{bcf} \Rightarrow \text{persistent} \\ \text{and} & \text{marked graph} \Rightarrow \text{free-choice.} \end{array}$$

We show:

– The properties 'on' and 'cf' are, essentially, the same.
– In the presence of liveness and boundedness, every output-nonbranching net is a marked graph.
– In the presence of liveness, and for k-markings with $k \geq 2$, every behaviourally conflict-free net is output-nonbranching.

Lemma 6 (Reducing cf-nets to on-nets). *For every conflict-free Petri net N with initial marking M_0, there is an output-nonbranching net N' with initial marking M_0' and isomorphic reachability graph.*

Proof. Consider a place s in N for which $|s^\bullet| > 1$. By the conflict-freeness property, $s^\bullet \subseteq {}^\bullet s$, which means that s is a side-condition of every output transition of s (though it may still have some input transitions to which it is not a side-condition).

We may split s into $|s^\bullet|$ places, each connected only to one of the output transitions of s by a side-condition loop, such that all transitions in ${}^\bullet s \setminus s^\bullet$ are still input transitions of each of the new places, and the marking of s is, by definition, also the marking of every one of the new places. Apparently, the reachability graph of the new net is isomorphic to that of the original one, but the new net has one place with two or more output transitions less than the original one.

We repeat this until all places s satisfy $|s^\bullet| \leq 1$. The result is an on-net N' with initial marking M_0', whose reachability graph is isomorphic to the original one. ■6

An (unmarked) net (S, T, F) is called structurally bounded if (S, T, F, M_0) is bounded for *every* marking M_0.

Lemma 7 (Characterisation of structural boundedness). *The following are equivalent (where C^T is the transposed of C):*

(i) *(S, T, F) is structurally bounded.*
(ii) *There exists a vector $x \in \mathbb{N}^{|S|}$ with $x > 0$ and $C^\mathsf{T} \cdot x \leq 0$.*
(iii) *There exists no vector $y \in \mathbb{N}^{|T|}$ with $C \cdot y \geq 0$ and $C \cdot y \neq 0$.*

Proof. (Sketch.) The equivalence between (ii) and (iii) is (a version of) Farkas' lemma [8]. If some x as in (ii) exists, we have $0 \leq x^\mathsf{T} \cdot M \leq x^\mathsf{T} \cdot M_0$ whenever M is reachable from M_0, and hence M_0 is bounded, for any M_0. Conversely, if a vector y as in (iii) exists, we may define a marking that has sufficiently many tokens so that a transition sequence with Parikh vector y can repeatedly be executed, leading to unboundedness. ■7

Lemma 8 (Reducing on-nets to marked graphs). *Let $N = (S, T, F, M_0)$ be a live, bounded, and output-nonbranching net. Then N is a marked graph.*

Proof. Let $N = (S, T, F, M_0)$ be live, bounded, and output-nonbranching. By liveness and boundedness, each connected component of N is strongly connected. Since N is output-nonbranching, it is also free-choice. From the structure theory of free-choice nets (cf. Theorem 5.6 in [2]), it follows that each connected component of (S, T, F) is structurally bounded, so that (S, T, F) is structurally bounded itself. Define $y = \underline{1}$, with index set T. Then, by the fact that N is output-nonbranching and its connected components are strongly connected, $C \cdot y \geq 0$. Assume now that N is not a marked graph. Then also $C \cdot y \neq 0$, since we have at least one place with more than one input transition. But this contradicts structural boundedness by Lemma 7. Hence N is indeed a marked graph. ■8

Lemma 9 (Reducing bcf-nets to on-nets). *Let $N = (S, T, F, M_0)$ be a live, behaviourally conflict-free net and let M_0 be a k-marking, for some $k \geq 2$. Then N is output-nonbranching.*

Proof. We prove the claim of this lemma for $k = 2$. Let N be live and behaviourally conflict-free, and let the initial marking, M_0, be a 2-marking. We show that N is output-nonbranching.
Assume, on the contrary, that N is not output-nonbranching. Then there is some structural conflict, i.e., there are a place s and two transitions t, t' with $t \neq t'$ such that $F(s, t) > 0$ and $F(s, t') > 0$. We plan to prove that this structural conflict can actually be realised, that is, that there is some marking M (reachable from M_0) which activates both t and t', contradicting behavioural conflict-freeness.

Because M_0 is a 2-marking, the set of initial tokens can be divided into green ones and red ones, such that every place initially either has no tokens, or at least one green token and at least one red token (we may, e.g., distribute equally many green and red tokens). We claim:

- when the red tokens are omitted from the net, transition t can be activated in the resulting net, i.e., using only the green tokens;
- symmetrically, transition t' can be activated using only the red tokens.

From this, it follows immediately that the structural s, t, t' conflict can be realised, since one can use the green tokens to activate t and, independently, the red tokens to activate t', leading to a marking in which both t and t' are enabled.

What remains to be proven is that t can be activated using only the green tokens. To show this, we consider a sequence of length n activating t, viz.,

$$M_0 \,[t_0 t_1 \dots t_{n-1}\rangle\, M \quad \text{with} \quad M\,[t\rangle,$$

where we can assume, w.l.o.g., that t does not occur in $t_0 t_1 \dots t_{n-1}$. Such a sequence exists by liveness. Our aim is to find, from this sequence, another one which also activates t and consists of adjacent pairs of same transitions, thus:

$$M_0 \,[u_0 u_0 \, u_1 u_1 \, \dots \, u_{m-1} u_{m-1}\rangle\, \widehat{M} \quad \text{with} \quad \widehat{M}\,[t\rangle.$$

In such a sequence, every second marking, i.e., every marking M_j reached after $u_0u_0 \dots u_{j-1}u_{j-1}$ $(0 \le j \le m)$ is a 2-marking; this is true, in particular, for $\widehat{M}$. It follows that the sequence where every second transition is omitted, i.e., $u_0u_1 \dots u_{m-1}$, is firable by moving green tokens only, and what is more, also activates t, since $\widehat{M}$ is a 2-marking. (Note that this depends essentially on plainness, i.e., all arc weights being no greater than 1.)

Now consider the sequence $t_0t_1 \dots t_{n-1}$, which leads to an activation of t from M_0. Starting with t_0, we will gradually transform this sequence into a sequence $u_0u_0u_1u_1 \dots u_{m-1}u_{m-1}$ as desired.

Case 1: Suppose that t_0 does not occur in $\{t_1, \dots, t_{n-1}\}$ (and also, by assumption, $t_0 \neq t$); that is, t_0 occurs exactly once in the sequence $t_0 \dots t_{n-1}$. Because M_0 is a 2-marking, t_0 can occur twice from M_0 (again, we use plainness of the net), and what is more, t_0 cannot be in structural conflict with *any* of the transitions in $\{t_1, \dots, t_{n-1}, t\}$ – because if it could, this would contravene structural conflict-freeness. Therefore, if we enlarge the sequence by adding another t_0 after the first one:

$$t_0t_0\,t_1t_2\,\dots\,t_{n-1},$$

the extended sequence remains executable from M_0 and still activates t in its final marking. We can now chop t_0t_0 off the left-hand end of the sequence and deal with the shorter sequence $t_1t_2 \dots t_{n-1}$ – of length $n-1$ – in the same way (note that the marking reached after t_0t_0 is again a 2-marking).

Case 2: Suppose that t_0 occurs as one of the t_j's $(1 \le j \le n-1)$, but still, by assumption, $t_0 \neq t$. Then the sequence $t_0t_1 \dots t_{n-1}$ is of the following form:

$$t_0\sigma_1t_0\sigma_2,$$

where we may, w.l.o.g., assume that σ_1 does not contain another occurrence of t_0 (though σ_2 may). By the same argument as before, t_0 cannot be in structural conflict with any of the transitions occurring in σ_1. Hence the second occurrence of t_0 can be permuted back through σ_1 to a place adjacent to the first occurrence:

$$t_0t_0\,\sigma_1\sigma_2.$$

The inductive step now proceeds as above, except that the remaining sequence, $\sigma_1\sigma_2$, is of length $n-2$.

No other cases remain. This shows that a sequence $u_0u_0u_1u_1 \dots u_{m-1}u_{m-1}$ activating t from M_0 can be found, as was claimed, finishing the proof for $k = 2$ altogether.

The proof is similar for $k > 2$, the general principle being that we create blocks of k adjacent same transitions.

■9

Theorem 2. *Let $N = (S, T, F, M_0)$ be a live, bounded, and behaviourally conflict-free net, let $M_0[\tau\rangle M$, and let M_0, M be k-markings. Then there exists a separation $\tau_1, \dots, \tau_k$ of τ.*

Proof. If $k = 1$, then there is nothing to prove (we may take $\tau_1 = \tau$). If $k \geq 2$, then N is output-nonbranching by Lemma 9, a marked graph by Lemma 8, and the result follows from Theorem 1. ■2

Figure 3 proves that the result cannot be generalised to free-choice nets, even if liveness and boundedness is assumed. The net shown in this figure is live, bounded, and free-choice. Moreover, the initial marking is a 2-marking, and the transition sequence $\tau = t_3t_1t_2t_5t_4t_5$ is repetitive (i.e., leads back to the initial marking). But it is not separable (for $k = 2$): t_2 needs a prior t_1 which, in turn, needs a prior t_3; t_4 needs a prior t_3 or t_2; but since t_2 and t_3 occur only once, all occurrences of t_1, t_2, t_3, t_4 (and consequently also the two occurrences of t_5) must occur in the same subsequence.

The question of separability remains open if liveness and/or boundedness are dropped in Theorem 2. Intriguingly, it also remains open for persistent nets, even if liveness and boundedness are assumed.

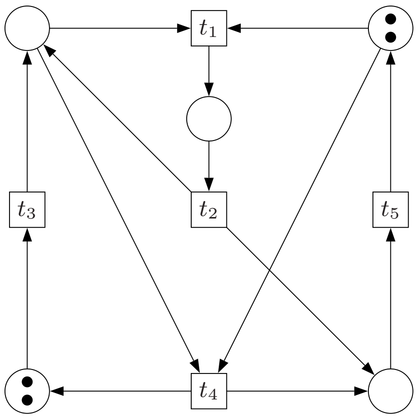


Fig. 3. A live and bounded free-choice net with a non-separable transition sequence

5 Outlook

In future, we aim at finding out whether the separability property can be proved for persistent nets, and at investigating separability in terms of concurrent, rather than interleaving, behaviour. Other research directions will be to check the usefulness of separability in applications where synchronic distances [5] play an important role, and to implement the separation property in model-checking systems that exploit the marking equation (e.g., [3]).

Acknowledgment

The first author gratefully acknowledges financial support by the research project UK/EPSRC Behavioural Synthesis of Systems with Heterogenous Timing, and support by the French/German DAAD/Procope project Comète.

References

1. F. Commoner, A.W. Holt, S. Even, A. Pnueli: Marked Directed Graphs. J. Comput. Syst. Sci. 5(5): 511-523 (1971).
2. J. Desel, J. Esparza: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, 242 pages, ISBN:0-521-46519-2 (1995).
3. J. Esparza, St. Melzer: Verification of Safety Properties Using Integer Programming: Beyond the State Equation. Formal Methods in System Design 16(2): 159-189 (2000).
4. H.J. Genrich, K. Lautenbach: Synchronisationsgraphen. Acta Inf. 2: 143-161 (1973).
5. U. Goltz: Synchronic Distance. Lecture Notes in Computer Science, Vol. 254: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course (W. Brauer, W. Reisig, G. Rozenberg, eds), Springer-Verlag, 338-358 (1987).
6. K. van Hee, N. Sidorova, M. Voorhove: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. Proc. ICATPN'2003, Eindhoven (van der Aalst / Best, eds.), Springer-Verlag LNCS Vol.2679, 337-356 (2003).
7. L.H. Landweber, E.L. Robertson: Properties of Conflict-Free and Persistent Petri Nets. JACM 25(3): 352-364 (1978).
8. A. Schrijver: Theory of Linear and Integer Programming. Wiley (1986).
9. H.C. Yen, B.Y. Wang, M.S. Yang: Deciding a Class of Path Formulas for Conflict-Free Petri Nets. Theory of Computing Systems, Vol. 30, No. 5: 475-494 (1997).

Grand Challenges of System Programming

Victor Ivannikov

Institute for System Programming
Russian Academy of Sciences
ivan@ispras.ru

In 1969 the Second All-Soviet Programming Conference took place here, in Akademgorodok. One of the hot issues discussed at that conference was the problem of crisis of programming, which was proposed by Andrei Ershov. Indeed, programs were becoming bulky and complicated, and were swarming with errors; the programmer's labor efficiency was thus low, and the development process hardly manageable. The laundry list of troubles can be continued. One can recall the dramatic story of developing the OS 360, which Brooks told in his "The Mythical Man Month". The world has changed drastically over the past years, much due to the advances in computer science. Even the mighty OS 360 is, by today's standards, an all-average program. But have all those troubles and problems been solved? No — and they've kept accumulating.

One of the essential reasons for this situation is the different rates of progress in the two opposing sectors of computer science: research versus development, software industry versus academic community. It's high time we spoke not of the crisis of programming in general, but of crisis in research. It's exactly how we can view the numerous works on the so called Grand Challenges, developed by such organizations as the British Computer Society, Computer Research Association, President's Information Technology Advisory Committee, etc.

I intend to touch upon only the few of the serious problems which system programming is facing.

Specifying and Verifying Programs in Spec#

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Spec# is research programming system that aims to provide programmers with a higher degree of rigor than in common languages today. The Spec# language extends the object-oriented .NET language C#, adding features like non-null types, pre- and postconditions, and object invariants. The language has been designed to support an incremental path to using more specifications. Some of the new features of Spec# are checked by a static type checker, some give rise to compiler-emitted run-time checks, and all can be subjected to the Spec# static program verifier. The program verifier generates verification conditions from Spec# programs and then uses an automatic theorem prover to analyze these.

In this talk, I will give an overview of Spec#, including a demo. I will then discuss in more detail some aspects of its design and our experience so far.

Joint work with Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, Manuel Fähndrich, Francesco Logozzo, Peter Müller, David A. Naumann, Wolfram Schulte, and Herman Venter.

Basic Protocols: Specification Language for Distributed Systems

Alexander Letichevsky

Department of Recursive Computers
Glushkov Institute of Cybernetics
Kiev, Ukraine
let@iss.org.ua

Verification of requirement specifications is an important stage of the software development process. Detection of inconsistency and incompleteness of requirement specifications, as well as discovering of wrong decisions at early stages of the design process decreases the cost of software quality. An approach to requirement verification has been considered in the papers [5–8]. The language of basic protocols is used there for specification of distributed concurrent systems and formalizing requirements for them.

Basic protocols combine well known Hoare triples with the model of interaction of agents and environments [1–4]. Each basic protocol is an expression of the type $\forall x(\alpha(x) \rightarrow< u(x) > \beta(x))$, where x is a (typed) list of parameters, $\alpha(x)$ and $\beta(x)$ are a precondition and a postcondition, respectively, and $u(x)$ is a finite process expression. Preconditions and postconditions are formulas of some logic language (usually first order one) called the *basic language.* This language is used to describe the properties of the states of a system represented as composition of an environment and agents inserted into this environment. The process $u(x)$ describes the behavior of the environment with inserted agents in the states that satisfy the precondition. A basic protocols specification (BP specification) of a system is defined by means of a set of basic protocols and an environment description, which determines the signature and interpretation of the basic language, the syntax of actions of the environment and agents, and the properties of possible initial states. The specified system is assumed to be an attributed labeled transition system, that is a transition system with transitions labeled by actions and states labeled by attribute labels. For a *concrete* model they are the interpretations of distinguished predicate and functional symbols of the signature of the basic language called *attributes* (propositional variables in model checking). For an *abstract* model the attribute labels are formulas of the basic language. In both cases the validity relation $s \models \alpha$ is defined, which means that the formula α is valid on the label of the state s.

The meaning of basic protocols can be defined in terms of temporal logics. Each basic protocol can be considered as a temporal logic formula which expresses the fact that the process $u(x)$ may be initiated only if (for appropriate values of parameters x) its precondition $\alpha(x)$ is valid on a system state, and if the protocol starts then after its successful termination the labeling satisfies the postcondition $\beta(x)$. A system that satisfies this requirement is called an *implementation* of the system of basic protocols. The set of possible interpretations can be restricted by introducing additional requirements depending on a subject domain.

Requirement specifications of distributed systems from real engineering practice, such as telecommunications, embedded and other kinds of distributed software systems, usually have the form very close to basic protocols and, as our experience shows, can be easily formalized in this form. Basic protocols also permit different levels of abstraction from concrete models with a given number of agents inserted into an environment and explicit attributes changing their values during evolving of the system states to abstract (symbolic) models with the states of the environment represented by the properties of the attributes.

Two abstract implementations $\mathbf{S}_P$ and $\mathbf{S}^P$ of BP specification defined by a set P of basic protocols and an environment description are considered. Both use two restrictions on transitions and the labeling of the specified system. The first restriction is expressed in terms of a *predicate transformer* – a transformation defined over formulas of the basic language. It transforms the condition γ, which characterizes the state labeling prior to applying a protocol, and the respective postcondition β into a new labeling $\mathbf{pt}(\gamma, \beta)$ (the transformed postcondition). A predicate transformer must satisfy the condition $\mathbf{pt}(\gamma, \beta) \rightarrow \beta$ and therefore it strengthens the postcondition w.r.t. the strengthened precondition.

The second restriction relates to possible interpretations of actions. It is defined in terms of permutability relation on the set of actions and is formulated as follows. The environment of the implementation controls all running protocols and a new protocol can be initiated only if its first action is permutable with all actions that can be performed by all running protocols. Moreover, an action performed by a protocol must be permutable with all actions that can be performed by all protocols initiated before it.

The restriction defined by permutability can be expressed in terms of composition of processes called *partially sequential* composition. In case of permutability of all actions of two basic protocols, their composition degenerates into a parallel composition, and if no actions are permutable — into a sequential composition of processes. Partially sequential composition generalizes the notion of weak sequential composition introduced by Renier [11] for the definition of formal semantics of MSC diagrams.

The states of abstract implementations are formulas of the basic language, and the behavior S_γ of a system in a state γ is defined by the equation:

$$S_\gamma = \sum_{p \in P(\gamma)} \mathbf{proc}(p) * (\mathbf{T}(\gamma, p) : \mathit{\Delta}) * S_{\mathbf{T}(\gamma,p)}.$$

The following notations are assumed in this formula. Let

$$p = \forall x(\alpha(x) \rightarrow < u(x) > \beta(x))$$

be a basic protocol. Its parameters $x = (x_1, x_2, \ldots)$ may have types and have particular value domains. Substitution of symbolic constants or values from the respective domains for parameters into the body $\alpha(x) \rightarrow< u(x) > \beta(x)$ of the basic protocol is called its *instantiation*. For an instantiated basic protocol $q = \forall x(\alpha(t) \rightarrow< u(t) > \beta(t))$ let us denote: $\mathbf{pre}(q) = \alpha(t)$, $\mathbf{post}(q) = \beta(t)$, $\mathbf{proc}(q) = u(t)$. Let P_{inst} be a set of instantiated basic protocols. Then

$$P(\gamma) = \{p \in P_{inst} \mid \gamma \rightarrow \mathbf{pre}(p)\}$$

for the system $\mathbf{S}_P$ and

$$P(\gamma) = \{p \in P_{inst} \mid \neg \models \neg(\gamma \wedge \mathbf{pre}(p))\}$$

for the system $\mathbf{S}^P$, $\mathbf{T}(\gamma, p) = \mathbf{pt}(\gamma, \mathbf{post}(p))$, $*$ denotes a partially sequential composition, Δ is a successfully terminated process, and $\alpha : s$ denotes a state or behavior s labeled by a condition α. Some additional constructions can be added to distinguish the states of successful termination.

The systems $\mathbf{S}_P$ and $\mathbf{S}^P$ are abstractions of concrete implementations and if basic protocols are formalization of requirement specifications, they precede the concrete implementation that must appear later. The question of how abstract implementations are connected with concrete ones was considered in [8]. To answer this question, the notion of abstraction relation was defined on the class of attributed systems. This notion generalizes some specific abstractions used in symbolic model checking [9,10] and is defined as follows.

Let S and S' be two attributed (not necessarily different) transition systems with common states and attribute labels and $\mathbf{BL}$ be the basic language. Define the abstraction relation $\mathbf{Abs} \subseteq S \times S'$ on the set of states as follows:

$$(s, s') \in \mathbf{Abs} \Leftrightarrow \forall(\alpha \in \mathbf{BL})((s \models \alpha) \Rightarrow (s' \models \alpha)).$$

We say that the system S is an *abstraction* (or an abstract model) of the system S' and the system S' is a *concretization* (or a concrete model) of the system S if a relation $\varphi \subseteq \mathbf{Abs}^{-1}$ exists, which is a relation of modeling (simulation). In other words, for any action a the following statement holds:

$$\forall(s \in S, s' \in S')((s', s) \in \varphi \wedge s \xrightarrow{a} t \Rightarrow \exists(t' \in S')(s' \xrightarrow{a} t' \wedge (t', t) \in \varphi)).$$

For systems with the set of initial and final states, the requirement of preserving the initial and final states is added.

This notion of abstraction means that each transition of an abstract model is forced by some transition of its concretization. It is also interesting to consider abstract models with the inverse property: each transition of a concrete model is forced by the respective transition of the abstract model. In other words, the relation $\varphi \subseteq \mathbf{Abs}^{-1}$ has the following property:

$$\forall(s \in S, s' \in S')((s', s) \in \varphi \wedge s' \xrightarrow{a} t' \Rightarrow \exists(t \in S)(s \xrightarrow{a} t \wedge (t', t) \in \varphi)).$$

In this case we say about an *inverse* abstract model S of the system S'.

Both kinds of abstract models are useful for this purpose. For a direct model it is true that if some property is reachable in its concretization then it is also reachable in the model. And if some property is reachable in the concretization, it is also reachable in the inverse model. Therefore direct models can be used for verification of a system and inverse models for test generation.

In [8], for some specific signature and BP specification P, there was defined a class $\mathbf{K}(P)$ of concrete models of P and the main result was proved: *a system* $\mathbf{S}_P$ *is a direct abstraction of all systems from the class* $\mathbf{K}(P)$ *and* $\mathbf{S}^P$ *is an inverse abstraction of all concrete systems from this class.*

The abstract models of basic protocols are implemented in the VRS system developed for Motorola [5–8] and have been successfully applied for verification of requirement specifications of real engineering projects. The tools of VRS include the following:

- *checking consistency and completeness of preconditions of BP specifications.* In the strongest case consistency means that preconditions of different protocols with the same starting actions in the process cannot intersect (never valid at the same time) and completeness means that the disjunction of preconditions for protocols with the same starting actions is always valid. Consistency provides determinism and completeness checks such properties as the absence of deadlocks.
- *proving safety conditions on abstract models.* Proving is fulfilled by induction using the deductive system of VRS or by modeling of direct or inverse abstractions.
- *proving reachability of properties.* Symbolic and concrete modeling is used.

The languages MSC, SDL, and UML with annotations are used for description of processes. Deductive tools include the proving procedure for first order typed predicate calculus integrated with linear inequalities for integers (Pressburger) and reals.

References

1. A. Letichevsky and D. Gilbert. A general theory of action languages. Cybernetics and System Analysis, 1, 1998.
2. A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.
3. A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science, 1827, 2004.
4. A. Letichevsky, Algebra of behavior transformations and its applications, in Proceedings of the Summer School on Structural Theory of Automata, Semigroups and Universal Algebra, Montreal July 8-18, 2003, To be published by Kluwer Acad. Press.
5. S. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert. Leveraging UML to Deliver Correct Telecom Applications. In L. Lavagno, G. Martin, and B.Selic, editors. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam, 2003.

6. A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, forthcoming in 2005.
7. J. Kapitonova, A. Letichevsky, V. Volkov, and T. Weigert. Validation of Embedded Systems. In R. Zurawski, editor. The Embedded Systems Handbook. CRC Press, Miami, forthcoming in 2005.
8. A.A.Letichevsky, J.V.Kapitonova, V.A.Volkov, A.A.Letichevsky, jr., S.N.Baranov, V.P.Kotlyarov, T.Weigert, System Specification with Basic Protocols, Cybernetics and System Analyses, 4, 2005.
9. E.M.Clarke, Jr., O.Grumberg, D.Peled, Model checking, MIT Press, 1999,
10. E.M.Clarke, O.Grumberg, and O.E.Long. Model checking and abstraction. ACM Transactions on Programming Languages and Systems, V.16, 5, p. 1512-1542.

Why Automata Models Are Sexy for Testers? (Invited Talk)

Alexandre Petrenko

Centre de Recherche Informatique de Montreal (CRIM)
550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada
Petrenko@crim.ca

Formal methods traditionally aim at verifying and proving correctness (a typical academic activity), while testing can only show the presence of errors (that is what practitioners do). Recently, there is an increasing interest in the use of formal models and methods in testing. In this talk, we first present a traditional framework of model–based testing, considering a variety of state-oriented (automata) models, such as Finite State Machines (FSM), Communicating FSM, Extended FSM, where input and output are coupled for each transition; and input/output automata (a.k.a. transition systems), where inputs are outputs are decoupled. We provide an overview of existing test derivation techniques based on automata models, while paying a special attention to the underlying testing assumptions and fault detection capability of the resulting tests.

We distinguish two testing scenarios, where an implementation under test is treated as a black-box and either a formal specification of the expected behavior or a set of desired properties is given, respectively, model-based and property-based testing. A property-based testing framework for distributed systems is also presented in the talk. The processes in a system are instrumented to generate events, such as send and receive of messages, local events and others. The collected events constitute a partially ordered event trace and some user-defined properties can be checked on the trace offline. We present an approach to property-based testing, where a trace of a distributed system is converted into a collection of communicating automata that serves as an input to a model checker that tests whether given properties are violated in the trace. We discuss possibilities of merging both scenarios and conclude by pointing to open problems.

An Universal Resolving Algorithm for Inverse Computation of Lazy Languages

Sergei Abramov[1], Robert Glück[2], and Yuri Klimov[3,⋆]

[1] Program Systems Institute, Russian Academy of Sciences
RU-152140 Pereslavl-Zalessky, Russia
abram@botik.ru
[2] DIKU, Department of Computer Science, University of Copenhagen
DK-2100 Copenhagen, Denmark
glueck@acm.org
[3] M.V. Keldysh Institute for Applied Mathematics, Russian Academy of Sciences
RU-125047 Moscow, Russia
yuri@klimov.net

Abstract. The Universal Resolving Algorithm was originally formulated for inverse computation of tail-recursive programs. We present an extension to general recursion that improves the efficiency and termination of inverse computation because partially produced output is used to reduce the search space. In addition, we present a transformation using a new unification-based equality operator. Examples demonstrate the advantages of the new technique. We found that these extensions can also improve inverse computation in the context of functional-logic languages.

1 Introduction

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function. Inverse computation is the calculation of the possible input of a program for a given output. The *Universal Resolving Algorithm* (URA) [2,3] is an algorithm for inverse computation in a first-order functional language. The algorithm is sound and complete with respect to the solutions defined by a given program. Termination and efficiency depends directly on the search space traversed when performing inverse computation. The original algorithm relied only on perfect driving [6] to reduce the search space and was restricted to a tail-recursive programming language.

In this paper we present an extension of the original algorithm to general recursion. This allows us to reduce the search space drastically when partially defined output becomes available. We show how termination and efficiency of inverse computation can be improved by intersection and unification-based equality. We demonstrate the gains of our method with several examples. This paper concerns the question on how to make inverse computation faster and more terminating. Another proposal [17] which approximates functional programs by grammars is complete, but sacrifices soundness for termination. It is well-known

⋆ Supported by the Russian Foundation for Basic Research grant No. 06-01-00574.

that an algorithm for inverse computation cannot be sound, complete and always terminating. URA is sound and complete, so we try to improve termination.

To summarize the contributions: We present an extension of the original URA [2,3] to general recursion and show two novel solutions that can drastically improve the efficiency and termination of the algorithm: (1) A new technique for cutting and backpropagation based on intersection of classes during perfect driving. (2) A novel unification-based equality operator that provides a surprisingly simple solution by an equivalence transformation of a given request for inverse computation into the source language. Our techniques aim at reducing the search space during inverse computation and can sometimes turn an infinite into a finite search. This is of interest beyond URA. We found that inverse computation in modern functional-logic languages can be improved by these techniques.

After reviewing the principles of inverse computation (Sect. 2), we explain the reduction of the search space (Sect. 3) and the semantics of unification-based equality (Sect. 4). Then we define a straightforward equivalence transformation (Sect. 5) and demonstrate the technique with several examples (Sect. 6). We conclude with related work (Sect. 7) and future work (Sect. 8).

2 Background: An Approach to Inverse Computation

This section summarizes the concepts behind the Universal Resolving Algorithm [2,3]. For a given program p written in programming language L and output d_{out}, *inverse computation* is the determination of an input ds_{in} such that $[\![p]\!]_L\, ds_{in} = d_{out}$. Here, ds_{in} is a list of values $[d_1, \ldots, d_n]$ and d_{out} is a single value. When additional information about the input domain is available, we may want to restrict the search space of the input for a given output. Conversely, we may want to specify a set of output values, instead of fixing a particular value d_{out}. We do so by specifying the input and output domains using an *input-output class* (io-class) cls_{io}. A class is a finite representation of a possibly infinite set of values. Let $\lceil cls_{io} \rceil$ be the set of input-output values represented by cls_{io}, then the correct solution $Inv(L, p, cls_{io})$ to an inversion problem is specified by

$$Inv(L, p, cls_{io}) = \{\ (ds_{in}, d_{out})\ \mid\ (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil,\ [\![p]\!]_L\, ds_{in} = d_{out}\ \} \quad (1)$$

where L is a programming language, p is an L-program, and cls_{io} is an input-output class. The *universal solution* $Inv(L, p, cls_{io})$ is the largest subset of $\lceil cls_{io} \rceil$ such that $[\![p]\!]_L\, ds_{in} = d_{out}$ for all elements (ds_{in}, d_{out}) of this subset.

In general, inverse computation using a program *invint* for inverse computation for a language L takes the form

$$[\![invint]\!]\ [p, cls_{io}]\ =\ ans \quad (2)$$

where p is an L-program and cls_{io} is an io-class. We say, cls_{io} is a *request* for inverse computation of L-program p and ans is the *answer*. When designing an algorithm for inverse computation, we need to choose a concrete representation for cls_{io} and ans. We use S-expressions known from Lisp as the value domain and represent the classes by *expressions with variables and restrictions* [2,3,16].

The *Universal Resolving Algorithm* (URA) [3,2] is an algorithm for inverse computation in a first-order functional language. The algorithm produces a universal solution, hence the first word of its name. The answer $ans = \{(\theta_1, \widehat{r}_1), \ldots\}$ produced by URA is a set of substitution-restriction pairs that represents set $Inv(L, p, cls_{io})$. The correctness of URA is given by

$$\bigcup_i \lceil (cls_{io}/\theta_i)/\widehat{r}_i \rceil \;=\; Inv(L, p, cls_{io}) \tag{3}$$

where $(cls_{io}/\theta_i)/\widehat{r}_i$ narrows the set of input-output values represented by io-class cls_{io} by applying substitution θ_i to it and then adding restriction $\widehat{r}_i$.

As an example, consider inverse computation of a program a2b (Sect. 4). Program a2b replaces each 'A by 'B in a list of symbols, leaving all other symbols unchanged. For example, $[\![\text{a2b}]\!]$ [['A, 'B, 'A]] = ['B, 'B, 'B]. Suppose that we have the list ['B] as output and want to find all inputs that can produce this output. For inverse computation of a2b, we specify the io-class

$$cls_{io} = \langle (\, \underbrace{[Xe_1]}_{\widehat{ds}_{in}},\; \underbrace{[\text{'B}]}_{\widehat{d}_{out}} \,),\; \underbrace{\;\emptyset\;}_{\widehat{r}_{io}} \,\rangle \tag{4}$$

where $\widehat{ds}_{in}$ specifies the input, $\widehat{d}_{out}$ the output, and $\widehat{r}_{io} = \emptyset$ is an empty restriction (no constraints on the domains of c-variables). Placeholders like Xe_1 are called *configuration variables* (c-variables); they range over the set of S-expressions. A restriction $\widehat{r}_{io}$ is a finite set of inequalities that constrains the domain of c-variables (*e.g.*, we might specify $\{Xe_1 \neq \text{'A}, Xe_1 \neq Xe_2\}$ as a restriction). A rewrite system can be used to normalize such kinds of constraints [16]. We distinguish between a value, d, and an expression, $\widehat{d}$, that represents a sets of values. Inverse computation with URA then takes the form:

$$[\![ura]\!]\ [\text{a2b}, cls_{io}] \;=\; ans\ . \tag{5}$$

In our example, the answer contains two substitution-restriction pairs each with a substitution for Xe_1 and an empty restriction: $ans = \{\, ([Xe_1 \mapsto [\text{'A}]], \emptyset), ([Xe_1 \mapsto [\text{'B}]], \emptyset) \,\}$. This tells us that ['B] is produced by input ['A] and ['B].

URA is based on the notion of a *perfect process tree* [6] that represents the computation of a program with *partially specified input* by a tree of all possible computation traces. The algorithm constructs, breadth-first and lazily, a perfect process tree for a given program p and input class $cls_{in} = \langle \widehat{ds}_{in}, \widehat{r}_{io} \rangle$ taken from the given request cls_{io}, and extracts the answer ans from the finite traces and leaves in the tree. The construction of a process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input [5]. It is important that each fork in a perfect tree partitions the input class cls_{in} into disjoint and exhaustive subclasses. URA is sound and complete [3], but does not always terminate because the process tree is not always finite. The algorithm is based on the idea of *driving* known from supercompilation [19]. We present now an important extension of the algorithm to general recursion that can improve termination and produce answers faster.

3 Reducing the Search Space

In this section we establish the idea of using an *mgu-based intersection* and the *constructor skeleton* of an expression as an approximation of the output. The following two examples show that intersection can drastically reduce the search space of URA by cutting infeasible branches and backpropagating bindings.

Tracing a program with partially specified input may confront us with conditionals that depend on unspecified values, and we have to consider the possibility that either branch of the conditional is entered with some input. For instance, if the program tests whether the value of a program variable is a pair, but the variable is bound to Xe_1, then we have two possibilities (*perfect split*): it is a pair Xe_2**:**Xe_3 or an atom. This leads to two new branches in the perfect process tree. Repeating these driving steps can lead to an infinite tree.

Cutting branches. Consider that we trace a2b along the following configurations. Recall that we are looking for input that produces the output ['B].

$$\begin{array}{lll} c_1 = \langle(\textbf{call}\ \text{a2b}\ [Xe_1]), \emptyset\rangle & \downarrow \theta_1 = [Xe_1 \mapsto Xe_2 : Xe_3],\ \theta_2 = [Xe_2 \mapsto \text{'A}] \\ c_7 = \langle\text{'B}\!:\!(\textbf{call}\ \text{a2b}\ [Xe_3]), \emptyset\rangle & \downarrow \theta_5 = [Xe_3 \mapsto Xe_6 : Xe_7],\ \theta_6 = [Xe_6 \mapsto \text{'A}] \\ c_9 = \langle\text{'B}\!:\!\text{'B}\!:\!(\textbf{call}\ \text{a2b}\ [Xe_7]), \emptyset\rangle & \downarrow \ldots \end{array}$$

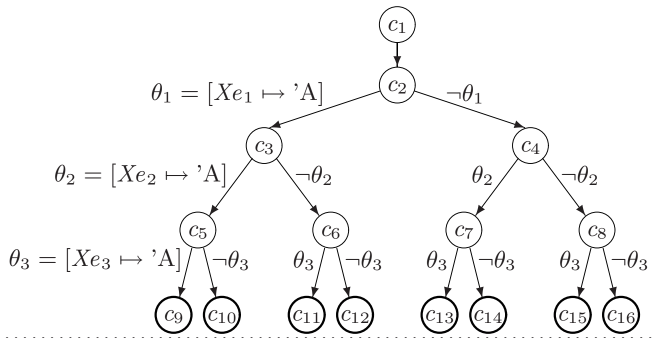
Clearly, the last configuration c_9 and its descendents can never lead to an answer because their output will always be a list with a length greater than one, 'B : 'B : •, where • stands for the unknown output of (**call** a2b $[Xe_7]$), while we are looking for a list of length one as output: 'B : [ ]. Instead of blindly continuing an infinite search, we examine whether the *partially computed output* at c_9 can possibly lead to the desired output and, if not, stop tracing. Intuitively, the current io-class at c_9 and the given io-class (4) do not 'unify': their intersection is empty. Thus, we can stop tracing at c_9 without loosing an answer. Cutting such unproductive branches improves efficiency and, in our example, the process tree becomes finite. Intersection $\star$ of the current io-class cls'_{io} and the initial io-class cls_{io} is empty (operation $\star$ will be defined below). The term ['A : 'A : Xe_7] in cls'_{io} is obtained by applying $\theta_1, \theta_2, \theta_5, \theta_6$ to $[Xe_1]$ of cls_{io}. We compute the intersection:

$$\underbrace{\langle([\text{'A}\!:\!\text{'A}\!:\!Xe_7], \text{'B}\!:\!\text{'B}\!:\!\bullet), \emptyset\rangle}_{\text{current } cls'_{io}} \star \underbrace{\langle([Xe_1], \text{'B}\!:\![\,]), \emptyset\rangle}_{\text{given } cls_{io}} = \emptyset\ . \tag{6}$$

Backpropagation. The second example adds a function f that is defined by

$$(\textbf{define}\ \text{f [x] ([x, (}\textbf{call}\ \text{a2b [x])]))}\ .$$

Function f is simple: it takes a list x as input and returns as output a list containing two elements: the original list x and the result of applying function a2b to x. We specify the output domain as $\widehat{d}_{out} = [[Xe_0, Xe_0, Xe_0], [\text{'B}, \text{'B}, \text{'B}]]$. For instance, it includes the value [['C, 'C, 'C], ['B, 'B, 'B]]. The three identical c-variables Xe_0 stand for three identical values. Also, we specify that the input must be a list of length three: $\widehat{ds}_{in} = [Xe_1, Xe_2, Xe_3]$. There are no restrictions



$c_1 = \langle(\textbf{call } \text{f } [Xe_1, Xe_2, Xe_3]), \emptyset\rangle$
$c_2 = \langle[[Xe_1, Xe_2, Xe_3],\ (\textbf{call } \text{a2b } [Xe_1, Xe_2, Xe_3])], \emptyset\rangle$
$c_3 = \langle[[\text{'A}, Xe_2, Xe_3],\ \text{'B}\textbf{:}(\textbf{call } \text{a2b } [Xe_2, Xe_3])], \emptyset\rangle$
$c_4 = \langle[[Xe_1, Xe_2, Xe_3],\ Xe_1\textbf{:}(\textbf{call } \text{a2b } [Xe_2, Xe_3])], \{Xe_1 \neq \text{'A}\}\rangle$
$c_5 = \langle[[\text{'A}, \text{'A}, Xe_3],\ \text{'B}\textbf{:}\text{'B}\textbf{:}(\textbf{call } \text{a2b } [Xe_3])], \emptyset\rangle$
$c_6 = \langle[[\text{'A}, Xe_2, Xe_3],\ \text{'B}\textbf{:}Xe_2\textbf{:}(\textbf{call } \text{a2b } [Xe_3])], \{Xe_2 \neq \text{'A}\}\rangle$
$c_7 = \langle[[Xe_1, \text{'A}, Xe_3],\ Xe_1\textbf{:}\text{'B}\textbf{:}(\textbf{call } \text{a2b } [Xe_3])], \{Xe_1 \neq \text{'A}\}\rangle$
$c_8 = \langle[[Xe_1, Xe_2, Xe_3],\ Xe_1\textbf{:}Xe_2\textbf{:}(\textbf{call } \text{a2b } [Xe_3])], \{Xe_1 \neq \text{'A}, Xe_2 \neq \text{'A}\}\rangle$
$c_9 = \langle[[\text{'A}, \text{'A}, \text{'A}],\ [\text{'B}, \text{'B}, \text{'B}]], \emptyset\rangle$
$\Rightarrow$ **Answer:** $([Xe_0 \mapsto \text{'A}, Xe_1 \mapsto \text{'A}, Xe_2 \mapsto \text{'A}, Xe_3 \mapsto \text{'A}], \emptyset)$
$c_{10} = \langle[[\text{'A}, \text{'A}, Xe_3],\ [\text{'B}, \text{'B}, Xe_3]], \{Xe_3 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{11} = \langle[[\text{'A}, Xe_2, \text{'A}],\ [\text{'B}, Xe_2, \text{'B}]], \{Xe_2 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{12} = \langle[[\text{'A}, Xe_2, Xe_3],\ [\text{'B}, Xe_2, Xe_3]], \{Xe_2 \neq \text{'A}, Xe_3 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{13} = \langle[[Xe_1, \text{'A}, \text{'A}],\ [Xe_1, \text{'B}, \text{'B}]], \{Xe_1 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{14} = \langle[[Xe_1, \text{'A}, Xe_3],\ [Xe_1, \text{'B}, Xe_3]], \{Xe_1 \neq \text{'A}, Xe_3 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{15} = \langle[[Xe_1, Xe_2, \text{'A}],\ [Xe_1, Xe_2, \text{'B}]], \{Xe_1 \neq \text{'A}, Xe_2 \neq \text{'A}\}\rangle$ $\Rightarrow$ No answers
$c_{16} = \langle[[Xe_1, Xe_2, Xe_3],\ [Xe_1, Xe_2, Xe_3]], \{Xe_1 \neq \text{'A}, Xe_2 \neq \text{'A}, Xe_3 \neq \text{'A}\}\rangle$
$\Rightarrow$ **Answer:** $([Xe_0 \mapsto \text{'B}, Xe_1 \mapsto \text{'B}, Xe_2 \mapsto \text{'B}, Xe_3 \mapsto \text{'B}], \emptyset)$

Fig. 1. Perfect process tree without backpropagation

on the c-variables, so $\widehat{r}_{io}$ is empty. Thus, we have the initial io-class:

$$cls_{io} = \langle(\underbrace{[[Xe_1, Xe_2, Xe_3]]}_{\widehat{ds}_{in}}, \underbrace{[[Xe_0, Xe_0, Xe_0], [\text{'B}, \text{'B}, \text{'B}]]}_{\widehat{d}_{out}}), \underbrace{\emptyset}_{\widehat{r}_{io}}\rangle . \qquad (7)$$

The process tree is finite because the length of the input list is fixed when we trace f with cls_{io} in (7), but the construction time is exponential in the length of the $\widehat{ds}_{in}$ since tracing explores all possibilities for the three elements of the list. For example, trace the computation along the configuration sequence (Fig. 1):

$$\begin{array}{ll}
c_1 = \langle(\textbf{call } \text{f } [Xe_1, Xe_2, Xe_3]), \emptyset\rangle & \downarrow \text{unfold} \\
c_2 = \langle[[Xe_1, Xe_2, Xe_3],\ (\textbf{call } \text{a2b } [Xe_1, Xe_2, Xe_3])], \emptyset\rangle & \downarrow \theta_1 = [Xe_1 \mapsto \text{'A}] \\
c_3 = \langle[[\text{'A}, Xe_2, Xe_3],\ \text{'B}\textbf{:}(\textbf{call } \text{a2b } [Xe_2, Xe_3])], \emptyset\rangle & \downarrow \neg\theta_2 = \{Xe_2 \neq \text{'A}\} \\
c_6 = \langle[[\text{'A}, Xe_2, Xe_3],\ \text{'B}\textbf{:}Xe_2\textbf{:}(\textbf{call } \text{a2b } [Xe_3])], \{Xe_2 \neq \text{'A}\}\rangle & \downarrow \ldots
\end{array}$$

$$\begin{tikzcd}
 & c_1 \arrow[d] & \\
 & c_2 \arrow[d, dotted, "{\theta' = [Xe_1 \mapsto Xe_0,\ Xe_2 \mapsto Xe_0,\ Xe_3 \mapsto Xe_0]}"'] & \\
 & c_2' \arrow[dl, "{\theta_1 = [Xe_0 \mapsto \text{'A}]}"'] \arrow[dr, "\neg\theta_1"] & \\
c_3 \arrow[d, "*"'] & & c_4 \arrow[d, "*"'] \\
c_5 & & c_6
\end{tikzcd}$$

$c_1 = \langle(\textbf{call}\ \text{f}\ [Xe_1, Xe_2, Xe_3]), \emptyset\rangle$

$c_2 = \langle \widehat{s}_2, \emptyset\rangle = \langle[[Xe_1, Xe_2, Xe_3],\ (\textbf{call}\ \text{a2b}\ [Xe_1, Xe_2, Xe_3])], \emptyset\rangle$

partially computed output has form $\widehat{d}'_{out} = skel(\widehat{s}_2) = [[Xe_1, Xe_2, Xe_3],\ Xe^{\diamond}]$, *necessary condition to meet desired output is given by substitution* θ' *(computed by mgu)*

$c_2' = \langle[[Xe_0, Xe_0, Xe_0],\ (\textbf{call}\ \text{a2b}\ [Xe_0, Xe_0, Xe_0])], \emptyset\rangle$

$c_3 = \langle[[\text{'A}, \text{'A}, \text{'A}],\ \text{'B}\textbf{:}(\textbf{call}\ \text{a2b}\ [\text{'A}, \text{'A}])], \emptyset\rangle$

$c_4 = \langle[[Xe_0, Xe_0, Xe_0],\ Xe_0\textbf{:}(\textbf{call}\ \text{a2b}\ [Xe_0, Xe_0])], \{Xe_0 \neq \text{'A}\}\rangle$

$c_5 = \langle[[\text{'A}, \text{'A}, \text{'A}],\ [\text{'B}, \text{'B}, \text{'B}]], \emptyset\rangle$

$\Rightarrow$ **Answer:** $([Xe_0 \mapsto \text{'A}, Xe_1 \mapsto \text{'A}, Xe_2 \mapsto \text{'A}, Xe_3 \mapsto \text{'A}], \emptyset)$

$c_6 = \langle[[Xe_4, Xe_4, Xe_4],\ [Xe_4, Xe_4, Xe_4]], \{Xe_0 \neq \text{'A}\}\rangle$

$\Rightarrow$ **Answer:** $([Xe_0 \mapsto \text{'B}, Xe_1 \mapsto \text{'B}, Xe_2 \mapsto \text{'B}, Xe_3 \mapsto \text{'B}], \emptyset)$

Fig. 2. Perfect process tree with backpropagation

None of the descendents of c_6 can ever lead to an answer because they all violate the requirement in $\widehat{d}_{out}$ that its first component is a list containing three identical elements (restriction $Xe_2 \neq$ 'A requires that the second element Xe_2 is different from the first 'A). Instead of tracing the program using only information from the input class, we also backpropagate information from the output class to the input class. Let us intersect the current io-class at c_2 with the given io-class (7):

$$\begin{array}{ll} & \langle(([[Xe_1, Xe_2, Xe_3]], [[Xe_1, Xe_2, Xe_3], \bullet]), \emptyset\rangle \\ \star & \langle(([[Xe_1, Xe_2, Xe_3]], [[Xe_0, Xe_0, Xe_0],\ [\text{'B}, \text{'B}, \text{'B}]]), \emptyset\rangle \ = \ \{(\theta, \emptyset)\} \end{array} \tag{8}$$

where substitution $\theta = [Xe_1 \mapsto Xe_0,\ Xe_2 \mapsto Xe_0,\ Xe_3 \mapsto Xe_0,\ \bullet \mapsto [\text{'B}, \text{'B}, \text{'B}]]$. The intersection is not empty and the result is a substitution-restriction pair $(\theta, \emptyset)$ that, when applied to any of the two io-classes, gives a new io-class that represents the domain of the intersection. We use $(\theta, \emptyset)$ to narrow configuration c_2 to $c_2' = \langle[[Xe_0, Xe_0, Xe_0],\ (\textbf{call}\ \text{a2b}\ [Xe_0, Xe_0, Xe_0])], \emptyset\rangle$. Because the input of a2b is now limited to lists containing three identical elements, backpropagating the result of the intersection into c_2 leads to a dramatic speed-up: the search time becomes linear in the length of the input list.

Method. The method to reduce the search space of URA is shown in Fig. 3. The central operation is the intersection ($\star$) of the approximated io-class cls'_{io}

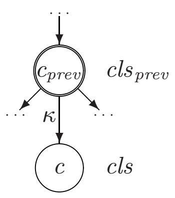
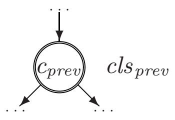
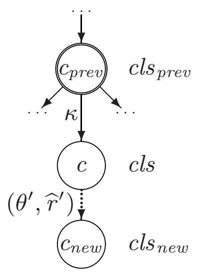
Given a request with initial io-class $cls_{io} = \langle (\widehat{ds}_{in}, \widehat{d}_{out}), \widehat{r}_{io} \rangle$ and program p:	
Current process tree: 	**Current node:** $c = \langle \widehat{s}, \widehat{r} \rangle$ current configuration, $cls = \langle \widehat{ds}, \widehat{r} \rangle$ current input class. **Approximate** the current io-class cls'_{io} of configuration c by $cls'_{io} = \langle (\widehat{ds}, skel(\widehat{s})), \widehat{r} \rangle$ where $skel(\widehat{s})$ is the known constructor skeleton of the output.
(1) Cutting: 	**if** $cls'_{io} \star cls_{io} = \emptyset$ (**intersection of io-classes**) **then** 1. Cut node c 2. Continue driving other branches.
(2) Backpropagation:  cls_{prev} κ cls $(\theta', \widehat{r}')$ cls_{new}	**else let** $cls'_{io} \star cls_{io} = \{(\theta, \widehat{r})\}$ 1. Define $(\theta', \widehat{r}')$ by removing from $(\theta, \widehat{r})$ all bindings and restrictions on c-variables that do not occur in the current input class cls. 2. Perform contractions on c and cls: $c_{new} = c/\theta'/\widehat{r}'$, $cls_{new} = cls/\theta'/\widehat{r}'$. 3. Add a new branch labeled $(\theta', \widehat{r}')$ and a new node with configuration c_{new} and input class cls_{new}. 4. Continue driving.

Fig. 3. Reduction of search space by cutting and backpropagation

with the given io-class cls_{io}. If the intersection is empty, then the current configuration can *never* lead to a valid answer; otherwise, the intersection returns a contraction $(\theta, \widehat{r})$ containing a substitution θ and restriction $\widehat{r}$ which may further constrain the current configuration. The process tree cannot become larger by performing the operations in Fig. 3, but it may have less edges. This can make URA faster and more terminating. We now describe the main operations:

1. *Intersection* ($\star$) of two io-classes is based on the *most general unifier* (mgu). The mgu examines the entire constructor skeletons in $\widehat{dd}_1$ and $\widehat{dd}_2$ of the two classes. If the mgu succeeds, it is necessary to check that the substitution $\theta = mgu(...)$ does not lead to a contradiction when applied to the restrictions $(\widehat{r}_1 + \widehat{r}_2)$ [3].[1] Thus, there are three cases: (i) the mgu fails, (ii) the mgu succeeds, but θ leads to a contradiction in the restrictions, and (iii) the mgu succeeds and θ is consistent with the restrictions and the intersection is not empty.

[1] Ex.: applying $\theta = [Xe_1 \mapsto \text{'A}]$ to $\widehat{r} = \{Xe_1 \neq \text{'A}\}$ leads to a contradiction: $\{\text{'A} \neq \text{'A}\}$.

Definition 1 (intersection of io-classes). *Let* cls_1, cls_2 *be two io-classes,* $cls_1 = \langle \widehat{dd}_1, \widehat{r}_1 \rangle$ *and* $cls_2 = \langle \widehat{dd}_2, \widehat{r}_2 \rangle$ *such that* $\mathrm{var}(cls_1) \cap \mathrm{var}(cls_2) = \emptyset$, *and let* $mgu(\widehat{dd}_1, \widehat{dd}_2)$ *denote the most general unifier of* $\widehat{dd}_1$ *and* $\widehat{dd}_2$*, if it exists, then define* io-class intersection *($\star$) by*

$$cls_1 \star cls_2 \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } mgu(\widehat{dd}_1, \widehat{dd}_2) \text{ fails} \\ \emptyset & \text{if } (\widehat{r}_1 + \widehat{r}_2)/\theta = \{\mathsf{contra}\} \text{ where } \theta = mgu(\widehat{dd}_1, \widehat{dd}_2) \\ \{(\theta, \widehat{r})\} & \text{otherwise, where } \theta = mgu(\widehat{dd}_1, \widehat{dd}_2),\ \widehat{r} = (\widehat{r}_1 + \widehat{r}_2)/\theta\,. \end{cases}$$

2. *Constructor skeleton* (*skel*). The output of the current configuration can be approximated by taking the constructor skeleton $skel(\widehat{s})$ of the current state $\widehat{s}$, that is, by replacing all function calls in $\widehat{s}$ with fresh c-variables. For example, if $\widehat{s}$ = 'B:'B:(**call** a2b ...) then $skel(\widehat{s})$ = 'B:'B:$Xe^{\diamond}$ where $Xe^{\diamond}$ is a fresh c-variable. The io-class cls'_{io} of the current configuration c can then be approximated by combining the current input class cls and the approximated output $skel(\widehat{s})$. The operation *skel* is a pure syntactic approximation of the output and does not use any semantic information about the functions defined in a program.

Both operations, intersection and constructor skeleton, are important for our method. If we do not use the intersection operation, but approximate it or only check whether the intersection of the two classes is empty, we might miss a chance to reduce the search space by not backpropagating enough information into the current configuration. If we delay the intersection operation or do not examine all of the known constructor skeleton at the current configuration, we might miss a chance to cut a node that never leads to an answer.

4 Dealing with MGU-Based Equality

We introduced an improvement for inverse computation by URA in the previous section and showed that intersection is a powerful operation to test *during tracing* whether two io-classes represent sets of values that share values. We used intersection to predict whether the current configuration may lead to an answer that lies in the given output domain or not. If the intersection is empty then the current configuration can never lead to a valid answer. This observation leads us to the second idea, namely, to the introduction of a new *mgu-based equality* in the source language which is different from the usual matching-based operations found in most functional-logic languages and lazy functional languages.

```
(define f [x]
   ([x, (call a2b [x])]))

(define a2b [x]
   (if (cons? x h t _)
      (if (equ? h 'A)
         ('B:(call a2b [t]))
         ( h :(call a2b [t])))
      []))
```

We use the first-order, lazy functional language Nested Typed S-Graph (NTSG) that extends the original language S-Graph [6] with nested function calls and a new non-atomic equality. The body of a function is an expression e which is either a function call, a conditional, a cons-pair (e : e), an atom 'z or a program variable x (Fig. 4). The semantics of NTSG is similar to the semantics of TSG that was given elsewhere [2,3]. Values can be

p	$::=$	q^+	Program
q	$::=$	$(\textbf{define}\ f\ x^*\ e)$	Definition
e	$::=$	$(\textbf{call}\ f\ e^*)\ \mid\ (\textbf{if}\ k\ e\ e)\ \mid\ (e\ :\ e)\ \mid\ 'z\ \mid\ x$	Expression
k	$::=$	$(\textbf{equ?}\ e\ e)\ \mid\ (\textbf{cons?}\ e\ xe\ xe\ xa)$	Condition
x	$::=$	$xe\ \mid\ xa$	Typed variable

Fig. 4. Abstract syntax of NTSG

Condition Equ?

$$\frac{e = e' \quad \mathit{passive}(e) \quad \mathit{passive}(e')}{\vdash_{\mathit{if}}\ (\textbf{equ?}\ e\ e')\ e_1\ e_2 \Rightarrow e_1}$$

$$\frac{\mathit{mgu}(\mathit{skel}(e), \mathit{skel}(e'))\ \text{fails}}{\vdash_{\mathit{if}}\ (\textbf{equ?}\ e\ e')\ e_1\ e_2 \Rightarrow e_2}$$

Transition for Conditional

$$\frac{\mathit{redex}(s) = (s', [\bullet \mapsto (\textbf{if}\ k\ e_1\ e_2)]) \quad \vdash_{\mathit{if}}\ k\ e_1\ e_2 \Rightarrow e}{\vdash_{\Gamma}\ s \to s'/[\bullet \mapsto e]}$$

Fig. 5. Excerpt of operational semantics: the equality test

tested and/or decomposed in two ways. Condition **equ?** checks the equality of two S-expressions and condition (**cons?** $e\ xe'\ xe''\ xa$) works in the following way: if e has the form $(e'\ :\ e'')$, then variable xe' is bound to head e' and variable xe'' to tail e''; if e is an atom, then variable xa is bound to this atom. For simplicity, we write '_' when a variable is not used (*e.g.*, in the first condition of function a2b where the else-branch returns an empty list). The original URA [2,3] allowed only an atomic equality test **eqa?**, while our extension uses **equ?** instead. We will now discuss the equality test in the context of a lazy language.

Operational semantics. The semantics of the equality test (**equ?** $e\ e'$) is straightforward (Fig. 5; other rules omitted due to limited space): The true-branch (e_1) is chosen if e and e' are passive and identical (*strict equality*). An expression e is *passive* iff it contains no function calls and no **if**-subexpressions.[2] The false-branch (e_2) is chosen if $\mathit{mgu}(\mathit{skel}(e), \mathit{skel}(e'))$ fails, that is, when the constructor skeletons of e' and e'' disagree in some position (*non-strict non-equality*). The operator *skel* replaces every function call and **if**-subexpression in an expression by a fresh c-variable. If the *mgu* fails then e and e' can never become equal even if all redexes in the two expressions are evaluated. This can speedup evaluation by detecting a failure early. When the *mgu* does not fail and at least one of the two expressions is not passive, neither strict equality nor

[2] In the operational semantics, when both operands are ground and passive, *mgu* reduces to pattern matching; only during driving, the mechanism is fully used.

Condition Equ?

$$\frac{\theta = mgu(\widehat{e}, \widehat{e}') \quad passive(\widehat{e}) \quad passive(\widehat{e}')}{\Vdash_{if} (\mathbf{equ?}\ \widehat{e}\ \widehat{e}')\ \widehat{e}_1\ \widehat{e}_2 \Rightarrow (\widehat{e}_1, \theta)}$$

$$\frac{\theta = mgu(skel(\widehat{e}), skel(\widehat{e}')) \quad \neg(passive(\widehat{e}) \wedge passive(\widehat{e}'))}{\Vdash_{if} (\mathbf{equ?}\ \widehat{e}\ \widehat{e}')\ \widehat{e}_1\ \widehat{e}_2 \Rightarrow ((\mathbf{if}\ (\mathbf{equ?}\ \widehat{e}\ \widehat{e}')\ \widehat{e}_1\ \widehat{e}_2), \theta)}$$

$$\frac{mgu(skel(\widehat{e}), skel(\widehat{e}'))\ \text{fails}}{\Vdash_{if} (\mathbf{equ?}\ \widehat{e}\ \widehat{e}')\ \widehat{e}_1\ \widehat{e}_2 \Rightarrow (\widehat{e}_2, \emptyset)} \qquad \frac{\theta = mgu(skel(\widehat{e}), skel(\widehat{e}'))}{\Vdash_{if} (\mathbf{equ?}\ \widehat{e}\ \widehat{e}')\ \widehat{e}_1\ \widehat{e}_2 \Rightarrow (\widehat{e}_2, \neg\theta)}$$

Transition for Conditional

$$\frac{redex(\widehat{s}) = (\widehat{s}', [\bullet \mapsto (\mathbf{if}\ k\ \widehat{e}_1\ \widehat{e}_2)]) \quad \Vdash_{if} k\ \widehat{e}_1\ \widehat{e}_2 \Rightarrow (\widehat{e}, \kappa) \quad \widehat{r}/\kappa \neq \{\,\mathsf{contra}\,\}}{\vdash_{\Gamma} \langle \widehat{s}, \widehat{r} \rangle \rightarrow \langle \widehat{s}'/[\bullet \mapsto \widehat{e}], \widehat{r} \rangle/\kappa}$$

Fig. 6. Excerpt of trace semantics for perfect process trees: the equality test

non-strict non-equality can be established (the expression that is not passive has to be evaluated further until it becomes passive or *mgu* fails). While strict equality is common in functional-logic languages, the use of *mgu* to detect a failure fast by examining the available constructor skeleton, and in an evaluation order independent way, is not (even in modern functional-logic languages like Curry [9] or Babel [11]). The function *redex* in the transition rule for conditionals (other transition rules omitted) picks the conditional according to some evaluation strategy and splits the current state s into a context s' containing a hole $\bullet$ and a substitution that binds the redex.

Perfect driving. URA traces the computation of a program with partially specified input and builds a perfect process tree representing all possible computation traces, as outlined in Sects. 2 and 3. In contrast to the operational semantics, the input to a program may contain c-variables (non-ground input) and there may not be enough information to decide which branch to choose when tracing conditionals. In this case, tracing has to follow both branches. The assumptions that lead us to choose a branch are returned as additional information from the rules for conditionals. The rules for equality return a pair $(\widehat{e}, \kappa)$ where $\widehat{e}$ is an expression and κ a contraction. A contraction κ is either a substitution θ or its negated form $\neg\theta$. The transition rule for the conditional **if** checks whether there is a contradiction between the new contraction and the current restriction. This is done be applying the new contraction κ to restriction $\widehat{r}$ and checking that there is no contradiction: $\widehat{r}/\kappa \neq \{\,\mathsf{contra}\,\}$. If there is a contradiction, then the branch is infeasible. We now describe the tracing semantics for **equ?** (Fig. 6).

The rule for selecting the false-branch (3rd rule) is identical to the one in the operational semantics: if some constructors in the constructor skeletons disagree, the false-branch must be selected (and no other rule applies). Because we are dealing with non-ground expressions that may contain c-variables, *mgu* may succeed and return a substitution θ. Then there are two possibilities and in

For all NTSG-programs p and for all input-output classes cls_{in}:

$$Inv(\text{NTSG}, p, cls_{io}) = Inv(\text{NTSG}, p', cls'_{io}) \tag{9}$$

where

$$\begin{aligned}
cls_{io} &= \langle(([\widehat{d}_1, ..., \widehat{d}_n], \widehat{d}_{out}), \widehat{r}\rangle \\
cls'_{io} &= \langle(([\widehat{d}_1, ..., \widehat{d}_n, \widehat{d}_{out}], \text{'True}), \widehat{r}\rangle \\
p' &= [\,(\textbf{define}\ \text{main}\ [\text{in}_1, ..., \text{in}_n, \text{out}] \\
&\qquad\quad (\textbf{call}\ \text{test}\ [(\textbf{call}\ \mathit{mainfct}(p)\ [\text{in}_1, ..., \text{in}_n]), \text{out}])), \\
&\qquad (\textbf{define}\ \text{test}\ [\text{res}, \text{out}] \\
&\qquad\quad (\textbf{if}\ (\textbf{equ?}\ \text{res out})\ \text{'True 'False}))\,] +\!\!+\ p
\end{aligned}$$

Fig. 7. Answer equality of a transformed request

either case two rules apply at the same time. This leads to a branching in the perfect process tree. First, both expressions $\widehat{e}$ and $\widehat{e}'$ are passive: the 1st and 4th rule apply and substitution θ and its negated version $\neg\theta$ are propagated into the then- and else-branch, respectively. Second, at least one of the expressions, $\widehat{e}$ or $\widehat{e}'$, is not passive: the 2nd and 4th rule apply and substitution θ and its negated version $\neg\theta$ are propagated. Because at least one of the expressions is not passive, we cannot yet enter the then-branch (as in the 1st rule). We need to drive a non-passive expression and again check the result.

5 Equivalence Transformation of Requests

Now we show another, surprisingly simple, solution based on the mgu-based equality test that we introduced in the previous section. Instead of implementing the method in Fig. 3 in URA, we perform an equivalence transformation of the given request for inverse computation into the source language of URA.

There is an implementation of URA for NTSG according to [3] for the given source language. Instead of modifying URA to implement the method described in Sect. 3, we perform an equivalence transformation of the given request. The transformation is shown in Fig. 7. Given a program p and input-output class cls_{in}, we transform them into a new program p' and a new input-output class cls'_{in}. The new program p' is constructed by adding two new functions to the original program.[3] The new main function is defined as a p's main function call nested to call of function test. Function test compares the result computed by p with the desired output out, and returns the corresponding Boolean value.

The new input-output class cls'_{in} is a reformatted version of cls_{in} where the desired output is fixed to 'True and the user desired output is now the last argument for the new main function of p'. The restriction $\widehat{r}$ remains unchanged.

Theorem 1 (answer equivalence of transformed request). *Given language* NTSG*, for all programs p and for all io-classes cls_{io}, equation* (9) *holds.*

[3] If the new functions ("main", "test") occur already in p then they are renamed.

This transformation achieves the effects described above. Instead of extending URA, we encode the inversion problem in the source language taking advantage of the equality test (Fig. 7). This is possible because the semantics of the equality test coincides with the desired *mgu*-based method in Fig. 3.

Why does it work? The transformation of the original request to the new request makes test **equ?** the root of the perfect process tree. This has two effects. First, the test demands the calculation of the components of p's output until a mismatch with the desired output out is found, which establishes 'False. This will stop any further development in the corresponding branch of the perfect process tree. This is the cutting operation. Second, any new contractions on c-variables obtained by **equ?** are applied to the current configuration. This achieves backpropagation. Another advantage is that the driving of p is guided by the equality test. This avoids the unnecessary computations which do not contribute to the goal of establishing an answer to the inversion problem. The effectiveness of this approach will be demonstrated in the next section.

6 Demonstration

We demonstrate the improved efficiency of inverse computation using the equivalence transformation in Fig. 7. The examples include inverse computation of a tree traversal function [12] and experiments comparing inverse computation in URA with equivalent requests in Curry [9], a modern functional-logic language.[4]

1. A breadth-first labeling of a tree with respect to a given list of values is a labeling of nodes in the tree with values in the list in breadth-first order. We implemented a program in NTSG which, given a binary tree, collects the values of the nodes by a breadth-first traversal. Inverse computation of the program then performs the desired breadth-first labeling. We performed two experiments: URA before and after the equivalence transformation of the request. Given a list with 13 values, the time to find the 132 trees labeled in breadth-first order is 216.05 secs; the search does not terminate. After the equivalence transformation of the request, the time to find the 132 trees is 6.90 secs (*that is 31.3 times faster*); after 15.43 secs the search terminates!

2. Modern functional-logic programming languages like Curry [9] and Babel [11] allow programs to be written in a functional programming style while their narrowing-based semantics can evaluate programs with non-ground input. Requests for inverse computation can be formulated in these languages using the equality operator available in these language (*e.g.*, Curry's `=:=`), much like the transformation in Fig. 7, and setting 'true' as desired output of 'test'. However, since narrowing in Curry and Babel is not based on perfect splitting of io-classes (they can overlap, duplicating search space) and the equality operators are not mgu-based (*e.g.*, in Curry, Babel), they do miss important chances for cutting

[4] All running times on CPU AMD Athlon 64 3500+ (2.2GHz), RAM 2GB, OS Debian Linux, The Glorious Glasgow Haskell Compilation System, version 6.4 (with -H1536m run-time option, *e.g.* 1.5 GB heap size). Compiler: Curry into Prolog from Portland Aachen Kiel Curry System (PACKS) 1.6.1-5 and SICStus Prolog 3.12.3.

and backpropagation. (Example omitted.) As a result, inverse computation can be less efficient and less work terminating than with URA. It will be an interesting task to add perfect driving and mgu-based equality to a language like Curry which is also meant as a platform for experiments in functional-logic languages.

7 Related Work

The Universal Resolving Algorithm presented in this paper is derived from *perfect driving* [6] and is combined with a mechanical extraction of the answers (*cf.* [1,14]) giving the algorithm the power comparable to SLD-resolution, but for a first-order functional language with tail-recursion (see [7]). The complete algorithm is given in [3]. The constraint system for perfect driving can be normalized by a rewrite system [16]. The idea for the algorithm was originally conceived in the context of the programming language Refal [18]. Logic programming inherently supports inverse computation. The use of an appropriate inference procedure permits the determination of any computable answer. Recently, work in this direction has been done regarding the integration of the functional and logic programming paradigm using narrowing, a unification-based goal-solving mechanism [8]; for a survey see [4]. The relation to functional-logic languages was already discussed in Sect. 6.

8 Conclusion and Future Work

We presented an extension of URA based on intersection that improves efficiency and termination of inverse computation and introduce a new mgu-based equality that allows us to achieve the same effect by mapping requests into the source language. By doing so, we found that such an equality operator might considerably improve inverse computation also in functional-logic languages. With the mgu-based equality we established a solution for dealing with equality under perfect driving. Our techniques work best for functions that produce some part of the output in each recursion because partially known results can be examined during the construction of the process tree and infeasible branches can be cut or additional information can be propagated back into the tree. This can drastically reduce the size of the tree and even turn an infinite into a finite tree. Our method might be viewed as a form of 'reverse' URA because output information is exploited to guide the construction of the perfect process tree. Some methods follow the traces in reverse order [13,14].

Further work is desirable in several directions. First, we plan to establish more empirical results of the algorithm presented in this paper. The algorithm is fully implemented in Haskell which serves our experimental purposes quite well. Second, recent works [10] on term rewrite systems define the notion of fully-collapsed jungles on graphs. We want to investigate the use of these techniques in the context of process tree construction as in [15]. Third, it would be interesting to examine the benefits of mgu-based equality by implementing it in a functional-logic system together with a constraint system that leads to perfect process trees.

References

1. S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).
2. S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 1837, 187–212. Springer-Verlag, 2000.
3. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
4. E. Albert, G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
5. A. P. Ershov. On the essence of compilation. In E. Neuhold (ed.), *Formal Description of Programming Concepts*, 391–420. North-Holland, 1978.
6. R. Glück, A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, A. Rauzy (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
7. R. Glück, M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, 165–181. Springer-Verlag, 1994.
8. M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
9. M. Hanus. Curry: an integrated functional logic language (version 0.8). Report, University of Kiel, 2003.
10. B. Hoffmann, D. Plump. Implementing term rewriting by jungle evaluation. *Informatique Théorique et Applications/Theoretical Informatics and Applications*, 25(5):445–472, 1991.
11. J. J. Moreno-Navarro, M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language Babel. *J of Logic Programming*, 12(3):191–223, 1992.
12. S.-C. Mu, R. Bird. Inverting functions as folds. In E. Boiten, B. Möller (eds.), *Mathematics of Program Construction*, LNCS 2386, 209–232. Springer-Verlag, 2002.
13. A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, et al. (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 249–260. Springer-Verlag, 1996.
14. A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.
15. J. P. Secher. Driving in the jungle. In O. Danvy, A. Filinsky (eds.), *Programs as Data Objects. Proceedings*, LNCS 2053, 198–217. Springer-Verlag, 2001.
16. J. P. Secher, M. H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, A. V. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 113–127. Springer-Verlag, 2000.
17. J. P. Secher, M. H. Sørensen. From checking to inference via driving and DAG grammars. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 41–51. ACM Press, 2002.
18. V. F. Turchin. Equivalent transformations of recursive functions defined in Refal. In *Tjeorija Jazykov i Mjetody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods). (Kiev-Alushta, USSR)*, 31–42, 1972. (In Russian).
19. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

Program Generation in the Equivalent Transformation Computation Model Using the Squeeze Method

Kiyoshi Akama[1], Ekawit Nantajeewarawat[2], and Hidekatsu Koike[3]

[1] Information Initiative Center, Hokkaido University, Hokkaido, Japan
akama@iic.hokudai.ac.jp

[2] Computer Science Program, Sirindhorn International Institute of Technology
Thammasat University, Pathumthani, Thailand
ekawit@siit.tu.ac.th

[3] Faculty of Social Information, Sapporo Gakuin University, Hokkaido, Japan
koike@sgu.ac.jp

Abstract. In the equivalent transformation (ET) computation model, a specification provides background knowledge in a problem domain, a program is a set of prioritized rewriting rules, and computation consists in successive reduction of problems by rule application. As long as meaning-preserving rewriting rules, called ET rules, with respect to given background knowledge are used, correct computation results are guaranteed. In this paper, a general framework for program synthesis in the ET model is described. The framework comprises two main phases: (1) equivalent transformation of specifications, and (2) generation of a program from an obtained specification. A method for program generation in the second phase, called the squeeze method, is presented. It constructs a program by accumulation of ET rules one by one on demand, with the goal of producing a correct, efficient, and non-redundant program.

1 Introduction

Equivalent transformation (ET) is one of the most fundamental principles of computation, and it provides a simple and general basis for verification of computation correctness. Computation by ET was initially implemented in experimental natural language understanding systems at Hokkaido University in the early 90's, and the idea was further developed into a new computation model, called the *ET model* [1,4]. A program in this model is a set of prioritized rewriting rules for meaning-preserving transformation of problems, and a problem solving process consists in successive rule application. Besides extensive use in the domain of first-order terms, the model has been applied in several data domains, including RDF and XML (e.g. in [7] and [16], respectively).

Advantages of the ET model are best seen from the viewpoint of program synthesis, where the possibility and effectiveness of generating correct and efficient programs from specifications are of central importance. Programs are clearly

1) $pal(X) \leftarrow rv(X, X)$
2) $rv([\,], [\,]) \leftarrow$
3) $rv([A|X], Y) \leftarrow rv(X, R), ap(R, [A], Y)$
4) $ap([\,], X, X) \leftarrow$
5) $ap([A|X], Y, [A|Z]) \leftarrow ap(X, Y, Z)$

Fig. 1. Definite clauses defining the predicates *pal*, *rv*, and *ap*

separated from specifications in this model. A specification provides background knowledge for associating declarative meanings with problems and specifies a set of problems of interest—no procedural semantics is associated with specifications. From a specification, a program consisting of procedural rewriting rules is constructed. The separation between programs and specifications greatly widens the possibility of program synthesis—several kinds of rewriting rules with varying procedural expressive power can be generated from a specification. This is in sharp contrast to program synthesis in declarative computation paradigms such as logic programming [11] and functional programming [8], where specifications are regarded as programs by assuming a certain predetermined procedural semantics and, consequently, program improvement can only be achieved by transformation of specifications (i.e., program transformation [13,14,15]).

The primary objective of this paper is to develop a basic method for program construction in the ET model. A general program synthesis framework in this model is presented. It consists of two phases: (1) equivalent transformation of specifications, and (2) generation of a set of prioritized rewriting rules from a specification. Methods and techniques from the wealth of literature on program transformation and partial deduction (e.g. [9,12,13,14,15]) readily lend themselves as tools for the first phase. For the second phase, a heuristic program generation method, called the *squeeze method*, is introduced.

The squeeze method generates a program by demand-driven accumulation of meaning-preserving rewriting rules, called *ET rules*. It capitalizes on several advantages of the fundamental structure of the ET model; e.g., the correctness and efficiency of an ET rule can be checked individually, and execution of a partial program always yields problem reduction that provides a meaningful clue to generation of new ET rules towards completing the program. These characteristic features facilitate componentwise program generation [6]—generating a correct and efficient program by creating individually correct and efficient program components (ET rules) one by one on demand—which appears to be an effective and indispensable approach to program synthesis. Based on the squeeze method, a program synthesis system has been implemented and used for constructing many nontrivial programs.

To begin with, computation by transformation of problems in the ET model is reviewed in Section 2. The general program synthesis framework and the squeeze method are presented in Sections 3 and 4, respectively. Although the ET model can deal with data structures of various kinds, the paper is deliberately confined to the domain of first-order terms for reasons of simplicity.

$$
\begin{array}{ll}
r_{pal}\colon & pal(*x) \Rightarrow rv(*x, *x).\\
r_{rv_1}\colon & rv([*a|*x], *y) \Rightarrow rv(*x, *v), ap(*v, [*a], *y).\\
r_{rv_2}\colon & rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y).\\
r_{ap_1}\colon & ap(*x, *y, [*a|*z]) \Rightarrow \{=(*x, [\,]), =(*y, [*a|*z])\};\\
 & \qquad\qquad\qquad\qquad \Rightarrow \{=(*x, [*a|*v])\}, ap(*v, *y, *z).\\
r_{rv_3}\colon & rv(*x, [*a|*y]) \Rightarrow \{=(*x, [*u|*v])\}, rv(*v, *w), ap(*w, [*u], [*a|*y]).\\
r_{ap_2}\colon & ap(*x, [*a], [*b, *c|*y]) \Rightarrow \{=(*x, [*b|*v])\}, ap(*v, [*a], [*c|*y]).\\
r_{ap_3}\colon & ap(*x, [*a], [*b]) \Rightarrow \{=(*x, [\,]), =(*a, *b)\}.\\
r_{rv_4}\colon & rv([\,], *x) \Rightarrow \{=(*x, [\,])\}.
\end{array}
$$

Fig. 2. Examples of rewriting rules

2 Computation in the ET Model

2.1 An Introductory Example

Assume as background knowledge a set consisting of the five definite clauses in Fig. 1, where *pal*, *rv*, and *ap* stand for "palindrome", "reverse", and "append", respectively. Consider the problem "find all ground terms t such that $[1|t]$ and $[2|t]$ are palindromes". This problem is represented in the ET model as a set consisting of a single definite clause

$$ans(X) \leftarrow pal([1|X]), pal([2|X]),$$

where *ans* stands for "answer", and this definite clause is intended to mean "X is an answer if both $[1|X]$ and $[2|X]$ satisfy the definition of *pal*". The rewriting rules in Fig. 2 are devised for solving this problem. A detailed description of their syntax and semantics is deferred until Subsection 2.3. Table 1 illustrates a sequence of problem transformation steps by successive application of these rules, where atoms to which the rules are applied are underlined and the rule applied in each step is given in the last column. The transformation sequence changes the initial problem (i.e., prb_0) into the singleton set $\{ans([\,]) \leftarrow\}$, which means "the empty list is an answer (unconditionally) to the problem and there exists no other answer". The correctness of this computation can be verified by proving that each rule in Fig. 2 is a meaning-preserving rule with respect to the predicate definitions in Fig. 1.

The rule r_{rv_2} in Fig. 2 makes replacement of two atoms simultaneously (see, e.g., its application to prb_4 in Table 1), and is called a *multi-head rule*. Every other rule in the figure replaces a single atom at a time, and is called a *single-head rule*. Each single-head rule in the figure operates as an unfolding rule using the definition of the predicate appearing in its left-hand side, and is called an *unfolding-based rule*. The rule r_{pal} is applicable to any *pal*-atom containing any arbitrary term, and is called a *general rule*. All other rules in the figure are applicable to atoms having certain specific patterns, and are called *specialized*

Table 1. Transformation of problems

Problem	Problem representation	Rule applied
prb_0	$\{ans(X) \leftarrow \underline{pal([1\|X])}, pal([2\|X])\}$	r_{pal}
prb_1	$\{ans(X) \leftarrow rv([1\|X],[1\|X]), \underline{pal([2\|X])}\}$	r_{pal}
prb_2	$\{ans(X) \leftarrow \underline{rv([1\|X],[1\|X])}, rv([2\|X],[2\|X])\}$	r_{rv_1}
prb_3	$\{ans(X) \leftarrow rv(X, A1), ap(A1,[1],[1\|X]), \underline{rv([2\|X],[2\|X])}\}$	r_{rv_1}
prb_4	$\{ans(X) \leftarrow \underline{rv(X, A1)}, ap(A1,[1],[1\|X]), \underline{rv(X, A2)}, ap(A2,[2],[2\|X])\}$	r_{rv_2}
prb_5	$\{ans(X) \leftarrow rv(X, A1), \underline{ap(A1,[1],[1\|X])}, ap(A1,[2],[2\|X])\}$	r_{ap_1}
prb_6	$\{ans([\,]) \leftarrow rv([\,],[\,]), ap([\,],[2],[2]),$ $ans(X) \leftarrow \underline{rv(X,[1\|A3])}, ap(A3,[1],X), ap([1\|A3],[2],[2\|X])\}$	r_{rv_3}
prb_7	$\{ans([\,]) \leftarrow rv([\,],[\,]), ap([\,],[2],[2]),$ $ans([A4\|A5]) \leftarrow rv(A5, A6), ap(A6,[A4],[1\|A3]), ap(A3,[1],[A4\|A5]),$ $\underline{ap([1\|A3],[2],[2, A4\|A5])}\}$	r_{ap_2}
prb_8	$\{ans([\,]) \leftarrow rv([\,],[\,]), \underline{ap([\,],[2],[2])}\}$	r_{ap_3}
prb_9	$\{ans([\,]) \leftarrow \underline{rv([\,],[\,])}\}$	r_{rv_4}
prb_{10}	$\{ans([\,]) \leftarrow\}$	–

rules. Employment of specialized rules allows *content-based control* of computation [4]—an appropriate reduction step can be decided based on the run-time contents of clauses occurring in a computation state.

2.2 Comparison with Computation in Logic Programming

When computation by SLD resolution is viewed in the ET framework, expansion of a node (generation of its children) in a search tree for finding SLD-refutations corresponds to an unfolding transformation step. Accordingly, computation in logic programming can be seen as computation using only one specific class of rewriting rules, i.e., single-head general unfolding-based rules. By employment of such a restricted class of rules alone, it is often difficult to achieve effective computation control, in particular, for preventing infinite computation or for improving computation efficiency.

As an example, consider the query illustrated in the preceding subsection, which is represented in logic programming as the goal clause $\leftarrow pal([1|X])$, $pal([2|X])$. It was shown in [4] that when executing this query, any logic program for checking palindromes enters infinite computation after giving $X = [\,]$, and thus fails to infer that the empty list is the "only" ground instance of X that satisfies the query. This difficulty is overcome in the ET model by content-based control of computation and the possibility of employing several types of rewriting rules, including specialized rules and multi-head rules. Consider, for example, the role of the multi-head rule r_{rv_2} in successful termination of the reduction sequence in Table 1. The application of this rule creates an additional information connection, i.e., via the common variable $A1$, between descendants of $pal([1|X])$ (i.e., the *rv*-atom and the first *ap*-atom in prb_5) and a descendant of $pal([2|X])$ (i.e., the second *ap*-atom in prb_5). Through this connection, the constraint on a term possibly substituted for X imposed by the first *pal*-atom and that imposed by the second *pal*-atom can be exchanged as a finite information pattern; i.e., if $X \neq []$, then by the *rv*-atom in prb_5, $A1 \neq []$, and then by the first and the second *ap*-atoms in prb_5, $A1 = [1|t]$ and $A1 = [2|t']$, respectively, for some terms t and t'. Consequently, the contradiction occurring when $X \neq []$ can be found in finite reduction steps. Such an additional information connection cannot be created by single-head rules.[1] Note that the multi-head rule r_{rv_2} is devised based on the functionality of the “reverse” relation, and its operation is completely different from unfolding.

Problem transformation in the ET model and program transformation in logic programming [13,15] have different objectives. The former is a method for problem solving; it aims to reduce a problem into a simplified form. The latter is a method for program improvement; it aims to derive a more efficient logic program from an initially given one. Since computation using a logic program always corresponds to computation using only single-head general unfolding-based rules, any palindrome-checking logic program obtained from program transformation still fails to terminate when executing the query considered above.

2.3 Syntax and Operational Semantics of Rewriting Rules

The class of rewriting rules used in this paper will now be described. Usual atoms are used in a rule to denote atom patterns in a definite clause. In addition, atoms of a special kind, called *executable atoms*, are used to denote built-in operations; e.g. an executable atom $=(t, t')$ denotes the operation “find the most general unifier of terms t and t'” (‘=’ denotes the unification operation).[2] An executable atom is evaluated by some predetermined evaluator, and if the evaluation succeeds, it yields a substitution as the result; e.g. the evaluation of $=([1|X], [Y, 2, Z])$ yields $\{X/[2, Z], Y/1\}$.

A rewriting rule r in this paper takes the form

$$\begin{array}{rcl} Hs & \Rightarrow & \{Es_1\}, Bs_1; \\ & \cdots & \\ & \Rightarrow & \{Es_n\}, Bs_n, \end{array}$$

where $n \geq 1$, *Hs* is a nonempty sequence of (usual) atoms, the Es_i are sequences of executable atoms, and the Bs_i are sequences of (usual) atoms. For each i ($1 \leq i \leq n$), the pair $\langle\{Es_i\}, Bs_i\rangle$ is called a *body* of r, and $\{Es_i\}$ and Bs_i are called an *execution part* and a *replacement part*, respectively. An execution part is optional. When *Hs* contains more than one atom, r is called a *multi-head rule.* It is called a *single-head rule* otherwise. Variables used in rewriting rules and

[1] Detailed analysis of the role of the multi-head rule r_{rv_2} in successful termination of the reduction sequence in Table 1 can be found in [4]. It is also shown in [4] that no finite successful reduction sequence for the above query can be obtained by using single-head rules alone.

[2] Arbitrary built-in deterministic operations, e.g. arithmetic functions, can be used as executable atoms. In this paper, only the unification operation is used.

those used in definite clauses are of different types. The former variables are used for representing patterns of terms, and, for the sake of syntactically clear distinction, they always have an asterisk prefixed to their names.

Given a definite clause C containing atoms $b_1, \ldots, b_m$, where $m \geq 1$, in its body, the rule r is *applicable* to C at $b_1, \ldots, b_m$ iff Hs matches these atoms by a substitution θ (i.e., $Hs\theta$ is the sequence $b_1, \ldots, b_m$). To apply r to the clause C, the pattern-matching substitution θ is additionally required to instantiate all variables that occur in r but not in Hs into distinct usual variables that do not occur in C. The application then replaces C with the clauses that are obtained as follows: for each i ($1 \leq i \leq n$), if the evaluation of $Es_i\theta$ succeeds and yields a substitution σ, then a clause obtained from $C\sigma$ by replacing $b_1\sigma, \ldots, b_m\sigma$ with $Bs_i\theta\sigma$ is constructed. The reader is referred to [3,4] for more elaborate operational semantics and a larger class of rewriting rules.

3 Program Synthesis in the ET Model

3.1 Specifications, Programs, and Program Correctness

Specifications. A *specification* in the ET model is a pair $\langle D, Q \rangle$, where D is a set of definite clauses, representing background knowledge, and Q is a set of problems of interest. Each problem in Q is also a set of definite clauses. It is required that for each problem $prb \in Q$, the predicates occurring in the heads of clauses in prb occur neither in D nor in the bodies of clauses in prb. Given a set A of definite clauses, the *meaning* of A, denoted by $\mathcal{M}(A)$, is the set $\bigcup_{n=1}^{\infty} T_A^n(\emptyset)$, where T_A is the usual one-step consequence operator determined by A (see, e.g., [11]). The *answer set* of a problem prb with respect to a specification $\langle D, Q \rangle$ is defined as

$$\mathcal{M}(D \cup prb) - \mathcal{M}(D),$$

i.e., the set of all ground atoms in $\mathcal{M}(D \cup prb)$ whose predicates occur in the heads of clauses in prb.

Programs. A *program* in the ET model is a set of prioritized rewriting rules. A rewriting rule r is said to be *applicable* to a problem prb iff r is applicable to some definite clause in prb, and is said to *transform* prb into a problem prb' *in one step* iff prb' is obtained from prb by applying r to a clause in prb one time.[3] A program P is said to be *applicable* to a problem prb iff some rewriting rule in P is applicable to prb, and is said to *transform* prb into a problem prb' *in one step* iff there exists a rewriting rule $r \in P$ such that

1. r is applicable to prb and r transforms prb into prb' in one step,
2. for any rewriting rule $r' \in P$, if r' takes priority over r, then r' is not applicable to prb.

[3] Application of a rewriting rule to a definite clause is described in Subsection 2.3.

Computation. A *computation* of a program P on a problem prb is a finite or infinite sequence $com = [prb_0, prb_1, prb_2, \ldots]$ of problems such that $prb_0 = prb$ and the following two conditions are satisfied:

1. For any two successive problems prb_i and prb_{i+1} in com, P transforms prb_i into prb_{i+1} in one step.
2. If com is finite, then P is not applicable to $last(com)$ (i.e., the last problem in com).

If com is finite and $last(com)$ consists only of unit clauses, then the *answer set obtained* from com is the set

$$\{g \mid ((a \leftarrow) \in last(com)) \;\&\; (g \text{ is a ground instance of } a)\},$$

otherwise the answer set obtained from com is undefined.

Program Correctness. A program P is *correct* with respect to a specification $\langle D, Q\rangle$ iff for any problem $prb \in Q$ and computation com of P on prb, the answer set obtained from com is defined and is equal to the answer set of prb with respect to $\langle D, Q\rangle$.

3.2 Program Synthesis Problems and a Sufficient Condition for Program Correctness

Program Synthesis Problems. A *program synthesis problem* in the ET model is formulated as follows:

> Given a specification $\langle D, Q\rangle$, construct a program P such that P is correct with respect to $\langle D, Q\rangle$ and P is sufficiently efficient.

ET Rules and a Sufficient Condition for Program Correctness. A rewriting rule is an *ET rule* with respect to a set D of definite clauses iff for any problems prb and prb', if the rule transforms prb into prb', then

$$\mathcal{M}(D \cup prb) = \mathcal{M}(D \cup prb').$$

It is shown in [4] that a program P is correct with respect to a specification $\langle D, Q\rangle$ if the following conditions are all satisfied:[4]

(ETR) P consists only of ET rules with respect to D.

(APP) For any problem $prb \in Q$, if there exists a problem prb' such that
- P transforms prb into prb' (in a finite number of steps),
- prb' contains some non-unit clause,

then P is applicable to prb'.

(TER) For any problem $prb \in Q$, every computation of P on prb is finite.

This sufficient condition for program correctness provides a basis for program generation in the ET model.

[4] (ETR), (APP), and (TER) are abbreviations for "ET rules only", "applicability of a program", and "termination of computation", respectively.

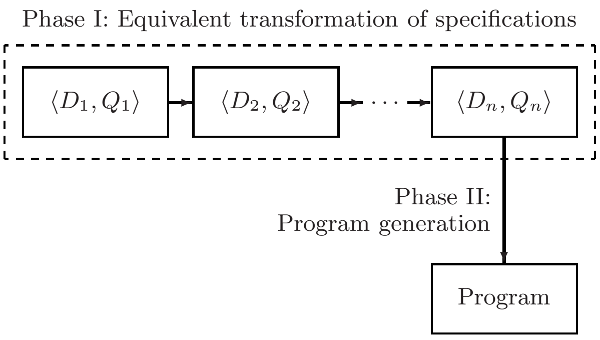

Phase I: Equivalent transformation of specifications
$\langle D_1, Q_1 \rangle$
$\langle D_2, Q_2 \rangle$
$\langle D_n, Q_n \rangle$
Phase II: Program generation
Program

Fig. 3. A two-phase program synthesis framework

3.3 A Two-Phase Program Synthesis Framework

As outlined in Fig. 3, program synthesis in the ET model consists of two main phases: (1) equivalent transformation of specifications, and (2) generation of a program from an obtained specification. In the first phase, an initially given specification is transformed into an equivalent specification that has a more suitable form for generation of efficient rewriting rules. Transformation methods and strategies from research works on program transformation [13,14,15], partial deduction [12], and conjunctive partial deduction [9], e.g. fold/unfold transformation, goal replacement, tupling, etc., can be employed in this phase. In many cases, only the background-knowledge part (i.e., D_i) of a specification is changed.[5]

The second phase is concerned with generation of a set of prioritized rewriting rules (a program) from a specification. A program generation method called the *squeeze method* is used. The squeeze method will be described in the next section.

Program synthesis in the logic programming model can be viewed as a special case of this two-phase framework. As explained in Subsection 2.2, computation in logic programming corresponds to ET-based computation using single-head general unfolding-based rules alone. Since a set of definite clauses (a background knowledge part) always determines a unique (up to variable renaming) set of single-head general unfolding-based rules, the second phase in this special case is very simple—program generation is restricted to only a fixed one-to-one correspondence between specifications and programs. With this restriction, improvement of programs can be achieved only by equivalent transformation of specifications in the first phase.[6] From an obtained specification, no search for more effective programs is made—the power of the second phase is not exploited.

[5] By applying data structure extension techniques, e.g. safe extension of specialization systems [2], the problems in Q_i can also be changed in the first phase.

[6] Since a set of definite clauses is regarded as a logic program, equivalent transformation of specifications corresponds to "program transformation" in the logic programming context.

```
repeat
1. run the current program under certain control of execution
2. if some obtained final clause is not a unit clause then
     begin
       2.1 select one or more atoms in the body of a non-unit final clause
       2.2 determine a general pattern of the selected atoms
       2.3 generate an ET rule for transforming atoms that conform
           to the obtained pattern
       2.4 assign a priority level to the obtained rule
       2.5 add the obtained rule to the current program
     end
until all obtained final clauses are unit clauses
```

Fig. 4. The squeeze method

4 Program Generation Using the Squeeze Method

4.1 The Squeeze Method

The squeeze method is shown in Fig. 4. It generates a program by accumulation of rules one by one on demand, with the goal of producing a correct, efficient, and non-redundant program.

Heuristic Parameters. Heuristics are used for suggesting a suitable rule to be added in each iteration. They are given via the following parameters:

[RUN]	Control of execution at Step 1.
[TAR]	Guidelines on selection of target atoms at Step 2.1.
[PAT]	Guidelines on determination of a general atom pattern at Step 2.2.

Rule Generation. The squeeze method is used both for the purpose of aiding human programmers and for that of automated program construction. An ET rule may be generated manually or automatically at Step 2.3. An algorithm for automatic generation of ET rules, based on meta-computation, has been developed in [10]—given a set D of definite clauses and an atom pattern as inputs, the algorithm generates an ET rule with respect to D for transforming atoms that conform to the input pattern.

Rule Prioritization. The number of rule bodies provides a basis for rule prioritization. In order to obtain an efficient program, rules with fewer bodies are preferable on the grounds that a problem reduction sequence is typically longer as the number of clauses in a computation state increases. Rules are prioritized accordingly at Step 2.4.

Minimized Redundancy. By the "demand-driven" characteristic of the method, redundant rules in a resulting program can be minimized—a new rule is generated only when a non-unit definite clause to which no existing rule is applicable is found, under the control of execution in use (specified by the parameter [RUN]).

Table 2. An example of program generation using the squeeze method

Iteration	Final problem	Atom(s) selected	Atom(s) pattern	Rule obtained	Priority assigned
1st	prb_0	$pal([1\|X])$	$pal(*x)$	r_{pal}	PR-1
2nd	prb_2	$rv([1\|X],[1\|X])$	$rv([*a\|*x],*y)$	r_{rv_1}	PR-1
3nd	prb_4	$rv(X,A1),rv(X,A2)$	$rv(*x,*y),rv(*x,*z)$	r_{rv_2}	PR-1
4th	prb_5	$ap(A1,[1],[1\|X])$	$ap(*x,*y,[*a\|*z])$	r_{ap_1}	PR-2
5th	prb_6	$rv(X,[1\|A3])$	$rv(*x,[*a\|*y])$	r_{rv_3}	PR-1
6th	prb_7	$ap([1\|A3],[2],[2,A4\|A5])$	$ap(*x,[*a],[*b,*c\|*y])$	r_{ap_2}	PR-1
7th	prb_8	$ap([\,],[2],[2])$	$ap(*x,[*a],[*b])$	r_{ap_3}	PR-1
8th	prb_9	$rv([\,],[\,])$	$rv([\,],*x)$	r_{rv_4}	PR-1
9th	prb_{10}	–	–	–	–

Based on the squeeze method, an experimental automatic program synthesis system has been implemented and used for constructing many nontrivial programs (including the program in Subsection 2.1), and computer-aided programming tools for supporting human programmers have also been developed.

4.2 Example

Assuming the background knowledge in Fig. 1, generation of the program considered in Subsection 2.1 will now be illustrated. The following parameters are used:

[RUN] Usual rule selection based on rule priority is employed under one constraint: employment of low-priority rules should be minimized.

[TAR] One or more atoms can be selected, using the following guidelines:
- Select an atom that has a specific structure; e.g. $ap([1|X],Y,Z)$ is preferable to $ap(X,Y,Z)$ since $[1|X]$ is more specific than X.
- Select atoms that have common variables; e.g. $ap(X,Y,Z)$ and $ap(X,V,W)$ with X as a common variable.
- A smaller number of selected atoms is preferable.

[PAT] A more general pattern is preferable as long as it does not lead to a rule with a larger number of bodies.

As shown in Table 2, with these parameters the squeeze method produces the rules in Fig. 2 within nine iterations. The construction process will now be described in more detail (with reference to prb_0–prb_{10} in Table 1).

The 1st iteration: The initial program contains no rule; running the program makes no change to the problem prb_0. Either $pal([1|X])$ or $pal([2|X])$ may be selected, and the atom pattern $pal(*x)$ is determined. An ET rule for this pattern is generated, and r_{pal} is obtained. Since r_{pal} has a single body, assign a high priority level, say PR-1, to it.

The 2nd iteration: The current program contains only r_{pal}. Following the first two problem reduction steps in Table 1, running this program yields prb_2 as the final problem. The two body atoms in prb_2 have the same pattern, and one of them is selected as the target atom. The atom pattern $rv([*a|*x], *y)$ is determined. An ET rule for this pattern is generated, and r_{rv_1} is obtained. Since r_{rv_1} also has a single body, the priority level PR-1 is assigned to it.

The 3rd iteration: Following the first four reduction steps in Table 1, running the current program results in the problem prb_4. The two rv-atoms in this problem are selected as the target atoms, and the pair of $rv(*x, *y)$ and $rv(*x, *z)$ is set as the target pattern.[7] The multi-head ET rule r_{rv_2} is devised based on the functionality of the "reverse" relation. Again, the priority level PR-1 is assigned to it.

The 4th iteration: Following the first five reduction steps in Table 1, the current program now yields prb_5 as the final problem. The first ap-atom in this problem is selected as the target atom, and $ap(*x, *y, [*a|*z])$ is set as the target atom pattern. An ET rule for this pattern is generated, and r_{ap_1} is obtained. Since it has more than one body, a lower priority level, say PR-2, is assigned to r_{ap_1}.

The 5th iteration: By the first six reduction steps of Table 1, the current program transforms prb_0 into prb_6. By the constraint imposed by the parameter [RUN], although prb_6 can be transformed further using r_{ap_1}, this transformation step is not made. Instead, a new rule is constructed. The first rv-atom in the second clause of prb_6 is selected as the target atom, and the atom pattern $rv(*x, [*a| * y])$ is determined. The ET rule r_{rv_3} is then generated, and the priority level PR-1 is assigned to it.

The 6th iteration onwards: Following the squeeze method three more iterations, the ET rules r_{ap_2}, r_{ap_3}, and r_{rv_4} are generated and added to the program in succession. The priority level PR-1 is given to each of them. When running the resulting program with the input problem prb_0, a problem consisting only of unit clauses is obtained, and the squeeze method ends.

By adjustment of the parameters of the method, a more efficient program can be generated. For example, if the parameter [RUN] is changed into "only single-body rules are used", then the two-body rule r_{ap_1} constructed in the 4th iteration above will not be applied and a search for alternatively suitable target atoms will be made. In this case, the parameter [TAR] recommends alternative selection of the two atoms $ap(A1, [1], [1|X])$ and $ap(A1, [2], [2|X])$ in prb_5 since they have $A1$ and X as common variables. As shown in the full version of this paper [5], this selection results in generation of a single-body rule (multi-head) for ap-atoms, and the obtained program produces a shorter problem reduction sequence (3 reduction steps less, compared with the sequence in Table 1).

[7] According to [TAR], any of the two ap-atoms in prb_4 is an alternative choice. However, selection of such an ap-atom would yield a two-body rule (see the 4th iteration). By selecting the two rv-atoms, a single-body rule, which is preferable, is obtained.

4.3 On the Correctness of the Squeeze Method

The correctness of the squeeze method can be analyzed based on the sufficient condition for program correctness given in Subsection 3.2 (i.e., (ETR), (APP), and (TER)). Using the squeeze method, a program is generated from a specification $\langle D, Q \rangle$ by repeatedly

1. selecting a problem from Q, and
2. accumulatively generating rewriting rules for solving the selected problem

until sufficient representative samples of the problems in Q have been selected. As long as only ET rules with respect to D are generated (at Step 2.3 of Fig. 4), (ETR) is satisfied. While (ETR) can be checked by examination of each rewriting rule individually, (APP) and (TER) are global properties—they involve analysis of the interactions between obtained rules. Nonetheless, by its demand-driven behavior, the squeeze method naturally directs a program generation process towards satisfaction of (APP)—a new rule is generated whenever an intermediate problem to which a program is not applicable is found. By assuming some well-founded ordering on atoms, (TER) can also be well controlled; e.g. only rules that transform definite clauses into simpler ones with respect to the well-founded ordering should be generated. Although it is difficult in general to assure strict correctness with respect to (APP) and (TER) by stepwise rule accumulation alone, the squeeze method works well in many cases and it provides a good structure for developing additional techniques and strategies for controlling (APP) and (TER), e.g. rule priority adjustment techniques and atom ordering strategies.

4.4 How the ET Model Supports the Squeeze Method

From an abstract viewpoint, the squeeze method can be seen as a componentwise program generation method that is applicable when its underlying computation model satisfies the following requirements:

1. Correctness and efficiency of program components can be discussed.
2. The correctness (respectively, efficiency) of a component can be verified (respectively, evaluated) independently of other components.
3. A correct (respectively, efficient) program can be constructed by accumulation of individually correct (respectively, individually efficient) components.
4. A partial program suggests appropriate components to be added towards completing the program.

In the ET model, rewriting rules are program components, and each ET rule is regarded as a correct component. The quality of "being an ET rule" of one rewriting rule does not depend on any other rewriting rule, and can be checked individually. Since a set of unit clauses obtained from a sequence of problem reduction steps using ET rules always yields a correct answer set, a correct program can be constructed by accumulation of ET rules with some control strategies for (APP) and (TER). An incomplete program in this model can always be executed; and when the execution terminates with a problem containing non-unit definite clauses, the body atoms of the obtained clauses always provide a clue to creation of new ET rules (e.g. the parameter [TAR] provides heuristics for selection of appropriate body atoms). In regard to the efficiency aspect, a rule with fewer bodies is considered as a more efficient component inasmuch as it narrows down a search space. Obviously, the number of rule bodies is an individual property of a rule. All the above requirements are thus satisfied.

In contrast, consider the logic programming model, where a set of definite clauses is regarded as a program. Correctness of a definite clause can be defined in such a way that it can be checked independently of other definite clauses. However, the efficiency of a definite clause cannot be evaluated individually—when a node in a partial SLD-tree is expanded using one predicate definition, each clause in that definition possibly yields one child node, and, hence, the number of all resulting child nodes is not known unless the predicate definition is complete. Moreover, when a partial logic program fails to prove a given true ground query, it is difficult to find an appropriate definite clause to be added—a partial SLD-tree can be very large and there are various possible choices of nodes and definite clauses from which a new branch should be created in the tree.

5 Concluding Remarks

A clear-cut separation between specifications and programs in the ET model along with the generality of the ET principle for computation correctness opens up the possibility of employing a very large class of rewriting rules—any rule whose application always results in meaning-preserving transformation with respect to given background knowledge can serve as an ET rule. As a consequence, various classes of rewriting rules, with varying expressive power, can be introduced. In the ETI system[8] developed at Hokkaido University, for example, rules with execution parts, rules with applicability conditions (possibly involving extra-logical predicates), and multi-head rules are provided.

Program synthesis in the ET model consists of two main phases—(1) equivalent transformation of specifications, and (2) generation of a set of prioritized rewriting rules from a specification. The second phase makes program synthesis in this model significantly different from that in declarative computation paradigms, in which specifications are regarded as programs and program synthesis is based solely on equivalent transformation of specifications. In conjunction with a large variety of possible rule classes, the second phase enhances the possibility of program improvement and optimization. Program synthesis in logic programming, for example, corresponds only to the first phase of the framework. Through the second phase, some problems that cannot be solved by logic programs can be dealt with in the ET model.

All methods and techniques of fold/unfold transformation, partial deduction, and conjunctive partial deduction, e.g. [9,12,13,14,15], are applicable to the first

[8] ETI is an interpreter system that supports ET-based problem solving. It is available at http://assam.cims.hokudai.ac.jp/etpro.

phase. An incremental program construction method for the second phase, called the squeeze method, is described in this paper. How the structure of the ET model supports componentwise program generation is discussed.

References

1. Akama, K., Shigeta, Y., Miyamoto, E.: Solving Logical Problems by Equivalent Transformation—A Theoretical Foundation. Journal of the Japanese Society for Artificial Intelligence **13** (1998) 928–935
2. Akama, K., Koike, H., Mabuchi, H.: Equivalent Transformation by Safe Extension of Data Structures. In: Bjørner, D., Broy, M., Zamulin, A. (eds.): Perspectives of System Informatics (PSI'01). Lecture Notes in Computer Science, Vol. 2244. Springer-Verlag (2001) 140–148
3. Akama, K., Nantajeewarawat, E., Koike, H.: A Class of Rewriting Rules and Reverse Transformation for Rule-Based Equivalent Transformation. Electronic Notes in Theoretical Computer Science **59**(4) (2001)
4. Akama, K., Nantajeewarawat, E.: Formalization of the Equivalent Transformation Computation Models. Journal of Advanced Computational Intelligence and Intelligent Informatics **10** (2006) 245–259
5. Akama, K., Nantajeewarawat, E., Koike, H.: Program Generation in the Equivalent Transformation Computation Model using the Squeeze Method. Technical Report, Information Initiative Center, Hokkaido University (January 2006)
6. Akama, K., Nantajeewarawat, E., Koike, H.: Componentwise Program Construction: Requirements and Solutions. WSEAS Transactions on Information Science and Applications **3**(7) (2006) 1214–1221
7. Anutariya, C. et al.: RDF Declarative Description (RDD): A Language for Metadata. Journal of Digital Information **2**(2) (2001)
8. Bird, R.: Introduction to Functional Programming. 2nd edn. Prentice Hall (1998)
9. De Schreye, D. et al.: Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments. Journal of Logic Programming **41** (1999) 231–277
10. Koike, H., Akama, K., Boyd, E.: Program Synthesis by Generating Equivalent Transformation Rules. In: Proc. 2nd International Conference on Intelligent Technologies, Bangkok, Thailand (2001) 250-259
11. Lloyd, J. W.: Foundations of Logic Programming. 2nd edn. Springer-Verlag (1987)
12. Lloyd, J. W., Shepherdson, J. C.: Partial Evaluation in Logic Programming. Journal of Logic Programming **11** (1991) 217–242
13. Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. Journal of Logic Programming **19**&**20** (1994) 261–320
14. Pettorossi, A., Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs. ACM Computing Surveys **28**(2) (1996) 360–414
15. Pettorossi, A., Proietti, M.: Synthesis and Transformation of Logic Programs using Unfold/fold Proofs. Journal of Logic Programming **41** (1999) 197–230
16. Wuwongse, V. et al.: A Data Model for XML Databases. Journal of Intelligent Information Systems **20** (2003) 63–80

A Versioning and Evolution Framework for RDF Knowledge Bases

Sören Auer and Heinrich Herre

University of Leipzig, 04109 Leipzig, Germany
auer@informatik.uni-leipzig.de
http://www.informatik.uni-leipzig.de/~auer/

Abstract. We present an approach to support the evolution of online, distributed, reusable, and extendable ontologies based on the RDF data model. The approach works on the basis of atomic changes, basically additions or deletions of statements to or from an RDF graph. Such atomic changes are aggregated to compound changes, resulting in a hierarchy of changes, thus facilitating the human reviewing process on various levels of detail. These derived compound changes may be annotated with meta-information and classified as ontology evolution patterns. The introduced ontology evolution patterns in conjunction with appropriate data migration algorithms enable the automatic migration of instance data in distributed environments.

1 Introduction

The goal of the envisaged next generation of the Web (called Semantic Web [2]) is to smoothly interconnect personal information management, enterprise application integration, and the global sharing of commercial, scientific, and cultural data[1]. In this vision, ontologies play an important role in defining and relating concepts that are used to describe data on the web [4]. In a distributed, dynamic environment such as the Semantic Web, it is further crucial to keep track of changes in its documents to ensure the consistency of data, to document their evolution, and to enable concurrent changes. In areas such as software engineering, databases, and web publishing versioning and revision control mechanisms have already been developed and successfully applied. In *software engineering* versioning is used to track and provide controls over changes to a project's source code. In *database systems* versioning is usually provided by a database log, which is a history of actions executed by a database management system. For *web publishing* the Web-based Distributed Authoring and Versioning (WebDAV) standard was released as an extension to the Hyper Text Transfer Protocol (HTTP) supporting versioning and with the intention of making the World Wide Web a readable and writable medium.

For revision control of semantic-web data, unfortunately these developed technologies are insufficient. In software engineering and web publishing revision control is based on unique serializations, enabled by their data models. Such unique

[1] http://www.w3.org/2001/sw/Activity

serializations are not available for Semantic Web knowledge bases, usually consisting of unordered collections of statements. Database logs on the other hand cope with a multitude of different interrelated objects of their data model (e.g. databases, tables, rows, columns/cells) in contrast to just statements of the RDF data model.

In this paper, we present an approach for the versioning of distributed knowledge bases grounded on the RDF data model with support for ontology evolution. Under ontology versioning we understand to keep track of different versions of an ontology and possibly to allow branching and merging operations. Ontology evolution additionally shall identify and formally represent the conceptual changes leading to different versions and branches. On the basis of this information, ontology evolution should support the migration of data adhering to a certain ontology version.

This paper is structured as follows: Our approach works on the basis of atomic changes which are determined by additions or deletions of certain groups of statements to or from an RDF knowledge base (Section 2). Such atomic changes are aggregated to more complex changes, resulting in a hierarchy of changes, thus facilitating the human reviewing process on various levels of detail (Section 3). The derived compound changes may be annotated with meta-information such as the user executing the change or the time when the change occurred. We present a simple OWL ontology capturing such information, thus enabling the distribution of change sets (Section 5). Assuming that there will be no control of evolution, it must be clarified which changes are compatible with a concurrent branch of the same root ontology. We present a compatibility concept for applying a change to an ontology on the level of statements (Section 4). To enable the evolution of ontologies with regard to higher conceptual levels than the one of statements we introduce evolution patterns (Section 6) and give examples for appropriate data migration algorithms (Section 7). We further give account of the successful implementation of the approach in Powl, summarize related work and give an outlook on planned directions for future work (Section 8).

2 Atomic Changes on RDF Graphs

To introduce our notion of atomic changes on RDF graphs we need recall some preliminary definitions from [5]. Some of the main building blocks of the semantic-web paradigm are *Universal Resource Identifier* (URI) and their RDF counterparts URI References, whose quite technical definitions we omit here.

Definition 1 (Literal). *A* Literal *is a string combined with either a language identifier (plain literal) or a datatype (typed literal).*

Definition 2 (Blank Node). *Blank Nodes are identifiers local to a graph. The set of* Blank Node*s, the set of all URI references, and the set of all literals are pairwise disjoint. Otherwise, the set of blank nodes is arbitrary.*

Definition 3 (Statement). *A* Statement *is a triple* (S, P, O)*, where*

- *S is either a URI reference or a blank node (Subject).*
- *P is a URI reference (Predicate).*
- *O is either a URI reference or a literal or a blank node (Object).*

Definition 4 (Graph). *A* Graph *is a set of statements.*

The set of nodes of an graph is the set of subjects and objects of triples in the graph. Consequently the blank nodes of a graph are the members of the subset of the set of nodes of the graph which consists only of blank nodes.

Definition 5 (Graph Equivalence). *Two RDF graphs G and G' are equivalent if there is a bijection M between the sets of nodes of the two graphs, such that:*

1. *M maps blank nodes to blank nodes.*
2. *$M(lit) = lit$ for all literals lit which are nodes of G.*
3. *$M(uri) = uri$ for all URI references uri which are nodes of G.*
4. *The triple (s, p, o) is in G if and only if the triple $(M(s), p, M(o))$ is in G'.*

Based on these definitions we want to discuss the possible changes on a graph. RDF statements are in [7] identified to be the smallest manageable piece of knowledge. This view is justified by the fact that there is no way to add, remove, or update a resource or literal without changing at least one statement, whereas the opposite does not hold. We adopt this view but require the smallest manageable pieces of knowledge to be somehow closed regarding the usage of blank nodes. Moreover we want to be able to construct larger changes out of smaller ones, and since the order of additions and deletions of statements to a graph may matter, we distinguish between Positive and Negative Atomic Changes.

Definition 6 (Atomic Graph). *A graph is atomic if it may not be split into two nonempty graphs whose blank nodes are disjoint.*

Obviously, a graph without any blank node is atomic if it consists of exactly one statement. Hence, any statement which does not contain a blank node as subject or object is an atomic graph.

Definition 7 (Positive Atomic Change). *An atomic graph C_G is said to be an* Positive Atomic Change *on a graph G if the sets of blank nodes occurring in statements of G and C_G are disjoint.*

The rationale behind this definition is the aim of applying the positive atomic change C_G to the graph G. Hence, a positive atomic change on a graph G can be applied to G to yield a new graph as a result. For this purpose we introduce a (partial) function $Apl^+(X, Y)$ whose arguments are graphs.

Definition 8 (Application of a Positive Atomic Change). *Let C_G be a positive atomic change on the graph G. Then the function Apl^+ is defined for the arguments G, C_G and it holds $Apl^+(G, C_G) = G \cup C_G = G'$ which is symbolized by $G \overset{C_G}{\rightarrow} G'$. We say that C_G is applied to the graph G with result G'.*

Application of the positive atomic change C_G to G yielding G' is just identifying the union of C_G and G with G'. Of course a graph may not only be changed by adding statements leading to the notion of a negative atomic change.

Definition 9 (Negative Atomic Change). *A subgraph C_G of G is said to be a* Negative Atomic Change *on G if C_G is atomic and contains all statements of G whose blank nodes occur in the statements of C_G.*

Analogously to the case of positive changes we introduce a function $Apl^-(G, C_G)$ which pertains to negative atomic changes.

Definition 10 (Application of a Negative Atomic Change). *Let C_G be a negative atomic change on the graph G. Then the function Apl^- is defined for the arguments G, C_G and is determined by $Apl^-(G, C_G) = G \backslash C_G = G'$ which is symbolized by $G \overset{C_G}{\rightarrow} G'$. We say that C_G is applied to G with result G'.*

These definitions require changes involving blank nodes to be somehow independent from the graph in the sense that blank nodes in the change and in the (remaining) graph do not overlap. This is crucial for changes being exchangeable between different RDF storage systems, since the concrete identifiers of the blank nodes may differ. It may have the negative effect though that large subgraphs, which are only interconnected by blank nodes, have to be deleted completely and added - slightly modified - afterwards.

3 Change Hierarchies

The evolution of a knowledge base typically results in a multitude of sequentially applied atomic changes. These are usually small, and may often contain only a single statement. On the other hand, in many cases multiple atomic changes form one larger 'logical' change. Consider for example the case where the arrival of the information of 'being of German nationality' for a person, results not only in adding this fact to the knowledge base, but also in using the right spelling for the persons name using umlauts. As shown in Example 1 this could result in three atomic changes. The information that those three changes somehow belong together should not be lost, as we would like to enable human users to observe the evolution of a knowledge base on various levels of detail. This could be achieved by constructing hierarchies of changes on a graph.

To achieve this goal first of all Atomic Changes are called *Changes of Level 0* and then changes of higher levels are defined inductively. Let At be the set of atomic changes. General changes, which are simply called changes, are defined as sequences over the set At. The set $Changes(At)$ of changes over At is the

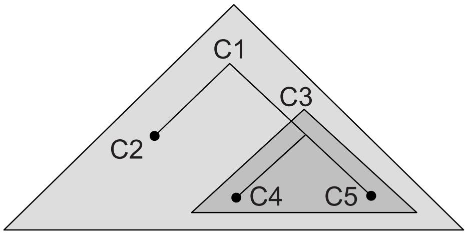


Fig. 1. Schematic visualisation of a change hierarchy. Black dots represent atomic changes and gray triangles compound changes.

smallest set containing the empty sequence () and closed with respect to the following condition: if $\{C_1, \ldots C_k\} \subseteq \mathit{Changes}(\mathit{At}) \cup \mathit{At}$, then $(C_1, \ldots, C_k) \in \mathit{Changes}(\mathit{At})$. An annotated change is an expression of the form C^A where $C \in \mathit{Change}(\mathit{At})$, and A is an annotation object. No restriction is imposed on the annotation object A which is attached to a change. In Section 5 we present a simple ontology schema, which may be used for capturing such change annotations.

The changes of level at least n, denoted by $\mathit{Ch}(n)$, are defined inductively. Every change has a level at least 0, i.e $\mathit{Ch}(0) = \mathit{Changes}(\mathit{At})$. If $C_1, \ldots, C_k$ are changes in $\mathit{Ch}(n)$, then $(C_1, \ldots, C_k) \in \mathit{Ch}(n+1)$. A change C is of level (exactly) n if $C \in \mathit{Ch}(n) \backslash \mathit{Ch}(n+1)$, i.e. C has level at least n but not level at least $n+1$. The application functions $\mathit{App}^+, \mathit{App}^-$ may be extended to a function $\mathit{App}(G, C)$ whose first argument is a graph, and second argument is a change. App is recursively defined on the level of the second argument C. Now we would like to apply a change C of level > 0 to a graph. Since C is a sequence of changes of smaller level, these changes – being components of C – can be consecutively applied to intermediate graphs. This is demonstrated in the following for the change from Example 1.

$C1$ is applied to some graph G containing information about people and results in a new revision of G, namely G':

$$G \overset{C1}{\rightarrow} G'$$

Since $C1$ consists of $C2$ and $C3$, $C1$ it may be resolved into:

$$G \overset{C2}{\rightarrow} G^{(1)} \overset{C3}{\rightarrow} G'$$

And finally since $C3 = (C4, C5)$:

$$G \overset{C2}{\rightarrow} G^{(1)} \overset{C4}{\rightarrow} G^{(2)} \overset{C5}{\rightarrow} G'$$

$C2$, $C4$, and $C5$ are atomic changes and may be applied as proposed in Definitions 8 and 10.

Example 1 (Change Hierarchy). *Consider the following update of the description of a person:*

```
Resource changed                                  (C1)
  Resource classified                             (C2)
    http://auer.cx/Soeren  hasNationality  German
  Labels changed                                  (C3)
    Label removed                                 (C4)
      http://auer.cx/Soeren  rdfs:label  "Soeren Auer"
    Label added                                   (C5)
      http://auer.cx/Soeren  rdfs:label  "Sören Auer"
```

$C1$ represents a compound change with $C1 = (C2, C3)$ and $C3 = (C4, C5)$; $C2$, $C4$, and $C5$ here are atomic changes. It may be visualized as in Fig. 1.

We call a change of a level $n > 1$ a *Compound Change*. As visualized in Fig. 1 it may be viewed as a tree of changes with atomic changes on its leafs. This enables the review of changes on various levels of detail (e.g. statement level, ontology level, domain level) and thus facilitates the human reviewing process.

A further advantage in addition to improved change examination is, that on their basis a knowledge transaction processing may be implemented. Assuming that a Relational Database Management System supporting transactions is used as a triple store for knowledge bases, every compound change may then be encapsulated within a database transaction. Meanwhile the repository will be blocked for other write accesses. Compound Changes thus should not be nested arbitrarily deep but up to some compound change, which was for example triggered by a user interaction. We call such a top-level compound change *Upper Compound Change*. Multiple, possibly semantically related compound changes can be collected in a *Patch* for easy distribution, for example in a Peer-to-Peer environment.

4 Change Conflict Detection

Tracking additions and deletions of statements as described in the last section enables the implementation of linear undo / redo functionality. In distributed or web-based environments usually several people such as knowledge engineers and domain experts contribute changes to a knowledge base. In such a setting it is highly demandable to rollback only certain earlier changes. Of course, this will not be possible for arbitrary changes.

Consider the case when some statements were added to a graph in the change C_1 and removed later in the change C_2. The rollback of the change C_1 should not be possible any longer after C_2 took place. In the opposite case when statements are removed from the knowledge base first and added again later, the rollback of the deletion should not be possible either. The following definitions clarify which atomic changes are compatible with a distinct knowledge base in this sense.

Definition 11 (Compatibility of a Positive Atomic Change with a Graph). *A Positive Atomic Change C_G is compatible with a graph G', iff C_G is not equivalent to some subgraph of G'.*

Definition 12 (Compatibility of a Negative Atomic Change with a Graph). *A Negative Atomic Change C_G is compatible with a graph G', iff C_G is equivalent to some subgraph of G'.*

If a positive (negative) atomic change C_G is compatible with some graph G' then it may be easily applied to G' by simply adding (respectively removing) the statements of C_G to G'. Possibly blank node identifiers have to be renamed in C_G if the same occurs in G'.

The notion of compatibility may be easily generalized to compound changes. Since the changes belonging to a compound change are ordered, every compound change may be broken up into a corresponding sequence of atomic changes $(C_1, \ldots, C_n)$. If we consider the compound change from Example 1, the corresponding sequence of atomic changes will be $(C2, C4, C5)$.

Definition 13 (Compatibility of a Compound Change with a Graph). *A compound change $C_{G'}$ is compatible with a graph G, iff*

– *the first atomic change in the corresponding sequence of atomic changes $(C_1, ..., C_n)$ is compatible with G and results in G^1*
– *every following atomic change C_i $(1 < i \leq n)$ from the sequence is compatible with the intermediate graph G^{i-1} and its application results in G^i.*

The compatibility is especially interesting if G' is some prior version of G, since it supports the decision if the change may be rolled back. However, this compatibility concept only deals with possible conflicts on the level of statements. In the remaining part of this section we point out directions how we can cope with incompatibilities on higher conceptual levels than the one of statements.

In [6] the impact of distinct change patterns on instance data is studied. Change patterns include all elementary operations on an OWL ontology such as adding, deleting of classes, properties or instances. The effects on instances are categorized into change patterns which result in information preserving, translatable or information-loss changes. If a compound change contains an atomic change matching a change pattern of one of the latter two categories, this can be indicated to the user and possible solutions could be offered (cf. Section 6 for details on ontology evolution patterns). If the graph represents some *Web Ontology Language* (OWL) knowledge base, furthermore a description logic reasoner may be used to check whether a model is consistent after a change is applied or not. Ideally an evolution enabled knowledge base editor provides an interface to dynamically plug-in functionality to check the applicability of a distinct change with respect to a certain graph.

5 Represention of Changes

To distribute changes on a graph (e.g. in a client server or peer-to-peer setting), a consistent representation of changes is needed. We propose to represent changes as instances of a class `log:Change`. Statements to be added or deleted by atomic changes are represented as reified statements and referenced by the properties `log:added` and `log:removed` from a change instance. The property `log:parentChange` relates a change instance to a compound change instance of higher level.

To achieve our goal of enhanced human change review, it should be possible to annotate changes with information, such as about the user making the change, the date and time on which the change took place, a human-readable documentation about why the change was made, and which effects it may have, just to mention a few. Table 1 summarizes important properties attached to `log:Change`. The complete OWL ontology schema for capturing the change information is provided at `http:/powl.sf.net/logOnt`.

Table 1. Properties for representing and annotating changes

Property	**Description**	**Example**
Action	A string or URI identifying predefined action classes.	”Resource changed”
User	A string or URI identifying the editing user.	`http://auer.cx/Soeren`
DateTime	The timestamp in xsd:DateTime format when the change took place.	”20050320T16:32:11”
Documentation	A string containing a human readable description of the change.	Nationality added and name typing corrected correspondingly.
ParentChange	Optional URI identifying a compound change this change belongs to.	

6 Evolution Patterns

The versioning and change tracking strategy presented so far is applicable to arbitrary RDF graphs but also enables the representation and annotation of changes on higher conceptual levels than the one of pure RDF statements. In this section we demonstrate how it may be used and extended to support consistent OWL ontology and instance data evolution.

OWL ontologies consist of classes arranged in a class hierarchy, properties attached to those classes, and instances of the classes filled with values for the properties. Now we classify changes operating on OWL ontologies according to specific patterns reflecting common change intentions. The positive atomic change `(hasAddress,rdf:type,owl:ObjectProperty)` for example can be classified to be an *object property addition*, since the predicate of the statement in the change is `rdf:type` and the object is `owl:ObjectProperty)`. Complementary there is a category of *object property deletions* for negative atomic changes with that predicate and object. Such categories of changes can be described more formally and generally by our notion of Evolution Patterns.

Definition 14 (Evolution Pattern). *A positive (negative) evolution pattern is a triple* $(X, G(X), A(X))$*, where* X *is a set of variables,* $G(X)$ *is a graph pattern characterizing a positive (resp. negative) change with the variables* X *and* $A(X)$ *being an appropriate data migration algorithm.*

Graph patterns are essentially graphs where certain URI references are replaced by placeholders (i.e. variables). The precise definition is omitted here but can be found in [8]. As an example we consider the following positive atomic change of adding a cardinality restriction to the property `nationality` attached to the class `Person`:

```
Person  owl:subClassOf      :_1
:_1     rdf:type            owl:Restriction
:_1     owl:onProperty      nationality
:_1     owl:maxCardinality  2
```

The corresponding evolution pattern will be $AddMaxCardinality = (X, G(X), A(X))$ with $X = (class, property, maxCardinality)$, the graph pattern $G(X)$ will be:

```
?class        owl:subClassOf      ?restriction
?restriction  rdf:type            owl:Restriction
?restriction  owl:onProperty      ?property
?restriction  owl:maxCardinality  ?maxCardinality
```

Finally, the data migration algorithm $A(class, property, maxCardinality)$ will iterate through all instances of *class* and remove property values of *property* exceeding $maxCardinality$.

Beside facilitating the review of changes on a knowledge base the classification of changes into such evolution patterns enables the automatic migration of instance data, even in settings where instance data is distributed. General evolution patterns can be constructed out of sequences of positive and negative evolution patterns. The modification of a `owl:maxCardinality` restriction can thus be made up by sequentially applying changes belonging to the negative evolution pattern $DelMaxCardinality$ and the positive evolution pattern $AddMaxCardinality$.

In [4] a taxonomy of change patterns for OWL ontologies and their possible effects on instance data is given. However, from our point of view these change patterns will not be sufficient to capture change intentions and to enable automatic instance data migration. Intentions of changes can be made explicit by precisely describing effects on instance data, e.g. by providing instance data migration algorithms. We illustrate possible intentions for class deletions and re-classifications in the next two subsections.

Class Deletions. The deletion of some entity from an ontology corresponds to the deletion of all statements from the graph where an URI referencing the entity occurs as subject, predicate, or object. The deletion of a distinct class thus will result in the following serious effects:

- former instances of the class are less specifically typed,
- former direct subclasses become independent top level classes,
- properties having the class as domain become universally applicable,
- properties having the class as range will lose this restriction,

In most cases some or all of these effects are not desired to be that rigorous, but have to be mitigated. Before actually deleting the class, we then have to cope with the following aspects of the classes usage.

- *What happens with instances of the class?* If instances of a class C should be preserved they may be reclassified to be instances of a superclass of C (labeled I_R). If C has no explicit direct superclass the instances may be classified to be instances of the implicit superclass `owl:Thing`. Otherwise all its instances may be deleted (I_D).
- *How to deal with subclasses?* Subclasses may be deleted (S_D), reassigned in the class hierarchy (S_R) or kept as independent top level classes (S_K).
- *How to adjust properties having the class as domain (or range)?* The domain (or range) of properties having the class as domain (or range) may be extended (i.e. changed to a superclass - P_E) or restricted (i.e. changed to a subclass - P_R). A further possibility is to delete those properties (P_D).

Some combinations of those evolution strategies obviously do not make sense (i.e. (I_D, S_D, P_R) - deleting all instances and subclasses and restricting the domain and range of directly attached properties) while others are heavily needed (see also Fig. 2):

- (I_R, S_R, P_E) - merge class with superclass
- (I_D, S_D, P_E) - cut class off
- (I_D, S_D, P_D) - delete complete subtree including instances and directly attached properties

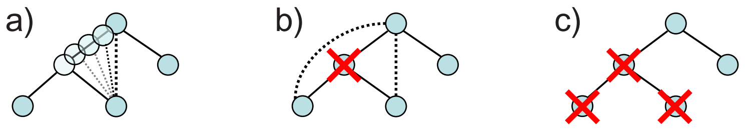


Fig. 2. Different intentions for deleting a class: a) merge with superclass, b) cut class off, c) delete subtree

As those different class deletions illustrate, different intentions to delete a class result in different combinations of data migration strategies and finally in different evolution patterns. Some other example for a complex ontology evolution pattern is the reclassification of a complete sub-class tree.

Reclassification. Often the distinction between abstract categories and concrete entities is not easy, resulting in different modeling possibilities, when it is required to stay within OWL DL: representation as classes or instances. In a later modeling or usage stage the selected representation strategy (classes or instances) may turn out to be suboptimal and reclassification is required.

If all classes in a whole class tree below a class C have no instances and directly attached properties, then they may be converted into instances. This can be done by defining a functional property P, which is used to preserve the hierarchical structure formerly encoded in the subclass-superclass relationship. Then for all classes C_i in the the subtree:

- add `(Ci,rdf:type,C)`,
- if C_i is a direct subclass of C, then delete the statement `(Ci,rdfs:subClassOf,C)`, else delete all statements `(Ci,rdfs:subClassOf,Cj)` and correspondingly add `(Ci,P,Cj)`.

Conversely, assuming we have a class C and a functional property P with C as domain and range, which does not reference instances in cycles. Then the instances of C then may be converted into subclasses of C as follows:

- every statement (I_1,P,I_2) is converted into (I_1,rdfs:subClassOf,I_2),
- if there is for I_1 no triple (I_1,P,I_2) add (I_1,rdf:type,C).

Beside class deletions and reclassification there are other ontology evolution patterns such as:

- *Move a property* A property P may be moved from a class C_1 to a referenced class C_2 (labeled `log:PropertyMove`).
- *"Widden" a restriction* For a property P we may increase the number of allowed values or decrease the number of required values.
- *"Narrow" a restriction* For a property P we may decrease the number of allowed values or increase the number of required values.
- *Split a class* A class C may be split into two new classes C_1 and C_2 related to each other by some property P (labeled `log:ClassSplit`).
- *Join two classes* Two classes C_1 and C_2 referencing each using a functional property may be joined.

These examples show that the basic change patterns from [4] are not sufficient to capture the intentions for ontology changes. To support independently, but synchronously evolving schema and instance data, as visualized at the example of splitting a class in Fig. 3, we propose to annotate compound (schema) changes with their respective evolution patterns. Corresponding data migration algorithms then can be used to migrate instance data agreeing to a former version of the ontology. However, it is future work to provide a complete library of ontology evolution patterns.

The annotation of compound changes with ontology evolution patterns can be easily achieved within the framework showcased in Section 5. The move of a property $P1$ from a class $C1$ to a class $C2$ referencing each other by a property $P2$ could be represented for example as follows:

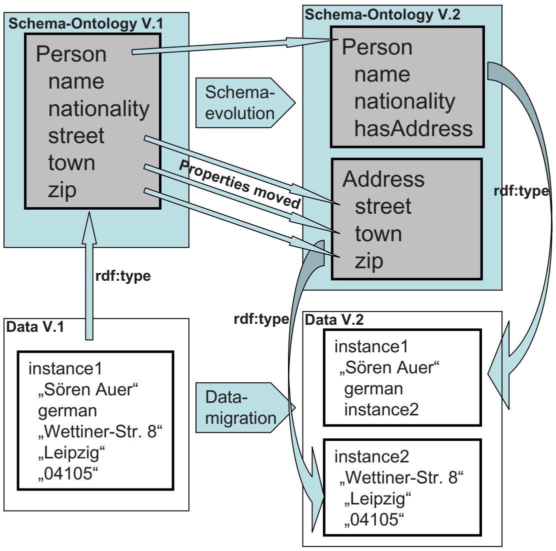


Fig. 3. Ontology evolution and instance data migration at the example of splitting a class

```
@prefix  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix  rdfs: <http://www.w3.org/2000/01/rdf-schema#
@prefix  log:  <http:/powl.sf.net/logOnt>

C1  rdf:type         log:PropertyMove
C1  log:pmProperty   P1
C1  log:pmFrom       C1
C1  log:pmTo         C2
C1  log:pmReference  P2
C1  log:removed      S1
C1  log:added        S2

S1  rdf:type         rdf:Statement
S1  rdf:subject      P1
S1  rdf:predicate    rdfs:domain
S1  rdf:object       C1

S2  rdf:type         rdf:Statement
S2  rdf:subject      P1
S2  rdf:predicate    rdfs:domain
S2  rdf:object       C2
```

7 Data Migration Strategies

One of the main advantages of using ontologies in a distributed environment as the World Wide Web is the reuse of structural information (schemata) encoded in an ontology. If such an ontology representing structural information evolves, ontologies containing data bound to this structural information have to be adopted as well. To automate this task as much as possible it is therefore desirable to have instance data migration algorithms for evolution patterns available. In the following two subsections we give examples for data migration algorithms for the common evolution patterns `log:PropertyMove` and `log:ClassSplit`.

Moving a Property. Assuming we have a change on a graph G belonging to the evolution pattern `log:PropertyMove` moving a directly attached property P_1 from a class C_1 to some other class C_2 using a property P_2 relating C_1 to C_2. A data migration algorithm can be given as follows:

```
foreach triple (?i1,rdf:type,C1) in G
  find triple (i1,P1,?v) in G
  foreach triple (i1,P2,?i2) in G
    add triple (i2,P1,v) to G
    del triple (i1,P1,v) from G
```

It moves the P_1 property values of instances of C_1 to the related instances of C_2.

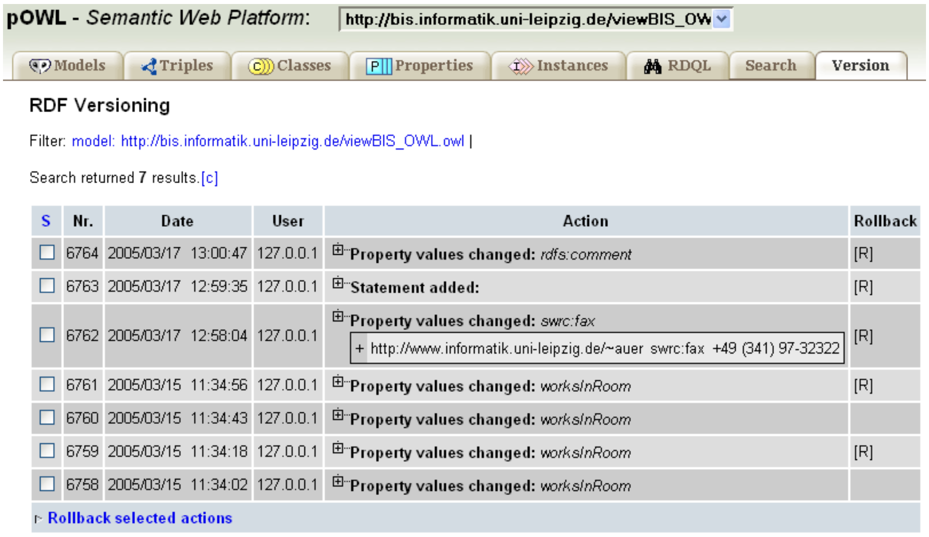
Splitting a Class. Since splitting a class requires to move properties, an appropriate data migration algorithm for the `log:ClassSplit` evolution pattern may make use of the `log:PropertyMove` data migration:

```
1   add triple (C1,rdf:type,owl:class) to G
2   foreach triple (?i1,rdf:type,C) in G
3     create new instance identifier i
4     add triple (i,rdf:type,C1) to G
5     add triple (i1,R,i) to G
6   foreach moved property P
7     PropertyMove(C,C1,P)
```

First a class $C1$ is created (line 1), thereafter for every instance of C a corresponding instance of C_1 is created, whereas the relation between both is established by the property R (lines 3-5) and finally the `log:PropertyMove` data migration algorithm is used for every moved property (lines 6,7).

8 Related Work and Summary

The versioning strategy described in this paper was implemented in the web application development framework Powl [1], which provides a comprehensive web user interface for collaborative knowledge base authoring as well as an application programming interface for PHP developers. To every change on the knowledge base using Powl, an optional versioning comment can be attached describing the change for review by humans. The user interface of Powl's versioning module then enables users to review changes chronologically, their compatibility with the current version is indicated and distinct changes may be rolled back. Changes may be filtered according to user, ontology, and date. Compound changes may be expanded up to the atomic change level indicating added (respectively removed) statements.


pOWL - Semantic Web Platform:
http://bis.informatik.uni-leipzig.de/viewBIS_OW
Models
Triples
Classes
Properties
Instances
RDQL
Search
Version

RDF Versioning

Filter: model: http://bis.informatik.uni-leipzig.de/viewBIS_OWL.owl |

Search returned 7 results.[c]

S	Nr.	Date	User	Action	Rollback
☐	6764	2005/03/17 13:00:47	127.0.0.1	⊞ Property values changed: *rdfs:comment*	[R]
☐	6763	2005/03/17 12:59:35	127.0.0.1	⊞ Statement added:	[R]
☐	6762	2005/03/17 12:58:04	127.0.0.1	⊞ Property values changed: *swrc:fax* + http://www.informatik.uni-leipzig.de/~auer swrc:fax +49 (341) 97-32322	[R]
☐	6761	2005/03/15 11:34:56	127.0.0.1	⊞ Property values changed: *worksInRoom*	[R]
☐	6760	2005/03/15 11:34:43	127.0.0.1	⊞ Property values changed: *worksInRoom*	
☐	6759	2005/03/15 11:34:18	127.0.0.1	⊞ Property values changed: *worksInRoom*	[R]
☐	6758	2005/03/15 11:34:02	127.0.0.1	⊞ Property values changed: *worksInRoom*	

▹ Rollback selected actions

Fig. 4. Reviewing changes with Powl

Other approaches targeting to support ontology evolution and versioning can be roughly divided into two categories:

– Approaches which are aware of the trace of changes which result in a new version and
– Approches which compare ontologies and compute differences or mappings between them.

Ognyanov and Kiryakov in [7] (falling in the first category) define a formal model for tracking changes in graph-based data models. Higher-level evaluation or classification of the updates are beyond the scope of their work. Those are studied and discussed in depth, for example, in [3]. Our contribution here is a way to easily relate low-level changes on the statement level to higher-level changes on the level of complex operations. In [6] (falling in the second category) automatic techniques based on heuristic comparisons for finding similarities and differences between versions are developed. [10] develop a merging method for ontologies following a bottom-up approach which offers a structural description of the merging process. These approaches are complementary to the presented one, since they are applicable even if ontology editors or storage systems do not support a finely grained change tracking. Ljiljana Stojanjovic's work [9] on ontology evolution gives an overview over current developments.

We presented a method for specifying complex changes by means of less complex changes and finally atomic changes on a graph. This method is especially suited to be implemented in ontology editors and storage systems. In a dynamic distributed environment sets of changes may then independently spread out from the originating ontologies. A user of some ontology may decide for every single change whether he accepts it or not. Assistance for this decision is provided by the compatibility concept between an ontology and a change. Annotation of changes on OWL ontologies with corresponding ontology evolution patterns further enables automatic data migration of independently stored instance data agreeing on the changed ontology. In this context the development of an exhaustive library of ontology evolution patterns with appropriate data migration algorithms is planned.

References

1. Sören Auer. Powl: A web based platform for collaborative semantic web development. In Sören Auer, Chris Bizer, and Libby Miller, editors, *Proceedings of the Workshop Scripting for the Semantic Web*, number 135 in CEUR Workshop Proceedings, Heraklion, Greece, 05 2005.
2. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, 284(5):34–43, May 2001.
3. Ying Ding, Dieter Fensel, Michel Klein, and Borys Omelayenko. Ontology management: survey, requirements and directions. Technical report, IST 1999-10132 Ontoknowledge Project, Deliverable 4, 2001.
4. Michel Klein, Dieter Fensel, Atanas Kiryakov, Natasha F. Noy, and Heiner Stuckenschmidt. Wonderweb deliverable D20. versioning of distributed ontologies, December 18 2002.
5. Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C Recommendation (http://www.w3.org/TR/rdf-concepts), 2 2004.
6. Natalya Fridman Noy, Sandhya Kunnatur, Michel C. A. Klein, and Mark A. Musen. Tracking changes during ontology evolution. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, November 7-11, 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2004.
7. Damyan Ognyanov and Atanas Kiryakov. Tracking changes in RDF(S) repositories. In Asunción Gómez-Pérez and V. Richard Benjamins, editors, *EKAW*, volume 2473 of *Lecture Notes in Computer Science*, pages 373–378. Springer, 2002.
8. Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Working Draft (http://www.w3.org/TR/rdf-sparql-query/), 2005.
9. Ljiljana Stojanovic. *Methods and Tools for Ontology Evolution.* PhD thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe (TH), 2004.
10. Gerd Stumme and Alexander Maedche. FCA-Merge: A bottom-up approach for merging ontologies. In *JCAI '01 - Proceedings of the 17th International Joint Conference on Artificial Intelligence, Seattle, USA, August, 1-6, 2001, San Francisco/CA: Morgen Kaufmann 2001/07/04*, 2001.

A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules

Thomas Baar and Slaviša Marković

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{thomas.baar,slavisa.markovic}@epfl.ch

Abstract. Refactoring is a powerful technique to improve the quality of software models including implementation code. The software developer applies successively so-called refactoring rules on the current software model and transforms it into a new model. Ideally, the application of a refactoring rule preserves the semantics of the model, on which it is applied. In this paper, we present a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class diagrams and OCL constraints. Our approach is based on a novel formalization of the OCL semantics in form of graph transformation rules. We illustrate our approach using the refactoring rule `MoveAttribute`.

Keywords: Refactoring, Semantic Preservation, UML, OCL.

1 Introduction

Modern software processes advocate the frequent application of so-called *refactoring rules* in order to improve the quality of software under development. A refactoring step is typically a small change made in a schematic way. Many approaches and tools have been developed for refactoring of implementation code but refactoring of more abstract software models, such as UML class diagrams (e.g. [1,2]) became only recently a research topic. In our previous paper [3] we have formalized refactoring rules for UML class diagrams and OCL invariants (called *UML/OCL models* in the remainder of this paper) using a graph-transformation based formalism. In this paper, we present a technique to prove the correctness of our refactoring rules.

There are two important criteria for the correctness of refactoring rules. Firstly, a rule should be *syntactic preserving*, i.e., whenever the rule is applicable on a source model then the target model obtained by the application of the rule is syntactically correct, i.e., the target model is an instance of the UML/OCL metamodel and obeys all of the metamodel's multiplicity constraints and well-formedness rules. Secondly, a rule should be *semantic preserving*, i.e., the semantics of source and target model should coincide. The proof of both syntactic and semantic preservation can be challenging (see [4]). This paper concentrates on proving semantic preservation.

A proof for semantic preservation must rely on a formal semantics of source and target models and a criterion for their semantic equivalence. For UML/OCL models, a formal semantics based on set theory is given in [5] but this semantics is clumsy when arguing on the semantic preservation of a graphically defined refactoring rule. For this reason, we propose here a novel formalization of OCL's semantics in form of graph-transformation rules. Moreover, we give a simple criterion for the semantic equivalence of two UML/OCL models and show how this criterion is met by the refactoring rule `MoveAttribute`.

The rest of the paper is structured as follows. In Section 2, we give based on an example a brief introduction to graph transformations. Section 3 applies graph transformations for the formalization of the `MoveAttribute` refactoring and defines a criterion for semantic preservation. The section closes with two, more complicated versions of `MoveAttribute` whose formalization requires the usage of semantic preconditions. Section 4 presents a graphical definition of OCL's semantics and applies this semantics for proving the semantic preservation of `MoveAttribute`. Section 5 concludes the paper.

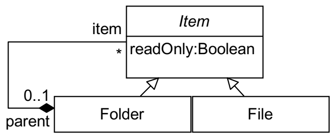
1.1 Related Work

In his seminal work [6], Opdyke gives a catalog of refactoring rules for C++ programs. Opdykes defines semantic preservation (also called *behavioral preservation* if implementation code is refactored) as "...if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same". In practice, it turned out that this simple criterion is hard to prove. Thus, more fine grained criteria such as *access preservation*, *update preservation*, and *call preservation* has been discussed in literature (an overview is given by Mens et al. in [7]).

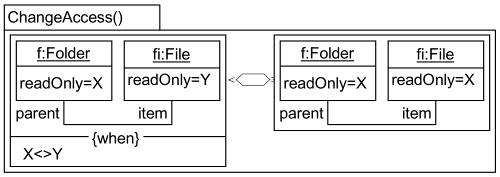
2 Graph Transformation Rules

A graph transformation rule defines how source models are transformed into target models. A model is seen here as a typed graph, more precisely, as an instance of the modeling language's metamodel (see App. A for the relevant part of the UML/OCL metamodel). We assume the reader to be familiar with the technique of metamodeling (a good introduction is [8]).

A graph transformation rule consists of two patterns called *left hand side* (LHS) and *right hand side* (RHS), which are denoted in a generalized form of object diagrams over the metamodel for the transformed modeling language. A graph transformation rule is applied on a given source model by (1) searching a LHS-matching region and (2) substituting the matched region by RHS under the same matching. If LHS matches with more than one region in the source model, one of the regions is non-deterministically chosen and rewritten by RHS. The application of the rule is repeated until the current model does not contain any LHS-matching region. A matching is a binding of all pattern variables to concrete values. Pattern variables are used in LHS and RHS in order to identify objects or as a representation of attribute values. The value of pattern variables are possibly restricted by the when-clause of the rule.



(a) FolderFile metamodel



(b) ChangeAccess transformation rule

Fig. 1. Metamodel and transformation rule

We illustrate the application of graph transformation rules on models written in a simple FileFolder-language, whose metamodel is given in Fig. 1(a). Instances of this metamodel are tree structures over folders and files. Each file or folder has an attribute `readOnly` of type `Boolean`. Suppose, a transformation should update for each file in the tree the value of its attribute `readOnly` with the `readOnly` value of its parent folder (if such a folder exists). Such a transformation is concisely formalized by the graph transformation rule `ChangeAccess` shown in Fig. 1(b).

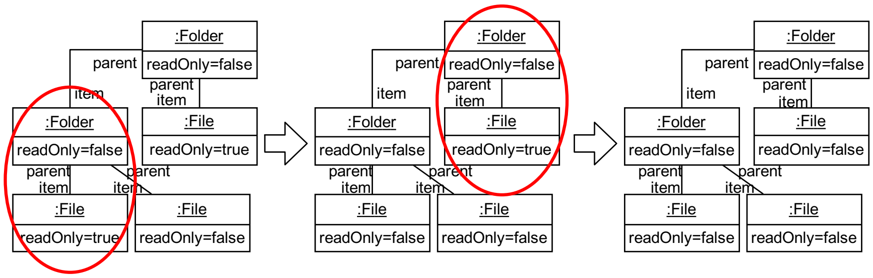


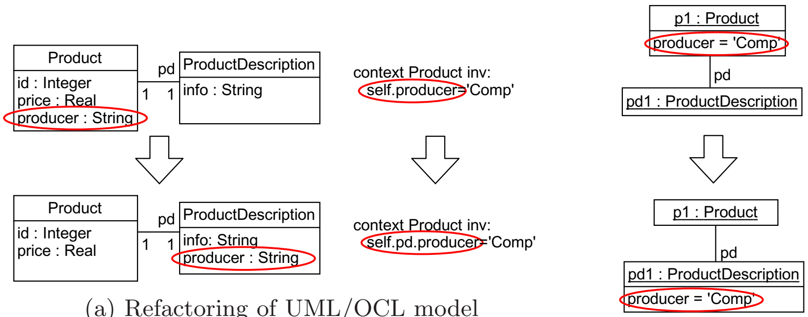
Fig. 2. Sequence of transformations

The LHS of `ChangeAccess` matches in a given source model with each pair of `File-Folder` instances that are connected by a parent-item link and whose values for attribute `readOnly` are different (see when-clause). Due to the RHS, the LHS-matching structure is rewritten by the same pair of `File-Folder` instances but the value for `readOnly` in the file has changed. The rule `ChangeAccess` is applied iteratively as long as LHS-matching structures can be found. Note how termination of this process is ensured by the when-clause. Figure 2 shows an application of `ChangeAccess` on a concrete source model.

3 Formalization of Semantic Preserving Refactoring Rules for UML/OCL

Research on refactoring has focused so far on implementation code but many refactoring rules for (object-oriented) implementation languages can be adapted to UML class diagrams and OCL constraints [3]. Since refactoring rules for UML/OCL models refer to the metamodel defining UML class diagrams and OCL expressions, we have included – for the sake of understandability – the relevant fragments of the metamodel in App. A.

Figure 3(a) shows the application of the refactoring rule `MoveAttribute` on a concrete UML/OCL model. The attribute `producer` is moved over an association with multiplicity 1 on both ends (called 1–1 association in the remainder of the paper) from class `Product` to `ProductDescription`. The attached OCL constraint has to be changed as well since the referred attribute `producer` is not owned any longer by class `Product`.



(a) Refactoring of UML/OCL model

(b) Refactoring of object diagram

Fig. 3. Application of `MoveAttribute` on an example

In the rest of this section we present a graph-transformation based formalization of the refactoring rule `MoveAttribute` and, as a new contribution of this paper, give a correctness criterion for the semantic preservation of UML/OCL refactoring rules. The section closes with a discussion on applying the correctness criterion on more complicated variants of the `MoveAttribute` rule, in which the attribute is moved over an 1–* or *–1 association.

3.1 Formalization of the Simple Form of `MoveAttribute`

In [3], we have already formalized a number of frequently used refactoring rules for UML class diagrams and analyzed their influence on OCL constraints attached to the refactored class diagram. The formalization of rule `MoveAttribute`

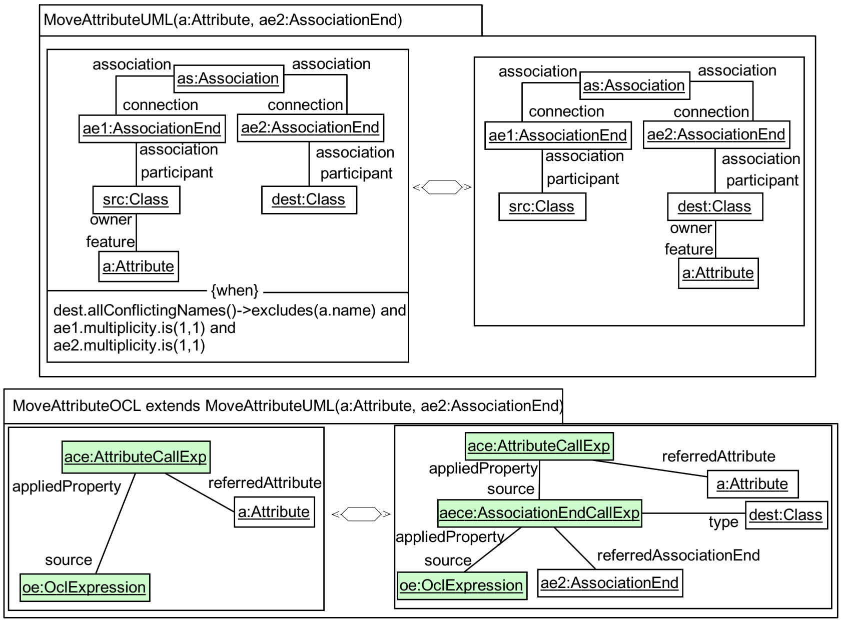


Fig. 4. Influence of `MoveAttribute` on class diagrams and OCL constraints

is presented in Fig. 4. The refactoring is split into two graph transformation rules, where the second one, which describes changes on OCL, extends the first rule, which formalizes the changes on the UML class diagram. The two parameters *a* and *ae2* of the first rule determine the attribute to be moved together with the association over which the attribute is moved (note that the parameter *ae2* identifies both the association and the destination class). The when-clause of the first rules prevents rule applications that would yield syntactically incorrect target models (an attribute must not be moved if its name is already used in the destination class). Furthermore, the when-clause explicates the assumption of moving the attribute over an 1–1 association.

Since the second rule is an *extension*, it can refer to elements from the extended rule, e.g. *a:Attribute*. Semantically, rule extension means that the second rule is applied as many times as possible in parallel to each single application of the first rule. For our example: Whenever attribute *a* is moved from class *src* to class *dest* each attribute call expression of form *oe.a*[1] is rewritten by *oe.ae2.a*.

3.2 A Correctness Criterion for Semantic Preservation

Semantic preservation, intuitively, means that source and target model express 'the same'. Established criteria for the refactoring of implementation code, where 'the same' usually means that the observable behavior of original and refactored

[1] Here, for the informal argumentation, the attribute call expression mentioned in `MoveAttributeOCL` is rendered in OCL's concrete syntax.

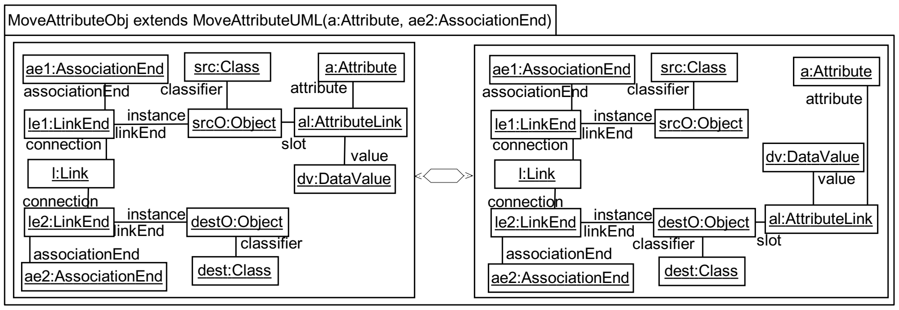


Fig. 5. Influence of `MoveAttribute` on object diagrams

program coincide, cannot be used for UML/OCL models, simply because the refactored UML class diagram with annotated OCL constraints is a static model of a system and does not describe behavior.

We propose to call a UML/OCL refactoring rule *semantic preserving* if the conformance relationship between the refactored UML/OCL model and its instantiations is preserved. An *instantiation* can be represented as an object diagram whose objects, links and attribute slots obey all type declarations made in the class diagram part of the UML/OCL model. An object diagram *conforms to* a UML/OCL model if all OCL invariants evaluate to true and all multiplicity constraints for associations of the class diagram are satisfied. A first – yet coarse and not fully correct (see below) – characterization of conformance preservation is that whenever an object diagram does/does not conform to the source model, it also does/does not conform to the target model.

This criterion, however, is still too coarse since it ignores the structural changes of instances of source and target model, e.g., applying `MoveAttribute` changes the owning class of the moved attribute (see Fig. 3(b) for illustration). In order to solve this problem, one has to bridge these structural differences of the model instances. This is realized by the transformation shown in Fig. 5.

Taking the structural differences between instances of source and target model into account, the semantic preservation can now be formulated as:

Definition 1 (Semantic Preservation of UML/OCL Refactorings) *Let cd_o be a class diagram, $constr_o$ be any of the constraints attached to it, od_o be any instantiation of cd_o, and $cd_r, constr_r, od_r$ be the refactored versions of $cd_o, constr_o, od_o$, respectively. The refactoring is called* semantic preserving *if and only if*

$$eval(constr_o, od_o) = eval(constr_r, od_r)$$

holds, where $eval(constr, od)$ denotes the evaluation of the OCL constraint $constr$ in the object diagram od.

3.3 Formalization of General Forms of `MoveAttribute`

The formalization of `MoveAttribute` covers so far a rather simple case: The attribute *a* is moved from the source to the destination class and in all attached OCL constraints, the attribute call expressions of form *oe.a* are rewritten to *oe.ae2.a*. Semantic preservation of the rule is rather intuitive because for each object *srcO* of source class *src* there exists a unique, corresponding object *destO* of destination class *dest* and the slot *al* for attribute *a* on *srcO* is moved to *destO* (see rule `MoveAttributeObj` in Fig. 5). Before we present in Section 4 a technique to prove semantic preservation, we want to formalize now some versions of rule `MoveAttribute` for other cases than moving over an 1–1 association. As we will see shortly, the semantic preservation of the more general forms of `MoveAttribute` can only be ensured if the conditions for applying the rule (formalized by the when-clause) also refer to object diagrams.

We discuss in the next Subsection 3.3.1 the case that the association keeps multiplicity 1 at the end of the destination class but has an arbitrary multiplicity at the opposite end of the source class. Subsection 3.3.2 discusses the opposite case with multiplicity 1 at the source end and arbitrary multiplicity at the destination end. The last case, arbitrary multiplicity at both ends, is not discussed here explicitly since this case is covered by combining the mechanisms used in the two other cases.

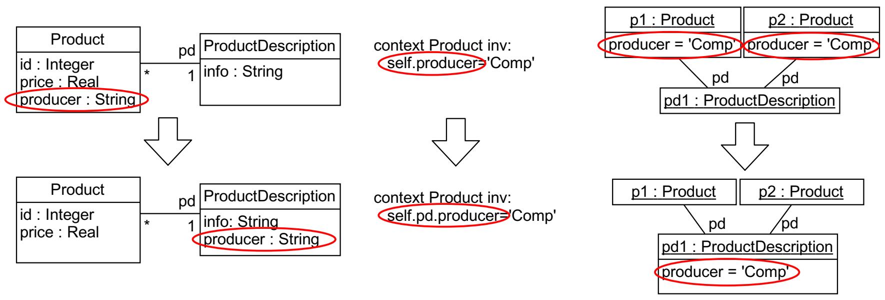


Fig. 6. Example refactoring if connecting association has multiplicities *–1

3.3.1 Multiplicities *–1

The UML and OCL part of the refactoring rule are basically the same as for moving the attribute over an 1–1 association. The only change is a new semantic precondition in order to ensure semantic preservation: All source objects (i.e., objects of the source class), which are connected to the same destination object (in Fig. 6, the source objects *p1*, *p2* are connected to the same object *pd1*), must share the same value for the moved attribute. For this reason, the when-clause of the UML part has changed compared to the previous version shown in Fig. 4 to:

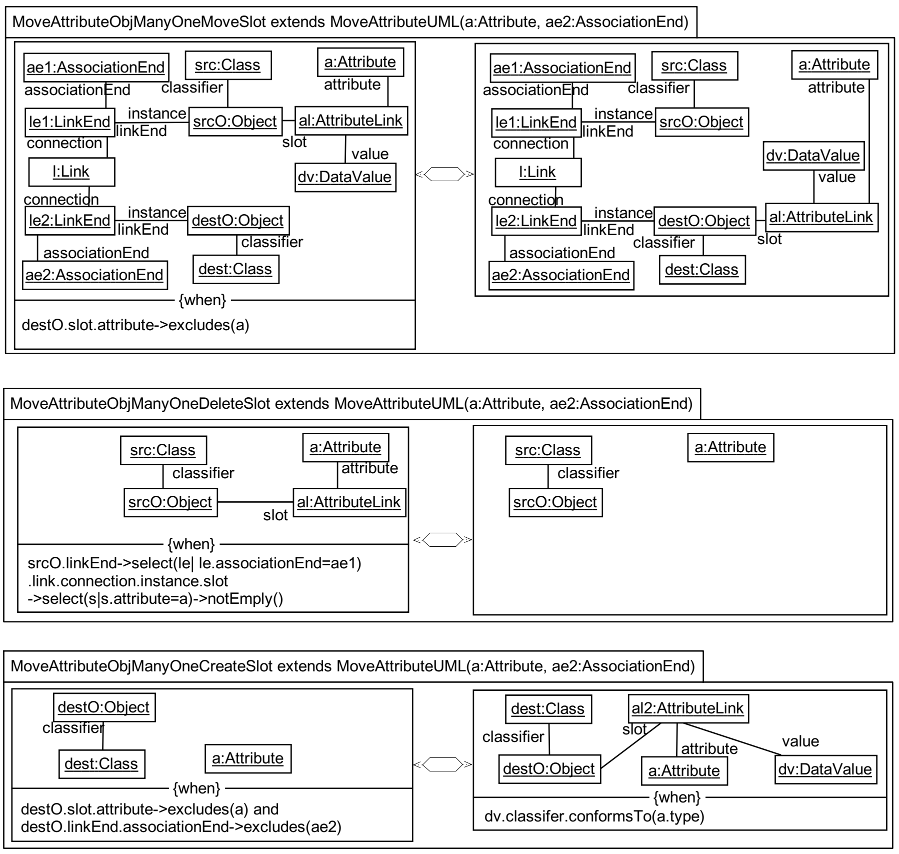

MoveAttributeObjManyOneMoveSlot extends MoveAttributeUML(a:Attribute, ae2:AssociationEnd)
ae1:AssociationEnd
associationEnd
src:Class
classifier
a:Attribute
attribute
le1:LinkEnd
instance
linkEnd
srcO:Object
al:AttributeLink
slot
value
connection
l:Link
dv:DataValue
connection
le2:LinkEnd
destO:Object
classifier
associationEnd
ae2:AssociationEnd
dest:Class
{when}
destO.slot.attribute->excludes(a)
MoveAttributeObjManyOneDeleteSlot extends MoveAttributeUML(a:Attribute, ae2:AssociationEnd)
{when}
srcO.linkEnd->select(le| le.associationEnd=ae1)
.link.connection.instance.slot
->select(s|s.attribute=a)->notEmply()
MoveAttributeObjManyOneCreateSlot extends MoveAttributeUML(a:Attribute, ae2:AssociationEnd)
al2:AttributeLink
destO.slot.attribute->excludes(a) and
destO.linkEnd.associationEnd->excludes(ae2)
dv.classifer.conformsTo(a.type)

Fig. 7. Object diagram part of refactoring rule if association has multiplicities *–1

```
dest.allConflictingNames()->excludes(a.name) and
ae2.multiplicity.is(1,1) and
dest.instance->forAll(do|
   do.linkEnd->select(le|le.associationEnd=ae2)
   ->collect(ae| ae.oppositeLinkEnd.instance)
   ->forAll(so1,so2|
      a.attributeLink->forAll(al1,al2|
          al1.instance=so1 and al2.instance=so2
          implies
          al1.value=al2.value)))
```

This *semantic precondition* seems, at a first glance, to be put at a wrong place. Is a refactoring of UML/OCL models not by definition a refactoring of the static structure of a system and done when developing the system? And at that time, are system states, i.e. the instantiations of the class diagram, not unavailable? Yes, this is a common scenario in which all refactoring rules, whose when-clause refers to object diagrams, are not applicable due to semantical problems a refactoring step might cause. But there are also other scenarios, e.g. where a class diagram describes a database schema and an OCL constraint can be seen as a selection criterion for database entries. Here, it would be possible to check whether the content of the database satisfies all semantic preconditions when applying the refactoring. If the refactoring rule is semantic preserving, one can deduce that a refactored database entry satisfies a refactored selection criterion if and only if the original selection criterion is satisfied by the original database entry.

The object diagram part of the refactoring shown in Fig. 7 reflects the fact that slots cannot be moved any longer naively, because the destination object would get in that case as many slots as it has links to source objects (but only one slot is allowed). The first two rules formalize that only one slot is moved to the destination object and all remaining slots at the linked source objects are deleted. The last rule `MoveAttributeObjManyOneCreateSlot` covers the case when a destination object is not linked to any source object. In this case, a slot for the moved attribute is created at the destination object and initialized with an arbitrary value (dv) of appropriate type.

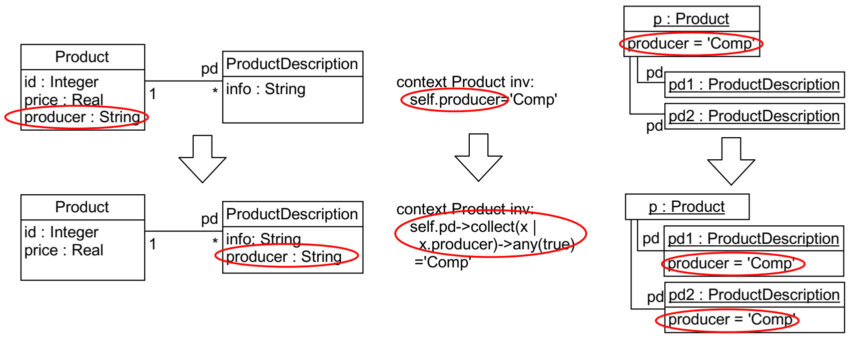


Fig. 8. Example refactoring if connecting association has multiplicities 1–*

3.3.2 Multiplicities 1–*

Compared with moving attribute over an 1–1 association, the refactoring has changed in the OCL part and in the object diagram part; the UML part has remained the same (except of a slight extension of the when-clause). In object diagrams, the slot for the moved attribute at each source object is copied to all the associated destination objects (see Fig. 8). Semantic preservation of the rule can only be ensured if for each source object at least one destination object exists, with which the source object is linked (otherwise, the information on the attribute value for the source object would be lost). Thus, the when-clause of the UML part has been rewritten as

```
dest.allConflictingNames()->excludes(a.name) and
ae1.multiplicity.is(1,1) and
src.instance->forAll(so|
        so.linkEnd->select(le|le.associationEnd=ae1)->notEmpty())
```

The object diagram part of the refactoring rule is changed as shown by the two upper rules in Fig. 9. The first rule copies the slot *al* for attribute *a* from the source object *srcO* to each of the linked destination objects *destO*. After this has been done, the second rule ensures deletion of slot *al* at the source object *srcO*. Note that this rule is essentially the same as the rule for deletion of slots in the previous subsection.

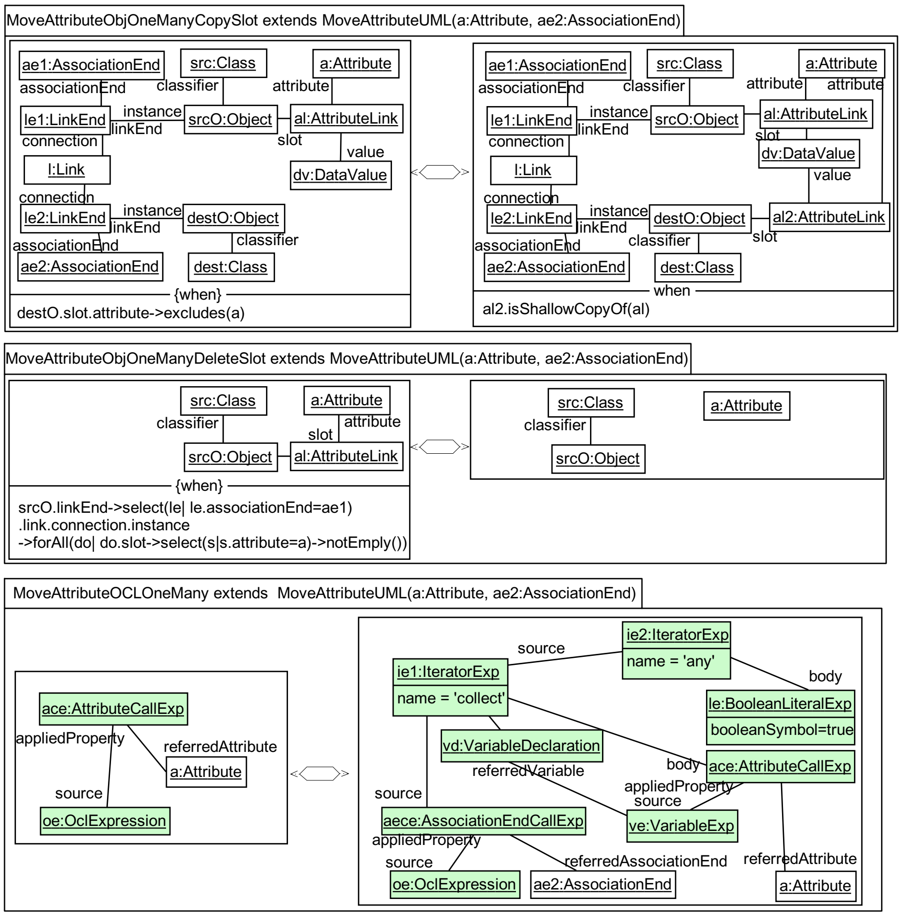


Fig. 9. Object diagram and OCL part of refactoring rule if connecting association has multiplicities 1–*

The third rule in Fig. 9 shows the OCL part of the refactoring rule. If the upper limit of the multiplicity at the destination class is greater than 1, the rewriting of *oe.a* to *oe.ae2.a*, as it was done in the previous versions of `MoveAttributeOCL`, would cause a type error since the type of subterm *oe.ae2* would be a collection type. However, since oe.ae2 is part of the attribute call expression *oe.ae2.a*, an object type would be expected.

In order to resolve this problem, the expression *oe.ae2* is wrapped by a collect()-expression, which is, in turn, wrapped by an any()-expression. Please note that, despite of the non-deterministic nature of any() in general, the rewritten OCL term oe.ae2−>collect(x|x.a)−>any() is always evaluated deterministically, because the subexpression oe.ae2−>collect(x|x.a) always evaluates in the refactored object diagram to a singleton set.

4 MoveAttribute Is Semantic Preserving

For a proof of the semantic preservation of a UML/OCL refactoring rule it is necessary to have a formal definition on how OCL constraints are evaluated. The evaluation function *eval* is defined with mathematical rigor in the OCL language specification [5]. The mathematical definition is, however, clumsy to apply in our scenario since it does not match the graph-based definitions we used so far for the formalization of our refactoring rules.

For this reason, we propose an alternative formalization of *eval* in form of graph-transformation rules. Due to the lack of space, we present here only the definition of *eval* for attribute call expressions and association end call expressions (a more complete version of OCL's semantics can be found in [9]). Fortunately, these two definitions are sufficient for proving the semantic preservation of `MoveAttribute` if the attribute is moved over an 1–1 association.

The formalization of *eval* given in Fig. 10 refers to a slightly extended version of the OCL metamodel in which the metaclass `OclExpression` has a new association to metaclass `Instance` (with multiplicity 0..1 and role `eval`). A link of this association from an object *oe:OclExpression* to an object *i:Instance* indicates that the expression *oe* is evaluated to *i*. If an expression does not have such a link to `Instance`, then this expression is not evaluated yet.

The first rule `EvalAttributeCallExp` defines the evaluation of expressions of form *oe.a* (where *a* denotes an attribute) in any object diagram that conforms to the underlying class diagram. The rule can informally be read as follows: Within the syntax tree of the OCL constraint to be evaluated, we search successively for expressions of form *oe.a* which are not evaluated yet (when-clause) but whose subexpression *oe* is already evaluated (to an object named *o*). Due to the type rules of OCL we know that object *o* must have a slot for attribute *a*. The lower part of the LHS shows the relevant part of the object diagram in which the OCL constraint is evaluated. The value of the slot for attribute *a* at object *o* is represented by variable *dv*. The RHS of rule `EvalAttributeCallExp` differs from LHS just by an added link from object *ac* (what represents expression *oe.a*) to *dv*. Informally speaking, the expression *oe.a* is now evaluated to *dv*. The second rule `EvalAssociationEndCallExp` is defined analogously. Based on this formalization we can state the following theorem:

Theorem 1 (Semantic Preservation of MoveAttribute). *Let* $cd_o, constr_o,$ od_o *be a concrete class diagram, a concrete OCL invariant, and a concrete object*

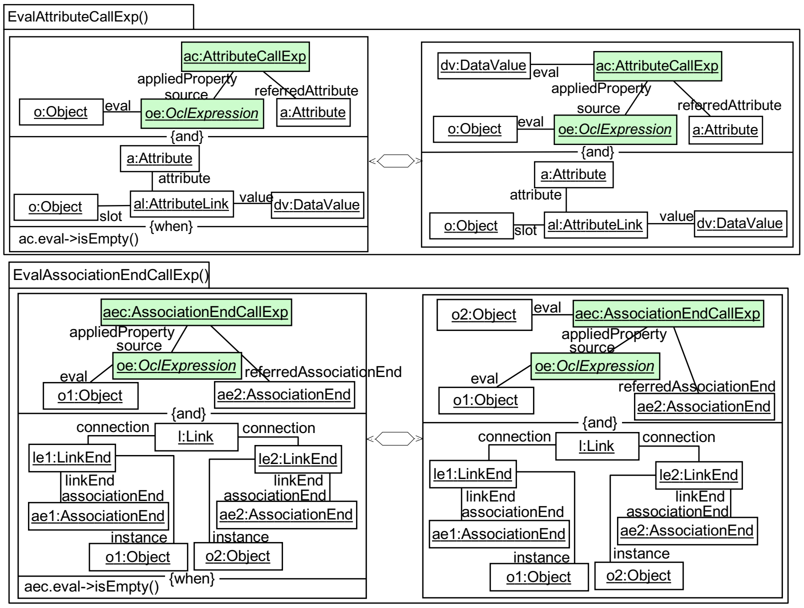


Fig. 10. Evaluation of OCL expressions (attribute call, association navigation)

diagram, respectively, and $cd_r, constr_r, od_r$ *their version after the refactoring of moving attribute a from class src to dest has been applied. Then,*

$eval(constr_o, od_o) = eval(constr_r, od_r)$

Proof: By construction, $constr_o$ and $constr_r$ differ only at places where $constr_o$ contains an expression form $oe.a$. The refactored constraint $constr_r$ has at the same place the expression $oe.ae2.a$. By structural induction, we show that these both expressions are evaluated to the same value. By induction hypothesis, we can assume that oe is evaluated for both expressions to the same value $srcO$. In object diagram od_o, object $srcO$ must have an attribute link for a, whose value is represented by dv. According to `EvalAttributeCallExp`, $oe.a$ is evaluated in od_o to dv. Furthermore, in both od_o and od_r the object $srcO$ is linked to an object $destO$ of class $dest$. According to `EvalAssociationEndCallExp`, the expression $oe.ae2$ is evaluated to $destO$ in od_r. Furthermore, we know by construction of od_r that $destO$ has an attribute slot for a with value dv. Hence, $oe.ae2.a$ is evaluated to dv.

5 Conclusions and Future Work

While the MDA initiative of the OMG has triggered recently much research on model transformations, there is still a lack of proof techniques for proving the semantic preservation of transformation rules. In the MDA context, this question has been neglected also because many modeling languages do not have an accessible formal semantics yet what seems to make it impossible to define criteria for semantic preservation. However, as our example shows, the semantic preservation of rules can also be proven if the semantics of source/target models is given only partially. In case of `MoveAttribute` it is enough to agree on the semantics of attribute call and association end call expressions.

In this paper, we define and motivate a criterion for the semantic preservation of UML/OCL refactoring rules. Our criterion requires to extend a refactoring rule by a mapping between the semantic domains (states) of source and target model. We argue that our running example `MoveAttribute` preserves the semantics according to our criterion. Our proof refers to the three graphical definitions of the refactoring rule (class diagram, OCL, object diagram) and to a novel, graphical formalization of the relevant parts of OCL's semantics.

As future work, we plan to apply our approach also on pure OCL refactoring rules, i.e., rules, which simplify the structure of complicated OCL expressions but do not change anything in the underlying class diagram (see [10]).

Acknowledgments. We would like to thank the anonymous reviewers for their very helpful comments. This work was supported by Swiss Science Foundation (SNF), contract number 200020-109492/1.

References

1. Dave Astels. Refactoring with UML. In *International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002.
2. Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.
3. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *MoDELS'05*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
4. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
5. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
6. William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
7. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.
8. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, second edition, 2005.
9. Slaviša Marković and Thomas Baar. An OCL semantics specified with QVT. In *MoDELS'06*, volume 4199 of *LNCS*, pages 660–674. Springer, October 2006.
10. Alexandre Correa and Cláudia Werner. Applying refactoringtechniques to UML/OCL. In *UML 2004*, volume 3273 of *LNCS*, pages 173–187. Springer, 2004.

A Metamodels

This appendix contains the relevant parts of the metamodels for UML 1.5 (including object diagrams) and OCL 2.0. For the sake of readability, the metaclasses from the OCL metamodel are rendered with gray rectangles.

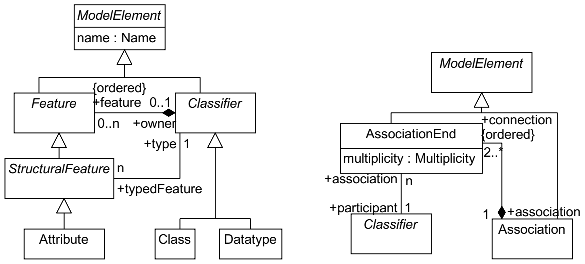


Fig. 11. UML - Core Backbone and Relationships

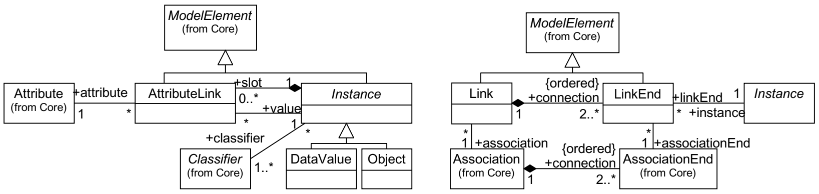


Fig. 12. UML - CommonBehavior Instances and Links

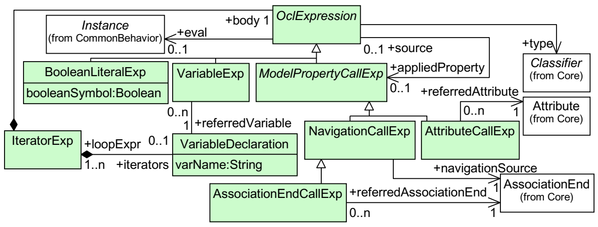


Fig. 13. OCL - Expressions

On the Usage of Concrete Syntax in Model Transformation Rules

Thomas Baar[1] and Jon Whittle[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
[2] Department of Information and Software Engineering
George Mason University
Fairfax VA 22030 USA
`thomas.baar@epfl.ch, jwhittle@ise.gmu.edu`

Abstract. Graph transformations are one of the best known approaches for defining transformations in model-based software development. They are defined over the abstract syntax of source and target languages, described by metamodels. Since graph transformations are defined on the abstract syntax level, they can be hard to read and require an in-depth knowledge of the source and target metamodels. In this paper we investigate how graph transformations can be made much more compact and easier to read by using the concrete syntax of the source and target languages. We illustrate our approach by defining model refactorings.

Keywords: Metamodeling, Model Transformation, Refactoring, UML.

1 Motivation

One of the key activities of model-based software development [1] is transformation between models. Model transformations are defined in order to bridge two different modeling languages (e.g., to transform UML sequence to UML communication diagrams) or to map between representations in the same language. A well-known example of the latter case is refactorings, i.e., transformations that aim at improving the structure of the source model [2,3].

Model transformations can be expressed in many formalisms (see [4] for an overview) but graph transformation based approaches [5] are especially popular due to their expressive power. Also the recently adopted OMG standard "Query, Views, Transformations (QVT)" is based on this technique [6]. The problem tackled in this paper is that model transformations written in a pure graph transformation notation can easily become complex and hard to read.

A transformation written in QVT consists of a set of transformation rules. Each rule has a left-hand-side (LHS) and right-hand-side (RHS) which define the patterns for the transformation rule's source and target models. A rule is applied on a given, concrete source model by matching a sub-model of the concrete model with the LHS of the rule and replacing the matched sub-model with the RHS, where any matchings are applied to the RHS before replacement. Additionally, all conditions imposed by the optional when-clause of the rule must be satisfied. The patterns defining the LHS and RHS are given in terms of the metamodels for the source and target modeling language (note that nowadays all major modeling languages, such as UML [7], are defined in the form of a metamodel). For the sake of simplicity in this paper (but our approach is not restricted to that), we will assume that the modeling languages for the source and target model coincide and thus each transformation rule refers only to the metamodel of one language.

A disadvantage of the graph transformation approach in defining model transformations is that the patterns LHS and RHS refer only to the abstract syntax of the modeling language and the more readable concrete syntax is not used in the transformation rule. Transformations written purely using abstract syntax are not very readable and require the reader to be familiar with the metamodel defining the abstract syntax. To overcome this problem, our approach is to write the transformation rules directly in the concrete syntax of the modeling language where possible. Unfortunately, this cannot be done directly since a number of subtleties of patterns in transformation rules have to be taken into account. In this paper, we make a distinction between the modeling language and the pattern language used to formulate the LHS and RHS. More precisely, we describe how the metamodel of the pattern language can be extracted from that of the modeling language. The extracted metamodel for the pattern language is then the basis to define a concrete syntax for patterns, that is similar to the concrete syntax of the original modeling language.

The rest of the paper is organized as follows. Section 2 gives some background information on defining modeling languages and model transformation techniques, with an emphasis on graph transformations. We show in Section 3 how to improve the readability of transformation rules by exploiting a concrete syntax derived from the source and target modeling language. Section 4 illustrates the strengths and some limitations of the approach by applying it to UML refactoring rules and Section 5 concludes the paper.

1.1 Related Work

The authors know of no other work in using concrete syntax for graph-based model transformations. There is a good deal of research in applying graph transformations to software engineering problems — see [8] for an introduction — such as code generation, viewpoint merging and consistency analysis. However, in all applications we have seen, the transformation rules are based on the abstract syntax of the source and target modeling languages.

Approaches addressing issues related to concrete syntax and transformations have been focused somewhat differently than our work. Papers on tool support for model transformations, e.g. [9], have discussed the problem of synchronizing a model and its visual representation after a transformation has been executed.

One approach is to extend the metamodels of the modeling languages with a metamodel for the visual representation of models (i.e., the concrete syntax) and to formulate the transformation rules based on this extended metamodel.

2 Defining Model Transformations

2.1 Metamodeling

A modeling language has three parts: (1) the abstract syntax that identifies the concepts of the language and their relationships, (2) the concrete syntax that defines how the concepts are represented in a human-readable format, and (3) the semantics of the concepts. This paper is only concerned with (1) and (2).

The abstract syntax of a modeling language is usually defined in the form of a metamodel. A metamodel is usually described by a (simplified form of a) UML class diagram [7] with OCL [10] invariants. The concepts of the language are defined by classes in the metamodel (i.e., meta-classes). Concept features are given as meta-attributes on meta-classes and relationships between concepts are given by meta-associations.

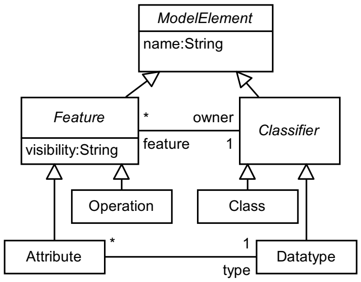
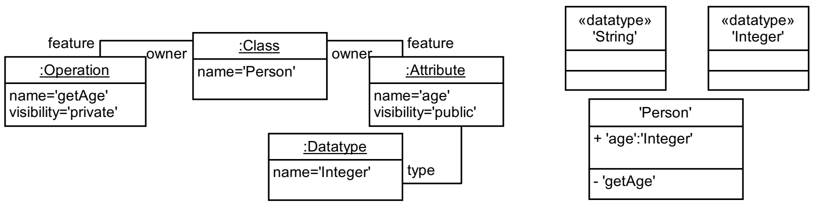


Fig. 1. Metamodel of simplified class diagrams, called *CDSimp*

Figure 1 shows the metamodel of a drastically simplified version of UML class diagrams, called *CDSimp*. The language *CDSimp* will serve as a running example in the remainder of this paper. The metamodel for *CDSimp* consists of metaclasses that correspond directly to concrete model elements, namely `Attribute`, `Operation`, `Class` and `Datatype`, as well as abstract metaclasses that do not have a concrete syntax representation but are introduced for structuring purposes: `ModelElement`, `Feature`, `Classifier`. For instance, the metaclass `ModelElement` declares a metaattribute `name` of type `String` that is inherited by all other metaclasses.

OCL invariants attached to the metamodel impose restrictions that every well-formed model must obey (thus, the invariants are also called *well-formedness rules*). Two invariants are relevant for the examples presented later in this paper.



(a) Instantiation of metamodel (b) Graphical notation using concrete syntax

Fig. 2. Two representations of the same class diagram

The first invariant says that the names of all features in a class or datatype are pairwise different and the second invariant restricts the values for visibility:

```
context Classifier inv UniqueFeatureName:
    self.feature->forAll(f1,f2| f1.name=f2.name implies f1=f2)

context Feature inv VisibilityDomain:
    self.visibility = 'public' or
    self.visibility = 'private' or
    self.visibility = 'protected'
```

Considering only the abstract syntax of a modeling language, one can say that a *model* written in this modeling language is just an instance of the language's metamodel that obeys the two given well-formedness rules (i.e. the two OCL invariants *UniqueFeatureName* and *VisibilityDomain* are evaluated in every model to true). A model can be depicted as an object diagram (cf. Figure 2(a)) but this is not very readable, because all concrete model concepts are reified as metaclasses in the metamodel. More readable for humans is a graphical representation of the same model that takes the *concrete syntax* of the language into account (cf. Figure 2(b)). The concrete syntax for *CDSimp* resembles that of UML class diagrams. The only difference is that string literals, such as the name of a class or attribute, are given in quoted form (e.g. 'Person' instead of Person). We will need this convention later on.

2.2 Concrete Syntax Definition

The concrete syntax of a language can be defined as a mapping from all possible instances of the language's metamodel into a representation format (in most cases, a visual language [11]). Despite the recent progress that has been achieved in formalizing diagrams (see, for example, OMG's proposal for Diagram Interchange [12] as an attempt to standardize all graphical elements that can possibly occur in diagrams), it is still current practice to define the concrete syntax of a modeling language informally. For the sake of brevity, we also give an informal definition here, but, as shown in [13], a conversion into a formal definition can be done straightforwardly. The language *CDSimp* has the following concrete syntax definition:

- Classes and datatypes are represented by rectangles with three compartments.
- The first compartment contains the name of the class/datatype. The name of datatypes is stereotyped with `<<datatype>>`.
- The remaining compartments contain the representation of all owned features (attributes are shown in the second, operations in the third compartment). A feature is represented by a line of the form:
  visiRepr ' ' *name* [':' *type*]
  where *visiRepr* is a representation of the feature's visibility ('+' for 'public', '-' for 'private', '#' for 'protected'), *name* is the actual name of the feature, and, in case of an attribute, *type* is the name of the attribute's type. Note that both visibility and name are mandatory parts of a feature representation.

Concrete syntax definitions are needed only for those concepts that are reified in a concrete model. For example, the abstract metaclass `Feature` does not have a concrete syntax definition.

2.3 Model Transformations

The exact format and semantics of model transformation rules is fully described in [6]. In this paper, we consider only the format of the patterns LHS and RHS in each transformation rule, and the relationship of LHS and RHS to the optional when-clause of the rule.

A pattern can be defined as a more general form of object diagram in which all objects are labeled by a unique variable with the same type as the object. Variables are also used in order to represent concrete values in objects for attributes. Unlike usual object diagrams, objects of abstract classes (e.g. `Classifier`) can occur in patterns.

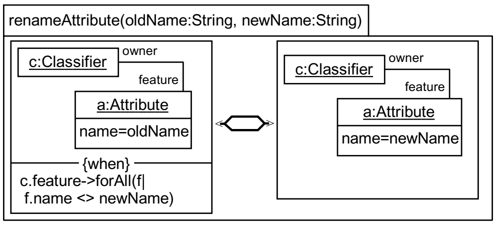


Fig. 3. QVT rule to rename an attribute within classifiers

Figure 3 shows an example transformation for renaming an attribute of a classifier. The pattern LHS specifies the subgraphs to be matched in the source model when the rule is applied. The LHS consists of two objects of type `Classifier` and `Attribute`, respectively, labeled with variables c and a, which are connected by a link instance of the owner-feature association. The value for metaattribute `name` in object a is the same as the value of the rule parameter *oldName* of type `String`. In addition, the when-clause requires object c to have no feature with name *newName*. The pattern RHS is identical to LHS with the exception that variable *oldName* is substituted by *newName*. Informally, the application of `renameAttribute` on a concrete source model involves the following steps: (1) find a classifier (i.e., since metaclass `Classifier` is abstract, a class or datatype) in the source model that has an attribute with name *oldName* (matching the LHS) but no feature with name *newName* (checking the when-clause) and then (2) rename the matching attribute from *oldName* to *newName* and do not make any other change in the model. These steps would be applied iteratively as often as possible. Note that the when-clause implicitly imposes the constraint that *newName* is different than *oldName*. This ensures termination of the rule application. The when-clause also ensures syntactical correctness of the target model. For example, it ensures that the well-formedness rule *UniqueFeatureName* is satisfied in the target.

3 Patterns In Concrete Syntax (PICS)

Graph transformation rules, such as those given by QVT and described in Section 2, are a very powerful mechanism to describe model transformations. Readability, however, can become a serious problem if the patterns LHS and RHS are given in object diagram syntax. The main idea of our approach is to alleviate this problem by exploiting the concrete syntax of the language whose models we want to transform. Unfortunately, we cannot apply the concrete syntax of the modeling language directly for the rendering of patterns because some important information of the pattern would be lost. We will, thus, first analyze the differences between a modeling language and the corresponding pattern language used in transformation rules. Then, the pattern language is defined by its own metamodel, which is, as shown in Section 3.2, a straightforward modification of the original metamodel for the modeling language. Based on the modified metamodel, we finally define a concrete syntax for the pattern language, which is called *PICS* (patterns in concrete syntax). The term *PICS metamodel* refers to the metamodel of the pattern language that has been derived from the metamodel of the modeling language.

3.1 Differences Between Models and Patterns

For defining a concrete syntax for pattern diagrams the following list of differences between models (seen as instances of the modeling language's metamodel) and patterns used in transformation rules has to be taken into account:

1. **Objects in patterns must be labeled**[1] with a unique variable (e.g. the label for `c:Class` is *c*).
2. **A pattern usually represents an incomplete model** whereas object diagrams are assumed to be complete, i.e., all well-formedness rules and multiplicities of the metamodel are obeyed. For example, the patterns LHS, RHS in `renameAttribute` (Figure 3) show neither the attribute `visibility` of object *a:Attribute* nor a link to its type (an object diagram could not drop this link due to multiplicity 1 of the corresponding association end at `Datatype`).
3. **Patterns can have objects whose type is an abstract class** whereas the type of objects in object diagrams is always a non-abstract class.
4. **Patterns can contain variables to represent attribute values in objects** whereas in object diagrams such values are always literals or ground-terms.

A pattern language is, due to these differences, more expressive than the language of object diagrams since each object diagram is also a pattern but not vice versa. Note, however, that the last difference is only a minor one. Variables for attribute values could easily be integrated into object diagrams as well if the value of attribute slots are always displayed according to some simple rules: (i) literals of type `String` must be enclosed by quotes and (ii) literals of all other types have to be pre-defined. If, under these conditions, a term occurs in an attribute slot that is neither a composed term nor a literal of type `String` or any other type, then this term would denote a variable.

3.2 Transforming the Original Metamodel to PICS Metamodel

The important differences between models and patterns (points (1) – (3) above) can be formalized by defining a metamodel for pattern diagrams. Fortunately, this metamodel can be automatically derived from the original metamodel by applying the following changes:

- Add attribute `label:String` with standard multiplicity 1..1 to each metaclass. This change captures the mandatory labels of objects in pattern diagrams (see difference (1) in above list of differences).
- Make all attributes in the metamodel optional (by giving them the attribute multiplicity 0..1) and change all association multiplicities from y..x to 0..x. Both changes reflect incompleteness of patterns (see difference (2)).
- Make all abstract classes non-abstract (see difference (3)).

Figure 4 shows the changes on the metamodel for *CDSimp*. The root class `ModelElement` has a new attribute `label` that is inherited by all other classes. The two other attributes `name` and `visibility` became optional by the attribute

[1] In many graph transformation systems including QVT the label is optional. We assume here the strict version since it will make it easier to rewrite a pattern using the concrete syntax.

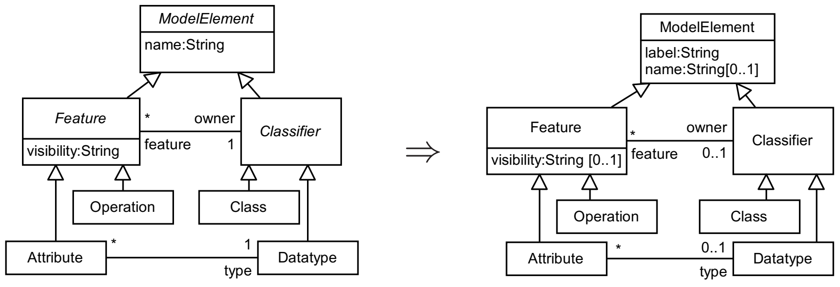
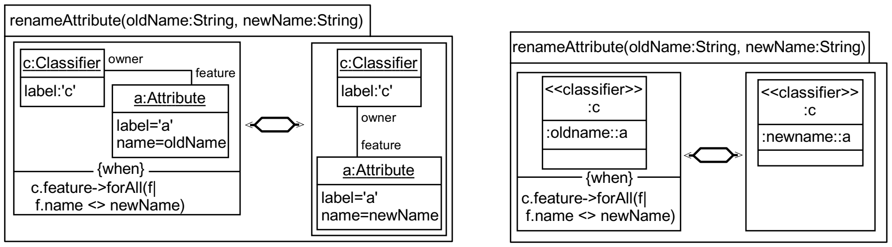


Fig. 4. Original language metamodel and derived PICS metamodel



(a) Rule as instance of PICS metamodel (b) After applying PICS concrete syntax

Fig. 5. Rule `renameAttribute` as instance of PICS metamodel and in concrete syntax

multiplicity 0..1. The abstract classes `ModelElement`, `Feature`, `Classifier` became non-abstract and finally all multiplicities on association ends were changed to the range 0..OrigMultiplicity (note that multiplicity * is not affected).

3.3 Defining Concrete Syntax for PICS Metamodel

After the pattern language has been formalized as the PICS metamodel, we can represent each pattern as an instance of the PICS metamodel. Figure 5(a) shows the transformation rule `renameAttribute` as an example. Please note that Figure 5(a) is just another representation of the original definition given in Figure 3 and conveys exactly the same information. Hence, each representation equivalent to Figure 5(a) is also equivalent to the original definition of the transformation rule.

Defining an equivalent representation for the instances of a metamodel is traditionally done by defining a concrete syntax for the metamodel. For our running example, a concrete syntax for the PICS metamodel shown in Figure 4 could be defined by modifying the concrete syntax for *CDSimp* as follows:

– Instead of the *name*, the first compartment of classes/datatypes shows a line of the form *name* ':' *label* where *name* denotes the value of the optional attribute `name` and *label* the value of the mandatory attribute `label`. Since *name* appears only optionally, a delimiter ':' between *name* and *label* is needed in order to ensure correct parsing. The delimiter must not occur in *name* and *label*.

– An attribute/operation having an owning classifier is shown by a text line in the second/third compartment of the owning classifier. The only difference to the concrete syntax of *CDSimp* is the usage of delimiter ':' to separate the line items (in order to handle optional occurrences) and that the label of the attribute/operation is added at the end of the line.

  In other words, the line has the form [*visiRepr*] ':' [*name*] [':' *type*] ':' *label* If an attribute/operation does not have an owning classifier (note the multiplicity 0..1 for the association between `Feature` and `Classifier` in the PICS metamodel) then the text line is shown outside any other classifier in a box.

– Instances of `Classifier` are rendered the same as classes/datatypes except that they have a stereotype `<<classifier>>` in the first compartment.

– Features (instances of `Feature`) are rendered the same way as attributes/ operations but, in order to distinguish them, they have to be marked as features. This could be done, for example, by preceding the text line with 'f:'.

– Instances of `ModelElement` are rendered by a one-compartment rectangle labeled with *name* ':' *label*.

The first two items explain how to adapt the renderings of metaclasses that are non-abstract both in the original metamodel of the modeling language *CDSimp* and in the PICS metamodel. The rendering in PICS is very similar to that in *CDSimp*. Merely the label of the object had to be added and a delimiter was introduced to identify the position of an element in a text line. The remaining items explain the rendering of metaclasses that were abstract in the original metamodel but became non-abstract in PICS. Since no rendering of these classes was defined for *CDSimp*, the new renderings for the PICS metamodel had to be invented. An application of the PICS concrete syntax is shown in Figure 5(b) for `renameAttribute`.

To summarize so far, we have defined the abstract syntax (using a metamodel) of the pattern language for defining patterns in the LHS and RHS of a graph transformation rule. Furthermore, we have shown on an example how to define concrete syntax for this pattern language based on the syntax of the associated modeling language.

3.4 Finding a Good Concrete Syntax for the Pattern Language

Although it is always possible to define a concrete syntax for the PICS metamodel (note that showing the instance of the metamodel just as an object diagram – see Figure 5(a) for an example – would be a trivial version of a concrete syntax) it is usually a challenge to find a non-ambiguous concrete syntax that is still similar to the concrete syntax of the modeling language whose models are being transformed. The definition of a good concrete syntax is of primary importance for the readability and understandability of the transformation rules written in PICS syntax.

There are basically two problems to tackle: (1) Handling of optional occurrences of attribute and links and (2) rendering of classes that were abstract in the metamodel of the modeling language.

The first problem was tackled in the above *CDSimp* example by using delimiters that allow to infer for a rendered object which of its attributes are rendered and which not. This technique, however, needs the assumption that the symbol used as delimiter (here ':') is not used otherwise in order to avoid ambiguity of the representation. Some initial tool support for detecting ambiguities in a concrete syntax definition is described in [14].

In order to solve the second problem, new icons/symbols have to be invented which raises the issue of similarity between the original modeling language and the pattern language. For some classes, e.g. `Feature` and `Classifier`, a suitable rendering can be defined as a straightforward generalization of the renderings of the subclasses. For other classes, e.g. `ModelElement`, this heuristic does not work just because the renderings of the subclasses are too diverse.

As future work, we plan to investigate pattern languages that allow mixing of abstract and concrete syntax. This could be done by leaving the concrete syntax definition for the pattern language incomplete. This would mean that some parts of patterns do not have a rendering in the concrete syntax, and would be done when there exists no rendering that is similar to the modeling language. For example, if a rule needs to refer to an abstract metaclass that has no rendering, we would allow this abstract metaclass to be referenced in the pattern definition. In essence, this allows patterns to mix concrete and abstract syntax. A mechanism would be required to control whether an element is concrete or abstract but this could be done, for example, in a similar way to the quote/anti-quote mechanism in LISP.

4 Case Study: UML Refactoring Rules in PICS Notation

In [15], a number of refactoring rules for UML class diagrams using the abstract syntax of class diagrams has been defined. We present, in the following, a rewriting of these refactoring rules[2] using our PICS approach.

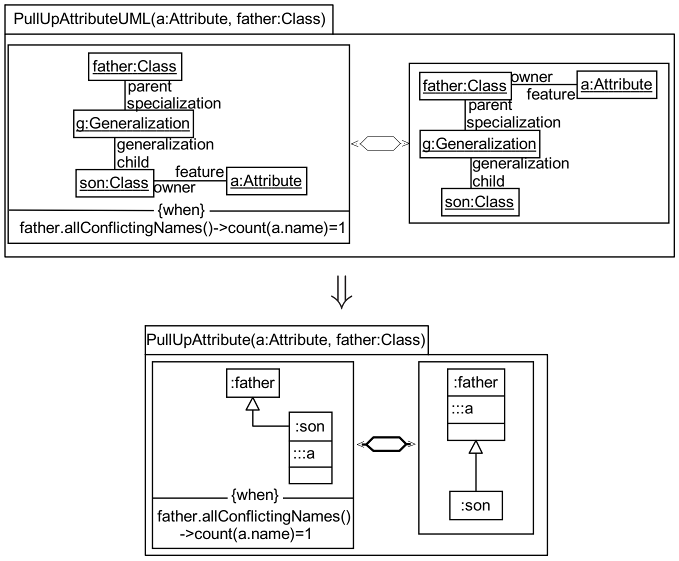
The refactoring rules are written with respect to the metamodel for UML 1.5 (see [15] for the relevant part of the metamodel). The refactoring rules are designed to preserve an important well-formedness rule of the UML 1.5 metamodel, namely that names for attributes, opposite association ends and owned elements are unique within each classifier and all its parents along the generalization hierarchy. In order to ensure this well-formedness rule, most refactoring rules have a when-clause that uses the following additional operation:

[2] More precisely, some improved variants of the rules given in [15] are taken here as a starting point.

```
context Classifier def: allConflictingNames():Bag(String)=
    self.allParents()->including(self)
   ->union(self.allChildren())
   ->iterate(c; acc:Bag(String)=Bag{}|
        acc->union(c.oppositeAssociationEnds().name)
           ->union(c.attributes().name)
           ->union(c.ownedElement.name))
```

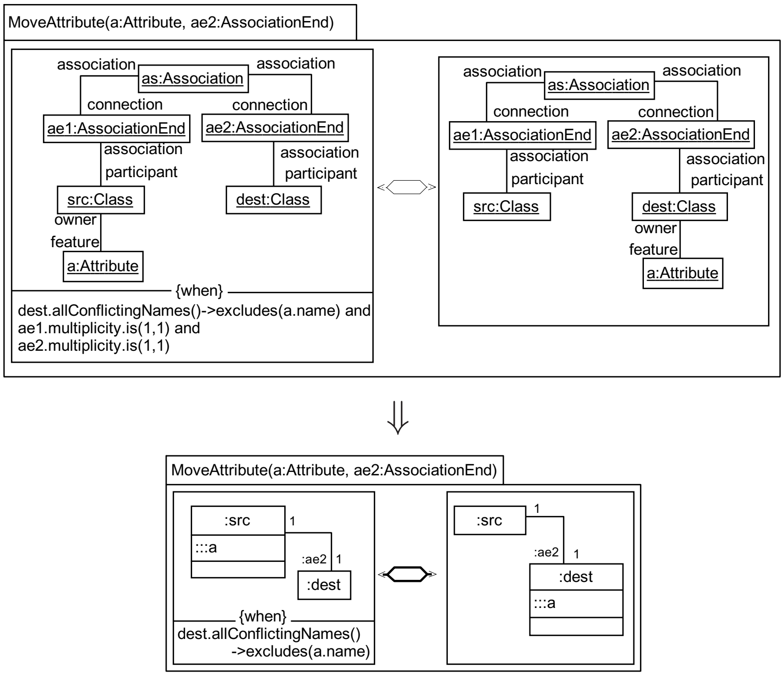
4.1 PullUpAttribute

The rule *PullUpAttribute* moves an attribute from a child class to a parent class. It can be rewritten straightforwardly using the same PICS syntax that has been used for *CDSimp*. Please note that we do not rewrite the when-clause. We show below the *PullUpAttribute* both in its original form and the rewritten version (all the examples will be presented in a similar fashion).



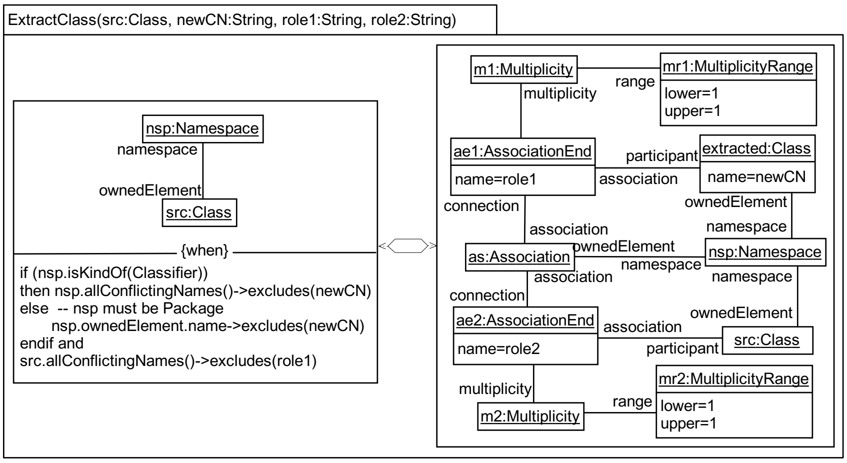
4.2 MoveAttribute

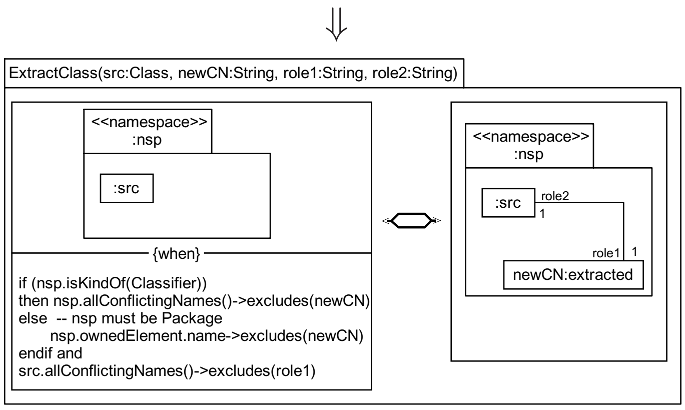
The refactoring *MoveAttribute* moves an attribute from a class on one end of an association to a class on the other end of the association. In the rewritten version, the when-clause is modified. The part of the when-clause stipulating the connecting association to be of multiplicity 1-1 is in the rewritten version rendered by an annotation 1 on the association ends, which is a standard technique in UML. More complicated forms of OCL constraints could be represented in a visual form. We do not discuss this topic here in-depth but refer the interested reader to [16] where graphical notations are defined as abbreviations for complex OCL expressions.



4.3 ExtractClass

The refactoring *ExtractClass* creates for a given class *src* a new class and connects it with *src* by an association with multiplicity 1-1. Furthermore, the created class and association should be placed in the same namespace as class *src*. Please note that the metaclass `Namespace` is abstract in the UML 1.5 metamodel. For this reason, a new rendering for `Namespace` has to be invented for the PICS syntax. Here, a package icon stereotyped with `<<namespace>>` has been chosen.




⇓
ExtractClass(src:Class, newCN:String, role1:String, role2:String)
<<namespace>> :nsp
:src
{when}
if (nsp.isKindOf(Classifier))
then nsp.allConflictingNames()->excludes(newCN)
else -- nsp must be Package
nsp.ownedElement.name->excludes(newCN)
endif and
src.allConflictingNames()->excludes(role1)
<<namespace>> :nsp
:src
role2
1
role1
1
newCN:extracted

5 Conclusion and Future Work

This paper addressed how to define model transformation rules in a more readable way by using the concrete syntax of source and target modeling languages when defining the LHS and RHS of the rules. The concrete syntax, however, had to be adapted to the peculiarities of patterns, mainly mandatory labeling of objects and optional occurrence of attributes and links. Another major problem is that the PICS concrete syntax has to invent a new rendering for metaclasses that were abstract in the original metamodel. An alternative, that has been only sketched in this paper, is to show these metaclasses in the abstract syntax notation, that is to allow mixing concrete and abstract syntax presentations within transformation rules. If an intuitive concrete syntax for patterns is found, then transformation rules can be presented in the same way as models of the source/target languages.

We have not addressed in this paper how the when-clause of transformations rules can be rendered in graphical form as well. There is a standard technique in graph-transformation literature how negative constraints can be made visible (known as non-application conditions (NACs)). Our approach could also be extended by the work of Stein, Hanenberg and Unland presented in [16] where visualizations of OCL constraints for the domain of metamodel navigation is discussed.

Acknowledgements. The authors would like to thank the anonymous reviewers for their very helpful comments on the initial version of this paper.

References

1. Stuart Kent. Model driven engineering. In *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
2. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

3. Martin Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.
4. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proc. OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
5. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.
6. OMG. MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01, Nov 2005.
7. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, second edition, 2005.
8. Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In A. Corradini, H. Ehrig, and H.-J. Kreowski und G. Rozenberg, editors, *First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 402–429. Springer, 2002.
9. Esther Guerra and Juan de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, pages 54–69, 2004.
10. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
11. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
12. OMG. Unified Modeling Language: Diagram interchange version 2.0. Convenience Document ptc/05-06-04, June 2005.
13. Frédéric Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In *Proc. European Conference on Model Driven Architecture (ECMDA-FA)*, volume 3748 of *LNCS*, pages 190–204. Springer, 2005.
14. Thomas Baar. Correctly defined concrete syntax for visual models. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings, MoDELS/UML 2006, Genova, Italy*, volume 4199 of *LNCS*, pages 111–125. Springer, October 2006.
15. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
16. Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query models. In *Proc. IEEE 7th International Conference on the Unified Modeling Language (UML 2004)*, volume 3273 of *LNCS*, pages 98–112. Springer, 2004.

TTCN-3 for Distributed Testing Embedded Software*

Stefan Blom[1], Thomas Deiß[2], Natalia Ioustinova[4], Ari Kontio[3], Jaco van de Pol[4,6], Axel Rennoch[5], and Natalia Sidorova[6]

[1] Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
[2] Nokia Research Center, Meesmannstrasse 103, D-44807 Bochum, Germany
[3] Nokia Research Center, Itämerenkatu 11-13, 00180 Helsinki, Finland
[4] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
[5] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589, Berlin, Germany
[6] Eindhoven Univ. of Techn., Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
Stefan.Blom@uibk.ac.at, thomas.deiss@nokia.com, ari.kontio@nokia.com, ustin@cwi.nl, Jaco.van.de.Pol@cwi.nl, axel.rennoch@fokus.fhg.de, n.sidorova@tue.nl

Abstract. TTCN-3 is a standardized language for specifying and executing test suites that is particularly popular for testing embedded systems. Prior to testing embedded software in a target environment, the software is usually tested in the host environment. Executing in the host environment often affects the real-time behavior of the software and, consequently, the results of real-time testing.

Here we provide a semantics for *host-based testing* with *simulated time* and a a simulated-time solution for *distributed* testing with TTCN-3.

Keywords: TTCN-3, distributed testing, simulated time.

1 Introduction

The Testing and Test Control Notation Version 3 (TTCN-3) is a language for specifying test suites and test control [17]. Its syntax and operational semantics are standardized by ETSI [4,5]. Previous generations of the language were mostly used for testing systems from the telecommunication domain. TTCN-3 is however a universal testing language applicable to a broad range of systems. Standardized interfaces of TTCN-3 allow to define test suites and test control on a level independent of a particular implementation or a platform [6,7], which significantly increases the reuse of TTCN-3 test suites. TTCN-3 interfaces provide support for distributed testing, which makes TTCN-3 particularly beneficial for testing embedded systems. TTCN-3 has already been successfully applied to test embedded systems not only in telecommunication but also in automotive and railway domains [2,9].

* This work is done within the project "TTMedal. Test and Testing Methodologies for Advanced Languages (TT-Medal) sponsored by Information Technology for European Advancement Programm (ITEA)" [15].

Modern embedded systems consist of many timed components working in parallel, which complicates testing and debugging. Potential software errors can be too expensive to test on a *target environment* where the system is supposed to work. In practice, embedded software is tested in the *host environment* used for developing the system. That allows to fix most errors prior to testing in the target environment.

The *host* environment differs from the *target* environment. When being developed, the actual system does not exist until late stages of development. *Environment simulations* are used to represent target environments. If the target operating system is not available, *emulating the target OS* is used to provide message communication, time, scheduling, synchronization and other services necessary to execute embedded software. *Monitoring* and *instrumentation* are used to observe the order and the external events of an SUT.

Ideally, using environment simulations, target operating system emulations, monitoring or instrumentation should not affect the real-time behavior of an SUT. In practice, developing simulators and emulators with high timing accuracy is often unfeasible due to high costs and time limitations imposed on the whole testing process. Monitoring without affecting real time behavior of an SUT is expensive and often requires a product-specific hardware-based implementation. In host-based testing, using simulators, emulating target OS, monitoring or instrumentations usually *affects* the real-time behavior of the SUT. If the effects significantly change timed behavior, real-time testing is not optimal and leads to inadequate test results.

Here we propose host-based testing with *simulated time* where the system clock is modelled as a logical clock and time progression is modelled by a tick-action. The calculations and actions within the system are considered to be *instantaneous*. The assumption about instantaneity of actions implies that time progress can never take place if there is still an untimed action enabled, or in other words, the time progress has the least priority in the system and may take place only when the system is *idle*. We refer to the time progress action as `tick` and to the period of time between two `tick`s as a time slice. We assume that the concept of timers is used to express time-dependent behavior. Further, we refer to this time semantics as *simulated time*.

In [2] we proposed host-based testing with *simulated time* for non-distributed applications. There we implemented simulated time on the level of TTCN-3 specifications. Here we provide a framework for host-based testing of *distributed* embedded systems with TTCN-3, where simulated time is implemented at the level of test adapters. The framework allows to use the same test suites for host-based testing with simulated time and for testing with real time in the target environment.

The rest of the paper is organized as follows. Section 2 provides a brief survey on the general structure of a distributed TTCN-3 test system. In Section 3 we provide the time semantics for host-based testing with simulated time. In Section 5, we give an overview of two case studies where simulated time has been used two test two systems: one from telecommunication and one from

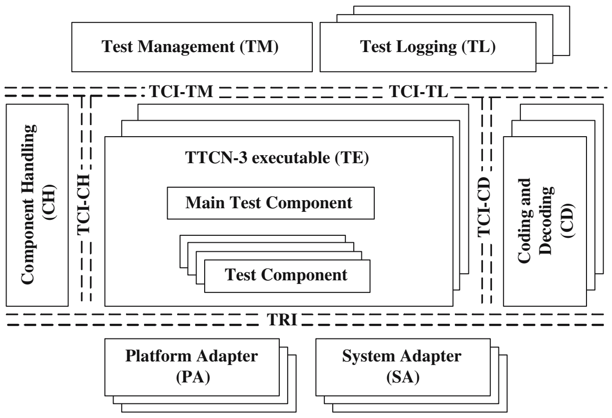


Fig. 1. General structure of a distributed TTCN-3 test system

transportation domain. In Sections 4, we present our testing framework. We conclude in Section 6 with discussing the obtained results.

2 TTCN-3 Test Systems

TTCN-3 is a language for the specification of test suites [8]. The specifications can be generated automatically or developed manually. A specification of a test suite is a TTCN-3 *module* which possibly imports some other modules. Modules are the TTCN-3 building blocks which can be parsed and compiled autonomously. A module consists of two parts: a definition part and a control part. The first one specifies test cases. The second one defines the order in which these test cases should be executed.

A test suite is executed by a TTCN-3 test system whose general structure is defined in [6] and illustrated in Fig. 1. The TTCN-3 executable (TE) entity actually executes or interprets a test suite. A call of a test case can be seen as an invocation of an independent program. Starting a test case leads to creating a *configuration*. A configuration consists of several test components running in parallel and communicating with each other and with an SUT by *message passing* or by *procedure calls*. The first test component created at the starting point of a test case execution is the main test component (MTC). For communication purposes, a test component owns a set of ports. Each port has **in** and **out** directions: infinite FIFO queues are used to represent **in** directions; **out** directions are linked directly to the communication partners.

The concept of timers is used in TTCN-3 to express time-dependent behavior. A timer can be either active or deactivated. An active timer keeps an information about the time left until its expiration. When the time left until the expiration becomes zero, the timer expires and becomes deactivated. The expiration of a timer results in producing a timeout. The timeout is enqueued at the component to which the timer belongs.

The Platform Adapter (PA) implements timers and operations on them. The SUT Adapter (SA) implements communication between a TTCN-3 test system and an SUT. It adapts message- and procedure-based communication of the TTCN-3 test system to the particular execution platform of the SUT. The run-time interface (TRI) allows the TE entity to invoke operations implemented by the PA and the SA.

A test system (TS) can be distributed over several test system instances TS_1, ..., TS_n each of which runs on a separate test device [14]. Each of the TS_i has an instance of the TE entity TE_i equipped with an SA_i, a test logging (TL) entity TL_i, a PA_i and a coder/decoder CD_i running on the node. One of TE's instances is identified to be the main one. It starts executing a TTCN-3 module and calculates final testing results.

The Test Management (TM) entity controls the order of the invocation of modules. Test Logging (TL) logs test events and presents them to the test system user. The Coding and Decoding (CD) entity is responsible for the encoding and decoding of TTCN-3 values into bitstrings suitable to be sent to the SUT. The Component Handling (CH) is responsible for implementing distribution of components, remote communication between them and synchronizing components running on different instances of the test system. Instances of the TE entity interact with the TM, the TLs, the CDs and the CH via the TTCN-3 Test Control Interface (TCI) [7].

3 Simulated Time in TTCN-3

Here we first define the time semantics for testing with simulated time and then proceed with concretizing it for TTCN-3 test systems.

The first choice to be made is between dense and discrete time. It is normally assumed that real-time systems operate in "real", continuous time (though some physicists contest against the statement that the changes of a system state may occur at any real-numbered time point). However, a less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when verification is concerned. Therefore we chose to work with discreet time.

We consider a class of systems where (i) the snapshots are taken with a speed that allows the system to see the important changes in the environment and (ii) external delays are significantly larger compared to the duration of normal computations within the system. If the system satisfies these *requirements*, the duration of computations within the system is *negligible* compared to the external delays and can be safely treated as *instantaneous* or zero-time.

The assumption about instantaneity of actions leads us to the conclusion that time progress can never take place if there is still an untimed action enabled, or in other words, the time-progress transition has the least priority in the system and may take place only when the system is *idle*: there is no transition enabled except for time progress and communication with the environment. It means that some actions are urgent, as a process may block the progress of time and enforce the execution of actions before some delay. This property is usually called *minimal delay* or *maximal progress* [13].

For testing purposes, we focus on *closed* systems (a test system together with an SUT) consisting of multiple components communicating with each other. We say that a *component* is *idle* if and only if it cannot proceed by performing computations, receiving messages or consuming timeouts. We refer to the idleness of a single component as *local idleness.* We say that a *system* is *idle* if and only if all components of the system are idle and there are no messages or timeouts that still can be received during the current time slice. We call such messages and timeouts *pending.* We refer to the idleness of the whole system as *global idleness.*

Definition 1 (Global Idleness). *We say that a closed system is* globally idle *if and only if all components are locally idle and there are no messages and no timeouts pending.*

If the system is globally idle, the *time progresses* by the action `tick` that decreases time left until expiration of active timers by one. If the delay left until the expiration of a timer reaches zero, the timer expires within the current time slice. Timers ready to expire within the same time slice expire in an arbitrary order. Further, we refer to this time semantics as *simulated time.*

The time semantics of TTCN-3 has been intentionally left open to enable the use of TTCN-3 with different time semantics [5]. Nevertheless, the focus has been on using TTCN-3 for real-time testing so not much attention has been paid to implementing other time semantics for TTCN-3 [17]. Existing standard interfaces TCI and TRI provide excellent support for real-time testing but lack operations necessary for implementing simulated time [6,7].

Our goal is to provide a solution for implementing simulated time for a distributed TTCN-3 test system. Developing a test suite for host-based testing costs time and efforts. Therefore, we want the test suites developed for host-based testing with simulated time to be reusable for real-time testing in the target environment. Therefore we provide a solution that can be implemented on the level of adapters, not on the level of TTCN-3 code. In this way, the same TTCN-3 test suites can be used both for host-based testing with simulated time and for real-time testing in the target environment. Although providing such a solution inevitably means extending the TRI and TCI interfaces, we try to keep these extensions minimal.

According to the definition of global idleness, we need to detect situations when all components of the system are locally idle and there are no messages and no timeouts pending. We reformulate this definition in terms of necessary and sufficient conditions for detecting global idleness of the closed system. For the sake of simplicity, we take into account only messages-based communication. Extending the conditions and the solution to procedure-based communication is straightforward.

The closed system consists of a TTCN-3 test system and an SUT. A *distributed* TTCN-3 test system (TS) consists of n test system instances running on different test devices. Further we refer to the test instances i as TS_i. Each of the TS_i consists of a TE_i, SA_i and PA_i. Global idleness requires all the entities to be

$$[\forall i = 1..n\colon (TE_i = idle) \wedge (PA_i = idle) \wedge (SA_i = idle)] \wedge SUT = idle \quad (1)$$
$$\textstyle\sum_{i=1..n} SA_i SentSUT = EnqdSUT \quad (2)$$
$$SentSUT = \textstyle\sum_{i=1..n} EnqdSA_i \quad (3)$$
$$\textstyle\sum_{i=1..n} TCISentTE_i = \sum_{i=1..n} TCIEnqdTE_i \quad (4)$$
$$\forall i = 1..n : TRISentTE_i = TRIEnqdSA_i PA_i \quad (5)$$
$$\forall i = 1..n : TRISentSA_i PA_i = TRIEnqdTE_i \quad (6)$$

Fig. 2. Global Idleness Condition

in the idle state (see condition 1 in Fig. 2). Condition 1 is necessary but not sufficient to decide on global idleness of the closed system. There still can be some message or timeout pending which can activate one of the idle entities.

"No messages or timeouts pending" means that all sent messages and timeouts are already enqueued at the input ports of the receiving components. When testing with TTCN-3, we should ensure that

- There are no messages pending between the SUT and the TS, i.e. all messages sent by the SA (*SASentSUT*) are enqueued by the SUT (*EnqdSUT*) and that all messages sent by the SUT (*SentSUT*) are enqueued by the SA (*EnqdSA*) (see conditions (2-3) in Fig. 2).
- There are no remote messages pending at the TCI interface, i.e. all messages sent by all instances of the TE entity via the TCI interface (*TCISentTE*) are enqueued at the instances of the TE entity (*TCIEnqdTE*) (see condition (4) in Fig. 2).
- There are no messages pending at the TRI interface, i.e. the number of messages sent by every TE$_i$ via the TRI (*TRISentTE*) should be equal to the number (*TRIEnqdSAPA*) of messages enqueued by the corresponding SA$_i$ and PA$_i$, and the number of messages sent by every SA$_i$ and PA$_i$ is the same as the number of messages enqueued by the corresponding TE$_i$ (see conditions (5-6) in Fig. 2).

It is straightforward to show that the system is still active if one of the conditions in Fig. 2 is not satisfied. If all conditions in Fig. 2 are satisfied then all entities of the test system and the SUT are idle and there are no timeouts/messages that still can be delivered and activate them, thus the closed system is globally idle.

Lemma 2. *A closed system is globally idle if and only if the conditions (1-6) in Fig. 2 are satisfied.*

Thus to implement the simulated time for TTCN-3, we need to detect situations where conditions (1-6) in Fig. 2 are satisfied and enforce time progression.

4 Distributed Idleness Detection and Time Progression in TTCN-3

Detecting global idleness of a distributed system is similar to detecting its termination. Our algorithm for simulated time is an extension of the well-known distributed termination detection algorithm of Dijkstra-Safra [3].

In the closed system, each component has a status that is either active or idle. Active components can send messages, idle components are waiting. An idle component can become active only if it gets a message or a timeout. An active component can always become idle.

In the Dijkstra-Safra's algorithm, termination detection was built into the functionality of components. We separate global idleness detection from normal functionality of a component by introducing an idleness handler for each component of the closed system. Since TTCN-3 is mainly used in the context of black-box testing where we can only observe external actions, we consider the SUT as a single component implementing certain interfaces in order to be tested with simulated time. In a distributed TTCN-3 test system, we consider instances of the TS as single components. We require synchronous communication between a component and its idleness handler to guarantee the correctness of the extension of the algorithm.

To decide on global idleness we introduce a *time manager*. The time manager can be provided as a part of SUT or as a part of the test system. The time manager and the idleness handlers are connected into a unidirectional *ring*.

Time Manager. The time manager initializes the global idleness detection, decides on global idleness and progresses time by sending an idleness token along the ring. The token consists of a global flag and a global message counter. The flag can be "IDLE" meaning that there are no active components in the closed system, "ACTIVE" meaning that maybe one of the components is still active, "TICK" meaning time progression and "RESTART" meaning reactivating the system in the next time slice. The counter keeps track of messages exchanged between the components.

The time manager initiates idleness detection by sending an idleness token with the counter equal to 0 and the flag equal to "IDLE" to the next idleness handler along the ring. The time manager detects global idleness if it receives back the idleness token with the counter equal to zero, meaning there are no messages pending between instances of the TS and the SUT, and the flag equal to "IDLE" meaning that all instances of the TSs and the SUT are idle. Otherwise it repeats idleness detection in the same time slice.

If the time manager detects global idleness, it progresses time by sending the token with flag "TICK" along the ring. After all instances of the TS and the SUT are informed about time progress, the manager reactivates the components of the system by sending the token with flag "RESTART" along the ring. That synchronizes all the TS's instances and the SUT on time progression. After the reactivation, the time manager restarts idleness detection in the new time slice.

Idleness handler for TS$_i$. We first consider the idleness handlers for TS$_i$. An idleness handler for the SUT is a simplified version of the TS$_i$ idleness handler. A fragment of the Java class `IdlenessHandlerTS` in Fig. 3 illustrates the behavior of an idleness handler for an instance of the TS$_i$. The class implements interface `Runnable` [1]. The idleness handler communicates with the other handlers via operation `IdlenessTokenSend()` that allows to receive an idleness token from

```
public synchronized void PAIdle(int TRISentPA, int TRIEnqdPA)
{TRISentSAPA+=TRISentPA; TRIEnqdSAPA+=TRIEnqdPA; idlePA=true; notify();}

public synchronized void SAIdle(int TRISentSA, int TRIEnqdSA,
int SASentSUT, int SUTEnqdSA)
{TRISentSAPA+=TRISentSA; TRIEnqdSAPA+=TRIEnqdSA;
SASUTcount+=SASentSUT-SUTEnqdSA; flagSA=true; idleSA=true; notify();}

public synchronized void TEIdle(int TCISentTE, int TCIEnqdTE,
int TRISentTE, int TRIEnqdTE)
{this.TRISentTE+=TRISentTE; this.TRIEnqdTE+=TRIEnqdTE;
 TCITEcount+=TCISentTE-TCIEnqdTE; flagTE=true; idleTE=true; notify();}

public synchronized void PAActivate(){idlePA=false; }
public synchronized void SAActivate(){idleSA=false; }
public synchronized void TEActivate(){idleTE=false; }

public synchronized void run(){ IdlenessToken msg=null;
for(;;) {if (idlePA & idleSA & idleTE &(TRISentTE==TRIEnqdSAPA)&
         (TRIEnqdTE==TRISentSAPA)&(buffer!= null))
         {msg=buffer; buffer=null;
         if (msg.tag==IdlenessToken.IDLE | msg.tag==IdlenessToken.ACTIVE)
         {if (flagTE | flagSA){msg.tag=IdlenessToken.ACTIVE;}
          if (flagTE){msg.count+=TCITEcount; TCITEcount=0; flagTE=false;}
          if (flagSA){msg.count+=SASUTcount; SASUTcount=0; flagSA=false;}
          }
         if (msg.tag==IdlenessToken.TICK)
         {TRISentTE=0; TRIEnqdTE=0; TRISentSAPA=0; TRIEnqdSAPA=0;
          SASUTcount=0; idlePA=false; flagSA=true; flagTE=true;pa.Tick();}
         if (msg.tag==IdlenessToken.RESTART){pa.Restart();}
         NextHandler.IdlenessTokenSend(msg);
         }
.....wait();}
}
```

Fig. 3. Idleness Handler for TS_i

one neighbor and propagate it further to the next one. For this purpose the idleness handler keeps the reference `NextHandler` to the next handler along the ring. The idleness handler decides on local idleness of the TS_i, propagates the idleness token along the ring and triggers time progression at the PA_i. The TS_i is locally idle iff the TE_i, the SA_i and the PA_i are idle and there are no messages/timeouts pending between the TE_i, the SA_i and the PA_i.

Messages exchanged by the TE_i, the SA_i and the PA_i via the TRI interface are internal wrt. the TS_i. Messages exchanged by the TE_i via the TCI interface and the messages exchanged by the SA_i with the SUT are external wrt. the TS_i. To keep information about external and internal messages, idleness handler maintains several local counters. `TRISentTE` and `TRIEnqdTE` keep the number of messages sent and enqueued by the TE_i via the TRI interface. `TRISentSAPA` and `TRIEnqdSAPA` provide analogous information for the SA_i and the PA_i. These four counters are necessary to detect local idleness of the TS_i. `TCITEcount` and `SASUTcount` keep the number of external messages exchanged by the TS_i via the TCI interface and with the SUT.

Two flags (for TE_i and SA_i) kept by the idleness handler show whether `TCITEcount` or/and `SASUTcount` respectively contain the up-to-date information that is not known to the idleness token. Since the PA_i communicates neither with the SUT nor with the other instances of the TS, information on messages exchanged by the PA_i is only important to detect local idleness of the TS_i. Therefore, there is no need for a flag for the PA_i. The idleness handler keeps information on the status of the TE_i, SA_i and PA_i in the variables `idleTE`, `idleSA` and `idlePA` respectively.

Initially, the statuses are *false* meaning TS_i is possibly active. The flags are initiated to *true*, meaning the idleness token does not have the up-to-date information about messages exchanged by the TS_i via TCI and messages exchanged by the TS_i and the SUT. The counters are initially zero.

To detect global idleness, the TE_i, the SA_i and the PA_i should support a number of interfaces. To detect idleness of a TE_i, we use TCI-operation `TEIdle( int TCISentTE, int TCIEnqdTE, int TRISentTe, int TRIEnqdTE)` called by a TE_i at the idleness handler when the TE_i becomes idle. The first two parameters keep track of external messages exchanged via the TCI and the last two parameters capture the same information for internal messages. Calling this operation leads to changing the value of `idleTE` to *true*, updating the local counters `TRISentTE`, `TRIEnqdTE` and `TCITEcount` and setting `flagTE` to *true*.

To detect local idleness of the PA_i, we use operation `PAIdle(int TRISentPA, int TRIEnqdPA)` called by PA at the idleness handler when an active PA_i becomes idle. Two parameters correspond to the number of messages sent and the number of messages received by the PA_i via the TRI respectively. Calling `PAIdle` at the idleness handler leads to changing variable `idlePA` to *true* and updating local counters `TRISentSAPA` and `TRIEnqdSAPA`.

To detect local idleness of an SA_i we use operation `SAIdle(int TRISentSA, int TRIEnqdSA, int SASentSUT, int SUTEnqdSA)` called by SA at the idleness handler when an active SA_i becomes idle. `TRISentSA` and `TRIEnqdSA` denote the numbers of internal messages sent and enqueued by the SA_i. Parameters `SASentSUT` and `SUTEnqdSA` keep analogous information about external messages exchanged between the SA_i and the SUT. Calling `SAIdle()` leads to changing the status of SA_i to *true*, updating the local counters and changing the flag of SA_i to *true*.

The TS_i can be activated by receiving an external message. To detect an activation, we use operations `TEActivate()` called by the CH at the idleness handler when a remote message is being enqueued at the idle TE_i, `SAActicvate()` called by the SA_i at the idleness handler when an idle SA_i gets a message or a timeout, and `PAActivate()` called by the PA_i at the idleness handler when an idle PA_i is activated. Calling these operation leads to updating the idleness status of the corresponding entity to *false*.

Checking local idleness of the TS_i is implemented by the method `run()`. Local idleness of the TS_i is detected iff status variables `idleSA`, `idlePA` and `idleTE` are *true* and all internal messages sent via the TRI interface have been enqueued. This is expressed by local idleness condition at the first if-statement of method `run()`.

If the local idleness conditions are satisfied and the idleness handler is in the possession of the idleness token with flag "IDLE" or "ACTIVATE" , the handler propagates up-to-date information about the external messages exchanged between instances of the TS and the external messages exchanged between the TS and the SUT by updating the idleness token and sending it further along the ring to the time manager.

If `flagTE` is *true* then the number of external messages exchanged by the TS_i via TCI has changed since the last detection round. Thus the idleness handler adds `TCIcount` to the counter of the idleness token. If `flagSA` is *true*, the number of messages exchanged with the SUT has change. Thus the idleness handler updates the token's counter by adding the number of messages sent by the SA_i to the SUT and subtracting the number of messages from the SUT enqueued by the SA_i. If at least one of the local flags is *true* the flag of the token changes to "ACTIVATE", meaning one of TS instances or the SUT may still be active.

If the idleness handler gets an idleness token with flag "TICK", it prepares for detecting idleness in the next time slice by setting all the flags to *true*, setting `idlePA` to *false*, calling operation `Tick()` at the PA_i, and sending the token to the next handler along the ring. Upon `Tick()`, the PA_i look-ups the timers ready to expire in the new time slice. If the idleness handler gets an idleness token with flag "RESTART", it calls operation `Restart()` at the PA_i and propagates the token to the next idleness handler. Upon `Restart()`, the PA_i expires the ready timers. The status of TE_i and of SA_i remains idle until explicit activation because both TE_i and of SA_i may remain idle during a time slice.

The solution proposed in this section strongly resembles the termination detection algorithm of Dijkstra-Safra when detection of messages pending on the level of TCI and communication with an SUT is concerned. The condition detected by an idleness handler in order to decide on local idleness of an instance of the TS, guarantees that all entities of the TS_i are idle and no messages/timeouts are pending on the level of TRI.

Corollary 3. *The solution for simulated time proposed in Section 4 detects global idleness iff the conditions (1-6) in Fig. 2 are satisfied.*

5 Case Studies

In this section we consider two case studies: one from the telecommunication domain and another one from the railway domain.

5.1 2G/3G Mobile Phone Application

Here we consider embedded software for a 2G/3G dual-mode mobile terminal that supports both WCDMA and GSM. GSM (Global System for Mobile Communication) [11] is a mobile network technology that has a global footprint in providing the second generation (2G) mobile services like voice, circuit-switched and packet-switched data and short message service (SMS). WCDMA (Wideband Code Division Multiple Access) [10] is one of the 3G mobile network technologies that meets the performance demands of mobile services like the Mobile Internet, including Web access, audio and video streaming, video and IP calls. WCDMA provides a cost efficient wireless technology for high data throughput.

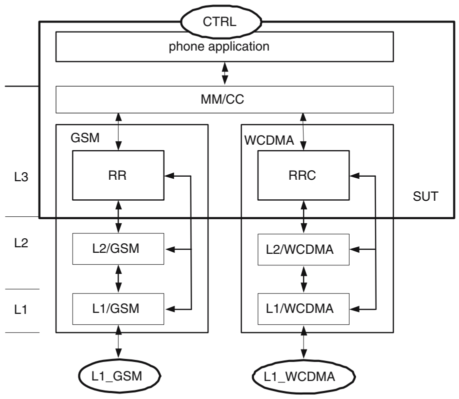


Fig. 4. Structure of embedded software for a 2G/3G dual-mode mobile terminal

Equipping the third generation mobile phones with both WCDMA and GSM technologies enables seamless, practically worldwide mobile service for their end-users [10,11].

The software for a dual-mode WCDMA/GSM phone implements an inter-working mechanism for both technologies (see Fig. 4). In case a phone user first establishes a voice call using WCDMA technology and then moves outside of WCDMA coverage, the software is able to provide voice call service over GSM. Handovers from WCDMA to GSM and vice versa are handled in such a way that no noticeable disturbance happens.

In this case study, an implementation of the third layer have been tested. It combines the functionality of the third layers of WCDMA and GSM respectively, solves WCDMA to GSM (and vice versa) handover issues and a mobile terminal application (see Fig.4). In order to access and to control the implementation of the third layer, the actual system under test (SUT) also includes layers 1 and 2 and a mobile terminal application. The SUT is a *timed* system. For example handover from WCDMA to GSM should be accomplished within certain time bounds. Otherwise handover would become visible to an end-user.

We have tested the SUT on a workstation, so the air interface connecting the mobile terminal to the network is simulated by an Ethernet connection. The network is mimicked by a test system that interacts with the SUT through the simulated interfaces. There are three points available to control and observe the SUT: CTRL can be used to control the phone driver on the top of the SUT, L1_GSM and L1_WCDMA are used to exchange messages between the SUT and the test system.

To test the implementation of the third layer of a 2G/3G mobile terminal, we mimic the GSM/WCDMA air interfaces by Ethernet connections, emulate services of the target operating system and simulate the mobile terminal hardware. The test system simulates behavior of layers 1-3 of the mobile network. The OS services have been emulated. We used host based testing with simulated time to check behavioral time-dependant features of the SUT.

At the time of developing the test system for this case study the TCI had not been defined yet, hence proprietary APIs of the TTCN-3 tool had to be used to implement it. The operations at this API are however comparable to the operations in the TCI relevant for message exchange and indicating idleness. Despite these technical differences to the approach in this paper, it is possible to achieve that the implementation of simulated time is not visible on the level of TTCN-3 code.

Testing the SUT with the developed test system sufficiently increased the possibilities for debugging the SUT. Throughout the test execution the test system the SUT could be suspended and inspected with a debugger. These time intervals could be arbitrarily long, but due to the usage of simulated time no timer expired in such an interval and testing could be continued after such a long interval.

5.2 Railway Interlockings

Railway control systems consist of three layers: infrastructure, logistic, and interlocking. The infrastructure represents a railway yard that basically consists of a collection of linked railway tracks supplied with such features as signals, points, and level crossings. The logistic layer is responsible for the interface with human experts, who give control instructions for the railway yard to guide trains. The interlocking guarantees that the execution of these instructions does not cause train collisions or derailments. Thus it is responsible for the safety of the railway system. If the interlocking considers a command as unsafe, the execution of the command is postponed until the command can be safely executed or discarded. Since the interlocking is the most safety-critical layer of the railway control system, we further concentrate on this layer.

Here we consider interlocking systems based on Vital Processor Interlocking (VPI) that is used nowadays in Australia, some Asian countries, Italy, the Netherlands, Spain and the USA [12]. A VPI is implemented as a machine which executes hardware checks and a program consisting of a large number of guarded assignments. The assignments reflect dependencies between various objects of a specific railway yard like points, signals, level crossings, and delays on electrical devices and ensure the safety of the railway system. An example of a VPI specification can be found in [16]. In the TTMedal project [15], we develop an approach to testing VPI software with TTCN-3. This work is done in cooperation with engineers of ProRail who take care of capacity, reliability and safety on Dutch railways. They have formulated general safety requirements for VPIs. We use these requirements to develop a TTCN-3 test system for VPIs.

The VPI program has several read-only input variables, auxiliary variables used for computations and several writable variables that correspond to the outputs of the program. The program specifies a *control cycle* that is repeated with a fixed period by the hardware. The control cycle consists of two phases: an active phase and an idle phase. The active phase starts with reading new values for input variables. The infrastructure and the logistic layer determine the values of the input variables. After the values are latched by the program, it uses them to compute new values for internal variables and finally decides on new outputs. The values of the output variables are transmitted to the infrastructure and to the logistic, where they are used to manage signals, points, level crossings and trains. Here we assume that the infrastructure always follows the commands of the interlocking. The rest of the control cycle the system stays idle.

The duration of the control cycle is fixed. Delays are used to ensure the safety of the system. A lot of safety requirements to VPIs are timed. They describe dependencies between infrastructure objects in a period of time. The objects of the infrastructure are represented in the VPI program by input and output variables. Thus the requirements defined in terms of infrastructure objects can be easily reformulated in terms of input and output variables of the VPI program. Hence VPIs are *time-critical systems*.

We have tested VPI software without access to the target VPI hardware. To execute VPI program, we simulated the VPI hardware/software interfaces and the VPI program itself. The duration of the control cycle of VPI program is fixed. The VPI program sees only snapshots of the environment at the beginning of each control cycle, meaning the program observes the environment as a *discrete* system. Timing constraints in a VPI program are expressed by time delays that are much longer than the duration of the control cycle. That leads us to the conclusion that we may safely use simulated time to test VPI software.

Based on the concept of the simulated time we have developed a test system for executing the test cases. The experiments showed that our approach to host-based testing with simulated time allows to detect violations of safety requirements in interlocking software.

6 Conclusion

In this paper we proposed a simulated-time framework for host-based testing of distributed systems with TTCN-3. Simulated time has been successfully applied to testing and verification of systems where delays are significantly larger than the duration of normal events in the system (see e.g. [2]). Our framework contributes to the repeatability of test results when testing embedded software and also solves some time-related debugging problems typical for distributed embedded systems. It allows to use the same test suites for simulated time and for real time testing. We also provide two case studies where host-based testing with simulated time has been applied to two systems: one from telecommunication domain and one from transportation domain.

References

1. K. Arnold, J. Gosling, and D. Holmes. *Java(TM) Programming Language*. Java Series. Addison Wesley, 2005.
2. S. Blom, N. Ioustinova, J. van de Pol, A. Rennoch, and N. Sidorova. Simulated time for testing railway interlockings with TTCN-3. In C. Weise, editor, *FATES'05*, LNCS to appear, pages 10–25. Springer, 2005.

3. E. W. Dijkstra. Shmuel Safra's version of termination detection. EWD998-0, Univ. Texas, Austin, 1987.
4. ETSI ES 201 873-1 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.
5. ETSI ES 201 873-4 V3.1.1 (2005-06). MTS; TTCN-3; Part 4: TTCN-3 Operational Semantics.
6. ETSI ES 201 873-5 V1.1.1 (2005-06). MTS; TTCN-3; Part 5: TTCN-3 Runtime Interface (TRI).
7. ETSI ES 201 873-6 V1.1.1 (2005-06). MTS; TTCN-3; Part 6: TTCN-3 Control Interface (TCI).
8. J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.
9. S. Hendrata. Standardisiertes Testen mit TTCN-3: Erhöhung der Zuverlässigkeit von Software-Systemen im Fahrzeug. *Hanser Automotive: Electronics+Systems*, (9-10):64–65, 2004.
10. H. Holma and A. Toskala. *WCDMA for UMTS- Radio Access for Third Generation Mobile Communications*. John Wiley and Sons, 2004.
11. H. Kaaranen, A. Ahtiainen, L. Laitinen, S. Naghian, and V. Niemi. *UMTS Networks: Architecture, Mobility and Services*. John Wiley and Sons, 2005.
12. U. Marscheck. Elektronische Stellwerke-internationale Überblick. *SIGNAL+DRAHT*, 89, 1997.
13. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548. Springer-Verlag, 1992.
14. I. Schieferdecker and T. Vassiliou-Gioles. Realizing Distributed TTCN-3 Test Systems with TCI. In D. Hogrefe and A. Wiles, editors, *TestCom*, volume 2644 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2003.
15. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.
16. F. J. van Dijk, W. J. Fokkink, G. P. Kolk, P. H. J. van de Ven, and S. F. M. van Vlijmen. Euris, a specification method for distributed interlockings. In W. Ehrenberger, editor, *Proc. 17th Conference on Computer Safety, Reliability and Security - SAFECOMP'98, Heidelberg*, volume 1516 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 1998.
17. C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, 2005.

Chase of Recursive Queries[⋆]

Nieves R. Brisaboa, Antonio Fariña, Miguel R. Luaces, and José R. Paramá

Laboratorio de Bases de Datos. Dept. de Computación. Univ. Da Coruña. Campus de Elviña s/n, 15071 A Coruña. Spain
Tf. +34 981167000. Fax. +34 981167160
`brisaboa@udc.es, fari@udc.es, luaces@udc.es, parama@udc.es`

Abstract. In this work, we present a semantic query optimization technique to improve the efficiency of the evaluation of a subset of SQL:1999 recursive queries.

Using datalog notation, we can state our main contribution as an algorithm that builds a program P' equivalent to a given program P, when both are applied over a database d satisfying a set of functional dependencies. The input program P is a linear recursive datalog program. The new program P' has less different variables and, sometimes, less atoms in rules, thus it is cheaper to evaluate. Using CORAL, P' is empirically shown to be more efficient than the original program.

Keywords: Recursive queries, Semantic Query Optimization.

1 Introduction

Although research in recursive queries has been carried out for the last three decades, the appearance of the SQL:1999 reaffirmed the necessity of research in this area, given that SQL:1999 includes queries with linear recursion. Previous standards of SQL did not include recursion, thus research in recursive query optimization might be able to provide the suitable algorithms, to be included in the query optimizers of the database management systems to speed up the execution of recursive queries.

Although our results are focused on the development of algorithms to be included in commercial object-relational database management systems, we use datalog syntax since it is easier to manipulate. The class of datalog programs considered in this paper can be translated into SQL:1999 queries straightforwardly.

Example 1. Let us suppose that we have a database with the following relations: $stretch(Number, From, To)$, meaning that a $flight$ with code $Number$ has a stretch from the airport $From$ to the airport To; $assig(Number, Employee)$ meaning that a certain $Employee$ is assigned to the flight $Number$.

Let us consider the query that computes the relation *conn(Number, From, To, First_Officer, Purser)*, meaning that flight *Number* connects the airport *From*

⋆ This work is partially supported by Grants TIC2003-06593 and TIN2006-15071-C03-03 of Ministerio de Educación y Ciencia and PGIDIT05SIN10502PR of Xunta de Galicia.

with the airport *To*, possibly using several stopovers. In addition, *conn* has the information of the *First_Officer* and the *Purser* of the flight. *conn* is the transitive closure of each flight code with some additional information about the flight crew.

$P : r_0 : conn(N, F, T, O, P) : -stretch(N, F, T), assig(N, O), assig(N, P)$

$r_1 : conn(N, F, T, O, P) : -stretch(N, F, Z), assig(N, O), assig(N, P), conn(N, Z, T, O, P)$ □

P is *linear*, which means it has only one recursive atom in the body of the recursive rule. Linear programs include most real life recursive queries, then much research effort has been devoted to this class of programs (see [17] for a survey of optimization techniques for this class of programs).

In addition, *P* is a *single recursive program* (sirup). This implies that it has only one recursive rule. Sirups is another class of programs considered by several researchers (see [6,8,1] for example).

The combination of both features (like in our example), is called *linear single recursive programs* (lsirup). These programs are the programs considered in this work, and they were also studied by several works (see [18,10,8] for example).

An interesting approach to optimize a recursive query is to see if we can transform the query, somehow, to make the recursion "smaller" and "cheaper" to evaluate. One possibility to do that is the *semantic query optimization* that uses integrity constraints associated with databases in order to improve the efficiency of the query evaluation [5]. In our case, we use functional dependencies (fds) to optimize linear recursive datalog programs.

In this paper we provide an algorithm to optimize single recursive datalog programs under fds. *The chase of datalog programs* ($Chase_F(P)$) is a modification of an algorithm introduced by Lakshmanan and Hernández [8]. It obtains from a linear single recursive program P a program P', equivalent to P when both are evaluated over databases that satisfy a set of functional dependencies F.

The chase of a datalog program P obtains an equivalent program P', where the recursive rule may has smaller number of different variables and, less number of atoms. That is, it obtains a program where the variables (in the recursive rule) are equated among them due to the effect of fds. Moreover, those equalizations of variables, sometimes reveal that an unbounded datalog program P is in fact (due to the fds) a bounded datalog program.

Example 2. Considering the program of Example 1, let us suppose that our company decides that the first officer should act as purser as well. This imposes a constraint specifying that one flight code only has one employee assigned, that is $assig : \{1\} \to \{2\}$. The set of functional dependencies F indicates that the values of the first argument determine the values in the second position in the set of facts over the predicate *assig*. For example, the atoms $assiged(ib405, peter)$ and $assig(ib405, mary)$ violate the fd $assig : \{1\} \to \{2\}$.

Using the algorithm of the *chase of datalog programs* shown in this paper, it is possible to compute the new program $Chase_F(P)$ that we call, for short, P':

$s_0 : conn(N, F, T, O, O) : -stretch(N, F, T), assig(N, O)$

$s_1 : conn(N, F, T, O, O) : -stretch(N, F, Z), assig(N, O), conn(N, Z, T, O, O)$

There are two combined beneficial effects. First, there are 6 different variables in r_0, but only 5 in s_0. Second, the number of predicates in the bodies of the rules also decreases: 3 and 4 in r_0 and r_1, respectively, but only 2 and 3 respectively in s_0 and s_1.

The reader can observe that in this case P' is very similar to the original program, the only difference is that some variables have been equalized. This equalization comes from the fact that the database fulfills the functional dependency $assig : \{1\} \rightarrow \{2\}$. Therefore, if during the evaluation of the query, an atom, let say $assig(N, O)$, is mapped to the ground fact $assig(IB405, peter)$, then another atom $assig(N, P)$ should be mapped to the same ground fact. Observe that $assig(N, O)$ and $assig(N, P)$ have the same variable in the first position, thus by $assig : \{1\} \rightarrow \{2\}$, O and P are necessarily mapped to the same ground term. $\square$

2 Related Work

Several strategies have been proposed to tackle the process of recursive queries. Bancilhon, Maier, Sagiv and Ullman [3] introduced a technique called *magic-sets* for rewriting linear queries taking advantage of binds between nodes in rule goal trees. There is a family of papers that try to reduce the work done by a query execution by remembering previous executions of queries that can have intermediate values useful for the current execution. These techniques are called *memoing* (see [4] for a survey).

Since in practice the great majority of recursions are linear, this class of queries has attracted much work. From a logic programming perspective, several works deal with the placement of the recursive atom in the body of the rules. "Right-linear" and "left-linear" give better performance in linear recursion than magic-sets [11].

The chase as a tool to optimize queries in the framework of datalog is also used by several researchers. Lakshmanan and Hernández [8] introduced an algorithm called *the chase of datalog programs* which is based in the use of the chase [9,2]. Recent data models have also adopted the chase to optimize queries. In the semistructured model, it has been also used as a rewriting technique for optimizing queries [7]. Popa et. al. [14] used it to optimize queries in the framework of the object/relational model.

3 Definitions

3.1 Basic Concepts

We assume the notation and definitions of [16] and then we only define the non-standard concepts. We use $EDB(P)$ to refer to the set of EDB predicate names in a datalog program P. We denote variables in datalog programs by capital letters, while we use lower case letters to denote predicate names. For simplicity, we do not allow constants in the programs. Let a_i be a atom, $a_i[n]$ is the term in the n^{th} position of a_i.

We say that a program P *defines* a predicate p, if p is the only IDB predicate name in the program. A *single recursive rule program (sirup)* [6] is a program that consists of exactly one recursive rule and several non-recursive rules and the program defines a predicate p. A *2-sirup* is a sirup that contains only one non-recursive rule (and one recursive rule).

A rule is *linear* if there is at most one IDB atom in its body. A *linear sirup (lsirup)* [18] is a sirup such that its rules are linear. A *2-lsirup* [18] is a 2-sirup such that its rules are linear. That is, a $2 - lsirup$ is a program defining a predicate p with one non-recursive rule and one recursive rule, which has only one IDB atom in its body.

Example 3. The program of Example 1 is a *2-lsirup*. □

For the sake of simplicity, many of the definitions will apply to $2 - lsirups$ although the algorithm presented in this paper is valid for $lsirups$ as well. In addition, from now on, we denote with r_0 the non-recursive rule in a $2 - lsirup$, and r_1 to denote the recursive rule.

Let P be a program, let r be a rule and let d be a database. Then, $P(d)$ represents the output of P when its input is d and $r(d)$ represents the output of r when its input is d. Let F be a set of functional dependencies, $SAT(F)$ represents the set of databases that satisfies F.

Let P_1 and P_2 be programs. $P_1 \subseteq_{SAT(F)} P_2$, if $P_1(d) \subseteq P_2(d)$ for all EDBs d in $SAT(F)$. $P_1 \equiv_{SAT(F)} P_2$, if $P_1 \subseteq_{SAT(F)} P_2$ and $P_2 \subseteq_{SAT(F)} P_1$.

A substitution is a finite set of pairs of the form X_i/t_i where X_i is a variable and t_i is a term, which is either a variable or a constant, and X_i and t_i are different. The result of applying a substitution, say θ, to an atom A, denoted by $\theta(A)$, is the atom A with each occurrence of X replaced by t for every pair X/t in θ. For example, consider $\theta = \{X/a, Y/b\}$ and the atom $p(X, Y)$, then $\theta(p(X, Y))$ is $p(a, b)$. A substitution θ can be applied to a set of atoms, to a rule or to a tree to get another set of atoms, rule or tree with each occurrence X replaced by t for every X/t in θ.

4 Expansion Trees

An *expansion tree* is a description for the derivation of a set of (intensional) facts by the application of one or more rules to an extensional database. First, we start with the definition of a tree generated *by only one rule*. Let r be the rule $q :- q_1, q_2, \ldots, q_k$. Then, a tree T can be built from r as follows: the node at the root of T is q, and q has k children, q_i, $1 \leq i \leq k$. We denote this tree as $tree(r)$.

Example 4. Using the program of Example 1, Figure 1 shows $tree(r_1)$.

In order to be a complete expansion tree, that is, an expansion tree describing the complete execution of a program, the tree should start with the application of a non-recursive rule.

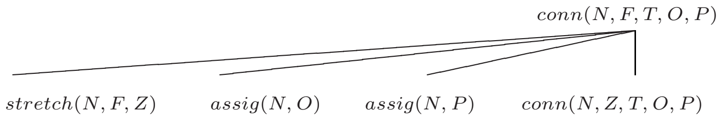


Fig. 1. $tree(r_1)$ □

Let S and T be two trees. Then, S and T are *isomorphic*, if there are two substitutions θ and α such that $S = \theta(T)$ and $T = \alpha(S)$.

The variables appearing in the root of a tree T are called *the distinguished variables of* T. All other variables appearing in atoms of T that are different from the distinguished variables of T are called *non-distinguished variables of* T.

Let S and T be two trees, where h_t denotes the head (the node at the root) of T. Assume that exactly one of the leaves of S is an IDB atom[1], denoted by p_s. The *expansion (composition)* of S with T, denoted by $S \circ T$ is defined if there is a substitution θ, from the variables in h_t to those in p_s, such that $\theta(h_t) = p_s$. Then, $S \circ T$ is obtained as follows: build a new tree, isomorphic to T, say T', such that T' and T have the same distinguished variables, but all the non-distinguished variables of T' are different from all of those in S. Then, substitute the atom p_s in the last level of S by the tree $\theta(T')$.

From now on, we use the expression $tree(r_j \circ r_i)$ to denote $tree(r_j) \circ tree(r_i)$ and, $tree(r_j^k)$ to denote the composition of $tree(r_j)$ with itself, k times. Given a $2{-}lsirup$ $P = \{r_0, r_1\}$, T_i denotes the tree $tree(r_1^i \circ r_0)$. T_i is a *complete* expansion tree since it describes the derivation of a set of IDB facts from an extensional database. Obviously, since P is a recursive program, we may construct infinitely many trees considering successive applications of the recursive rule. We call $trees(P)$ the infinite ordered collection of trees $\{T_0, T_1, T_2, T_3, \ldots\}$.

Example 5. Using the program of Example 1, Figure 2 shows T_2.

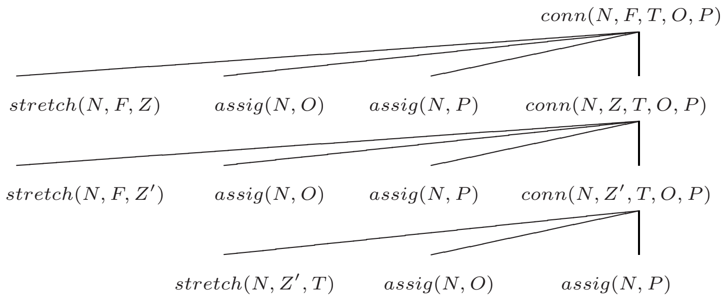


Fig. 2. The tree T_2 using the program of Example 1

[1] That is, the case of the trees generated by *lsirups*, since in the recursive rule of such programs, there is only one IDB predicate.

From now on, we shall consider only complete expansion trees. For the sake of simplicity we shall refer to expansion trees simply as trees.

Let T be a tree. *The level of an atom in* T is defined as follows: the root of T is at level 0, the level of an atom n of T is one plus the level of its parent. *Level* j of T is the set of atoms of T with level j. The last level of a tree T_i is the level $i+1$. We say that two levels i and k (in a tree T_j) are *separated by* w *levels* if $|i-k| = w$ and $i \leq j+1$ and $k \leq j+1$.

4.1 TopMost and Frontier of a Tree

TopMost and frontier of a tree are two rules that can be extracted from any tree. Let T be a tree: *the frontier of* T (also known as *resultant*), denoted by $frontier(T)$, is the rule $h \;:-\; l_1, \ldots, l_n$, where h is the root of T and $l_1, \ldots, l_n$ is the set of leaves of T; *the topMost of* T, denoted by $topMost(T)$, returns the rule $h \;:-\; c_1, \ldots, c_n$, where h is the root of T and $c_1, \ldots, c_n$ is the set of atoms that are the children of the root.

Example 6. Using the tree of Figure 2:

$frontier(T_2)$: $conn(N, F, T, O, P)$:- $stretch(N, F, Z), assig(N, O), assig(N, P)$
$stretch(N, F, Z'), assig(N, O), assig(N, P)$ $stretch(N, Z', T), assig(N, O), assig(N, P)$

$topMost(T_2)$:$conn(N, F, T, O, P)$:- $stretch(N, F, Z), assig(N, O), assig(N, P), conn(N, Z, T, O, P)$
□

Observe that the frontier of a tree in $trees(P)$ is a non-recursive rule, while the topMost may be a recursive one. Let P be a $2-lsirup$, d an extensional database and T a tree in $trees(P)$. $T(d)$ represents the result of applying the rules used to build T to the input extensional database d in the order specified by T. That is, $T(d)$ can be seen or computed as $frontier(T)(d)$[2]. Let T and Q be two trees, $T \equiv_{SAT(F)} Q$ means that $T(d) = Q(d)$ for any extensional database d in $SAT(F)$.

5 Chase of a Tree

The chase [9,2] is a general technique that is defined as a nondeterministic procedure based on the successive application of dependencies (or generalized dependencies) to a set of tuples (that can be generalized to atoms).

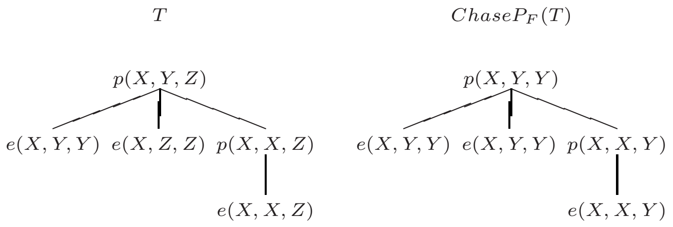
Let us consider the following: Let F be a set of fds defined over $EDB(P)$, for some program P. Let T_i be a tree in $trees(P)$. Let $f = p : \{n\} \rightarrow \{m\}$ be a fd in F. Let q_1 and q_2 be two atoms in the leaves of T_i such that the predicate name of q_1 and q_2 is p, $q_1[n] = q_2[n]$ and $q_1[m] \neq q_2[m]$. Note that n can be a set of positions. In addition, observe that $q_1[m]$ and $q_2[m]$ are variables since we are assuming that programs do not contain constants. An *application of the fd* f *to* T_i is the uniform replacement in T_i of $q_1[m]$ by $q_2[m]$ or vice versa. By uniform, we mean that $q_1[m]$ is replaced by $q_2[m]$ (or vice versa) all along the tree.

[2] Observe that if T is in $trees(P)$, the body of the frontier of a tree only contains EDB atoms.

5.1 Partial Chase of a Tree

The *partial chase of* T *with respect to* F, denoted by $ChaseP_F(T)$, is obtained by applying every fd in F to the atoms that are the leaves of T except the atoms in the last level, until no more changes can be made. Observe that although the atoms which are taken into account for the computation of the chase do not include the atoms in the last level, if a variable is renamed by the chase, such change is applied all along the tree.

Example 7. Let F be $e : \{1\} \rightarrow \{2\}$:



□

Lemma 1. *Let* P *be a* $2 - lsirup$*, let* F *be a set of fds over* $EDB(P)$*. There is a tree* T_k *such that for any tree* T_l *with* $l > k$*,* $topMost(ChaseP_F(T_l))$ *is isomorphic to* $topMost(ChaseP_F(T_k))$*.*

Proof. Note that for all i, x such that $i > x > 0$, T_i includes all the atoms of T_{i-x} that are considered by the partial chase, then any equalization in $ChaseP_F(T_{i-x})$ is also included in $ChaseP_F(T_i)$. Therefore, there is a limit in the equalizations produced in the topMost given that all trees in $trees(P)$ with more than two levels have the same topMost, and this topMost has a finite number of variables. □

The inclusion of the last level of the tree introduces equalizations that are more difficult to model. Lemma 1 would not be true is such a case. We explored this possibility in [13].

Lemma 2. *Let* P *be a 2-lsirup. Let* T_i *be a tree in* $trees(P)$*, and let* F *be a set of fds over* $EDB(P)$*. Then,* $T_i \equiv_{SAT(F)} ChaseP_F(T_i)$*.*

The proof can be done readily, we do not include it by lack of space.

6 The Chase of Datalog Programs

In [12], there is a method to find T_k, the tree such that for any tree T_l with $l > k$, $topMost(ChaseP_F(T_l))$ is isomorphic to $topMost(ChaseP_F(T_k))$. The basic idea is sketched below.

Let us consider a tree T_i in $trees(P)$ and two atoms $q_{j,k}$ and $q_{l,m}$ of T_i. $q_{j,k}$ is in the k^{th} position (numbering the atoms of its level from left to right) of level j. Similarly, $q_{l,m}$ is in the m^{th} position of level l. Now, let us suppose that the variables $q_{j,k}[n]$ and $q_{l,m}[n]$ are equalized by the $ChaseP_F(T_i)$. Then in T_{i+1}, $q_{j+1,k}[n]$ is equalized to $q_{l+1,m}[n]$ during the $ChaseP_F(T_{i+1})$. When we find a tree, say T_p, that for any equalization during its partial chase, say $q_{j,k}[n]$ equalized to $q_{l,m}[n]$, in $ChaseP_F(T_{p-r})$, where $1 \leq r \leq 2\mathcal{N}$, $q_{j-r,k}[n]$ is equalized to $q_{l-r,m}[n]$, then we have found T_k. Now the question is to find $\mathcal{N}$.

6.1 The Computation of $\mathcal{N}$

To compute $\mathcal{N}$ we need to provide a previous tool.

Let P be a $2 - lsirup$. Let p_h and p_b be the IDB atoms in the head and in the body of r_1, the recursive rule of P. The *Expansion Graph of a program P* is generated with this algorithm.

1. If the arity of the IDB predicate in P is k, add k nodes named $1, \ldots, k$.
2. Add one arc from the node n to the node m, if a variable X is placed in the position n of p_h, and X is placed in the position m of p_b.
3. Add one arc from the node n without target node, if a variable X is placed in the position n of p_h, and it does not appear in p_b.
4. Add one arc without source node and target node m, if a variable X is placed in the position m of p_b and it does not appear in p_h.

Example 8. Let $P = \{r_0, r_1\}$ where r_1 contains the following IDB atoms:

$p(A, B, C, D, E, F, G, H, I, J, K, L, M) : - \ldots p(B, A, E, C, D, F, W, G, G, X, J, L, L)$

In Figure 3, we can see the expansion graph of P. □

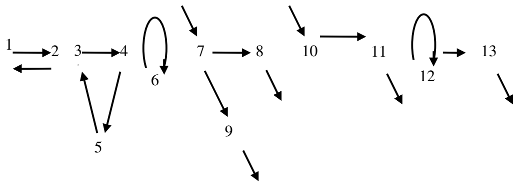


Fig. 3. Expansion Graph of P

Let G be the expansion graph of a *lsirup* P, then $\mathcal{N}$ is the least common multiple of the number of nodes in each path in G.

Example 9. The graph in Figure 3 has $\mathcal{N}$= 6 (6= least common multiplier of 2, 3, 1, 2, 2, 2).

6.2 The Algorithm

Assuming that we have found T_k, the *chase* of a $2 - lsirup$ P w.r.t. a set of fds F is obtained with the algorithm shown in Figure 4.

Our algorithm is based in Lakshmanan and Hernández' algorithm [8], but our algorithm obtains better results. This improvement comes from the terminating condition. Their algorithm stops when it finds two consecutive trees with the

```
**Chase** (P: a $2 - lsirup$, F: a set of functional dependencies over EDB(P))
**For** any tree $T_i$ with $i < k$ such that
            $topMost(ChaseP_F(T_k))$ is not isomorphic to $topMost(ChaseP_F(T_i))$
    **Output** $frontier(ChaseP_F(T_i))$;
**Output** $topMost(ChaseP_F(T_k))$;
```

Fig. 4. Chase of a datalog program

same topMost after the partial chase. However, it is clear that after two consecutive trees with the same topMost after the partial chase, there would be bigger trees that may introduce more equalizations in the topMost after the partial chase. Our algorithm stops in a tree T_k such that it is sure that any bigger tree than T_k would not introduce any other equalization in the topMost after the partial chase. Hence, our algorithm introduces more equalities in the recursive rule of the new program.

Theorem 1. *Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. The $Chase_F(P)$ is equivalent to P when both are evaluated over databases in $SAT(F)$.*

Proof. In order to prove this theorem, we have to prove that $P' \subseteq_{SAT(F)} P$ and $P \subseteq_{SAT(F)} P'$.

We start with the proof of $P' \subseteq_{SAT(F)} P$. Let NR be the set of non-recursive rules in P', and let R be the set of recursive rules in P'. Let s be a rule in NR, by the algorithm in Figure 4, $s = frontier(ChaseP_F(T_i))$ for some tree T_i in $trees(P)$. Then, by Lemma 2, $\{s\} \subseteq_{SAT(F)} r_1^i \circ r_0$, and thus $\{s\} \subseteq_{SAT(F)} P$. Let r be a rule in R. Therefore, $r = \theta(topMost(T_j))$, where θ is the substitution defined by $ChaseP_F(T_j)$ and T_j is a tree in $trees(P)$. Since r is a recursive rule and P only has one recursive rule, then $j > 0$ and $topMost(T_j) = r_1$. Therefore, by construction, using the algorithm of the Chase of datalog programs, $r = \theta(r_1)$, and hence $r \subseteq r_1$. Thus, we have shown that for any rule r in P', $\{r\} \subseteq_{SAT(F)} P$.

Now, we tackle the other direction of the proof; $P \subseteq_{SAT(F)} P'$. We are going to prove that any fact q produced by P, when P is applied to an extensional database d in $SAT(F)$, is also produced by P', when P' is applied to d.

Let us assume that q is in $T_i(d)$, that is, q is obtained after the application to d of r_0 once, and i times r_1. We are going to prove that q is in $P'(d)$. We prove it by induction on the number of levels of the tree T_i (in $trees(P)$) that if q is in $T_i(d)$, then q is in $P'(d)$.

Basis i=0, q is in $T_0(d)$. Then q is in $P'(d)$. Observe that P' always contains $frontier(ChaseP_F(T_0))$, since the $topMost_F(ChaseP_F(T_0))$ cannot be isomorphic to the topMost of the partial chase of any other tree in $trees(P)$ since r_0 is the only non-recursive rule of P. Then, necessarily the algorithm always outputs $frontier(ChaseP_F(T_0))$. Therefore, by Lemma 2, if q is in $T_0(d)$ then q is in $ChaseP_F(T_0)(d)$, and then q is in P'.

Induction hypothesis (IH): Let assume that $\forall q \in T_i(d)$, $1 \leq i < k$, $q \in P'(d)$.

Induction step: i=j=k. q is in $T_j(d)$. Assume q is not in any $T_m(d)$, $0 \leq m < j$, otherwise the proof follows by the IH. Thus, there is a substitution θ such that

q is $\theta(p_j)$, where p_j is the root of T_j and where $\theta(t_l) \in d$ for all the leaves t_l of T_j. Therefore, q is also in $\{frontier(T_j)\}(d)$.

We have two cases:

Case 1: $frontier(ChaseP_F(T_j))$ is one of the non-recursive rules of P'. Then by Lemma 2 q is in $P'(d)$.
Case 2: $frontier(ChaseP_F(T_j))$ is not one of the non-recursive rules of P'. Thus, by Lemma 2 $q \in \{ChaseP_F(T_j)\}(d)$ (assuming that $d \in SAT(F)$). Let γ be the substitution defined by the $ChaseP_F(T_j)$.

Let T_{sub} be the subtree of T_j that is rooted in the node at the first level of T_j that is, the recursive atom at that level. T_{sub} has one level less than T_j, therefore T_{sub} is isomorphic to T_{j-1}. Observe that this follows from the fact that in P there is only one recursive rule and one non-recursive rule.

Let q_{sub} be an atom in $T_{sub}(d)$, since T_{sub} is isomorphic to T_{j-1}, then q_{sub} is in $T_{j-1}(d)$. Hence, by IH $q_{sub} \in P'(d)$. It is easy to see that $q \in \{topMost(ChaseP_F(T_j))\}(d \bigcup q_{sub})$, that is, $q \in \{\gamma(r_1)\}(d \bigcup q_{sub})$.

By construction of P', in P' there is a rule $s_t = \theta(r_1)$, where θ is the substitution defined by the partial chase of T_k. By Lemma 1 and construction of the algorithm in Figure 4, $\gamma \equiv \theta$, otherwise $frontier(ChaseP_F(T_j))$ would be one of the non-recursive rules in P'.

We have already shown that q_{sub} is a fact in $P'(d)$. Therefore, since $s_t(d \cup q_{sub})$ obtains q, thus we have proven that if q is in $T_j(d)$ then q is also in $P'(d)$. $\square$

7 Empirical Results

We used CORAL [15], a deductive database, in order to compare the running time of the original program versus the optimized one. We ran 20 different programs over databases of different sizes. The datalog programs were synthetic queries developed by us. CORAL is an experimental system, this is a limitation, since the maximum database size is restricted, because CORAL loads all tuples in memory and then, if the database has a certain size, an overflow arises.

The computation time needed to obtain the optimized datalog program, using a program in C++ in a 200-MHz Pentium II with 48 Mbytes of RAM, takes on average 0.17 seconds with a varianza of 0.019. This is a insignificant amount of time when the database to which it is applied the query has a normal size.

The average running time of the optimized program is the 43.95% of that of the original one with a varianza of 0.10. The confidence interval of this improvement, with a confidence level of 95%, is $[28.82\%, 59.10\%]$. That is, the optimized program is between 1.7 and 3.5 times faster than the original one.

8 Conclusions and Future Work

Given a *lsirup* P and a set of fds F, we provide an algorithm that obtains a new program P' equivalent to P when both are applied over databases in $SAT(F)$. In addition, we have shown that the algorithm is correct. The algorithm shown in this paper is based in the partial chase, that does not consider atoms in the last level of the chased trees. As a future work, it would very interesting the inclusion of the last level in the computation of the chase. In that case, the chase would introduce more equalities in the alternative optimized program.

References

1. S. Abiteboul. Boundedness is undecidable for datalog programs with a single recursive rule. *Information Processing Letters*, (32):282–287, 1989.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–16, Cambridge, Massachusetts, 24–26 Mar. 1986.
4. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Washington, DC*, pages 16–52. ACM, May 1986.
5. U. S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kauffmann Publishers, 1988.
6. S. S. Cosmadakis and P. C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 280–293, 1986.
7. A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, pages 225–241, 2003.
8. L. V. S. Lakshmanan and H. J. Hernández. Structural query optimization - a uniform framework for semantic query optimization in deductive databases. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems*, pages 102–114, 1991.
9. D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.
10. J. Naughton. Data independent recursion in deductive databases. In *Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 267–279, 1986.
11. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. *ACM SIGMOD RECORD*, 18(2), June 1989. Also published in/as: 19 ACM SIGMOD Conf. on the Management of Data, (Portland OR), May.-Jun.1989.
12. J. R. Paramá. *Chase of Datalog Programs and its Application to Solve the Functional Dependencies Implication Problem.* PhD thesis, Universidade Da Coruña, Departmento de Computación, A Coruña, España, 2001.
13. J. R. Paramá, N. R. Brisaboa, M. R. Penabad, and A. S. Places. A semantic approach to optimize linear datalog programs. *Acta Informatica.* In press.
14. L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far. In *SIGMOD*, pages 273–284, 2000.
15. R. Ramakrishnan, P. Bothner, D. Srivastava, and S. Sudarshan. Coral: A databases programming language. Technical Report TR-CS-90-14, Kansas State University, Department of Computing and Information Sciences, 1990.

16. J. D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.
17. J. D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.
18. M. Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proc. Seventh ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 341–351, 1988.

Improving Semistatic Compression Via Pair-Based Coding[⋆]

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[2], and José R. Paramá[1]

[1] Database Lab., Univ. da Coruña, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain
{brisaboa,fari,parama}@udc.es

[2] Dept. of Computer Science, Univ. de Chile, Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. In the last years, new semistatic word-based byte-oriented compressors, such as Plain and Tagged Huffman and the Dense Codes, have been used to improve the efficiency of text retrieval systems, while reducing the compressed collections to 30–35% of their original size.

In this paper, we present a new semistatic compressor, called *Pair-Based End-Tagged Dense Code (PETDC)*. PETDC compresses English texts to 27–28%, overcoming the optimal 0-order prefix-free semistatic compressor (Plain Huffman) in more than 3 percentage points. Moreover, PETDC permits also random decompression, and direct searches using fast Boyer-Moore algorithms.

PETDC builds a vocabulary with both words and pairs of words. The basic idea in which PETDC is based is that, since each symbol in the vocabulary is given a codeword, compression is improved by replacing two words of the source text by a unique codeword.

1 Introduction

The grown in size and number of text databases during the last decade makes compression even more attractive. Compression exploits the redundancies in the text to represent it using less space [1]. It is well-known that CPU speed has been growing faster than disk and network bandwidth during the last years. Therefore, reducing the size of the text, even at the expense of some CPU time, is useful because it reduces the I/O time needed to load it from disk or to transmit it thought a network.

Joining compression and block addressing indexes [12] improves the efficiency of that retrieval structures. Those indexes are smaller than standard inverted indexes because their entries do not point to exact word positions. Instead of that, entries point to those blocks where a word appears. This has a main drawback: some searches (i.e. phrase search) need traversing the text in the pointed

[⋆] This work is partially supported (for the Spanish group) by MCyT (PGE and FEDER) grant(TIC2003-06593), MEC (PGE and FEDER) grant (TIN2006-15071-C03-03), Xunta de Galicia Grant (PGIDIT05SIN10502PR) and (for the third author) by Millennium Nucleus Center for Web Research, grant (P04-067-F), Mideplan, Chile.

blocks, what usually implies decompressing the block. However, if the text is compressed with a technique that permits *direct search* in the compressed text, scanning the compressed blocks is much faster.

A good compressor for text databases has to join two main properties: *i)* to permit direct search into the compressed text by compressing the search pattern and then looking for this compressed version, and *ii)* to allow local decompression, which permits to decompress any portion of the compressed file without the need of decompressing it from the beginning. From the two main families of compressors (semistatic and adaptive compressors) only the semistatic ones join those two properties. Adaptive compressors, such as those from the well-known Ziv-Lempel family [15,16] learn the data distribution of the source symbols as they compress the text and therefore, the mapping "source symbol $\leftrightarrow$ codeword" is adapted as compression progresses. Albeit there exist methods to search text compressed with adaptive compressors [14,8], they are not very efficient. Decompression is usually needed during searches as the code given to a source symbol may vary along the text. Semistatic compressors (such as Huffman [10]) perform a first pass over the source text to obtain the distinct source symbols and to count their number of occurrences. Then, they associate each source symbol with a code that do not change during the second pass (where each source symbol is replaced by the corresponding codeword). Since the mapping source symbol $\leftrightarrow$ codeword does not vary, direct searching is allowed.

Classic Huffman is a well-known technique. It is a character-based method that generates an optimal *prefix* [1] coding. Unfortunately, it is not well-suited for text databases because of its poor compression ratio (around 60%). In [11], Moffat proposed using words instead of characters along with a Huffman coding scheme. As a result, compression ratio was reduced to around 25–30%. Moreover, using words instead of characters, gave the key to the integration of compression and text retrieval systems, since words are also the atoms of those systems.

Based on Moffat's idea, in [7] two word-based byte-oriented Huffman codes were presented. The first one, named Plain Huffman (PH) is a Huffman-based code that uses bytes instead of bits as target alphabet. By using bytes instead of bits, decompression speed was improved at the expense of compression ratio, which grew up to around 30–35%. The second compressor, named Tagged Huffman (TH), uses the first bit of each byte to mark the beginning of a codeword. Therefore, only 7 bits of each byte can be used to create the codewords, and a loss in compression of around 3 percentage points exists. However, since the beginning of a codeword can be recognized, random decompression is allowed and direct searches can be done by using a fast Boyer-Moore matching algorithm [2].

In [5,4] we presented End-Tagged Dense Code (ETDC), a statistical semistatic technique that maintains the good capabilities of Tagged Huffman for searches while improving its compression ratio and using a simpler and faster coding scheme.

[1] In a prefix code, no codeword is a prefix of another, a property that ensures that the compressed text can be decoded as it is processed, since a lookahead is not needed.

In this paper, we present a modification over ETDC (see next section) that we call *Pair-Based End-Tagged Dense Code (PETDC)*, which improves its compression ratio (around 28–30%) by exploiting the co-occurrence of words in the source text. Moreover, PETDC permits direct searching the compressed text, as well as fast random decompression. The paper is structured as follows: In Section 2, ETDC is shown. Then PETDC is described in Section 3. In Section 4, empirical results measuring the efficiency of PETDC in compression ratio, compression and decompression speed are given. Finally, some conclusions end the paper.

2 Related Work: End-Tagged Dense Code

End-Tagged Dense Code (ETDC) [5,4] is a semistatic compression technique, which is the basis of the new PETDC presented in this paper.

Plain Huffman Code [7] is a word-based Huffman code that assigns a sequence of bytes (rather than bits) to each word. In Tagged Huffman [7], the first bit of each byte is reserved to flag whether the byte is the first of its codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag bit is not useful by itself to make the code a prefix code. The tag bit permits direct searching the compressed text by just compressing the pattern and then running any classical string matching algorithm like Boyer-Moore [13,9]. In Plain Huffman this does not work, as the pattern could occur in the text not aligned to any codeword [7].

Instead of using a flag bit to signal the *beginning* of a codeword, ETDC signals the *end* of the codeword. That is, the highest bit of any codeword byte is 0 except for the last byte, where it is set to 1.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits of each byte.

ETDC obtains a better compression ratio than Tagged Huffman while keeping all its good searching and decompression capabilities. On the other hand, ETDC is easier to build and faster in both compression and decompression.

In general, ETDC can be defined over symbols of b bits, although in this paper we focus on the byte-oriented version where $b = 8$.

Definition 1. *Given source symbols with decreasing probabilities* $\{p_i\}_{0 \leq i < n}$ *the corresponding codeword using the End-Tagged Dense Code is formed by a sequence of symbols of* b *bits, all of them representing digits in base* 2^{b-1} *(that is, from 0 to* $2^{b-1} - 1$*), except the last one which has a value between* 2^{b-1} *and* $2^b - 1$*, and the assignment is done in a sequential fashion.*

That is, the first word is encoded as $\underline{1}0000000$, the second as $\underline{1}0000001$, until the 128th as $\underline{1}1111111$. The 129th word is coded as $\underline{0}0000000{:}\underline{1}0000000$, 130th as $\underline{0}0000000{:}\underline{1}0000001$ and so on until the $(128^2 + 128)^{th}$ word $\underline{0}1111111{:}\underline{1}1111111$. Note that the code depends on the rank of the words, not on their actual frequency. As a result, only the sorted vocabulary must be stored with the compressed text to allow the decompressor to recover the source text.

It is clear that the number of words encoded with 1, 2, 3 etc, bytes is fixed (specifically 128, 128^2, 128^3 and so on) and does not depend on the word frequency distribution. Generalizing, being k the number of bytes in each codeword ($k \geq 1$) words at positions i:

$$2^{b-1}\frac{2^{(b-1)(k-1)}-1}{2^{b-1}-1} \leq \ i \ < 2^{b-1}\frac{2^{(b-1)k}-1}{2^{b-1}-1}$$

will be encoded with k bytes. These clear limits mark the change points in the codeword lengths and will be relevant in the PETDC that we present in this paper.

But not only the sequential procedure is available to assign codewords to the words. There are simple *encode* and *decode* procedures that can be efficiently implemented, because the codeword corresponding to symbol in position i is obtained as the number x written in base 2^{b-1}, where $x = i - \frac{2^{(b-1)k}-2^{b-1}}{2^{b-1}-1}$ and $k = \left\lfloor \frac{\log_2\left(2^{b-1}+\ (2^{b-1}-1)i\right)}{b-1} \right\rfloor$, and adding 2^{b-1} to the last digit.

Function *encode* obtains the codeword $C_i = encode(i)$ for a word at the i-th position in the ranked vocabulary. Function *decode* gets the position $i = decode(C_i)$ in the vocabulary for a codeword C_i. Both functions take just $O(l)$ time, where $l = O(\log(i)/b)$ is the length in digits of codeword C_i. Those functions are efficiently implemented through just bit shifts and masking.

End-Tagged Dense Code is simpler, faster, and compresses 7% better than Tagged Huffman codes. In fact, ETDC only produces an overhead of about 2% over Plain Huffman. On the other hand, since the last bytes of codewords are distinguished, ETDC has all the search capabilities of Tagged Huffman code. Empirical comparisons between ETDC and Huffman codes can be found in [5,4].

3 Pair-Based End-Tagged Dense Code

PETDC is a semistatic compressor based on ETDC. As in all semistatic compressors, a first pass over the source text is performed in order to gather the different words of the source text (vocabulary) and their number of occurrences. After that first pass, an encoding scheme is used to assign a codeword to each symbol in the vocabulary. A second pass over the source text replaces each source symbol by the codeword associated to it. In addition, the compression process ends by storing the vocabulary in such a way that, for each codeword, we can obtain its corresponding original symbol.

However, there are some differences between PETDC and other semistatic compressors such as ETDC, PH, etc. During the first pass, PETDC obtains an initial vocabulary by gathering the different words and their number of occurrences in the source text. Moreover, PETDC collects also all the different pairs of words that appear adjacent in the source text and counts their number of occurrences. PETDC aims to take advantage of the co-occurrence of words in the text by including some pairs in the vocabulary (which is composed by both single-words and pairs). Its main idea is simple: In classic semistatic compressors, each symbol in the vocabulary has a unique codeword assigned by the encoding scheme (in this case, ETDC is used). Therefore, replacing two source words by a unique codeword during the second pass may need less bytes than replacing two single words by two codewords. Example 1 clarifies this situation.

Example 1. Let us compress the sequence of words: ADCBACDCCDABABA CDC with ETDC, assuming that special bytes of only 3 bits are used, and that our words occupy 1 byte each. Therefore, the mapping word-codeword generated by the encoding schema is: $C \leftarrow \underline{1}00$, $A \leftarrow \underline{1}01$, $D \leftarrow \underline{1}10$, $B \leftarrow \underline{1}11$. Hence, all the words can be encoded with only one byte, and the size of the compressed text is 18 bytes. In this case, the vocabulary consists of only 4 words. As a result, the size of the compressed file is $18 + 4 = 22$ bytes.

Let us add to the vocabulary the most frequent pair of words 'BA', and compress the same text again. In this case, the vocabulary contains the symbols 'C', 'D', 'BA', 'A', and 'B', while the number of occurrences is 6, 4, 3, 2, and 0 respectively. Now the compressed text uses 15 bytes, and the vocabulary needs 6 bytes. Therefore, the compressed file occupies $15 + 6 = 21$ bytes. □

From Example 1 we show that processing some pairs of words as a unique source symbol reduces the size of the compressed text. However, as a drawback, the size of the vocabulary grows when a pair is added. Therefore, the existence of a trade-off between compressed text size and vocabulary size has to be taken into account when pairs are added to the vocabulary.

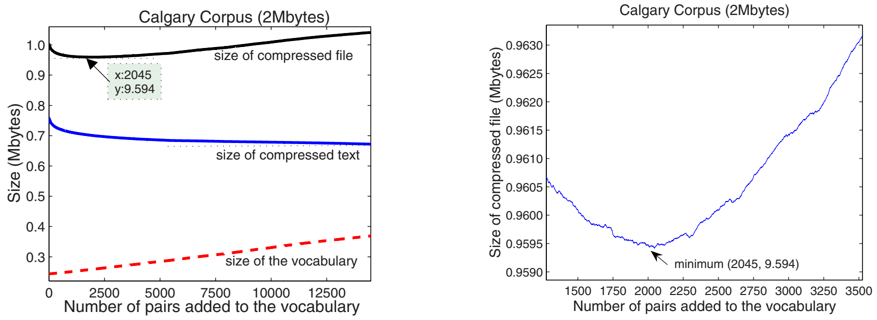
3.1 Deciding Which Pairs Should Be Added to the Vocabulary

Adding all the different pairs to the vocabulary is not a good idea because the vocabulary would grow too much. In Figure 1(a), we show the evolution of the size of a compressed file (as the sum of the size of the compressed data and the size of the vocabulary) depending on the number of pairs added. As expected, including the most frequent pairs in the vocabulary improves compression. However, at some point, the gain obtained by replacing two words by a unique codeword does not compensate the growth of the vocabulary size.

In Figure 1(b) it is shown that the previous curve has multiple local minima. This fact prevents us of looking for a heuristic to speed up the process by just breaking the addition of pairs to the vocabulary when the addition of a new pair worsens the compression. Instead of that, PETDC process all the pairs and applies a heuristic to determine which ones have to be added.

Used heuristic. Let us assume that a pair $\alpha\beta$, composed of two words α and β, is a candidate to be added to the vocabulary. Let us define f_x as the number of occurrences of a word or pair x. Let us also define C_x as the codeword that the encoding scheme[2] assigns to x, and let $|C_x|$ be the length of that codeword. The heuristic is based on comparing the number of bytes needed to encode all the

[2] The codeword assigned to a word by ETDC depends only on the rank in the sorted vocabulary.



(a) whole process. (b) area close to maximum compression.

Fig. 1. Evolution of compressed file as pairs are added

occurrences of α and β in the text in two cases: *i)* The pair is skipped ($skip_{bytes}$), and *ii)* the pair is added to the vocabulary (add_{bytes}). Once those values are computed, the pair $\alpha\beta$ is added to the vocabulary if $skip_{bytes} > add_{bytes}$ and skipped otherwise. Values $skip_{bytes}$ and add_{bytes} are given by the two following expressions:

$$skip_{bytes} = f_\alpha * |C_\alpha| + f_\beta * |C_\beta|$$
$$add_{bytes} = f_{\alpha\beta} * |C_{\alpha\beta}| + (f_\alpha - f_{\alpha\beta}) * |C'_\alpha| + (f_\beta - f_{\alpha\beta}) * |C'_\beta| + K$$

Where C'_α and C'_β are the codewords assigned to the words α and β assuming that the pair $\alpha\beta$ is added, and therefore their number of occurrences is $f_\alpha - f_{\alpha\beta}$ and $f_\beta - f_{\alpha\beta}$ respectively. The term 'K' is an estimation of the number of bytes needed to store any pair into the vocabulary. In general, $K = 5$.

Particular cases. There are two special situations that arise when pairs of words are considered:

– After adding a pair $\alpha\beta$ to the vocabulary it is necessary to ensure that any pair ending in α or beginning in β will not be included later. This happens because, by an efficiency issue, we do not store all the words that precede or follow any occurrence of $\alpha\beta$ in the text. As a result, given the text '$\gamma\underline{\alpha\beta}\delta\alpha\mu$', adding the pair $\alpha\beta$ implies that the pairs $\gamma\alpha$, $\beta\delta$, and $\delta\alpha$ cannot be added to the vocabulary. This is done by just marking α as "disabled as first word of pair" and marking β as "disabled as last word of pair", and finally checking those flags before adding a pair to the vocabulary.
– Sequences of the same word: The appearance of sequences of the same word α such as $\alpha_1\alpha_2\alpha_3\alpha_4$ might lead us to count erroneously the number of occurrences of the pair $\alpha\alpha$. Note that $\alpha_2\alpha_3$ is not a valid pair if the pair $\alpha_1\alpha_2$ is chosen. To avoid this problem, when a sequence is detected we only count the occurrences of the pairs that start in odd positions.

3.2 Data Structures and Compression Procedure

The data structures used by the compressor are sketched in Figure 2. There are two well-defined parts: *i)* Data structures that make up the vocabulary, and *ii)* data structures needed to hold the candidate pairs.

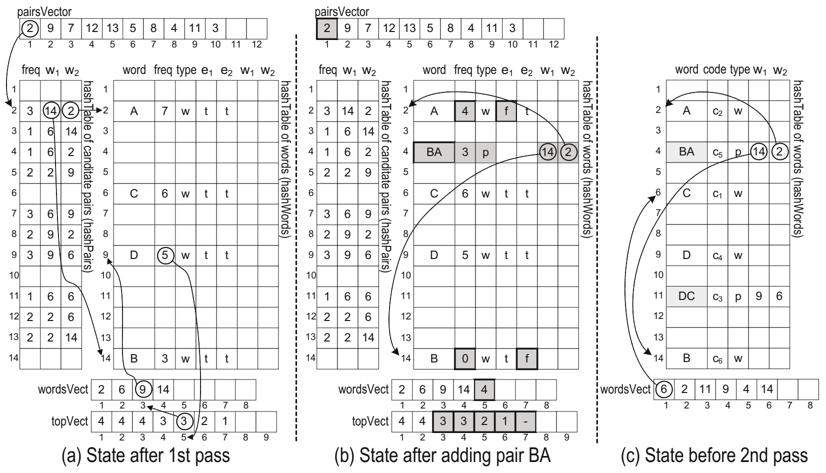

pairsVector
freq w1 w2
word freq type e1 e2 w1 w2
hashTable of candiate pairs (hashPairs)
hashTable of words (hashWords)
wordsVect
topVect
(a) State after 1st pass
(b) State after adding pair BA
word code type w1 w2
(c) State before 2nd pass

Fig. 2. Structures used in PETDC for text "ADCBACDCCDACADABABACDC"

– The vocabulary of the compressor consists of: A hash table used to locate a word quickly (*hashWords*) and two vectors: *wordsVect* and *topVect*. The hash table hashWords contains eight fields: *i) type* tells if an entry is either a word 'w' or a pair 'p', *ii)* if the entry has type 'w', *word* stores the word in ascii, *iii) freq* counts the number of occurrences of the entry, *iv-v)* e_1 *and* e_2 flag if the word is enabled to be the first or second component of a pair respectively, *vi-vii)* w_1 *and* w_2 store, for an entry of type 'p', pointers to the words that form the pair, and *viii) code* stores the codeword assigned to each entry of the vocabulary after the code generation phase.

Vector *wordsVect* serve us to maintain the vocabulary sorted by frequency. Then, slot 1 of *wordsVect* points to the entry of *hashWords* where the most frequent word (or pair) in the source text is stored. Assuming that *wordsVect* is sorted decreasingly by frecuency, vector *topVect[f]* keeps track of the first entry in *wordsVect* whose frequency is f.

– Managing the candidate pairs includes also the use of two main data structures: *i)* A hash table *hashPairs* with fields *freq*, w_1, and w_2, used to give a fast access to each candidate pair, and *ii)* a vector *pairsVector* that maintains all the candidate pairs sorted, in the same way as *wordsVect*.

Compressing with PETDC. Compression consists of five main phases:

– *First pass along the text.* As shown, during this pass, PETDC obtains the different v single-words and the different p candidate pairs that appear in the text. Moreover, it also counts their occurrences. The process costs $O(n)$, being n the number of words in the text. When the first pass ends, vectors *pairsVect* and *wordsVect* are sorted by decreasing frequency. Finally, *topVect* is initialized. Starting from element n downto 1, $topVec[i] = j$ if j is the first entry in $wordsVect$ such that $hashWords[wordsVect[j]].freq = i$. If $\nexists j$ such that $hashWords[wordsVect[j]].freq = i$, then $topVect[i] = topVect[i+1]$. The overall cost of this first phase is $O(n)+O(v\ log(v))+O(p\ \log(p))+O(v) = O(n) + O(p\ \log(p))$. Since $n \gg p$, we empirically proved that it costs $O(n)$.

– *Choosing and adding candidate pairs.* During this phase, $pairsVector$ is traversed ($O(p)$). A candidate pair $\alpha\beta$ is either added to the vocabulary or discarded, by applying the heuristic explained in Section 3.1. To compute that heuristic we need to know the current position[3] of α and β in the vocabulary to know the size of their respective codewords ($|C_x|$). We also need to assume that the pair $\alpha\beta$ is added, and we need to compute the new ranks of $\alpha\beta$, α, and β in the new ordered vocabulary. Since maintaining the vocabulary ordered upon inserting a pair is too expensive, we only maintain $topVect$ updated in such a way that, given a frequency f, $topVect[f]$ stores the rank, in a sorted vocabulary, of the first entry of frequency f. Then, being x an entry with frequency f_x, we estimate $|C_x|$ as $|C_{(topVect[f_x])}|$. It costs $O(F)$ time, where F is the frequency of the second most frequent entry of the vocabulary. Of course, $\alpha\beta$ is also inserted into $hashWords$ and into $wordsVector$. The overall cost of this phase is $O(p_a F + p) = O(p_a F)$, being p_a the number of pairs added to the vocabulary. Figure 2(b) shows the result of adding the pair “BA” to the vocabulary.
– *Code Generation Phase.* The only data structures needed in this phase are depicted in Figure 2(c). The vocabulary (with v' entries) is ordered by frequency and the encoding scheme of ETDC is used. Encoding takes $O(v')$ time. As a result, $hashWords$ will contain the mapping $entry_i \rightarrow code_i\ \forall i \in 1 \ldots v'$. The cost of this phase is $O(v' \log v')$.
– *Second pass.* The text is traversed again reading two words at a time and replacing source words by codewords. If the read pair $\alpha\beta$ belongs to $hashWords$ then the codeword $C_{\alpha\beta}$ is output and two new words $\gamma\delta$ are read. Otherwise C_α is output and the only the following word γ is read to form a new pair $\beta\gamma$. This phase takes $O(n)$ time.
– *storing the vocabulary.* As in ETDC, the vocabulary is stored along with the compressed data to permit decompression. A bitmask is used to save the type of entry. Then the v' entries of the vocabulary follow that bitmask. A single-word is written in ascii (ending in ‘\0’). To store a pair $\alpha\beta$ we write the relative positions of α and β in the vocabulary (encoded with the on-the-fly $C_w = getCode(i)$ function used in [3] in order to save space). Finally, the whole vocabulary is encoded with character-based Huffman.

Decompressing text compressed with PETDC. Decompression starts by loading the vocabulary into a vector. Then each codeword is easily parsed due

[3] Using the encoding scheme of ETDC, we can compute in $O(\log i)$ time the codeword $C_i = getCode(i)$ where i is the rank of a word w_i in the vocabulary.

to the flag bit that marks the end of a codeword. For each codeword C_i, the function $i = decode(C_i)$ [3] is used to obtain the entry i that contains either the word or the pair associated to C_i.

Searching text compressed with PETDC. In text compressed with PETDC, a word α can appear alone or as a part of one or more pairs $\alpha\beta$, $\gamma\alpha$,... Therefore searches will usually imply using a multi-pattern matching algorithm. When we load the vocabulary, we can easily generate for each single-word α, a list with the codewords of the pairs in which it appears. After that, an algorithm from the Boyer-Moore family such as *Set Horspool* [9,13] is used to search for those codewords and for the codeword C_α.

4 Empirical Results

We compare PETDC against other semi-static word-based compressors such as PH and ETDC and against two well-known compressors such as Gnu *gzip*[4], a Ziv-Lempel compressor and Seward's *bzip2* [5], a compressor based on the Burrows-Wheeler transform [6]. We used some large text collections from TREC-2 [6], namely AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as some from TREC-4: Congressional Record 1993 and Financial Times 1991 to 1994. We used the spaceless word model [7] to create the vocabulary; that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

Our first experiment is focused in comparing compression ratio. Table 1 shows in the two first columns the corpora used and their size. Columns three to six gives information about applying PETDC such as the number of candidate pairs, the number of occurrences of the most frequent pair, the number of pairs added, and the number of entries (pairs and words) of the vocabulary. The last five columns show compression ratio (in percentage) for the compressors used. It can be seen that PETDC improves the compression of PH and ETDC by around 3 and 4 percentage points respectively. Gaps against *gzip* grow up to $6-7$ percentage points. Finally, PETDC is overcome by *Bzip2* in around $1-3$ percentage points.

We focus now on comparing PETDC against other alternatives in compression and decompression speed. An isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 512 MB single-channel DDR-400Mhz was used in our tests. It ran Mandrake Linux 9.2. The compiler used was gcc version 3.3.1 and `-O9` compiler optimizations were set. Time results measure CPU user time in seconds. As it is shown in Table 2, PETDC pays the extra-cost of managing pairs during compression, being around 2.5 times slower than ETDC and PH, and around $1.5-2$ times slower than *gzip*. However, it is much faster than *bzip2*. In decompression, the extra-cost of PETDC consists only in processing

[4] `http://www.gnu.org`.
[5] `http://www.bzip.org`.
[6] `http://trec.nist.gov`.

Table 1. Compressing with PETDC and comparison in compression ratio with others

Corpus	Size (bytes)	cand. pairs	highest freq.	pairs added	entries vocab	Compression ratio (%) PETDC	PH	ETDC	gzip	bzip2
Calgary	2,131,045	53,595	2,618	4,705	35,700	41.31	42.39	43.54	37.10	29.14
FT91	14,749,355	239,428	18,030	12,814	88,495	30.72	33.13	34.02	36.42	27.06
CR	51,085,545	589,692	70,488	38,090	155,803	27.66	30.41	31.30	33.29	24.14
FT92	175,449,235	1,592,278	222,505	113,610	397,671	28.46	31.52	32.33	36.48	27.10
ZIFF	185,200,215	2,585,115	344,262	194,731	418,132	29.04	32.52	33.41	33.06	25.11
FT93	197,586,294	1,629,925	236,326	124,547	415,976	27.79	31.55	32.43	34.21	25.32
FT94	203,783,923	1,666,650	242,816	123,583	418,601	27.78	31.50	32.39	34.21	25.27
AP	250,714,271	1,574,819	663,586	118,053	355,674	28.80	31.92	32.73	37.32	27.22
ALL_FT	591,568,807	3,539,265	709,233	314,568	891,308	27.80	31.34	32.22	34.94	25.91

Table 2. Comparison in compression and decompression time

Corpus	Compression time (seconds)					Decompression time (seconds)				
	PETDC	PH	ETDC	gzip	bzip2	PETDC	PH	ETDC	gzip	bzip2
Calgary	0.59	0.33	0.37	0.26	0.88	0.06	0.06	0.06	0.04	0.32
FT91	3.07	1.48	1.47	1.71	6.33	0.30	0.25	0.25	0.20	2.24
CR	8.85	3.97	4.01	5.83	21.26	0.78	0.66	0.65	0.64	7.52
FT92	33.24	13.86	13.85	20.28	76.47	2.90	2.39	2.50	2.36	29.42
ZIFF	33.47	13.84	13.87	20.20	79.70	2.83	2.35	2.32	2.20	27.00
FT93	35.81	15.61	15.30	21.34	80.21	3.17	2.73	2.75	2.62	32.58
FT94	37.11	15.84	15.73	22.08	88.96	3.33	2.81	2.87	2.63	33.78
AP	48.50	20.06	20.26	30.66	105.48	4.11	3.41	3.41	3.43	39.42
ALL_FT	113.26	45.29	45.12	64.82	254.45	9.63	8.00	8.12	7.49	89.67

the bitmask in the header of the vocabulary file and rebuilding the pairs from the pointers to single-words. Therefore, the loss of speed against PH, ETDC, and *gzip* is small (under 20%), and PETDC becomes around $6 - 8$ times faster than *bzip2*.

5 Conclusions

We have presented a new semistatic pair-based byte-oriented compressor that we named *Pair-Based End-Tagged Dense code(PETDC)*. It takes advantage of using both words and pairs of words (exploiting the co-occurrence of words) to improve the compression obtained by similar word-based semistatic techniques such as PH or ETDC.

Dealing with pairs has a cost in compression speed (PETDC is around 2.5 times slower than ETDC) and in decompression (PETDC is 20% slower than ETDC). However, the new technique is able to reduce English texts to 27–28% of its original size (over 3 percentage points better than PH). Moreover, it maintains the ability of performing *direct searches* and *random decompression* of a portion of the text.

To sum up, PETDC is a technique well-suited to use in Text Databases due to its good compression ratio and decompression speed, as well as for its good search capabilities. The main drawback with respect to others might be its medium compression speed. However, in a text retrieval scenario a medium compression speed is only a minor problem since compression is done only once.

Acknowledgments. We want to thank Àngel Yàñez Miragaya for his help in the implementation of PETDC.

References

1. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* P.Hall, 1990.
2. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
3. N. Brisaboa, A. Fariña, G. Navarro, and José R. Paramá. Simple, fast, and efficient natural language adaptive compression. In *Proceedings of the 11th SPIRE*, LNCS 3246, pages 230–241, 2004.
4. N. Brisaboa, A. Fari na, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 2006. To appear.
5. N.R. Brisaboa, E.L. Iglesias, G. Navarro, and José R. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th ECIR*, LNCS 2633, pages 468–481, 2003.
6. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
7. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.
8. M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. In *Proceedings of the 27th ACM-STOC*, pages 703–712, 1995.
9. R. N. Horspool. Practical fast searching in strings. *SPE*, 10(6):501–506, 1980.
10. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
11. A. Moffat. Word-based text compression. *SPE*, 19(2):185–198, 1989.
12. G. Navarro, E.S. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *IR*, 3(1):49–77, 2000.
13. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge Univ. Press, 2002.
14. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proceedings of the 11th CPM*, LNCS 1848, pages 166–180, 2000.
15. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TIT*, 23(3):337–343, 1977.
16. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 24(5):530–536, 1978.

An Agent-Based Architecture for Dialogue Systems[⋆]

Mark Buckley and Christoph Benzmüller

Dept. of Computer Science, Saarland University
{markb|chris}@ags.uni-sb.de

Abstract. Research in dialogue systems has been moving towards reusable and adaptable architectures for managing dialogue execution and integrating heterogeneous subsystems. In this paper we present a formalisation of ADMP, an agent-based architecture which supports the development of dialogue applications. It features a central data structure shared between software agents, it allows the integration of external systems, and it includes a meta-level in which heuristic control can be embedded.

1 Introduction

Research in dialogue systems has been moving towards reusable and adaptable architectures for managing dialogue execution and integrating heterogeneous subsystems. In an architecture of this type, different theories of dialogue management can be formalised, compared and evaluated. In this paper we present a formalisation of ADMP[1], an architecture which uses software agents to support the development of dialogue applications. It features a central data structure shared between agents, it allows the integration of external systems, and it includes a meta-level in which heuristic control can be embedded.

We have instantiated the system to support dialogue management. Dialogue management involves maintaining a representation of the state of a dialogue, coordinating and controlling the interplay of subsystems such as domain processing or linguistic analysis, and deciding what content should be expressed next by the system. ADMP applies the information state update (ISU) approach to dialogue management [1]. This approach uses an information state as a representation of the state of the dialogue, as well as update rules, which update the information state as the dialogue progresses. The ISU approach supports the formalisation of different theories of dialogue management.

The framework of our research is the DIALOG project[2], which investigates flexible natural language dialogue in mathematics, with the final goal of natural tutorial dialogue between a student and a mathematical assistance system. In

⋆ This work was supported by the DAAD (German Academic Exchange Service), grant number A/05/05081 and by the DFG (Deutsche Forschungsgemeinschaft), Collaborative Research Centre 378 for Resource-adaptive Cognitive Processes.

[1] The Agent-based Dialogue Management Platform.

[2] http://www.ags.uni-sb.de/dialog/

the course of a tutorial session, a student builds a proof by performing utterances which contain proof steps, thereby extending the current partial proof. The student receives feedback from the DIALOG system after each proof step. This feedback is based on the computations and contribution of numerous systems, such as a domain reasoner or a natural language analysis module. The integration of these modules and the orchestration of their interplay as well as the selection of a next dialogue move which generates the feedback is the task of the dialogue manager.

The work presented in this paper is motivated by an initial prototype dialogue manager for the DIALOG demonstrator [2]. After its development we were able to pinpoint some features which we consider necessary for the DIALOG system, and which the platform presented here supports. The overall design of ADMP is influenced by the design of Ω-Ants [3], a suggestion mechanism which supports interactive theorem proving and proof planning. It uses societies of software agents, a blackboard architecture, and a hierarchical design to achieve concurrency, flexibility and robust distributed search in a theorem proving environment.

Although ADMP has been developed to support dialogue systems, it can be seen as a more general architecture for collaborative tasks which utilise a central data store. For example, we have used ADMP to quickly implement a lean prototype resolution prover for propositional logic.

Our work is related to other frameworks for dialogue management such as TrindiKit, a platform on top of which ISU based dialogue applications can be built. TrindiKit provides an information state, update rules and interfaces to external modules. Another such framework is Dipper [4], which uses an agent paradigm to integrate subsystems.

This paper is structured as follows. In Section 2 we give an overview of the DIALOG project and the role a dialogue manager plays in this scenario. Section 3 outlines the architecture of ADMP. Section 4 presents the formalisation of the system, and Section 5 concludes the paper.

2 The DIALOG Project

The DIALOG project is researching the issues involved in automating the tutoring of mathematical proofs through the medium of flexible natural language. In order to achieve this a number of subproblems must be tackled. An *input analyser* [5] must perform linguistic analysis of utterances. These typically contain both natural language and mathematical expressions and exhibit much ambiguity. In addition to the linguistic analysis the input analyser delivers an underspecified representation of the proof content of the utterance. Domain reasoning is encapsulated in a *proof manager* [6], which replays and stores the status of the student's partial proof. Based on the partial proof, it must analyse the correctness, relevance and granularity of proof steps, and try to resolve ambiguous proof steps. Pedagogical aspects are handled by a *tutorial manager* [7], which decides when and how to give which hints.

These three modules, along with several others such as a natural language generator, collaborate in order to fully analyse student utterances and to compute system utterances. Their computation must be interleaved, since they work with shared information, and this interplay is orchestrated by the dialogue manager. Fig. 1 shows the modules involved in the DIALOG system.

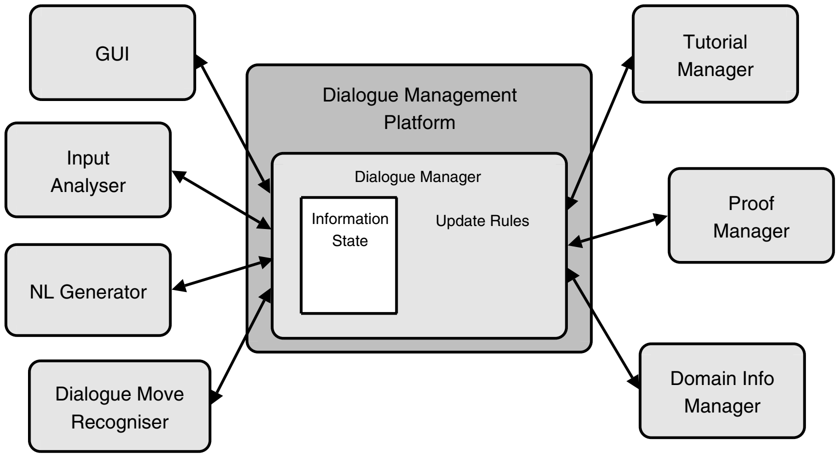


Fig. 1. The DIALOG system

We illustrate how the system works with an example from the DIALOG corpus [8] in Fig. 2, where K stands for the complement operation and U is the universal set. The student has made a correct step (`Stud1`) and thus has begun building a partial proof of the theorem. Later in the proof he makes an irrelevant step (`Stud2`). We now consider how the modules of the system interact to generate the response in (`Sys2`).

The student's utterance first goes to the input analyser, which calculates its linguistic meaning and underspecified proof content. The proof content in this case is a representation of the content of the formula which is given in the utterance. This is then taken up by the proof manager. In the given proof context it assigns the proof step category `correct`, since the formula holds under the assumptions, but also `irrelevant`, because the step does not lead to a proof, or at least does not appear in a proof of the theorem. Simultaneously the *dialogue move recogniser* uses the linguistic meaning to determine the function of the utterance in the dialogue, and encodes this function in a dialogue move [9]. A dialogue move is a multi-dimensional abstraction of the functions of an utterance in a dialogue, such as *question* or *assertion*. In the example the utterance is a *domain contribution* , since it asserts a new piece of information in the theorem proving task.

A *domain information manager* determines what domain information was addressed by the proof step by accessing static mathematical knowledge

`Sys0`: Bitte zeigen Sie: Wenn $A \subseteq K(B)$, dann $B \subseteq K(A)$
Please show that: If $A \subseteq K(B)$ *then* $B \subseteq K(A)$

`Stud1`: $U/A = K(A)$
$U/A = K(A)$

`Sys1`: Das ist richtig! Und weiter?
That's correct! And now?

...

`Stud2`: also ist $A \not\subseteq B$
Therefore it holds that $A \not\subseteq B$

`Sys2`: Nein, das bringt nichts. Wissen Sie, was Sie mit der wenn-dann-Beziehung anfangen müssen?
No, that doesn't help. Do you know what to do with the if-then relation?

Fig. 2. Excerpt from the Dialog corpus, session soc20k

resources. The tutorial manager uses a combination of these results to add hinting information to the dialogue moves. In this case it decides to explicitly indicate the inappropriateness (*"No"*) and irrelevance (*"that doesn't help"*) of the step. Furthermore, a combination of a student and tutor model result in an explicit hint, namely to draw the student's attention to dissolving the if-then relation which is the head of the theorem.

In general, the result of each module's computation is a contribution of content to some system dialogue move. The final step is that a *natural language generator* generates the utterances constituting the system's response in (`Sys2`) from these dialogue moves. Since a module's computations depend only on information stored in a subset of the information state, their execution order is only partially constrained. This means that many computations can and should take place in parallel, as in the case of the proof manager and dialogue move recogniser in the example above.

Dialog is an example of a complex system in which the interaction of many non-trivial components takes place. This interaction requires in turn non-trivial control to facilitate the distributed computation which results in the system response. This control function resides in the dialogue manager. As shown in Fig. 1, the dialogue manager forms the hub of the system and mediates all communication between the modules. It furthermore controls the interplay of the modules.

We realised a first Dialog demonstrator in 2003. It includes a dialogue manager built on top of Rubin [10], a commercial platform for dialogue applications. This dialogue manager integrates each of the modules mentioned above and controls the dialogue. It provides an information state in which data shared between modules is stored, input rules which can update the information state based on input from modules, and interfaces to the system modules.

However, we identified some shortcomings of this first dialogue manager for the demonstrator, and these have formed part of the motivation for the development of Admp:

– The modules in the system had no direct access to the information state, meaning they could not autonomously take action based on the state of the dialogue.
– The dialogue manager was static, and neither dialogue plans nor the interfaces to modules could be changed at runtime.
– There was also no way to reason about the flow of control in the system.

ADMP solves these problems by using a software agent approach to information state updates and by introducing a meta-level. The meta-level is used to reason about what updates should be made, and provides a place where the execution of the dialogue manager can be guided.

3 Architecture

The central concepts in the architecture of ADMP are information states and update rules, and these form the core of the system. An information state consists of slots which store values, and can be seen as an attribute-value matrix. It is a description of the state of the dialogue at a point in time, and can include information such as a history of utterances and dialogue move, the results of speech recognition or a representation of the beliefs of dialogue participants. Update rules encode transitions between information states, and are defined by a set of preconditions, a list of sideconditions, and a set of effects. Preconditions constrain what information states satisfy the rule, sideconditions allow arbitrary functions to be called within the rule, and effects describe the changes that should be made to the information state in order to carry out the transition that the rule encodes.

An update rule is embodied by an update rule agent, which carries out the computation of the transition that the update rule encodes. These check if the current information state satisfies the preconditions of the rule. When this is the case, they compute an *information state update* representing the fully instantiated transition. An information state update is a mapping from slotnames in the information state to the new values they have after the update is executed. We introduce information state updates as explicit objects in ADMP in order to be able to reason about their form and content at the meta-level.

As an example, we consider the information state in (1), a subset of the information state of the DIALOG system[3]. Here the user's utterance is already present in the slot `user_utterance`, but the linguistic meaning in the slot `lm` has not yet been computed. The slot `lu` stores a representation of the proof content of the utterance, and `eval_lu` stores its evaluated representation.

(1)
$$\text{IS}\begin{bmatrix} \texttt{user\_utterance} & \text{"also ist } A \not\subseteq B\text{"} \\ \texttt{lm} & \texttt{""} \\ \texttt{lu} & \texttt{""} \\ \texttt{eval\_lu} & \texttt{""} \end{bmatrix}$$

[3] In general an information state will contain richer data structures such as XML objects, but for presentation we restrict ourselves here to strings.

The update rule in (2) represents transitions from information states with a non-empty `user_utterance` slot to information states in which the `lm` and `lu` slots have been filled with the appropriate values.

(2) $$\frac{\{\texttt{non\_empty(user\_utterance)}\}}{\{\texttt{lm} \rightarrow \texttt{p}\ ,\ \texttt{lu} \rightarrow \texttt{q}\}} < \begin{array}{l}\texttt{r} := \texttt{input\_analyser(user\_utterance)},\\ \texttt{p} := \mathit{extract\_lm}(\texttt{r}),\\ \texttt{q} := \mathit{extract\_lu}(\texttt{r}) >\end{array}$$

In ADMP's update rule syntax this rule is defined as:

(3)
```
(ur~define-update-rule
 :name "Sentence Analyser"
 :preconds ((user_utterance :test #'ne-string))
 :sideconds ((r :function input_analyser
                :slotargs (user_utterance))
             (p :function extract-lm :varargs (r))
             (q :function extract-lu :varargs (r))
             )
 :effects ((lm p) (lu q))
)
```

The precondition states that the slot `user_utterance` must contain a non-empty string. When this is the case, the rule can fire. It carries out its sideconditions, thereby calling the function `input_analyser`, which performs the actual computation and calls the module responsible for the linguistic analysis of utterances. Rule (2) thus represents the input analyser. The result of this computation is an object containing both the linguistic meaning of the utterance and an underspecified representation of the proof content. The functions *extract_lm* and *extract_lu* access the two parts and store them in the variables `p` and `q`, respectively. The information state update that the rule computes maps the slot name `lm` to the linguistic meaning of the utterance and the slot name `lu` to its proof content.

Rule (4) represents the proof manager, and picks up the proof content of the utterance in the slot `lu`.

(4) $$\frac{\{\texttt{non\_empty(lu)}\}}{\{\texttt{eval\_lu} \rightarrow \texttt{r}\}} < \texttt{r} := \texttt{pm\_analyse(lu)} >$$

The proof manager augments the information in `lu` by attempting to resolve underspecification and assign correctness and relevance categories, and the resulting update maps `eval_lu` to this evaluated proof step. A similar update rule forms the interface to the dialogue move recogniser, which uses the linguistic meaning of the utterance in `lm` to compute the dialogue move it represents. Since these two computations are both made possible by the result of the update from the input analyser, they can run in parallel.

Fig. 3 shows the architecture of ADMP. On the left is the information state. Update rules have in their preconditions constraints on some subset of the information state slots and are embodied by update rule agents, which are shown here next to the information state. When an update rule agent sees that the preconditions of its rule hold, the rule is applicable and can fire. The agent then executes each of the sideconditions of the rule, and subsequently computes the information state update that is expressed by the rule's effects. The resulting information state update is written to the update blackboard, shown in the middle of the diagram.

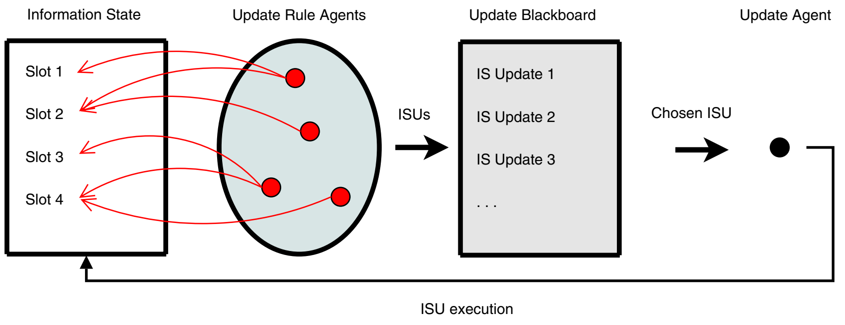


Fig. 3. The architecture of ADMP

The update blackboard collects the proposed updates from the update rule agents. These agents act in a concurrent fashion, so that many of them may be simultaneously computing results; some may return results quickly and some may perform expensive computations, e.g. those calling external modules. Thus the set of entries on the update blackboard can grow continually. On the far right of the diagram is the update agent, which surveys the update blackboard. After a timeout or some stimulus it chooses the heuristically preferred update (or a combination of updates) and executes it on the current information state. This completes a transition from one information state to the next.

Finally the update agent resets the update rule agents. Agents for whom the content of the slots in their preconditions has not changed can continue to execute since they will then be computing under essentially the same conditions (i.e. the information that is relevant to them is the same). Agents for whom the slots in the preconditions have changed must be interrupted, even if their preconditions still happen to hold. This is because they are no longer computing within the correct current information state.

4 A Formal Specification of ADMP

We now give a concise and mathematically rigorous specification of ADMP. We introduce the concepts and terminology necessary to guarantee the well-definedness of information states and update rules, and we give an algorithmic description of the update rule agents and the update agent.

Information States and Information State Updates. First, we fix some data structures for the slot names and the slot values of an information state. In our scenario it is sufficient to work with strings in both cases (alternatively we could work with more complex data structures). Let $\mathcal{A}$ and $\mathcal{B}$ be alphabets.

We define the *language for slot names* as $\mathit{SlotId} := \mathcal{A}^*$ and the *language for slot values* as $\mathit{SlotVal} := \mathcal{B}^*$. In our framework we want to support the checking of certain properties for the values of single slots. Thus we introduce the notion of a Boolean test function for slot values. A *Boolean test function* is a function $f \in \mathcal{BT} := \mathit{SlotVal} \to \{\top, \bot\}$.

Next, we define information state slots as triples consisting of a slot name, a slot value, and an associated Boolean test function. The set of all possible *information state slots* is $\mathit{Slots} := \mathit{SlotId} \times \mathcal{BT} \times \mathit{SlotVal}$. Given an information state slot $u = (s, b, v)$, the slot name, the test function, and the slot value can be accessed by the following projection functions: $\mathit{slotname}(u) := s$, $\mathit{slotfunc}(u) := b$ and $\mathit{slotval}(u) := v$.

Information states are sets of information state slots which fulfil some additional conditions. Given $r \subseteq \mathit{Slots}$, we call r a *valid information state* if $r \neq \emptyset$ and for all $u_1, u_2 \in r$ we have $\mathit{slotname}(u_1) = \mathit{slotname}(u_2) \Rightarrow u_1 = u_2$. We define $\mathcal{IS} \subset \mathcal{P}(\mathit{Slots})$ to be the set of all valid information states. The set of all slot names of a given information state $r \in \mathcal{IS}$ can be accessed by a function $\mathit{slotnames} : \mathcal{IS} \to \mathcal{P}(\mathit{SlotId})$ which is defined as follows

$$\mathit{slotnames}(r) = \{s \in \mathit{SlotId} \mid \exists\, u \in r \,.\, \mathit{slotname}(u) = s\}$$

We define a function $\mathit{read} : \mathcal{IS} \times \mathit{SlotId} \to \mathit{SlotVal}$ to access the value of a slot in an information state where $\mathit{read}(r, s) = \mathit{slotval}(u)$ for the unique $u \in r$ with $\mathit{slotname}(u) = s$.

In our framework information states are dynamically updated, i.e. the values of information state slots are replaced by new values. Such an *information state update* is a mapping from slots to their new values. The set of all valid information state updates μ is denoted by $\mathcal{ISU}$, the largest subset of $\mathcal{P}(\mathit{SlotId} \times \mathit{SlotVal})$ for which the following restriction holds: $\forall (s_1, v_1), (s_2, v_2) \in \mu \,.\, s_1 = s_2 \Rightarrow v_1 = v_2$ for all $\mu \in \mathcal{ISU}$. We define $\mathcal{ISU}_\bot := \mathcal{ISU} \cup \{\bot\}$. An information state update $\mu \in \mathcal{ISU}$ is *executable* in an information state $r \in \mathcal{IS}$ if the slot names addressed in μ actually occur in r and if the new slot values suggested in μ fulfil the respective Boolean test functions, i.e.

$$\mathit{executable}(r, \mu) \text{ iff } \forall (s, v) \in \mu \,.\, \exists\, u \in r \,.\, \mathit{slotname}(u) = s \wedge \mathit{slotfunc}(u)(v) = \top$$

We overload the function *slotnames* from above and analogously define it for information state updates. Information state updates are executed by a function $\mathit{execute\_update} : \mathcal{IS} \times \mathcal{ISU} \to \mathcal{IS}$. Given an information state $r \in \mathcal{IS}$ and an information state update $\mu \in \mathcal{ISU}$ we define

$$\mathit{execute\_update}(r, \mu) = \begin{cases} r & \text{if not } \mathit{executable}(r, \mu) \\ r^- \cup r^+ & \text{otherwise} \end{cases}$$

where

$$r^- := (r \setminus \{(s, b, v) \in r | s \in \mathit{slotnames}(\mu)\}$$
$$r^+ := \{(s', b', v') \mid (s', v') \in \mu \wedge \exists u \in r \,.\, s' = \mathit{slotname}(u) \wedge b' = \mathit{slotfunc}(u)\}$$

Update Rules. Update rules use the information provided in an information state to compute potential information state updates. They consist of preconditions, sideconditions and effects.

The preconditions of an update rule identify the information state slots that the rule accesses information from. For each identified slot an additional test function is provided which specifies an applicability criterion. Intermediate computations based on information in the preconditions are performed by the sideconditions of the update rules. For this, a sidecondition may call complex external modules, such as the linguistic analyser or the domain reasoner. The results of these side-computations are bound to variables in order for them to be accessible to subsequent sideconditions and to pass them over from the sideconditions to the effects of a rule. We now give a formal definition of each part in turn.

Let $s \in \mathit{SlotId}$ and $b \in \mathcal{BT}$. The tuple (s, b) is called an *update rule precondition.* The set of all update rule preconditions is denoted by $\mathcal{C} := \mathit{SlotId} \times \mathcal{BT}$. We define projection functions $\mathit{pc\_slotname} : \mathcal{C} \to \mathit{SlotId}$ and $\mathit{pc\_testfunc} : \mathcal{C} \to \mathcal{BT}$ such that $\mathit{pc\_slotname}(pc) = s$ and $\mathit{pc\_testfunc}(pc) = b$ for all $pc = (s, b)$. An information state $r \in \mathcal{IS}$ *satisfies* an update rule precondition $pc = (s, b)$ if the function b applied to the value of the slot in r named s returns $\top$, i.e. $\mathit{satisfies}(r, pc)$ iff $\exists u \in r \,.\, \mathit{pc\_testfunc}(pc)(\mathit{slotval}(u)) = \top \wedge \mathit{slotname}(u) = \mathit{pc\_slotname}(pc)$. We overload the predicate *satisfies* and define it for sets of preconditions $\mathcal{C}' \subseteq \mathcal{C}$ and information states $r \in \mathcal{IS}$ as follows: $\mathit{satisfies}(r, \mathcal{C}')$ holds if each precondition in $\mathcal{C}'$ is satisfied by r.

Let $v \in \mathit{Var}$ be a variable where Var is a set of variables distinct from the languages $\mathcal{A}^*$ and $\mathcal{B}^*$, let $(v_1 \ldots v_m) \in \mathit{Var}^m$ be an m-tuple of variables, let $(s_1 \ldots s_n) \in \mathit{SlotId}^n$ be an n-tuple of slot names, and let $f : \mathit{SlotVal}^n \to \mathit{SlotVal}^m \to \mathit{SlotVal}$ be a function[4] $(n, m \geq 0)$. A *single sidecondition* is thus given by the quadruple $(v, (s_1, \ldots, s_n), (v_1, \ldots, v_m), f)$. The set of all single sideconditions is denoted by $\mathcal{D} := \mathit{Var} \times \mathit{SlotId}^n \times \mathit{Var}^m \times (\mathit{SlotVal}^n \to \mathit{SlotVal}^m \to \mathit{SlotVal})$.

Given the set $\mathcal{D}$ of single sideconditions sc_i, the sideconditions of an update rule are now modelled as lists $l := <sc_1, \ldots, sc_n>$, $n \geq 0$. We further provide projection functions $\mathit{sc\_var} : \mathcal{D} \to \mathit{Var}$, $\mathit{sc\_slottuple} : \mathcal{D} \to \mathit{SlotId}^n$, $\mathit{sc\_slotnames} : \mathcal{D} \to \mathcal{P}(\mathit{SlotId})$, $\mathit{sc\_vartuple} : \mathcal{D} \to \mathit{Var}^m$, $\mathit{sc\_varnames} : \mathcal{D} \to \mathcal{P}(\mathit{Var})$ and $\mathit{sc\_func} : \mathcal{D} \to (\mathit{SlotVal}^n \to \mathit{SlotVal}^m \to \mathit{SlotVal})$, such that for all $sc = (v, (s_1, \ldots, s_n), (v_1, \ldots, v_m), f) \in \mathcal{D}$ it holds that $\mathit{sc\_var}(sc) = v$, $\mathit{sc\_slottuple}(sc) = (s_1, \ldots, s_n)$, $\mathit{sc\_slotnames}(sc) = \{s_1, \ldots, s_n\}$, $\mathit{sc\_vartuple}(sc) = (v_1, \ldots, v_m)$, $\mathit{sc\_varnames}(sc) = \{v_1, \ldots, v_m\}$ and $\mathit{sc\_func}(sc) = f$.

A *sidecondition list* l is called valid if two conditions hold: for all $sc_i, sc_j \in l$ with $i \neq j$ we must have $\mathit{sc\_var}(sc_i) \neq \mathit{sc\_var}(sc_j)$ and for all $sc_i \in l$ we must have $\mathit{sc\_varnames}(sc_i) \subseteq \{v | \exists\, sc_j \in l \,.\, j < i \wedge v = \mathit{sc\_var}(sc_j)\}$. The set of all valid sidecondition lists is denoted as $\mathcal{D}_l$.

Let $s \in \mathit{SlotId}$ and $v \in \mathit{Var}$ be a variable. The tuple (s, v) is called an *update rule effect.* The set of all update rule effects is denoted by $\mathcal{E} := \mathit{SlotId} \times \mathit{Var}$.

[4] We assume the right-associativity of $\to$.

We provide projection functions $e\_slotname : \mathcal{E} \rightarrow SlotId$ and $e\_var : \mathcal{E} \rightarrow Var$ such that $e\_slotname((s, v)) = s$ and $e\_var((s, v)) = v$.

Let $\mathcal{U}$ be a set of rule names (distinct from $\mathcal{A}^*$, $\mathcal{B}^*$, and Var). An *update rule* is a quadruple $\nu \in \mathcal{UR} := \mathcal{U} \times \mathcal{P}(\mathcal{C}) \times \mathcal{D}_l \times \mathcal{P}(\mathcal{E})$. An update rule $\nu = (n, c, d, e) \in \mathcal{UR}$ is *well-defined* w.r.t. the information state r if

1. the slotnames mentioned in the preconditions actually occur in r, i.e, for all $pc \in c$ we have $pc\_slotname(pc) \in slotnames(r)$,
2. each slot that is accessed by a sidecondition function has been mentioned in the preconditions, i.e., $(\bigcup_{d_i \in d} sc\_slotnames(d_i)) \subseteq \{s \in SlotId \mid \exists\ pc \in c\ .\ pc\_slotnames(pc) = s\}$,
3. the variables occurring in the effects have been initialised in the sideconditions, i.e., $\{v \in Var \mid \exists\ e_i \in e\ .\ e\_var(e_i) = v\} \subseteq \{v \in Var \mid \exists\ sc \in d\ .\ sc\_var(sc) = v\}$, and
4. the slotnames in the effects refer to existing slots in the information state r, i.e., $\{s \in SlotId \mid \exists e_i \in e\ .\ e\_slotname(e_i) = s\} \subseteq slotnames(r)$.

Let $\nu = (n, c, d, e) \in \mathcal{UR}$ be an update rule and $r \in \mathcal{IS}$ be an information state. ν is called *applicable* in r if ν is well-defined w.r.t. r and $satisfies(r, c)$ holds. This is denoted by $applicable(r, \nu)$.

Update Rule Agents. Update rule (software) agents encapsulate the update rules, and their task is to compute potential information state updates. The suggested updates are not immediately executed but rather they are passed to an update blackboard for heuristic selection. Update rule agents may perform their computations in a distributed fashion.

An update rule agent embodies a function $execute\_ur\_agent : \mathcal{UR} \rightarrow (\mathcal{IS} \rightarrow \mathcal{ISU}_\perp)$. The function $execute\_ur\_agent(\nu)$ takes an update rule ν and returns a function (lambda term) representing the computation that that rule defines. The new function can then be applied to a given information state in order to compute a suggestion for how to update this information state. For each update rule we obtain a different software agent.

We introduce a macro `sc_evaluate` which abbreviates the retrieval of the values in the variables and slotnames in the body of sidecondition and the computation of the value which is to be stored in the sidecondition's variable. We use **function_call** to apply a function to the arguments which follow it and **value_of** to retrieve the value stored in a variable.

```
sc_evaluate($sc$) =
  let $(s_1, \ldots, s_n) := sc\_slottuple(sc)$
    let $(v_1, \ldots, v_m) := sc\_vartuple(sc)$
      let $(t_1, \ldots, t_m) := (\mathbf{value\_of}(v_1), \ldots, \mathbf{value\_of}(v_m))$
        $\mathbf{function\_call}(sc\_func(sc), (read(r, s_1), \ldots, read(r, s_n)), (t_1, \ldots, t_m))$
```

We now define *execute_ur_agent* as

```
$execute\_ur\_agent(\nu = (n, c, d, e)) =$
  $\lambda r$ . **if** $applicable(r, \nu)$
      **then**
        **let** $<sc_1, \ldots, sc_n> := d$
          **let** $sc\_var(sc_1) := \texttt{sc\_evaluate}(sc_1)$
            **let** $sc\_var(sc_2) := \texttt{sc\_evaluate}(sc_2)$
                  $\vdots$
              **let** $sc\_var(sc_n) := \texttt{sc\_evaluate}(sc_n)$
                $\{(s, v) | \exists (s, sc\_var(sc_i)) \in e \,.\, v = \textbf{value\_of}(sc\_var(sc_i))\}$
      **else** $\perp$
```

Update Blackboard and Update Agent. An *update blackboard* is modelled as a set of information state updates $w \in \mathcal{UB} := \mathcal{P}(\mathcal{ISU})$, and stores proposed updates to the current information state. The *update agent* investigates the entries on the update blackboard, heuristically chooses one of the proposed information state updates and executes it. We assume a user-definable function $choose : \mathcal{UB} \rightarrow \mathcal{ISU}$ which realises the heuristic choice based on some heuristic ordering criterion $>_{\mathcal{UB}} : \mathcal{ISU} \times \mathcal{ISU}$. A simple example of a partial ordering criterion $>_{\mathcal{UB}}$ is

$$\mu_1 >_{\mathcal{UB}} \mu_2 \text{ iff } slotnames(\mu_2) \subseteq slotnames(\mu_1)$$

In fact, *choose* may be composed of several such criteria, and clearly the overall behaviour of the system is crucially influenced by them. The update agent now embodies a function $update\_agent : \mathcal{UB} \times (\mathcal{UB} \rightarrow \mathcal{ISU}) \times \mathcal{IS} \rightarrow \mathcal{IS}$ which is defined as

$$update\_agent(w, choose, r) = execute\_update(r, choose(w))$$

5 Conclusion

In this paper we have presented a formalisation of Admp, a platform for developing dialogue managers using the information state update approach. We were motivated by the need to integrate many complex and heterogeneous modules in a flexible way in a dialogue system for mathematical tutoring. These modules must be able to communicate and share information with one another as well as to perform computations in parallel.

Admp supports these features by using a hierarchical agent-based design. The reactive nature of the update rule agents allows for the autonomous concurrent execution of modules triggered by information in the information state. This furthermore obviates the need for a strict pipeline-type control algorithm often seen in dialogue systems, since agents can execute without being explicitly called. Interfacing the dialogue manager with system modules is also simplified by using the agent paradigm, because adding a new module involves only declaring a new update rule. Finally, the meta-level provides a place where overall control can take place if needed.

Admp thus allows the formalisation of theories of dialogue in the information state update approach, offering the functionality of related systems like TrindiKit and Dipper. However by introducing an explicit heuristic layer for overall control it allows reasoning about the execution of the dialogue manager which these two systems do not support.

An instantiation of Admp is achieved by declaring an information state, a set of update rules which operate on the information state, and a *choose* function, whereby a developer can fall back to a default function such as suggested in the previous section. A user-defined *choose* function should compute valid $\mathcal{ISU}$s, also in the case where $\mathcal{ISU}$s from the update blackboard are merged. As an example, a conservative merge strategy would simply reject the merging of pairs of $\mathcal{ISU}$s whose slotname sets intersect. Update rule agents and the update agent are automatically generated from the update rule declarations.

We have recently implemented Admp and given an instantiation for the Dialog system which uses eleven update rules and requires no declaration of control structure. We have also shown that we can implement a propositional resolution prover in Admp with four agents and five information state slots, which corresponds to just 40 lines of code. Extensions such as a set of support strategy can be realised simply by adding agents, possibly at runtime.

We foresee as future work the extension of our agent concept to include for instance resource sensitivity, and the investigation of further default heuristics for the dialogue scenario. Other interesting work is to turn the specification given in this paper into a formalisation within a higher-order proof assistant such as ISABELLE/HOL, HOL or OMEGA and to verify its properties.

References

1. Traum, D., Larsson, S.: The information state approach to dialogue management. In van Kuppevelt, J., Smith, R., eds.: Current and new directions in discourse and dialogue. Kluwer (2003)
2. Buckley, M., Benzmüller, C.: A Dialogue Manager supporting Natural Language Tutorial Dialogue on Proofs. Electronic Notes in Theoretical Computer Science (2006) To appear.
3. Benzmüller, C., Sorge, V.: Ω-Ants – An open approach at combining Interactive and Automated Theorem Proving. In Kerber, M., Kohlhase, M., eds.: 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus-2000), AK Peters (2000)
4. Bos, J., Klein, E., Lemon, O., Oka, T.: Dipper: Description and formalisation of an information-state update dialogue system architecture. In: Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue, Sapporo, Japan (2003)
5. Horacek, H., Wolska, M.: Interpreting Semi-Formal Utterances in Dialogs about Mathematical Proofs. Data and Knowledge Engineering Journal **58**(1) (2006) 90–106

6. Benzmüller, C., Vo, Q.: Mathematical domain reasoning tasks in natural language tutorial dialog on proofs. In Veloso, M., Kambhampati, S., eds.: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, Pennsylvania, USA, AAAI Press / The MIT Press (2005) 516–522
7. Tsovaltzi, D., Fiedler, A., Horacek, H.: A Multi-dimensional Taxonomy for Automating Hinting. In Lester, J.C., Vicari, R.M., Paraguaçu, F., eds.: Intelligent Tutoring Systems, 7th International Conference (ITS 2004). Number 3220 in LNCS, Springer (2004) 772–781
8. Benzmüller, C., Fiedler, A., Gabsdil, M., Horacek, H., Kruijff-Korbayová, I., Pinkal, M., Siekmann, J., Tsovaltzi, D., Vo, B.Q., Wolska, M.: A Wizard-of-Oz experiment for tutorial dialogues in mathematics. In: Proceedings of the AIED Workshop on Advanced Technologies for Mathematics Education, Sydney, Australia (2003) 471–481
9. Allen, J., Core, M.: Draft of DAMSL: Dialogue act markup in several layers. DRI: Discourse Research Initiative, University of Pennsylvania (1997)
10. Fliedner, G., Bobbert, D.: A framework for information-state based dialogue (demo abstract). In: Proceedings of the 7th workshop on the semantics and pragmatics of dialogue (DiaBruck), Saarbrücken (2003)

Encoding Mobile Ambients into the π-Calculus

Gabriel Ciobanu[1] and Vladimir A. Zakharov[2]

[1] "A.I.Cuza" University, Faculty of Computer Science
and Romanian Academy, Institute of Computer Science
Blvd. Carol I nr.8, 700505 Iaşi
gabriel@iit.tuiasi.ro

[2] Faculty of Computational Mathematics and Cybernetics,
Lomonosov State University, Moscow, Russia
zakh@cs.msu.su

Abstract. We present an encoding of the mobile ambients without communication into a subset of the π-calculus, namely the localized sum-free synchronous π-calculus. We prove the operational correspondence between the two formalisms. A key idea of the encoding is the separation of the spatial structure of mobile ambients from their operational semantics. The operational semantics is given by a universal π-process *Ruler* which communicates with a π-calculus term $Structure_A$ simulating the spatial structure of a mobile ambient A by means of channels. We consider the presented encoding as a first step toward designing a fully abstract translation of the calculus of mobile ambients into the π-calculus and thus developing a uniform framework for the theory of mobile computations.

1 Introduction

We consider two important models of the mobile computations, namely the π-calculus and mobile ambients. These formalisms are well suited for capturing two different aspects of mobility: the π-calculus is a process algebra where communication channels can "move" along other channels, whereas mobile ambients are suitable to represent such issue as migration of processes between boundaries.

The π-calculus [14] is a widely accepted model of interacting systems with dynamically evolving communication topology. Its mobility is expressed by the changing configuration and connectivity among processes. This mobility increases the expressive power of the π-calculus, and the π-calculus is a general model of computation which takes interaction as a primitive. The π-calculus models computing by reduction and interactive matching.

Another successful formalism for mobility is provided by mobile ambients [2]. The ambient calculus is a model for reasoning about properties of mobile processes and a programming language prototype for the Web. An ambient captures both the idea of process and the idea of location. The ambient calculus reflects the ability to bound the scope of process interaction and to provide migration of processes between boundaries. This formalism is well suited for expressing such issues of mobile computations as working environment, an access to information and resources, mutual arrangement of mobile processes and devices, etc.

Although both the π-calculus and the calculus of mobile ambients are Turing-complete [2,14] and they have almost the same field of application (mobile computations), it is widely believed (see [5]) that π-calculus does not directly model phenomena such as the distribution of processes within different localities, their migrations, or their failures. At the same time the π-calculus provides a solid and useful foundation for concurrent programming languages [7,15] and it is also supplied with a set of comprehensive techniques and tools for verification and analysis [6,18]. Therefore, it is worthwhile to take those advantages of the π-calculus that could be useful for manipulation, implementation, verification, etc. of mobile ambients.

In a number of papers (see [2,3,11,19]) it has been demonstrated that the calculus of mobile ambients can be used for simulating the π-calculus computations. On the other hand, Fournet, Levy and Schmitt [9] have translated mobile ambients into the distributed join-calculus. The atomic steps of mobile ambient computation are decomposed into several elementary steps, each involving only local synchronization. By combining this translation with the encoding of distributed join calculus into the join calculus [8] and then with the encoding of the join calculus into asynchronous π-calculus [7] one could obtain the translation of mobile ambients into the asynchronous π-calculus. But to the best of our knowledge no efforts were made to trace this chain of encodings from the beginning to the end. An attempt to build a straightforward translation from mobile ambients into a subset of synchronous π calculus has been undertaken by Brodo, Degano and Priami [1]. In this paper to imitate the spatial structure of mobile ambients some very rigid restrictions on the structural congruence rules of the π-calculus are imposed. It may be said that in [1] the encoding of mobile ambients into the π-calculus has been achieved on the pure syntactic level.

In our paper we also try to assess the capability of the π-calculus to encode mobile ambients. The topic is interesting because mobile ambients can be considered a higher-order calculus of fundamental character. Moreover, an encoding of mobile ambients into π-calculus appears challenging, in particular because distributed conformance changes must be effectuated over varying numbers of encoded agents and capabilities (as encoded ambients migrate or open up themselves). The main objective of our research is to build such a straightforward translation from the calculus of mobile ambients to the π-calculus which could preserve the behavioural properties of the processes. This translation coupled with that of [2,3,19] may form a basis for the development of a uniform theory of mobile computations.

As the starting point in this paper we present a rather simple variant of the straightforward encoding of mobile ambients into the π-calculus to demonstrate the practicability of our expectancies. A key idea of the encoding is based on the separation of the spatial structure of mobile ambients from their operational semantics. The operational semantics of mobile ambients is given by a universal π-process *Ruler* which plays the role of an interpreter routine. Each mobile ambient A is encoded into a π-calculus term $Structure_A$ which simulates the spatial structure of A by means of channels. Each step of the encoding is explained in some details (a complete elucidation of all constructions used in our encoding, their intended meaning and behaviour could be readily found in [4]). We also provide an operational correspondence between the two calculi.

To emphasize the key ideas of our encoding we confine ourselves with its most simple (sequential) variant which assumes the using of the unique *Ruler* for the whole system. As it can be readily seen from the description this encoding can be improved in such a way that it becomes both distributed and compositional. This can be achieved by supplying every π-process *Node* corresponding to an ambient with its own interpreter *Ruler*.

The structure of the paper is as follows. Section 2 presents the pure mobile ambients, and provides a short description of the π-calculus. The core of the paper is represented by Section 3; it presents the translation of mobile ambients into π-processes. We introduce the tree-wire processes, then we describe the simulation of capabilities consumption; the behaviour of an ambient is simulated by a *Ruler* process. Finally the encoding is defined in terms of two relations $\models_0$ and $\models$. The completeness and soundness of the encoding are presented in Section 4.

2 Mobile Ambients and the π-Calculus

We give here a short description of pure mobile ambients; more information can be found in [2]. Given an infinite set of names $\mathcal{N}$ (ranged over by $m, n, \ldots$) we define the sets $\mathcal{A}$ of MA-processes (denoted by $A, A', B, \ldots$) and capabilities (denoted by $M, M', \ldots$) as follows

$$M \;::=\; in\ n \;\;|\;\; out\ n \;\;|\;\; open\ n$$
$$A \;::=\; \mathbf{0} \;\;|\;\; A|B \;\;|\;\; !A \;\;|\;\; M.A \;\;|\;\; n[A] \;\;|\;\; (\nu n)\ A$$

Processes of the form $M.A$ and $n[A]$ are called *actions* and *ambients*, respectively. The free names of MA-process A are denoted by $fn(A)$. Hereafter, it will be convenient to assume an external ambient $\top \notin fn(A)$. For every process we define the set of top-level subprocesses as follows:

$TL(\mathbf{0}) = \emptyset$; if $A = M.B$ or $A = n[B]$, then $TL(A) = \{A\}$;
if $A = B_1|B_2$, then $TL(A) = TL(B_1) \cup TL(B_2)$;
if $A =!B$ or $A = (\nu n)\ B$, then $TL(A) = TL(B)$.

For each action $A = M.B$ or ambient $A = n[B]$ we will say that $TL(B)$ is the *lower context* of A and A is the *upper context* of every subprocess from $TL(B)$.

The *structural congruence* $\equiv_a$ on MA-processes is the least congruence satisfying the following requirements:

$(\mathcal{A}, |, \mathbf{0})$ is a commutative monoid;
if $n \notin fn(A)$ then $(\nu m)\ A \equiv_a (\nu n)\ A\{n/m\}$, and $(\nu n)\ (B|A) \equiv_a A|(\nu n)\ B$;
if $n \neq m$ then $(\nu n)\ m[A] \equiv_a m[(\nu n)\ A]$;

$(\nu n)\ \mathbf{0} \equiv_a \mathbf{0}$, $(\nu n)\ (\nu m)A \equiv_a (\nu m)\ (\nu n)A$, $!A \equiv_a A|!A$.
The operational semantics of pure ambient calculus is defined in terms of a reduction relation $\rightarrow_a$ by the following axioms and rules.

Axioms:

(In) $\quad n[in\ m.A|A']|m[B] \ \rightarrow_a\ m[n[A|A']|B]$;

(Out) $\quad m[n[out\ m.A|A']|B] \ \rightarrow_a\ n[A|A']|m[B]$;

(Open) $\quad open\ n.A|n[B] \ \rightarrow_a\ A|B$.

Rules:

(Res) $\quad \dfrac{A \ \rightarrow_a\ A'}{(\nu n)\ A \ \rightarrow_a\ (\nu n)\ A'}$; $\quad$ *(Comp)* $\quad \dfrac{A \ \rightarrow_a\ A'}{A|B \ \rightarrow_a\ A'|B}$;

(Amb) $\quad \dfrac{A \ \rightarrow_a\ A'}{n[A] \ \rightarrow_a\ n[A']}$; $\quad$ *(Struc)* $\quad \dfrac{A \equiv_a A',\ A' \ \rightarrow_a\ B',\ B' \equiv_a B}{A \ \rightarrow_a\ B}$.

Now we review the syntax and operational semantics of synchronous monadic π-calculus without sum and τ-prefix operators. Given the infinite set $\mathcal{V}$ of variables (ranged over by $x, y, \ldots$) and the infinite set $\mathcal{N}$ of channel names (ranged over by $m, n, \ldots$) we define the set $\mathcal{P}$ of π-processes (denoted $P, Q, \ldots$) as follows

$M \ ::=\ x \ \mid\ n$

$P \ ::=\ \mathbf{0} \mid P|Q \mid !P \mid \overline{M}\langle M'\rangle.P \mid M(x).P \mid [M = M'](P, Q) \mid (\nu n)\ P$

The set of free names in a π-process P is denoted by $fn(P)$. We restrict our consideration to the monadic π-calculus; therefore an expression $M(x_1, x_2, \ldots, x_n)$ (or $\overline{M}\langle M_1, M_2, \ldots, M_n\rangle$) should be read as denoting $M(x_1).M(x_2).\ldots M(x_n)$ (or, resp., $\overline{M}\langle M_1\rangle.\ \overline{M}\langle M_2\rangle.\ \ldots\ \overline{M}\langle M_n\rangle$). Accordingly, $[M_1 = N_1 \wedge M_2 = N_2].\ (P, Q)$ is just a shorthand of $[M_1 = N_1].\ ([M_2 = N_2].\ (P, Q),\ Q)$.

A π-process P is called *localized* if P has no subprocesses of the form $M(x).Q$, $[x = M](Q, R)$, or $[M = x](Q, R)$ such that x occurs in Q as the subject of an input prefix $x(y).Q'$. As demonstrated in [16], the locality property is useful for the analysis of the π-calculus terms, and for developing distributed implementations of the language.

The *structural congruence* $\equiv_\pi$ on π-processes is the least congruence satisfying the following requirements:

- $(\mathcal{P}, |, \mathbf{0})$ is a commutative monoid;
- if $n \notin fn(P)$ then $(\nu m)\ P \equiv_\pi (\nu n)\ P\{n/m\}$, and $(\nu n)\ (P|Q) \equiv_\pi P|(\nu n)\ Q$;
- $(\nu n)\ (\nu m)\ P \equiv_\pi (\nu m)\ (\nu n)\ P$, $!P \equiv_\pi P|!P$, $[M = M'](P, Q) \equiv_\pi [M' = M](P, Q)$.

The operational semantics of π-calculus is given in the form of a one-step reduction relation $\rightarrow_\pi$ by the following axioms and rules.

Axioms:

(Comm) $\quad \overline{n}\langle m\rangle.P \mid n(x).Q \ \rightarrow_\pi\ P|Q\{m/x\}$;

(Match) $\quad [n = n](P, Q) \ \rightarrow_\pi\ P$;

(MisMatch) $\quad [n = m](P, Q) \ \rightarrow_\pi\ Q$, where $n \neq m$.

Rules:

$$(Res)\quad \frac{P \ \rightarrow_\pi \ P'}{(\nu n)\ P \ \rightarrow_\pi \ (\nu n)\ P'}; \qquad (Comp)\quad \frac{P \ \rightarrow_\pi \ P'}{P|Q \ \rightarrow_\pi \ P'|Q};$$

$$(Struc)\quad \frac{A \equiv_\pi A',\ A' \ \rightarrow_\pi \ B',\ B' \equiv_\pi B}{A \ \rightarrow_\pi \ B};$$

For the sake of clarity we will widely use recursive definitions of π-processes. This does not requires an extension of the syntax, since recursion can be easy codified by means of replication $!P$ (see [14] for details).

We denote by $\rightarrow_\pi^*$ the transitive and reflexive closure of $\rightarrow_\pi$. A π-process P is called *deterministic* if for every pair of π-calculus terms Q_1, Q_2 such that $P \rightarrow_\pi Q_1$ and $P \rightarrow_\pi Q_2$, we have $Q_1 \equiv_\pi Q_2$. Otherwise P is called *non-deterministic*. Whenever P is deterministic (non-deterministic) and $P \ \rightarrow_\pi \ Q$ we will use the notation $P \mapsto_\pi Q$ (respectively $P \hookrightarrow_\pi Q$) for a one-step reduction. We will write $P \hookrightarrow_\pi^i Q$ for a chain of i non-deterministic reduction steps and $\mapsto_\pi^*$ for the transitive and reflexive closure of $\mapsto_\pi$. If $P \rightarrow_\pi^* Q$ and there are no processes R such that $Q \rightarrow_\pi R$, then we say that $P \rightarrow_\pi^* Q$ is a *terminating run of* P.

3 Encoding of the Ambient Calculus into the π-Calculus

In this section we describe a relationship between MA-processes and π-processes. This relationship may be thought of as a non-deterministic encoding of pure ambients into π-processes: with every MA-process A it associates a set $[\![A]\!]$ of π-processes. In the next section we prove the operational correspondence of the encoding by demonstrating that each π-process from $[\![A]\!]$ corresponds to the behaviour of A. The only purpose of considering $[\![\cdot]\!]$ as a relation is to simplify the proofs. When using our translation in applications, one may take a single (minimum-size) π-process from $[\![A]\!]$ as a true image of A.

A specific feature of pure ambient calculus is that an MA-process A has a spatial tree-like structure which serves a dual function. On the one hand, mobile ambients control the run of processes in A by bounding the scope of actions. On the other hand, the mobile ambients of A are acted upon by the spatial structure of A. A similar idea of decomposing the ambient process into a tree and actions is used in [12] to define the normal semantics for mobile ambients. When translating A into a π-process we separate these functions of mobile ambients. An ambient A is translated into a π-process $Proc_A = Structure_A | Ruler | Environment$ which is a composition of three π-processes $Structure_A$, $Ruler$ and $Environment$. The process $Structure_A$ is designed according to the following principles.

1. The mobile ambients and capabilities from A are represented by individual subprocesses Amb_i and Act_j; we will call these π-calculus terms *nodes*. The spatial structure of A is maintained by means of specific tree-wire subprocesses TW_k that are used for communication between nodes that represent ambients and actions. Thus, we have
$$Structure_A = Amb_1 | \dots | Amb_N | Act_1 | \dots | Act_M | TW_1 | \dots | TW_L$$
where Amb_i, Act_j, and TW_k represent generic notations for ambients, capabilities, and tree-wire structure.

2. Each subprocess Amb_i is associated with some mobile ambient $n[P]$ in A. It keeps the name n of the ambient and provides communication between the ambient and its upper and lower contexts.
3. Each subprocess Act_j is associated with some action of the form $in\ n.P$, $out\ n.P$ or $open\ n.P$ in A. It keeps the type of capability (in, out or $open$) and the name n and also provides communication between the action and its upper and lower context.
4. A subprocess TW_k is a set of wires arranged into a tree-like structure. TW_k delivers requests from its leaf nodes to the root and sends back replies from the root to the leaves. A tree-wire subprocess is intended to provide message exchange between the nodes and to accumulate consumed capabilities and dissolved ambients of A.
5. When a capability is consumed or an ambient is dissolved, a corresponding node becomes passive. A passive node rearranges to a wire and adds itself to some tree-wire. Thus, the wires of $Structure_A$ take account of computation steps generated by A. Since there are many different ways to derive the same mobile ambient term A, it may be encoded into a whole set of π-calculus terms which have the same nodes and differ only in structures of their tree-wires.

The subprocess *Ruler* does not depend on A. It is a universal π-process intended for simulating the operational semantics of mobile ambients. Here the *Ruler* is presented as a central handler. It is also possible to have *Ruler* acting as a virtual machine at each location; in this way the encoding becomes distributed. An execution of *Ruler* conforms to the following scenario.

1. *Ruler* selects from $Structure_A$ an arbitrary triple of nodes Act, Amb_{i_1} and Amb_{i_2} and collects the information about their types, names and links.
2. If (1) the type of Act is in, (2) Act is linked with Amb_{i_1}, (3) Act stores the same name as Amb_{i_2}, and (4) Amb_{i_1} and Amb_{i_2} are linked with the same node in $Structure_A$, then the subprocess $Act|Amb_{i_1}|Amb_{i_2}$ corresponds to a mobile ambient pattern $n[in\ m.P|Q]|m[R]$. In this case *Ruler* simulates the implementation of an entry instruction by switching the link of Amb_{i_1} to Amb_{i_2} and converting the node Act into a wire. This changes the entry-pattern $n[in\ m.P|Q]|m[R]$ into $m[n[P|Q]|R]$.
3. If (1) the type of Act is out, (2) Act is linked with Amb_{i_1}, (3) Act keeps the same name as Amb_{i_2}, and (4) Amb_{i_1} is linked with Amb_{i_2}, then $Act|Amb_{i_1}$ $|Amb_{i_2}$ corresponds to a pattern $m[n[out\ m.P|Q]|R$. In this case *Ruler* simulates the implementation of an exit instruction by converting the node Act into a wire, and directing the link of Amb_{i_1} to the same destination where the link of Amb_{i_2} is directed to. This changes the exit-pattern $m[n[out\ m.P|Q]|R$ into $m[R]|n[P|Q]$.
4. If (1) the type of Act is $open$, (2) Act keeps the same name as Amb_{i_1} and (3) both Act and Amb_{i_1} are linked with Amb_{i_2}, then $Act|Amb_{i_1}|Amb_{i_2}$ corresponds to a pattern $m[open\ n.P|n[Q]]$. In this case *Ruler* simulates the implementation of an open instruction by converting both Act and Amb_{i_1} into wires. This changes the open-pattern $m[open\ n.P|n[Q]]$ into $m[P|Q]$.

5. If none of the above cases holds, then *Ruler* tries another triple of nodes Act', Amb_{j_1} and Amb_{j_2} from $Structure_A$.

The subprocess *Environment* plays a role of a virtual external environment of mobile ambients. It bounds every MA-process and can not be dissolved. When *Ruler* simulates an open operation, it can select *Environment* for Amb_{i_2}.

Now we are ready to present the formal description of the processes involved in $Structure_A$, *Ruler* and *Environment* and define the encoding relation between pure mobile ambients and π-processes.

3.1 Tree-Wire Processes

A spatial structure of mobile ambients is represented by specific π-processes which are called *tree-wires*. A tree-wire is parallel composition of some basic processes which are called *wires*. A wire serves the message passing from one agent to another and back. By setting up an appropriate correspondence on the names of agents one could compose wires into any tree-like communication structure.

For $x \neq y$, we define a wire process by $W(x,y) = !\ (\nu u)\ x(v).\ \overline{y}\langle u\rangle.\ u(t).\ \overline{v}\langle t\rangle$. A wire has two parameters x and y. The parameter x is a name of a channel for communication with low-level components of a system, whereas y is a name of a channel for communication with its top-level component. The wire $W(x,y)$ receives a request $x(v)$ from one of the low-level components, re-address it to the top-level component $\overline{y}\langle u\rangle$, then receives a reply $u(t)$ via a private channel u, and finally re-address this reply $\overline{v}\langle t\rangle$ to the low-level component. It should be noticed that it may be the case when several low-level components at the same time try to communicate with a top-level component via $W(x,y)$. Then the top-level component can serve all the low-level components one by one. To avoid broadcasting every time a new private channel u is selected.

A tree-wire process $TW_{I,k}$, where I is a set of names and k is a name such that $k \notin I$, is any process which is composed of basic wires $TW_{I,k} = W(x_1,y_1)|\dots|W(x_m,y_m)$ in such a way that this parallel composition has a tree-like communication structure and provides message exchange between the set of leaf nodes I and the root k.

Definition 1. *The set of tree-wires is the minimal set of π-processes satisfying the following requirements.*

1. *Every wire $W_{x,y}$ is a tree-wire $TW_{\{x\},y}$.*
2. *If $W_{x,y}$ is a wire, $TW_{I,x}$ is a tree-wire and $y \notin I$, then the term (νx) $(W_{x,y}$ $|TW_{I,x})$ is a tree-wire $TW_{I,y}$.*
3. *If $TW_{I,y}$ and $TW_{J,y}$ are tree-wires such that $I \cap J = \emptyset$, then the term $TW_{I,y}|TW_{J,y}$ is a tree-wire $TW_{I\cup J,y}$.*

When dealing with a tree-wire $TW_{I,y}$, where $I = \{x_1, x_2, \dots, x_n\}$, we use the shorthand notation $(\nu I)(P|TW_{I,k})$ for (νx_1) $(\nu x_2)\dots(\nu x_n)$ $(P|TW_{I,k})$.

Proposition 1. *If $TW_{I,y}$ is a tree-wire, then $y \notin I$ and $fn(TW_{I,y}) = I \cup \{y\}$.*

When appending a tree-wire to a tree-wire we get again a tree-wire.

Proposition 2. *Let TW_{I_1,y_1} and TW_{I_2,y_2} be a pair of tree-wires such that $I_1 \cap I_2 = \emptyset$, $y_1 \notin I_2$, $y_2 \in I_1$. Then $(\nu y_2)\ (TW_{I_1,y_1}|TW_{I_2,y_2})$ is a tree-wire TW_{I_3,y_1}, where $I_3 = (I_1 - \{y_2\}) \cup I_2$.*

A tree-wire delivers requests from its leaf nodes to the root and replies from the root to leaf nodes.

Proposition 3. *Let $TW_{I,y}$ be a tree-wire and $x \in I$. Then the π-process*

$$(\nu v)\ \overline{x}\langle v\rangle.\ v(z).\ z|TW_{I,y}|y(u).\ \overline{u}\langle t\rangle$$

has a deterministic terminating run

$$(\nu v)\ \overline{x}\langle v\rangle.\ v(z).\ z|TW_{I,y}|y(u).\ \overline{u}\langle t\rangle \ \mapsto^* \ t|TW_{I,y}$$

3.2 Ambients and Actions

The main difficulty of encoding pure ambient processes into π-processes is that of simulating the consumption of capabilities, dissolving the boundaries of ambients, and changing the structure accordingly. In an MA-process, when an action *in* $n.P$, *out* $n.P$ or *open* $n.P$ is executed, the corresponding capability just disappears from the process (it is consumed). The same effect manifests itself when the boundary of an ambient named $m[Q]$ is dissolved. But when simulating actions and ambients as individual π-subprocesses of $Structure_A$ it is not possible just to reduce the consumed capabilities or dissolved ambients to inactive processes $\mathbf{0}$ since in this case we lose the links between the processes in $Structure_A$. The simplest way to make such processes inactive while preserving a tree-like structure of links between the remaining processes is to convert consumed capabilities and dissolved ambients into wires and use them merely to maintain links between the active processes. In this case the process $Structure_A$ corresponding to an MA-process A will depend not only on the spatial structure of A, but also on the way A is computed (the history of A). That is why instead of using deterministic encoding which maps every MA-process into a single π-process we introduce an encoding relation $\models$ which associates every MA-process A with a set of π-processes $[\![A]\!]$. Each process $Structure_A$ from $[\![A]\!]$ keeps along with the spatial structure of A the possible history of A, i.e. the way the MA-process A can be computed from the other processes. This history is represented by wires which keep track of consumed capabilities and dissolved ambients. The history of A does not influence the functionality of $Structure_A$; its only purpose is to maintain the links between active nodes of $Structure_A$.

Definition 2. *The formal description of a π-process $Node(a, n, u, d, s, l)$ is*

$$\begin{array}{lcll}
Node(a,n,u,d,s,l) & = & Reply(d,s,l)|Main(a,n,u,d,s,l) & \\
Reply(d,s,l) & = & d(y).\ [y = l] & (1)\\
 & & (\ \ d(u).\ \ \overline{s}\langle l\rangle.\ W_{d,u}\ , & (2)\\
 & & \quad \overline{y}\langle l\rangle.\ Reply(d,s,l)\ \ ) & (3)
\end{array}$$

$$\begin{array}{lll} Main(a,n,u,d,s,l) & = & (\nu v)\ (\nu w)\ (\nu k) & (4)\\ & & \overline{s}\langle v\rangle.\ v(x). & (5)\\ & & \overline{u}\langle w\rangle.\ w(l_u). & (6)\\ & & \overline{x}\langle a,n,l,u,d,l_u\rangle.\ v(y,z). & (7)\\ & & [y=l] & (8)\\ & & (\ \ \overline{d}\langle l\rangle.\ \overline{d}\langle z\rangle\ , & (9)\\ & & \quad \overline{d}\langle k\rangle.\ k(w'). & (10)\\ & & \quad \overline{s}\langle l\rangle.\ Main(a,n,z,d,s,l)\ \ ) & (11) \end{array}$$

The π-processes *Act* and *Amb* associated with actions *cap* $n.P$, where *cap* $\in \{in, out, open\}$, and ambients $n[P]$ in MA-process A have a similar arrangement. An action represented by a capability *cap* n is encoded into the π-process $Node(cap, n, up, down, s_{act}, label)$, where *up* and *down* are names of channels used for communication with upper and lower contexts of the action, s_{act} is a channel name shared by all action-type nodes of $Structure_A$ for communications with *Ruler*, and *label* is an individual label of the action in A. An ambient n is encoded into the π-process $Node(amb, n, up, down, s_{amb}, label)$, where *up*, *down* and *label* have the same meaning as above, *amb* is the key word for distinguishing ambients from capabilities, and s_{amb} is a channel name shared by all ambient-type nodes of $Structure_A$ for communications with *Ruler*.

The π-process $Node(a, n, u, d, s, l)$ is a recursive process composed of two subprocesses $Reply(d, s, l)$ and $Main(a, n, u, d, s, l)$. The subprocess *Reply* serves the dual function of providing communication with the lower context of a node (which is a set of nodes) and also of converting (if necessary) the node into a wire. The subprocess *Main* keeps the information about the node (its type, name and context) and communicates with *Ruler*. More details about the intended meaning of some fragments of *Reply* and *Main* can be found in [4].

3.3 Simulating the Operational Semantics of Pure Ambients

The π-process $Structure_A$ represents only the spatial structure of MA process A. The behaviour of A is simulated by a universal π-process *Ruler* which does not depend on A. This process has two parameters s_{act} and s_{amb} as channel names for receiving submissions from nodes corresponding to actions and ambients. The received submissions indicate the readiness of the nodes to participate in the simulation of some MA operation (entering, exiting or opening).

Definition 3. *The formal description of a π-process Ruler is as follows.*

$$\begin{array}{lll} Ruler(s_{act}, s_{amb}) & = & (\nu x_0)\ (\nu x_1)\ (\nu x_2)\ (\nu y) & (12)\\ & & s_{act}(v_0).\ s_{amb}(v_1).\ s_{amb}(v_2). & (13)\\ & & \overline{v}_0\langle x_0\rangle.x_0(t_c, n_c, l_c, u_c, d_c, ul_c). & (14)\\ & & \overline{v}_1\langle x_1\rangle.x_1(t_{a,1}, n_{a,1}, l_{a,1}, u_{a,1}, d_{a,1}, ul_{a,1}). & (15)\\ & & \overline{v}_2\langle x_2\rangle.x_2(t_{a,2}, n_{a,2}, l_{a,2}, u_{a,2}, d_{a,2}, ul_{a,2}). & \\ & & [t_c = in \wedge n_c = n_{a,2} \wedge ul_c = l_{a,1} \wedge ul_{a,1} = ul_{a,2}] & (16)\\ & & (& \\ & & \quad \overline{v}_0\langle l_c, u_c\rangle.\overline{v}_1\langle y, d_{a,2}\rangle.\overline{v}_2\langle y, u_{a,2}\rangle\ , & (17)\\ & & \quad [t_c = out \wedge n_c = n_{a,2} \wedge ul_c = l_{a,1} \wedge ul_{a,1} = l_{a,2}] & (18) \end{array}$$

$$\begin{aligned}
&(\\
&\quad \overline{v}_0\langle l_c, u_c\rangle.\overline{v}_1\langle y, u_{a,2}\rangle.\overline{v}_2\langle y, u_{a,2}\rangle\ , && (19)\\
&[t_c = open \wedge n_c = n_{a,1} \wedge ul_c = l_{a,2} \wedge ul_{a,1} = l_{a,2}] && (20)\\
&\quad (\ \ \overline{v}_0\langle l_c, u_c\rangle.\overline{v}_1\langle l_{a,1}, u_{a,1}\rangle.\overline{v}_2\langle y, u_{a,2}\rangle\ , && (21)\\
&\qquad \overline{v}_0\langle y, u_c\rangle.\overline{v}_1\langle y, u_{a,1}\rangle.\overline{v}_2\langle y, u_{a,2}\rangle\ \ ) && (22)\\
&)\\
&).s_{act}(z_0).\ s_{amb}(z_1).\ s_{amb}(z_2).\ Ruler(s_{act}, s_{amb}) && (23)
\end{aligned}$$

Environment: The environment is considered as a top-most ambient encompassing MA-process A. But unlike conventional ambients it can not be dissolved and no ambient can exit out of it. The environment plays the role of an upper context for unguarded actions and ambients; it can also participate in the simulation of open operations as Amb_2. Moreover, for the sake of uniformity it is convenient to compose an environment out of two ambient-type processes. Only one of these processes can actually participate in simulation of some MA-operation. The other is just a dummy which gives *Ruler* a possibility to operate till at least one active action-type node remains in $Structure_A$.

Definition 4. *The formal description of a π-process Environment is*

$$\begin{aligned}
Environment(env, \top, d, s_{amb}) \quad &= \quad Top(env, \top, d, s_{amb})\ | \\
&\qquad (\nu\ d')\ Top(env, \top, d', s_{amb}) && (24)\\
Top(env, \top, d, s) \quad &= \quad (\nu v)\ (\nu l)\ (\nu u)\ (\nu w) && (25)\\
&\qquad \overline{s}\langle v\rangle.\ v(x). && (26)\\
&\qquad \overline{x}\langle env, \top, l, u, d, w\rangle. && (27)\\
&\qquad v(y, z).\ \overline{s}\langle l\rangle.\ Top(env, \top, d, s) && (28)
\end{aligned}$$

3.4 The Intended Meaning of the Encoding Constructions

We present here some details about our encoding. First we briefly explain the intended meaning of some fragments of *Reply* and *Main* from the formal description of $Node(a, n, u, d, s, l)$:

(1) $Reply(d, s, l)$ can receive via the channel d either a request from the lower context of the node, or an instruction from *Main* which alters the whole node into a wire. In the former case y is evaluated into a private channel name for emitting at y the label l of the node. In the latter case the main process evaluates y into the label l (see line (9)) which does not match any private name. Therefore, after receiving the label l from *Main* the process *Reply* is reduced to the line (2), whereas after receiving a request from the lower context of the node it is reduced to the line (3).

(2) After receiving the label l of the node from the main subprocess of the node, *Reply* receives an updated channel name for communication with the upper context of the node (see line (9)), sends a synchronization message to *Ruler* indicating thus the completion of the instruction processing, and evolves into a wire which connects the lower and the upper contexts of the node. This may happen when the node becomes passive since the corresponding capability is consumed or the ambient boundary is dissolved.

(3) If *Reply* receives a request from the lower context of the node it considers the value of y as a private name of a channel and sends via this channel the label l of the node. As a consequence, the label of the node becomes available to its lower context. Afterwards *Reply* reverts to the original state.

(4) The subprocess $Main(a, n, u, d, s, l)$ uses the following private names:
   - v as a name of a channel for receiving acknowledgments and instructions from $Ruler$,
   - w as a name of a channel for receiving the label of a node that precedes our node in $Structure_A$,
   - k as an arbitrary fresh name different from the label of the node to switch *Reply* to the line (3).

(5) $Main$ begins with sending a message to *Ruler* to indicate the readiness of the node to participate in the simulation of some MA operation. *Ruler* will consider this message as a private name of a channel for communication with the node. If *Ruler* selects the node for simulating MA operation it sends via v another private name (see lines (14),(15)). This name will be used as a channel for sending to *Ruler* additional information: the name, the type and the environment of the node.

(6) The node sends a request to its upper context to know the label of preceding node in the $Structure_A$. The upper context replies via w and evaluates l_u to the label of the predecessor see (see lines (1), (3) and Proposition 3).

(7) The node sends to *Ruler* its type, name and label, the channel names for communication with the upper and lower context, the label of its predecessor in $Structure_A$. After processing this information *Ruler* replies via v to the node and informs it about its new status (active or passive) and its new upper context (see lines (17), (19), (21) and (22)). If the node does not match a pattern for simulating an MA operation or corresponds to a component of MA which is not consumed or dissolved along the operation, then *Ruler* emits at v some private name which does not match l. The node should consider this private name as an instruction to remain active. Otherwise *Ruler* evaluates y into the label l of the node and this is considered as an instruction to alter the node into a wire. In both cases *Ruler* evaluates z into a name of a new channel for communication with the upper context of the node (since the context of the node may be also changed as a result of simulation of MA operation).

(8) $Main$ checks the instruction.

(9) If the node becomes passive due to the consumption of a capability or dissolution of an ambient, then *Reply* is instructed to become a wire and receives an updated channel name for communication with its upper context. In this case the main subprocess of the node is reduced to $\mathbf{0}$.

(10) Otherwise the subprocess *Reply* is informed that the node remains active. The input action $k(w')$ is used just for the sake of uniformity.

(11) Afterwards the main subprocess sends to *Ruler* a synchronization message which indicates completion of the instruction processing, updates its upper context and reverts to the original state.

The intended meaning of the lines in the definition of $Ruler(s_{act}, s_{amb})$ is as follows.

(12) The subprocess *Ruler* uses the following private names:
- x_0, x_1, x_2 for communication with the nodes selected for simulating MA operations, and
- y for instructing the selected nodes to remain active.

(13) *Ruler* selects non-deterministically three nodes representing an action and a pair of ambients in an MA-process that could participate in the simulation of MA operation. Selection is put into effect by receiving requests via public channels s_{act} and s_{amb} (see line (5)). The first selected node is an action-type node since the request from this node is received via the channel s_{act} which is shared by action-type-nodes only. Two others are ambient-type nodes. The requests include private channel names for communication with the selected nodes.

(14) Using this private channel, *Ruler* sends a fresh channel name x_0 to the action-type node. The node considers this message as an inquiry about its characteristics (type t_c, name n_c, label l_c, channel names u_c, d_c for communication with upper and lower contexts, label ul_c of the preceding node). It delivers the required names to *Ruler* via the private channel x_0 (see line (7)).

(15) As in the case of actions (see line (14)), *Ruler* asks the selected ambient-type nodes to provide the information on the names $n_{a,1}$ and $n_{a,2}$, labels $l_{a,1}$ and $l_{a,2}$, channel names for communication with lower contexts $d_{a,1}$ and $d_{a,2}$, and labels of the preceding nodes $ul_{a,1}$ and $ul_{a,2}$ of these nodes.

(16) From this point *Ruler* begins to check which operation on MA can be executed by means of the capability represented by the selected action-type node on the ambients represented by the selected ambient-type nodes. There are three conditions to ensure that *Ruler* simulates the application of entering reduction step to the MA process. First, the selected action-type node labeled with l_c should represent an action which has the capability for entering ($t_c = in$) into the ambient named $n_{a,2}$ ($n_c = n_{a,2}$); secondly, it lies on the top level in the ambient named $n_{a,1}$ ($ul_c = l_{a,1}$); and finally, the ambients named $n_{a,1}$ and $n_{a,2}$ are siblings ($ul_{a,1} = ul_{a,2}$).

(17) The entering reduction step is simulated by changing the communication net in the π-process $Structure_A$ which represents the spatial structure of MA-process A. Since the capability represented by the selected action-type node is consumed, *Ruler* sends to this node its label l_c via the private channel. After receiving this message the node evolves into a wire (see lines (7),(8),(9),(1),(2)). Since the ambient named $n_{a,1}$ enters the sibling ambient named $n_{a,2}$, the corresponding ambient-type node has to change the upper context. *Ruler* sends to this node the channel name $d_{a,2}$ for communication with its new upper context which is the node corresponding to the ambient named $n_{a,2}$. The upper context of the node corresponding to the ambient named $n_{a,2}$ remains the same, and *Ruler* acknowledges this by communicating back the value $u_{a,2}$. A private name y which does not match

the labels $l_{a,1}$ and $l_{a,2}$ is sent to the ambient-type nodes as an instruction to remain active (see lines (7),(8),(10),(11),(1),(3)).

(18) If the selected nodes do not match an entry-pattern, then *Ruler* checks for an exit-pattern. There are three conditions to ensure that *Ruler* simulates the application of exiting reduction step to the MA process. First, the selected action-type node labeled with l_c should represent an action which has the capability for exiting ($t_c = out$) out of the ambient named $n_{a,2}$ ($n_c = n_{a,2}$); secondly, it lies on the top level in the ambient named $n_{a,1}$ ($ul_c = l_{a,1}$); and finally, the ambient named $n_{a,1}$ lies on the top level of the ambient named $n_{a,2}$ ($ul_{a,1} = l_{a,2}$).

(19) Since the capability represented by the selected action-type node is consumed, *Ruler* sends to this node its label l_c to evolve the node into a wire (see lines (7),(8),(9), (1),(2)). Since the ambient named $n_{a,1}$ is transformed into a sibling of the ambient named $n_{a,2}$, it changes the upper context from $u_{a,1}$ to $u_{a,2}$. The upper context of the node corresponding to the ambient named $n_{a,2}$ remains the same. *Ruler* sends a private name y which does not match the labels $l_{a,1}$ and $l_{a,2}$ to instruct the ambient-type nodes to remain active (see lines (7),(8),(10),(11),(1),(3)).

(20) If the selected nodes do not match exit-pattern, then *Ruler* checks for an open-pattern. If the selected action-type node labeled with l_c represents an action which has the capability for dissolving the boundary ($t_c = open$) of the ambient named $n_{a,1}$ ($n_c = n_{a,1}$), lies on the top level in the ambient named $n_{a,2}$ ($ul_c = l_{a,2}$), and the ambient named $n_{a,1}$ also lies on the top level of the ambient named $n_{a,2}$ ($ul_{a,1} = l_{a,2}$), then *Ruler* simulates the application of an opening reduction step to the MA process.

(21) To simulate an opening MA-operation, *Ruler* sends l_c to the action-type node and $l_{a,1}$ to the ambient-type node named $n_{a,1}$ to evolve these nodes into wires (see lines (7),(8),(9),(1),(2)) since the capability is consumed and the boundary of the ambient is dissolved. *Ruler* sends a private name y which does not match the label $l_{a,2}$ to instruct the ambient-type node named $n_{a,2}$ to remain active (see lines (7),(8),(10),(11),(1),(3)).

(22) If the selected nodes do not match any pattern, then *Ruler* informs them via private channels to remain active and to keep their channels for communication with upper contexts unchanged.

(23) After this *Ruler* waits till the nodes participated in this round of simulation send their synchronization messages (the labels) to indicate that instructions sent to them (see lines (17), (19), (21), or (22)) are performed (see lines (2) and (11)), reverts to the initial state and tries another triple of nodes.

Finally, we comment briefly on the intended meaning for *Environment*.

(24) The environment is composed of two processes *Top*. They have the same functionality, but only the first one has a global name for communication with its lower context (top-level actions and ambients encompassed by the environment). Nevertheless both processes can communicate with *Ruler* via the channel s_{amb} shared by ambient-type nodes. The name $\top$ is an arbitrary

name which is different from any free name in an ambient encompassed by the environment.

(25) A process Top uses the following private names:
- v as a name of a channel for receiving acknowledgments and instructions from *Ruler*,
- l, u, w as dummy names that are used only for the sake of uniformity in communications with *Ruler* (they stand for the label, upper context and label of the predecessor of the node that are of no importance for the environment).

(26) The environment processes begin with sending to *Ruler* their requests for participation in the simulation of MA operations. When participation is granted Top receives a private channel name for communication with *Ruler* (see line (15)).

(27) Using this channel Top sends to *Ruler* the information about its type (a keyword *env*), name (it should be different from any name in $Structure_A$), channel names for communication with its upper context (since it does not exist any private name is possible) and lower context (d), the label of the preceding node (it does not exist also and Top uses any private name for this purpose).

(28) As any other node participating in the simulation of MA operation as seen in line (7), Top receives a pair of names (y and z) which are interpreted as instructions for changing its status and updating its context. But since the environment can not be dissolved and it has no upper context, these names do not affect on its functionality. This input is used only for the sake of uniformity which gives *Ruler* a possibility not to distinguish Top as a specific node. Therefore, Top just acknowledges the receipt of these names by sending a synchronization message to *Ruler* and reverts to the original state.

3.5 Encoding of Pure Ambients into π-Processes

The encoding of pure ambients in π-processes is defined in terms of two relations $\models_0$ and $\models$. We use $\models_0$ for constructing $Structure_A$ corresponding to MA process A and $\models$ for constructing the ultimate π-process out of $Structure_A$, *Ruler* and *Environment*.

Definition 5. *The encoding relation* $\models_0$ *is defined inductively by the following axioms and rules. In every pair* $[P, k]$ *to the right of* $\models_0$ *the second component* k *stands for the free channel name used in the* π*-process* P *for communication with its upper context.*

Axioms:

Ax1 (Simple Inactivity) $\mathbf{0} \models_0 [\mathbf{0}, k]$, where $k \in \mathcal{N}$;

Ax2 (Tree-wire) $\mathbf{0} \models_0 [(\nu I)\ TW_{I,k}, k]$, where $I \subset \mathcal{N},\ k \in \mathcal{N}$;

Rules:

R1 (Add tree-wire) $$\frac{A \models_0 [P, k]}{A \models_0 [(\nu\ I)\ (W_{I,m}|P), m]} \quad ,$$

where $k \in I, fn(P) \cap I \subseteq \{k\},\ m \notin fn(P) \cup I$;

R2 (Composition) $\dfrac{A_1 \models_0 [P_1, k]\ ,\ A_2 \models_0 [P_2, k]}{A_1 | A_2 \models_0 [P_1 | P_2, k]}$;

R3 (Restriction) $\dfrac{A \models_0 [P, k]}{(\nu n)\ A \models_0 [(\nu\ n)\ P, k]}$; R4 (Replication) $\dfrac{A \models_0 [P, k]}{!A \models_0 [!P, k]}$;

R5 (Action) $\dfrac{A \models_0 [P, k]}{cap\ n.\ A \models_0 [(\nu\ k)\ (\nu\ l)(Node(cap, n, m, k, s_{act}, l)|P), m]}$,

where $cap \in \{in, out, open\},\ l, m \notin fn(P) \cup \{n\}$;

R6 (Ambient) $\dfrac{A \models_0 [P, k]}{n[A] \models_0 [(\nu\ k)\ (\nu\ l)\ (Node(amb, n, m, k, s_{amb}, l)|P), m]}$,

where $l, m \notin fn(P) \cup \{n\}$.

The encoding relation $\models$ is defined by the single rule

R0 (MA-to-π)

$$\frac{A \models_0 [Structure_A, k]}{A \models (\nu\ \Sigma)\ (Structure_A | Ruler(s_{act}, s_{amb}) | Environment(env, \top, k, s_{amb}))}$$

where $\nu\Sigma$ stands for the prefix

$$(\nu\ in)\ (\nu\ out)\ (\nu\ open)\ (\nu\ amb)\ (\nu\ env)\ (\nu\ s_{act})\ (\nu\ s_{amb})\ (\nu\ \top)\ (\nu\ k)\ ,$$

and $\top$ is any name from $\mathcal{N} - fn(A)$.

Proposition 4.

1. *Let A, B be two MA-processes such that $A \equiv_a B$, and $A \models P$. Then there exists a derivation $B \models Q$ such that $P \equiv_\pi Q$.*
2. *If $A \models P$, then $fn(A) = fn(P)$.*

4 Operational Correspondence

In this section we will demonstrate that the encoding of pure ambients into π-calculus is complete and sound. By completeness we mean that any π-process P associated with an MA-process A through the encoding relation $A \models P$ admits only those π-calculus reductions $P \rightarrow_\pi^* P'$ that can be interpreted in terms of pure ambient reductions $A \rightarrow_a A'$ such that $A' \models P'$. Soundness means that any reduction $A \rightarrow_a A'$ corresponds to some chain of π-calculus reductions $P \rightarrow_\pi^* P'$ of P such that $A' \models P'$. Thus, we may speak of a homomorphic embedding of pure mobile ambients into π-calculus.

Theorem 1 (Completeness). *Let A_0, A_1 be MA-processes and P_0 be a π-process such that $A_0 \rightarrow_a A_1$ and $A_0 \models P_0$. Then there exists a chain of π-calculus reduction steps*

$$P_0 \hookrightarrow_\pi^3 P_1' \mapsto_\pi^* P_1$$

such that $A_1 \models P_1$.

The proof follows straightforward from the description of processes $Main$, *Reply*, *Ruler*, Top and Proposition 3. The only non-deterministic steps in the reduction $P_0 \rightarrow_\pi^* P_1$ are three communications steps when *Ruler* selects nodes representing a capability and a pair of ambients for simulating a reduction step $A_0 \rightarrow_a A_1$. Afterwards the reduction of P_0 is completely deterministic until all communication actions in the the bodies of subprocesses *Ruler*, $Main$ and *Reply* are executed to an end.

Theorem 2 (Soundness). *Let A_0 be an MA-process and P_0 be a π-process such that $A_0 \models P_0$. Let $P_0 \rightarrow_\pi^* P$ be a chain of π-calculus reduction steps. Then there exist an integer N, $0 \le N \le 2$, a sequence of π-calculus terms $P'_1, P_1, P'_2, P_2, \dots, P'_n, P_n$ and a sequence of pure ambient terms $A_1, A_2, \dots, A_n$ such that the following conditions hold*

1. *$P \hookrightarrow_\pi^N P'_n \mapsto_\pi^* P_n$;*
2. *The chain of π-calculus reductions $P_0 \rightarrow_\pi^* P \rightarrow_\pi^* P_n$ can be partitioned as follows:*
   $P_0 \hookrightarrow_\pi^3 P'_1 \mapsto_\pi^* P_1 \hookrightarrow_\pi^3 P'_2 \mapsto_\pi^* P_2 \hookrightarrow_\pi^3 \cdots \hookrightarrow_\pi^3 P'_{n-1} \mapsto_\pi^* P_{n-1} \hookrightarrow_\pi^{3-N} P \hookrightarrow_\pi^N P'_n \mapsto_\pi^* P_n$
   such that
   (a) $A_i \models P_i$ for every i, $0 \le i \le n$;
   (b) for every i, $0 \le i < n$, either $A_i \equiv_a A_{i+1}$ or $A_i \rightarrow_a A_{i+1}$.

The intended meaning of this theorem is as follows. Suppose that a π-process P_0 encodes an MA-process A_0, and it can be reduced to a π-process P. Then either P encodes an MA-process A_n, or P is in an "intermediate" form and it can be further reduced to a π-process P_n which encodes A_n. In the latter case, the reduction of P to P_n is a composition of

– N non-deterministic reduction steps $P \hookrightarrow_\pi^N P'_n$, where $0 \le N \le 2$; these steps complete, if necessary, a non-deterministic selection of nodes representing an action and ambients in MA-process (see Section 3.4.(13)), and
– a finite number of deterministic reduction steps $P'_n \mapsto_\pi^* P_n$ corresponding to the interaction between the *Ruler* and the selected processes $Node$ (see Section 3.4. (14)-(23)).

When reduction of P to P_n is completed, the whole chain of π-calculus reductions $P_0 \rightarrow_\pi^* P \rightarrow_\pi^* P_n$ becomes a step-by-step simulation of some MA computation $A_0 \rightarrow_a^* A_n$.

The proof of this theorem is by induction on the number of non-deterministic steps $\hookrightarrow_\pi$ in a reduction of P_0. Each triple of non-deterministic steps in such reduction is followed by a chain of deterministic reduction steps that either simulate the execution of some MA-reduction step if the selected nodes in a π-process P_i comply to one of MA reduction rules, or restore P_i otherwise.

We may note that our translation has diverging reductions whenever the selected nodes do not conform any MA reduction rule. In this case we may obtain an infinite chain $P \hookrightarrow_\pi^3 P' \mapsto_\pi^* P \hookrightarrow_\pi^3 P' \mapsto_\pi^* P \hookrightarrow_\pi^3 \dots$

Proposition 5. *If A is an MA-process and P is a π-process such that $A \models P$, then we can see that P is a localized π-calculus term.*

5 Conclusion

Both ambient calculus and π-calculus are very expressive and allow the natural embedding into them of many other universal models of computations such as Turing machine and λ-calculus. In [2] and [3], the π-calculus was encoded into the full ambient calculus; [19] has proposed an encoding of the π-calculus only into the pure ambients calculus. In this paper we provide an encoding of the pure mobile ambients into the localized sum-free synchronous monadic π-calculus. The encoding uses the matching and mismatching operators. The encoding is close to an interpreter, and it can be used for implementing the ambient calculus. We prove the completeness and soundness of our encoding by showing that each reduction step of an MA process can be uniformly simulated by chains of π-calculus reductions of corresponding the π-terms. The encoding gives a possibility to analyze some properties of mobile ambients by means of static analysis and congruence-checking machinery developed for the π-calculus. The fact that we restrict ourselves with localized and sum-free π-terms alleviates substantially the analysis. Moreover, our encoding does not involve any sophisticated structures that can affect the precision of such analysis.

The encoding can be extended to a full ambient calculus, adding a communication channel per ambient. This implies a "merging" of channels when an ambient is opened; we may use the same mechanism: the *Ruler* randomly selects an input and an output, checks if they belong to the same ambient, and performs communication. On the other hand, it is worth noticing that our encoding is slightly more general in the sense that the target language is even simpler. We can use the asynchronous π-calculus, which is even simpler than the synchronous π-calculus, by using the standard encoding [10].

We consider our encoding in combination with the earlier results of [2,3,19] as the first efforts towards bridging the gap between the calculus of mobile ambients and the π-calculus. For the sake of clarity we present here the most simple (sequential) variant of the encoding: the unique *Ruler* serves the whole system. It can readily be imagined that the encoding could be made more advanced by combining every π-process *Node* corresponding to an ambient with its own interpreter *Ruler*. In this case a distributed interaction between the components of a system can be achieved, and the encoding becomes compositional. To study the behavioural properties of mobile ambients that are preserved by such an advanced encoding would be our next step. The ultimate aim is to build a fully abstract translation which preserves behavioural equalities between processes, such as reduction barbed congruence [13,17]. If such a translation should be obtained, it provides a sound basis for a uniform framework of the theory of mobile computations.

References

1. L. Brodo, P. Degano, C. Priami. Reflecting mobile ambients into the π-calculus. In *Proceeedings of Global Computing'03*, LNCS, **2872**, 2003, p.25–56.
2. L. Cardelli, A. Gordon. Mobile Ambients, *Theoretical Computer Science*, **240**, N 1, 2000, p.177–213.
3. W. Charatonik, A. Gordon, J.-M. Talbot. Finite-control mobile ambients. In *Proceedings of ESOP'02*, LNCS, **2305**, 2002, p.295–313.
4. G. Ciobanu, V.A. Zakharov. Embedding mobile ambients into the π-calculus. "A.I.Cuza" University of Iasi, Faculty of Computer Science, Technical Report TR 05-07. Available from http://thor.info.uaic.ro/ tr/tr05-07.pdf
5. S. Dal-Zilio. Mobile processes: a commented bibliography. In *Proceedings of MOVEP 2000*, LNCS, **2067**, 2000, p. 206–222.
6. G. Ferrari, S. Gnesi, U. Montanari, N. Pistore, R. Gioia. Verifying mobile processes in the HAL environment. In *Proceedings of CAV'98*, LNCS, **1427**, 1998, p. 511–515.
7. C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, ACM Press, 1996, p. 372-385.
8. C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, D. Remy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, LNCS **1119**, 1996, p. 406–421
9. C. Fournet, J.J. Levy, A. Schmitt. An Asynchronous Distributed Implementation of Mobile Ambients. In *Proceedings of the International IFIP Conference TCS*, LNCS, **1872**, 2000, p. 348–364.
10. K. Honda, M. Tokoro. An object calculus for asynchronous communication. In *Proceedings ECOP'91*, LNCS, **512**, 1991, p.133–147.
11. F. Levi, D. Sangiorgi. Controlling inference in ambients. In *POPL'00*, ACM Press, 2000, p. 372-385.
12. F. Levi, S. Maffeis. An abstract interpretation framework for mobile ambients. In *Proceedings of SAS'01*, LNCS, **2126**, 2001, p.395–411.
13. M. Merro, Z. Nardelli. Behavioral theory for mobile ambients. *Journal of the ACM*, **52**, N 6, 2005, p. 961–1023.
14. R. Milner. *Communicating and mobile systems: the π-calculus*, Cambridge University Press, 1999.
15. B.C. Pierce, D.N. Turner. Pict: a programming language based on the pi-calculus. In *"Proof, Language and Interaction: Essays in Honour of Robin Milner"*, MIT Press, 1997.
16. D. Sangiorgi, D. Walker, *The π-calculus: a theory of mobile processes*, Cambridge University Press, 2001.
17. D. Sangiorgi, D. Walker. On barbed equivalence in π-calculus. In *Proceedings of CONCUR'01*, LNCS, **2154**, 2001, p. 292–304.
18. B. Victor, F. Moller. The Mobility Workbench a tool for the π- calculus. In *Proceedings of CAV'94*, LNCS, **818**, 1994, p. 428-440.
19. P. Zimmer, On the expressiveness of pure safe ambients. *Mathematical Structures in Computer Science*, **13**, N 5, 2003, p. 721–770.

Characterizations of CD Grammar Systems Working in Competence Mode by Multicounter Machine Resources

Liliana Cojocaru

Rovira i Virgili University of Tarragona
Research Group on Mathematical Linguistics
Pl. Imperial Tarraco 1, 43005, Spain
liliana.cojocaru@estudiants.urv.cat

Abstract. In this paper we focus on characterizations of *cooperating distributed grammar systems* (henceforth CDGSs) working in $\{\leq k, = k, \geq k\}$-competence mode, $k \geq 1$, defined in [1], by means of time, space and several other resources of a multicounter machine such as number of counters, reversals or 0-tests. We show that CDGSs working in $\{\leq k, = k, \geq k\}$-competence mode, $k \geq 1$, can be simulated by *one-way nondeterministic multicounter machines* in linear time and space, with a linear-bounded number of reversals. If we impose restrictions on the capacity of each counter to perform 0-testings, we find that CDGSs working in $\{\leq 1, = 1\} \cup \{\geq k | k \geq 1\}$-competence mode can be simulated by *one-way partially blind multicounter machines* in quasirealtime. It is known from [8] that quasirealtime partially blind multicounter machines accept the family of *Petri net languages*. Consequently, various decision problems for partially blind multicounter machines are decidable. With respect to them, several decidability results, for CDGSs working in competence mode, grammars and systems with regulated rewriting, that emerge from the above simulations are presented, too.

1 Introduction

In this paper we deal with *cooperating distributed grammar systems* that use a special type of cooperation protocol based on the level of competence of a component to rewrite a certain number of nonterminals occurring in the underlying sentential form. The new protocol has been recently introduced in [1].

Informally, a component is $\leq k$*-competent*, $= k$*-competent* or $\geq k$*-competent*, $k \geq 1$, on a certain sentential form if it is able to rewrite at *most*, *exactly*, or at *least* k distinct nonterminals occurring in the sentential form, respectively. Once a component starts to be f-competent on a sentential form, $f \in \{\leq k, = k, \geq k\}$, $k \geq 1$, it has to continue the derivation as long as it is f-competent on the newest sentential form. The formal definition of competence-based protocols is provided in Section 2.

In [1] it is proved that competence-based protocols lead CDGSs to work at least at the level of *forbidding random context grammars* for the case of

$\{\leq 1, = 1\}$-competence mode, at the level of *ET0L systems* for ≥ 1-competence mode, at the level of *random context ETOL systems* for $\geq k$-competence mode, $k \geq 2$, and finally they are as powerful as *random context grammars* for the case of $\{\leq k, = k\}$-competence mode, $k \geq 2$, i.e., they characterize the class of *recursively enumerable languages.*

In Section 3 we give several characterizations of CDGSs that work in $\{\leq k, = k, \geq k\}$-competence mode, $k \geq 1$, by means of time, space and several other resources of a multicounter machine such as number of counters, number of reversals or 0-tests. For $f \in \{\leq k, = k, \geq k\}$, $k \geq 1$, we present a simulation of CDGSs working in f-competence mode by one-way nondeterministic *multicounter machines* in linear time and linear space, with a linear-bounded number of reversals. With respect to this simulation we give an equivalent result for *nondeterministic Turing machines.* For CDGSs working in $\{\leq 1, = 1\}$-competence mode or in $\geq k$-competence mode, $k \geq 1$, the simulation is done in quasirealtime by one-way partially blind multicounter machines.

We conclude in Section 4 with several decidability results for CDGSs working in competence mode, grammars and systems with regulated rewriting, that emerge from the above simulations.

2 CD Grammar Systems Working in Competence Mode

The aim of this section is to introduce the basic notions and notations that concern CDGSs that work in competence mode. For basic results on CDGSs the reader is referred to [3] and [1]. Let X be a finite set of alphabet letters, $|X|$ be the cardinal number of X, and X^* be the set of all strings over the alphabet X. The empty word is denoted by λ, the number of occurrences of the symbol $a \in X$, in the string $w \in X^*$, is denoted by $|w|_a$, and the length of a string $w \in X^*$ is denoted by $||w||$.

Definition 1. A *cooperating distributed grammar system* of degree s, is an $(s+3)$-tuple $\Gamma = (N, T, \alpha, P_1, \dots, P_s)$, where N and T are disjoint alphabets, the *nonterminal* and the *terminal* alphabet, respectively. $P_1, \dots, P_s$ are finite sets of context-free rules over $N \times (N \cup T)^*$, called the system components, and $\alpha \in (N \cup T)^+$ is the system axiom.

Definition 2. Let $\Gamma = (N, T, \alpha, P_1, \dots, P_s)$ be a CDGS, dom(P_i)=$\{A \in N | A \to z \in P_i\}$ be the domain of the component P_i, $1 \leq i \leq s$, and alph$_N$(x)=$\{A \in N | |x|_A > 0\}$. We say that P_i is *≤k-competent*, *=k-competent* or *≥k-competent* on a sentential form x, $x \in (N \cup T)^*$, if and only if $|\text{alph}_N(\text{x}) \cap \text{dom}(P_i)|$ is *less* than k, *equal* with k, or *greater* than k, respectively.

Informally, we say that a component P_i has the level of competence $\leq k$, $= k$ or $\geq k$, $k \geq 1$, on a certain sentential form x, $x \in (N \cup T)^*$, if the component P_i is *≤k-competent*, *=k-competent*, *≥k-competent* on a sentential form x, $x \in (N \cup T)^*$, i.e., it is able to rewrite at *most*, *exactly*, or at *least* k distinct nonterminals occurring in the sentential form, respectively. We denote by clev$_i$(x) the level of competence of P_i on x. The cooperation protocol, based on the capability of a component P_i to be *≤k-competent*, *=k-competent*, *≥k-competent* on $x \in (N \cup T)^*$, is defined in [1] as follows.

Definition 3. Let $\Gamma = (N, T, \alpha, P_1, \ldots, P_s)$ be a CDGS, $x, y \in (N \cup T)^*$, and $1 \leq i \leq s$.
1. We say that y is derived from x in the *≤k-competence mode*, denoted by $x \Rightarrow_i^{\leq k\text{-comp.}} y$, iff there is a derivation $x = x_0 \Rightarrow_i x_1 ... \Rightarrow_i x_{m-1} \Rightarrow_i x_m = y$ such that (a) $\text{clev}_i(x_j) \leq k$ for $0 \leq j < m$ and (i) $\text{clev}_i(x_m) = 0$ or (ii) $y \in T^*$,
(b) $\text{clev}_i(x_j) \leq k$ for $0 \leq j < m$ and $\text{clev}_i(x_m) > k$;
2. We say that y is derived from x in the *=k-competence mode*, denoted by $x \Rightarrow_i^{=k\text{-comp.}} y$, iff there is a derivation $x = x_0 \Rightarrow_i x_1 ... \Rightarrow_i x_{m-1} \Rightarrow_i x_m = y$ such that (a) $\text{clev}_i(x_j) = k$ for $0 \leq j < m$ and $\text{clev}_i(x_m) \neq k$ or
(b) $\text{clev}_i(x_0) = k$, $\text{clev}_i(x_j) \leq k$ for $1 \leq j \leq m$, and $y \in T^*$;
3. We have $x \Rightarrow_i^{\geq k\text{-comp.}} y$ iff there is a derivation $x = x_0 \Rightarrow_i x_1 ... \Rightarrow_i x_{m-1} \Rightarrow_i x_m = y$ such that (a) $\text{clev}_i(x_j) \geq k$ for $0 \leq j < m$ and $\text{clev}_i(x_m) < k$ or
(b) $\text{clev}_i(x_0) \geq k$, and $y \in T^*$.

Let $M = \{\leq k\text{-comp.}, = k\text{-comp.}, \geq k\text{-comp.} | k \geq 1\}$, and let $\Rightarrow^f$ denote $\Rightarrow_i^f$, for $1 \leq i \leq s$, $f \in M$. We denote by $\Rightarrow^{*f}$ the reflexive and transitive closure of $\Rightarrow^f$.

Definition 4. The *language generated* by Γ in f-mode of derivation, $f \in M$, is

$$L_f(\Gamma) = \{w \in T^* | \alpha \Rightarrow^{*f} w\}.$$

The family of languages generated by CDGSs in f-mode, $f \in M$, is denoted by $\mathcal{L}(\text{CD, CF}, f)$.

Definition 5. Let $\Gamma = (N, T, S, P_1, \ldots, P_s)$ be a CDGS, and D be a derivation in f-mode, $D : S = w_0 \Rightarrow_{P_{i_1}}^{=n_1} w_1 \Rightarrow_{P_{i_2}}^{=n_2} w_2 ... \Rightarrow_{P_{i_m}}^{=n_m} w_m = w$, $f \in M$, where P_{i_j} performs n_j steps, $1 \leq i_j \leq s$, $1 \leq j \leq m$. Let w be an arbitrary word in $L_f(\Gamma)$. The *Szilard word* of w associated with the terminal derivation D, is defined as: $\gamma_w(D) = i_1^{n_1} i_2^{n_2} ... i_m^{n_m}$, where i_j is the label of the component P_{i_j}, and n_j is the number of times the component P_{i_j} brings its contributions on the sentential form when it is activated in the f-mode. The *Szilard language* associated with the derivation D in Γ is defined as: $Sz(\Gamma, f) = \{\gamma_w(D) | w \in L_f(\Gamma)$ and D a terminal derivation in f-mode, $f \in M\}$.

We denote by $SZ(\text{CD, CF}, f)$ the family of Szilard languages $Sz(\Gamma, f)$ associated to CDGSs with context-free components working in f-mode, $f \in M$.

3 On the Time, Space and Reversal Complexity

In this section we focus on the characterization of CDGSs working in $\{\leq k, = k, \geq k\}$-comp.-mode, $k \geq 1$, by means of time, space, number of counters, number of reversals or 0-tests of a multicounter machine.

3.1 Multicounter Machines

Informally, a *multicounter machine*, abbreviated MC, is an accepting device composed of a finite state control, an input head, an input tape, and a finite number of semi-infinite storage tapes that function as counters capable to store any integer. If the input head is allowed to move only to the right the machine is a *one-way* (or on-line) *multicounter*, otherwise is a *2-way* (or off-line) *multicounter* machine. Next we deal only with one-way MC machines.

If X is the input alphabet, then an input for a one-way MC is a string of the form $\natural w \natural$, where $\natural$ is the input delimiter and $w \in (X - \{\natural\})^*$. The machine starts in an initial state, with the input head placed on the left delimiter with all counters set on zero. Each time the input head reads a symbol from the input tape, on a particular state q, the machine checks the configuration of each counter, increments each counter by +1, -1, or 0, moves the head on the right and changes the state q into q'. The input is accepted if the machine gets into a final state, having the input head placed on the right delimiter and all counters empty. In the case that the machine has at most one choice of action on any configuration, we have a *deterministic multicounter*. Otherwise, the machine is said to be *nondeterministic*. Each choice of action is defined by a *transition function*, denoted as δ.

The most important resources of multicounter machines are the *time*, i.e., the number of steps performed by the MC machine during the computation, the *space*, i.e., the sum of the maximum absolute values of the contents of each counter during the computation, the *number of counters*, the *number of reversals*, i.e., the number of alternations from increasing to decreasing and vice-versa performed during the computation, and the *number of 0-tests*.

For the formal definition of MC machines and other complexity and hierarchy results on counter languages the reader is referred to [6] and [7].

Theorem 1. *The Szilard language $Sz(\Gamma,f)$ attached to a CDGS Γ, working in f-mode, $f \in M$, is recognizable by one-way nondeterministic m-counter machine, in linear time and linear space, within linear bounded reversals and linear 0-tests, where m is the number of the system nonterminals.*

Proof. Let $\Gamma = (N, T, \alpha, P_1, \ldots, P_s)$ be a CDGS working in f-mode, $f \in M$. Next we give the demonstration for $=k$-comp.-mode, $k \geq 1$. Let us consider the ordered set of nonterminals $N = \{A_1, A_2, ..., A_m\}$, $m \geq k$, and let $P = \{1, 2, ..., s\}$ be the set of labels attached to each component in the system. Let $\mathcal{M}$ be the one-way nondeterministic m-counter machine, having as alphabet the set P and as input a word $\gamma_w(D) \in P^*$ of length n. Each counter corresponds to a nonterminal in N. For each $1 \leq i \leq m$, we denote by C_i the counter associated with A_i. In order to keep control of the constant number of nonterminals occurring in the axiom α we will consider an initial state of the form $q_\alpha = q_{[x_1]\ldots[x_{m-1}][x_m]}$, in which[1] $x_i = |\alpha|_{A_i}$, for $1 \leq i \leq m$.

The machine starts in q_α with all counters set on zero, having the input head placed on the first delimiter $\natural$. From the state q_α the machine goes into the

[1] Within a cell each x_i, $1 \leq i \leq m$, is a symbol, while "outside" the cell x_i is a number.

state[2] $q_{[x_1-1]\ldots[x_m]}$, by reading λ and increasing the value of the counter C_1 by +1. The operation continues until the system reaches the state $q_{[0][x_2]\ldots[x_m]}$, using x_1 number of transitions through which the first counter is increased by $x_1 = |\alpha|_{A_1}$ times. When the first cell of the state $q_{[x_1]\ldots[x_m]}$ had been "reduced" to 0 the same operation continues with the second cell "storing" the value x_2, by increasing the second counter with the value $x_2 = |\alpha|_{A_2}$, and so on. These operations end when all the m cells from the initial state had been reduced to 0 and each counter C_i had been increased by $x_i = |\alpha|_{A_i}$, $1 \leq i \leq m$. All the transitions[3] proceed by reading the empty string. Let us denote as q_0 the state $q_{[0]\ldots[0][0]}$. From now on the multicounter machine simulates the work of the CDGS in $=k$-comp.-mode, as follows.

Step 1. Let us suppose that the first symbol from $\gamma_w(D)$ is the label e corresponding to the component P_e, such that $|\text{dom}(P_e)| \geq k$. Firstly $\mathcal{M}$ checks whether P_e is $=k$-competent on α, i.e., $|\text{dom}(P_e) \cap \text{alph}_N(\alpha)| = k$, where $\text{alph}_N(\alpha) = \{A \in N \,|\, |\alpha|_A > 0\}$. This is done by checking in the state q_0, through the δ function that defines $\mathcal{M}$, whether reading e there are exactly k counters, that correspond to the k nonterminals from $\text{dom}(P_e) \cap \text{alph}_N(\alpha)$, that have positive values. If this condition does not hold $\gamma_w(D)$ is rejected. This can be done by not defining the δ function when reading e in other configurations than those mentioned before. ♢

Step 2. Let us consider that each nonterminal $A_l \in \text{dom}(P_e) \cap \text{alph}_N(\alpha)$ is rewritten by a rule of the form $A_l \to rhs(A_l)$. Then C_l is decreased by 1, and the value of each counter C_i corresponding to A_i occurring in $rhs(A_l)$ is increased by 1, as many times as the letter A_i occurs in $rhs(A_l)$. ♢

Note that, when a component P_e is $=k$-competent on a sentential form x, $\mathcal{M}$ nondeterministically chooses which nonterminal from x is rewritten by P_e, and which rule that rewrites the chosen nonterminal is applied. In order to emphasize that the component P_e has been activated[4] we let the machine $\mathcal{M}$ ends the procedure from Step 2 in state q_e.

For the next label in $\gamma_w(D)$, denoted as e', the machine checks, in state q_e by reading λ, whether the former component P_e, is still $=k$-competent on the newest sentential form. This is done analogously as in Step 1, taking x instead of α. If P_e is $=k$-competent on x and $e = e'$, then the computation continues as in Step 2. If P_e is $=k$-competent on x, and $e \neq e'$, then $\gamma_w(D)$ is rejected.

If $P_{e'}$ is $=k$-competent and P_e is not, the same operations on counters, described at Step 2, are nondeterministically performed for the component $P_{e'}$. The procedure ends in the state $q_{e'}$ in order to mark that the component $P_{e'}$ became active. If $P_{e'}$ is not $=k$-competent then $\gamma_w(D)$ is rejected.

[2] Here by $[x_1 - 1]$ we understand a notation and not the subtraction operation.

[3] These transitions cannot "go below zero", and due to this for a cell marked by 0 there is no transition that modifies this cell.

[4] Following the definition of the $=k$-comp.-mode this grammar should be active as long as it is $=k$-competent, even if several other components are $=k$-competent in the newest sentential form, too.

Finally, $\gamma_w(D)$ is accepted if and only if the machine ends the computation with the input head placed on the last delimiter $\natural$, with all counters set on zero.

Clearly, the number of 0-tests is linearly related to the length of the Szilard word $\gamma_w(D)$. Each time the input head reads a label from $\gamma_w(D)$, decreases one counter and increases a constant number of several other counters, therefore the number of reversals cannot be constant or more than linear.

Let us denote as $max(C_i)$ the maximum absolute value stored in the counter C_i, then $max(C_i) \leq |\alpha|_{A_i} + l_i$, where l_i is the number of times the counter C_i is increased by 1 during the computation. It is clear that each l_i is upper bounded by $n/2$. Therefore, $\sum_{i=1}^{m} max(C_i) \leq |\alpha| + mn/2$. These remarks justify our assertion that the simulation is done in linear time and linear space, within $\mathcal{O}(n)$ reversals and $\mathcal{O}(n)$ 0-tests, in terms of the length of the Szilard word.

For the other cases the simulation works analogously without major improvements. For the $\leq k$-comp.-mode, for instance, when checking the $\leq k$-competence of the component P_e, we should allow to $\mathcal{M}$ nondeterministically verify whether at most k counters corresponding to nonterminals from $\text{dom}(P_e) \cap \text{alph}_N(\alpha)$, have positive values. For the next label e' from $\gamma_w(D)$, when checking the $\leq k$-competence of $P_{e'}$ and the non $\leq k$-competence of P_e on the newest sentential form x, where P_e has been previously applied, we check whether there are at most k positive counters corresponding to at most k nonterminals from $\text{dom}(P_{e'})$ $\cap$ $\text{alph}_N(x)$, and all counters corresponding to nonterminals from $\text{dom}(P_e) \cap \text{alph}_N(x)$ are zero, where $\text{alph}_N(\text{x})=\{A \in N||x|_A > 0\}$. $\square$

In [2], and [6] several trade-offs between time, space and number of reversals of a MC machine, as well as their relations to Turing machines time and space are presented. Accordingly, from Theorem 1 we have the following result.

Theorem 2. *The Szilard language Sz(Γ,f) attached to a CDGS Γ, working in f-mode, $f \in M$, is recognizable by a nondeterministic Turing machine in $\mathcal{O}(n^2 logn)$ time and $\mathcal{O}(nlogn)$ space.*

Next our approaches are based on the notion of *cycles* introduced in [5], [10], and [11] in order to describe the structure of the words belonging to languages recognizable by multicounter machines and used in order to prove closure properties for these languages. Each time a MC machine reads a group of identical symbols whose number is greater than the number of states, the device enters in a *cycle*, i.e., a part of the computation delimited by two equal states q (the beginning and the end of the cycle) such that no further state q and no two equal states different from q occur in the part of computation from q to q. We say that this part of the computation is a *cycle* with *state characteristic* q, *reading head characteristic* h, i.e., the symbol read by the reading head, and *counter characteristic* c, for each counter, i.e., the difference between the counter contents at the beginning and at the end of the cycle. For a multicounter machine with s states the number of cycles with different characteristics is bounded by the constant $s \cdot s \cdot (2s+1)^k$.

Following the above structural characterization of the languages accepted by a multicounter machine, Theorem 1 implies a kind of cyclicity phenomenon in the structure of the words belonging to the Szilard language associated with a CDGS working in competence mode in terms of the above *state*, *head* and *counter characteristics*. Based on this remark we have.

Lemma 1. *Let Γ be a CDGS. The Szilard word $\gamma_w(D)$ of a word $w \in L_f(\Gamma)$, $f \in M$, associated with a terminal derivation D performed in f-mode, can be written in terms of a multicounter state, head and counter characteristics, of the form $\gamma_w(\mathcal{M}) = z_1 o_1^{n_1} z_2 o_2^{n_2} ... z_c o_c^{n_c} z_{c+1}$, where z_i, $1 \le i \le c+1$, are finite subwords containing no cycles, o_j are finite cycles and n_j, $1 \le j \le c$, represents the number of times o_j is consecutively repeating.*

Note that in the structure of the Szilard word given by Lemma 1, each segment z_i and o_j is a sequence of the form $z_i^1 ... z_i^{p_i}$ and $o_j^1 ... o_j^{r_j}$, $1 \le i \le c+1$ and $1 \le j \le c$, respectively. According to the computation performed by the machine $\mathcal{M}$, described in Theorem 1, each z_i^p and o_j^r, where $1 \le p \le p_i$ and $1 \le r \le r_j$, is a symbol that encodes three parameters, the *state characteristic* q of $\mathcal{M}$, the *reading head characteristic* h, i.e., the symbol read by $\mathcal{M}$ in state q, and the *counter characteristic* c, i.e., the content of the counter in state q.

The length of $\gamma_w(\mathcal{M})$ might be greater than the length of $\gamma_w(D)$ due to the fact that several z_i^p and o_j^r symbols might have a head characteristic 0, i.e., the multicounter machine from Theorem 1 reads λ. The relation between the length of $\gamma_w(D)$ and the length of $\gamma_w(\mathcal{M})$ is the following one $||\gamma_w(\mathcal{M})|| = d + ||\gamma_w(D)||$ in which $d \le q||\gamma_w(D)||$, because the machine $\mathcal{M}$ works in linear time of delay d and q is a constant.

In the sequel we denote as $\dot{c}$, $\dot{z}$, and $\dot{o}$, the maximum of c, the maximum of the length of z_i, $1 \le i \le c+1$, and the maximum of the length of o_j, $1 \le j \le c$, taken over all groups of Szilard words with the same structure given by Lemma 1, respectively. In the sequel our aim is to refine the class of counters that simulate CDGSs, in order to find decidability results for them.

3.2 Blind and Partially Blind Multicounter Machines

Informally, a *blind counter* is a particular multicounter that does not depend on the counters configuration, but only on states and input. A blind counter is unable to test the signs of its counters. The computation ends when the machine enters in a final state with empty counters.

A *partially blind multicounter* (henceforth pBMC) is a blind counter for which its counters store only natural numbers, they cannot be checked for zero, but the computation is blocked whenever at least one of counters becomes negative.

A (partially blind) MC machine works in *quasirealtime* if there exists a constant d such that the length of each part of any computation in which the reading head is stationary is bounded by d.

Blind MCs are strictly less powerful than quasirealtime pBMCs. Quasirealtime pBMCs accept the family of Petri net languages defined in [9]. They are strictly less powerful than linear time pBMCs and the family of quasirealtime one-way nodeterministic MCs. For the formal definition of blind MCs and pBMCs the reader is referred to [8].

Theorem 3. *The Szilard language $Sz(\Gamma, f)$ attached to a CDGS Γ, working in f-comp.-mode, $f \in \{\leq 1, = 1\}$, can be recognized in quasirealtime by one-way nondeterministic pBMC, with at most $r = \dot{z}(\dot{c}+1) + \dot{o}\dot{c}$ counters.*

Proof. Let $\Gamma = (N, T, \alpha, P_1, \ldots, P_s)$ be a CDGS working in $\{\leq 1, = 1\}$-comp.-mode. Let us consider the ordered set of nonterminals $N = \{A_1, A_2, ..., A_m\}$, and let $P = \{1, 2, ..., s\}$ be the set of labels attached to each component in the system. Let $\mathcal{B}$ be a one-way nondeterministic pBMC with at most $r = \dot{z}(\dot{c}+1)$ + $\dot{o}\dot{c}$ counters, that has as alphabet the set P, and as input the Szilard word $\gamma_w(D) \in P^*$ of length n. According to Lemma 1, $\gamma_w(D)$ can be also rewritten of the form $\gamma_w(\mathcal{M}) = z_1 o_1^{n_1} z_2 o_2^{n_2} ... z_c o_c^{n_c} z_{c+1}$ in which the segments z_i and o_j are of the form $z_i^1 ... z_i^{p_i}$ and $o_j^1 ... o_j^{r_j}$, $1 \leq i \leq c+1$ and $1 \leq j \leq c$, respectively. According to the computation performed by the machine $\mathcal{M}$, described in Theorem 1, each symbol z_i^p and o_j^r, $1 \leq p \leq p_i$ and $1 \leq r \leq r_j$, is characterized by three parameters, the *state characteristic* q of $\mathcal{M}$, the *reading head characteristic* h, and the *counter characteristic* c. Using the head characteristic we can make a direct relation between the Szilard word $\gamma_w(D)$ written as a sequence of labels from P and the Szilard word $\gamma_w(\mathcal{M})$ of the machine $\mathcal{M}$, written as a sequence of symbols characterized by the above multicounter parameters.

The r counters are associated only with nonterminals $A_i \in N$. The first m counters are associated with (different) nonterminals occurring in the axiom. The other counters are used accordingly with the length of the segments z_i and o_i from the Szilard word. Furthermore, we consider each state of $\mathcal{B}$ being of the form $q_{[]...[][]}$, in which the i^{th} cell $[]$ stands for the i^{th} nonterminal A_i, $1 \leq i \leq m$, and the last cell stands for the label from P corresponding to the component that is currently simulated.

Next we give the simulation for $=$ 1-comp.-mode. For $\leq$ 1-comp.-mode the simulation works analogously without any major improvements.

The machine starts with all counters set on zero, having the input head placed on the first delimiter $\natural$, and the initial state $q_\alpha = q_{[x_1]...[x_{m-1}][x_m][0]}$, in which[5] $x_i = |\alpha|_{A_i}$, for $1 \leq i \leq m$. The last zero-cell in q_α is kept for the grammar system component that is activated at each step of the computation. As in the previous theorem from the state q_α the machine goes into the state $q_{[x_1-1]...[x_m][0]}$, by reading λ and increasing the value of the counter C_1 by $+1$. The operation continues until the system reaches the state $q_{[0][x_2]...[x_m][0]}$, using x_1 number of states through which the first counter is increased by $x_1 = |\alpha|_{A_1}$ times. Due to the fact that this time we work with blind counters, when working with them we are not able to see the value stored in them. In order to be able to manipulate this lack each cell in the state that is reduced to 0 will be finally marked by $+$. In this way we can predict which counter could be non-empty and that the corresponding nonterminal could exist in the sentential form. Thus the state $q_{[0][x_2]...[x_m][0]}$ becomes $q_{[+][x_2]...[x_m][0]}$.

When the first cell of the state $q_{[x_1]...[x_m][0]}$ had been "reduced" to $+$ the same operation continues with the second cell "storing" the value x_2, by increasing the second counter with the value $x_2 = |\alpha|_{A_2}$, and so on. These operations end when

[5] Here x_i, $1 \leq i \leq m$, have the same significance as in the previous theorem.

all the m cells storing a "non-zero value", in the initial state had been reduced to $+$ and each counter C_i had been increased by $x_i = |\alpha|_{A_i}$, $1 \leq i \leq m$. Let us denote as q_c the final state reached at the end of this procedure. The state q_c has all cells corresponding to nonterminals occurring in α marked by $+$, all cells corresponding to nonterminals not-occurring in α marked by 0, and the last cell still 0, because none of the system components has been activated so far. As in the previous theorem, all the transitions proceed by reading the empty string. From now on the simulation of the CDGS in the $= 1$-comp.-mode is performed as follows.

Let e, $e \in P$, be the first symbol that occurs in $\gamma_w(D) \in P^*$. Firstly $\mathcal{B}$ checks whether P_e is $= 1$-competent on α. Because the machine cannot see its counters, to perform this operation, it will use the counters configuration codified into the current state. The $= 1$-competence of P_e on α, is checked as in *Step 1.*

Step 1. Let us consider that $\text{dom}(P_e) = \{A_{i_1}, ..., A_{i_p}\}$, where $p \leq m$. The component P_e is $= 1$-competent on α if in the state q_c only one of the ${i_j}^{th}$ cells, that correspond to nonterminals $A_{i_j} \in \text{dom}(P_e)$, $1 \leq j \leq p$, is marked by $+$, no matter how the other $m - p$ cells, that correspond to nonterminals that are not in $\text{dom}(P_e)$ have been marked so far. If the above condition does not hold $\gamma_w(D)$ is rejected[6]. Otherwise, the machine continues the computation as in *Step 2.* $\diamondsuit$

Step 2. Let us suppose that the cell corresponding to the A_{i_j} nonterminal from P_e has the property from *Step 1.* The system nondeterministically chooses which rule from P_e that rewrites A_{i_j} is applied. Let us consider that this is of the form $A_{i_j} \rightarrow rhs(A_{i_j})$. Then the counter C_{i_j} associated with A_{i_j} is decreased by 1. The value of each counter C_i that corresponds to each nonterminal A_i occurring in $rhs(A_{i_j})$ are increased by 1, as many times as A_i occurs in $rhs(A_{i_j})$. Each i^{th} cell, corresponding to nonterminal A_i occurring in $rhs(A_{i_j})$ from state q_c is marked by $+$, in the case that it has not been marked so far. $\diamondsuit$

The pBMC machine ends the procedure described in Step 2 in state q_e, in which the i_j^{th} cell together with the cells corresponding to the new nonterminals introduced so far in the axiom or in the newest sentential form x, are marked by a $+$ symbol. In order to emphasize that the component P_e has been activated, the last cell of the state q_e is marked by an e symbol.

With respect to the definition of the $= 1$-comp.-mode, the component P_e should be active as long as it is $= 1$-competent on x. A new label $e' \neq e$ will be accepted only and only if P_e is not anymore $= 1$-competent on x. In *Step 3* we explain how the simulation goes on when reading a new symbol from $\gamma_w(D)$.

Step 3. Let us suppose that the next symbol in $\gamma_w(D)$ is e'. Firstly the existence of A_{i_j} in the sentential form x is checked[7] (subtract 1, add 1 to the counter C_{i_j}

[6] This can be done by letting the δ function that defines $\mathcal{B}$, to read the label e only in those states in which only one nonterminal from $\text{dom}(P_e)$ is marked by $+$.

[7] Due to the fact that A_{i_j} had been rewritten once, it might not exist in x.

associated with A_{i_j}). If the computation does not block the cell corresponding to A_{i_j} in the state q_e is kept as $+$. If the computation blocks then the simulation continues as in *Step 4*.

If the symbol A_{i_j} still exists in the sentential form then the $= 1$-competence of P_e is checked as in *Step 1*. This time this operation is performed by reading λ but having the last cell of the state q_e marked by e. If P_e is $= 1$-competent and $e' = e$, then the simulation continues as in *Step 2*. It is blocked otherwise, by rejecting $\gamma_w(D)$. If P_e is not $= 1$-competent on x then the $= 1$-competence of the component P'_e is checked as in *Step 1*, by reading e'. If P'_e is $= 1$-competent on x and $e' \neq e$ the simulation continues as in *Step 2*, by substituting e with e'. It blocks otherwise. $\diamondsuit$

Below we give the procedure when zeroing the cells in states.

Step 4. Let us suppose that at *Step 3*, $\mathcal{B}$ has to guess whether the symbol A_{i_j} has been wholly rewritten, i.e., it does not exist anymore in the sentential form x. If the label e is the head characteristic of an element belonging to an z_i segment, after the guessing procedure, the counter C_{i_j} associated with $A_{i_j} \in \mathrm{dom}(P_e)$ is left as it is, and never be used during the simulation. To keep control for the next symbol A_{i_j} introduced in the sentential form another counter, different from C_{i_j}, will be used. If the label e is the head characteristic of an element belonging to an o_i cycle, then the computation proceeds as follows.

Let us consider that until the end of the cycle o_i, the machine performs a number of q guessing procedures. At each guessing $\mathcal{B}$ leaves a counter, that could be empty or not (depending on the correctness of the guess). With respect to the order in which the above counters have been left during the very first running of o_i, we will consider the ordered set of them $\mathcal{C} = \{C_i^1, C_i^2, ..., C_i^q\}$. Part of these counters are loaded during the execution of the cycle o_i, others before the cycle o_i. Accordingly, the set $\mathcal{C}$ can be parted into two subsets, the set $\mathcal{C}_1 = \{C_{i,1}^1, C_{i,1}^2, ..., C_{i,1}^{q_1}\}$ of counters used before the machine $\mathcal{B}$ enters in o_i, and $\mathcal{C}_2 = \{C_{i,2}^1, C_{i,2}^2, ..., C_{i,2}^{q_2}\}$ the set of counters that are used and loaded only within the cycle o_i. We have $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$, $\mathcal{C}_1 \cup \mathcal{C}_2 = \mathcal{C}$. The counters from the first set start to be used in the cycle o_i with nonempty contents (being charged before $\mathcal{B}$ enters in o_i). At each guessing procedure they are left and never be used again during the computation. It is clear that the number of these counters is finite, due to the finiteness length of the z_i part. Counters from the second set start to be used in the o_i segment with empty contents. At each guess they are left and never be used again during the current reading of o_i, and in none of the z_i segments or different than o_i cycles that follow o_i. These counters are "recyclable". The recycling process of the set $\mathcal{C}_2$ is performed as follows.

At each consecutive repetition of the cycle o_i the machine reuses the counter $C_{i,2}^1$ that keeps control of a certain nonterminal A, exactly in the same point of the cycle o_i in which $C_{i,2}^1$ had been firstly used to store A, during the very first execution of o_i. The counter $C_{i,2}^1$ is left at the first guessing procedure performed within o_i, and will never be used until the end of this cycle and in none of the z_i segments or different than o_i cycles. To keep control for the next symbol A_{i_j} introduced in the sentential form during the same session of o_i, another counter different from C_{i_j}, will be used. In the same way the next counters will be reused until the end of the cycle o_i.

It is clear that if the counters $C^j_{i,2}$, $1 \leq j \leq q_2$ have been left empty due to a correct guessing procedure performed during the first reading of the cycle o_i they will be correctly reused, with the chance of leaving them empty again for the next guessing procedure that concerns the nonterminal associated to it, performed at the next execution of o_i. If one of the guessing procedure was not correct, i.e., one of the counters of $\mathcal{C}_2$ enters in the next execution of o_i with a nonempty content then, there is no chance to have it empty at the end of the next guessing procedure performed within o_i (due to the periodicity of the increasing and decreasing operation performed at each repetition of o_i).

The same thing happens each time another cycle o_i is read $1 \leq i \leq c$. A successful computation ends only and only if all counters have been left empty, i.e., all guesses done by $\mathcal{B}$, during the simulation were correct.

When a counter C_{i_j} associated to a nonterminal A is left, to keep control for the next nonterminals A introduced in the sentential form another counter will be used. Each time a new counter is introduced or reused, the cell associated with the nonterminal A from the current state, will be zeroed. After zeroing a cell in a state the procedure that checks the $=1$-competence of the new component, for which its label is read by the input head, is performed again as in *Step 1* following accordingly *Step 2* and *Step 3*. ◇

To be observed that for all guesses performed by $\mathcal{B}$ during the scanning of all z_i segments a maximum number of $\dot{z}(\dot{c}+1)$ counters is required. For the guesses performed during the scanning of all o_i cycles, the machine needs a maximum of $\dot{o}\dot{c}$ counters. In this way the number of counters required by the above pBMC is finite, and it is upper bounded by $\dot{z}(\dot{c}+1) + \dot{o}\dot{c}$.

Finally, the computation continues as explained before, i.e., *Step 1*, *Step 2* and *Step 3* with the guessing procedure described in *Step 4*, until all symbols from $\gamma_w(D)$ are read. The Szilard word $\gamma_w(D)$ is accepted if the system reaches a final state with all counters empty.

It is easy to observe that the partially blind r-counter machine works in quasirealtime. The reading head is stationary only when $\mathcal{B}$ checks the $=1$-competence of a component P_e previously applied in the simulation. This is done by reading λ with the last cell in the current state marked by e, and it is performed each time, using a constant number of stationary steps. □

Theorem 4. *The Szilard language Sz(Γ,f) attached to a CDGS Γ, working in $\{\geq k|k \geq 1\}$-comp.-mode can be recognized in quasirealtime by one-way nondeterministic pBMC, with at most $r = \dot{z}(\dot{c}+1) + \dot{o}\dot{c}$ counters.*

Proof. Let $\Gamma = (N, T, \alpha, P_1, \ldots, P_s)$ be a CDGS working in $\{\geq k|k \geq 1\}$-comp.-mode, with the ordered set of nonterminals $N = \{A_1, ..., A_m\}$ and the set of labels associated with components $P = \{1, 2, ..., s\}$. Let $\mathcal{B}$ be a one-way nondeterministic pBMC with at most $r = \dot{z}(\dot{c}+1) + \dot{o}\dot{c}$ counters, having as alphabet the set P, and as input the Szilard word $\gamma_w(D) \in P^*$ of length n. According to Lemma 1, $\gamma_w(D)$ can be rewritten as $\gamma_w(\mathcal{M}) = z_1 o_1^{n_1} z_2 o_2^{n_2} ... z_c o_c^{n_c} z_{c+1}$. The r counters are associated only with nonterminals in N. The first m counters are used to keep control of the nonterminals occurring in the axiom.

The system starts in the initial state $q_\alpha = q_{[x_1]...[x_{m-1}][x_m][0]}$, where $x_i = |\alpha|_{A_i}$, for $1 \leq i \leq m$. As in the procedure explained in Theorem 3, the input head is stationary until each counter C_i, $1 \leq i \leq m$, is increased accordingly with the number of nonterminals A_i, occurring in the sentential form α. The state q_c in which $\mathcal{B}$ ends this procedure has all cells corresponding to nonterminals occurring in α marked by $+$, all cells corresponding to nonterminals not-occurring in α marked by 0, and the last cell marked also by 0, because none of the system components has been activated so far. From now on the simulation of the CDGS in the $= 1$-comp.-mode is performed as follows.

When reading the first letter from $\gamma_w(D) \in P^*$, let us suppose that this is $e \in P$, $\mathcal{B}$ tests whether the component P_e, such that $|\text{dom}(P_e)| \geq k$, is $\geq k$-competent on the sentential form, i.e., the axiom contains at least k distinct nonterminals from $\text{dom}(P_e)$. The procedure that decides whether P_e is $\geq k$-competent is described in *Step 1.*

Step 1. Let us denote as $S_j = \{A_{i_1}^{(j)}, ..., A_{i_k}^{(j)}\}$, $1 \leq j \leq C_p^k$, all possible subsets[8] of k elements from $\text{dom}(P_e)$, $|\text{dom}(P_e)| = p$, $p \geq k$. For each j, $1 \leq j \leq C_p^k$, in state q_c, reading the label e, each counter $C_{i_l}^{(j)}$, $1 \leq l \leq k$, is 0-tested (subtract 1, add 1). Here is an example how this operation proceeds using the δ function: $\delta(q_c, e) = (q_{e_-}, ..., -1_{C_{i_1}^{(j)}}, ..., -1_{C_{i_k}^{(j)}}, ...)$ for the 0-testing and $\delta(q_{e_-}, \lambda) = (q_e, ..., +1_{C_{i_1}^{(j)}}, ..., +1_{C_{i_k}^{(j)}}, ...)$ to restore the counters. $\diamondsuit$

It is clear that the machine reaches the state q_e only for those subsets S_j, for which each counter $C_{i_l}^{(j)}$, $1 \leq l \leq k$, has a positive value, otherwise, the machine blocks. In this way we ensure that the component P_e is $\geq k$-competent, $k \geq 1$, on α, or on some other sentential forms on which has been tested[9].

In state q_e the system nondeterministically chooses which rule from P_e that rewrites a nonterminal from $\text{dom}(P_e)$, occurring or not in S_j, is applied. In this way the system has the possibility to nondeterministically rewrite a nonterminal from $\text{dom}(P_e)$, that it might not have been included in S_j, but it might exist in the sentential form. However, the simulation fails when decreasing a counter corresponding to a nonterminal that does not occur in the sentential form.

Step 2. Let us consider that the chosen rule from $\text{dom}(P_e)$ is of the form $A \rightarrow rhs(A)$. Then the counter associated with A is decreased by 1, and the value

[8] The cardinal number of all possible subsets is given by the formula $C_p^k = p!/k!(p-k)!$

[9] We conclude that if the pBMC does not block, α contains at least, but not exactly, k distinct nonterminals from $\text{dom}(P_e)$. The lack of this procedure is that, in the case that $\mathcal{B}$ blocks, we do not know with how many counters it failed. That is why this method cannot be used to test the $= k$ or $\leq k$-competence, $k \geq 1$, of a component.

of each counter C_i that corresponds to a nonterminal A_i occurring in $rhs(A)$ is increased by 1, as many times as A_i appears in $rhs(A)$. $\diamondsuit$

According to the definition of the $\geq k$-comp.-mode, P_e should be active as long as it is $\geq k$-competent on the newest sentential form, denoted as x. Due to this, at the next step, in q_e the system checks whether P_e is still $\geq k$-competent on x, by using the 0-testing procedure described at *Step 1*. If the 0-test fails for all subset $S_j \subseteq \text{dom}(P_e)$, then P_e is no longer $\geq k$-competent on x, and the machine continues the simulation by reading the next label e' with the 0-testing procedure described at *Step 1*. On the other hand, when reading a new symbol from $\gamma_w(D)$, a guessing procedure is performed for the nonterminal that has been previously rewritten, i.e., A for this example. This procedure is performed as in *Step 4*, Theorem 3, by taking A instead of A_{i_j}. For the same reasons, explained in Theorem 3, the simulation requires a maximum of $\dot{z}(\dot{c}+1) + \dot{o}\dot{c}$, counters. Finally the simulation, done in quasirealtime, ends and accepts the Szilard word, only in a final state having all counters empty. $\square$

4 Consequences of the Main Results

For CDGSs working in competence mode, from [1] we have the following results:

1) $\mathcal{L}$(CD, CF, f-comp.) = $\mathcal{L}$(RC, CF), $f \in \{\leq k, = k\}$, $k \geq 2$,
2) $\mathcal{L}$(fRC, CF) $\subseteq \mathcal{L}$(CD, CF, f-comp.), $f \in \{\leq 1, = 1\}$,
3) $\mathcal{L}$(CD, CF, $\geq$ 1-comp.) = $\mathcal{L}$(ETOL),
4) $\mathcal{L}$(CD, CF, $\geq k$-comp.) = $\mathcal{L}$(RC, ETOL), $k \geq 2$.

It is known from [8] that quasirealtime pBMC accept the family of Petri net languages defined in [9]. Hence, decidability problems, such as finiteness, emptiness and membership, solvable for Petri nets are decidable for quasirealtime pBMC machines, too. From Theorem 3, Theorem 4 and the above remark we have.

Theorem 5. *The finiteness, emptiness and membership problems are decidable for the class of Szilard languages SZ(CD, CF, f), $f \in \{\leq 1, = 1\} \cup \{\geq k | k \geq 1\}$.*

As a consequence of the above theorem we have.

Theorem 6. *The finiteness and emptiness problems are decidable for the class of languages generated by CDGSs that work in f-comp.-mode, $f \in \{\leq 1, = 1\} \cup \{\geq k | k \geq 1\}$.*

Proof. The decidability of the emptiness problem for the class of languages $\mathcal{L}$(CD, CF, f), where $f \in \{\leq 1, = 1\} \cup \{\geq k | k \geq 1\}$, follows directly from Definition 5 and Theorem 5.

To prove the decidability of the finiteness problem for the same class of languages, we provide the pBMC machines from Theorem 3 and Theorem 4 with one more counter, let us denote it as T. When the machine $\mathcal{B}$ meets the very first cycle in the structure of a Szilard word, from $\text{Sz}(\Gamma, f)$ attached to a CDGS Γ, that contains a label of a component for which the nondeterministically chosen rule, of the form $A \to rhs(A)$, contains in the right-hand side at least one terminal,

T is increased by one and will never be used until the end of the computation. It is clear that this pBMC machine accepts only Szilard languages associated with languages that are finite, and rejects all Szilard languages associated with languages that are infinite[10]. Analogously, we can find a pBMC machine that accepts Szilard languages associated with languages that are infinite and rejects all Szilard languages associated with languages that are finite. □

Furthermore, in [4] it is proved the equality between the class of languages generated by ordered grammars (O) and forbidding random context grammars (fRC), with or without λ rules. From the inclusion 2), Theorem 6 and the above remark, the finiteness and emptiness problems are decidable for O and fRC grammars, too. From the relation 4) and Theorem 6 these problems are decidable also for RC ET0L systems. Thus we consider that we solved several decision problems left open in [4] related to the above rewriting grammars and systems.

Acknowledgments

The author is indebted to Prof. Jürgen Dassow and Prof. Juraj Hromkovič for many interesting and helpful discussions on the final version of this paper.

References

1. ter Beek, M.H., Csuhaj-Varjú, E., Holzer, M., Vaszil, G.: On Competence in CD Grammar Systems. In Calude, C.S., Calude, E., Dinneen, M.J., (Eds.), Developments in Language Theory, 8th International Conference, Auckland, New Zealand, December 13-17, Proceedings. LNCS 3340, Springer-Verlag (2004) 76–88
2. Chan. T.: Reversal-Bounded Computations. PhD. Thesis, December 1980, TR 80-448, Department of Computer Science, Cornell University, Ithaca, New York (1980)
3. Csuhaj-Varjú, E., Dassow, J., Kelemen, J., Păun, G.: Grammar Systems. A Grammatical Approach to Distribution and Cooperation, Gordon and Breach (1994)
4. Dassow, J., Păun. G.: Regulated Rewriting in Formal Language Theory, Springer-Verlag, Berlin (1989)
5. Ďuriš, P., Hromkovič, J.: One-Way Simple Multihead Finite Automata are not Closed under Concatenation. Theoretical Computer Science **27** (1983) 121–125
6. Fischer, P.C., Meyer, A. R., Rosenberg, A.L.: Counter Machines and Counter Languages. Mathematical System Theory **2** (1968) 265–283
7. Greibach, S.A.: Remarks on the Complexity of Nondeterministic Counter Languages. Theoretical Computer Science **2** (1976) 269–288
8. Greibach, S.A.: Remarks on Blind and Partially Blind One-Way Multicounter Machines. Theoretical Computer Science **7** (1978) 311–324
9. Hack, M.: Petri Net Languages, Computation Structures. Group Memo 124. Project MAC, Massachusetts Institute of Technology (1975)
10. Hromkovič, J.: Hierarchy of Reversal Bounded One-Way Multicounter Machines. Kybernetika **22(2)** Academia Praha (1986) 200–206
11. Hromkovič, J.: Reversal-Bounded Nondeterministic Multicounter Machines and Complementation. Theoretical Computer Science **51** (1987) 325–330

[10] In this case, a language is infinite if there exists at least one cycle in the structure of the Szilard word that "pumps" in a sentential form at least one terminal.

Deriving State-Based Implementations of Interactive Components with History Abstractions

Walter Dosch and Annette Stümpel

Institute of Software Technology and Programming Languages
University of Lübeck, Lübeck, Germany
{dosch,stuempel}@isp.uni-luebeck.de
http://www.isp.uni-luebeck.de

Abstract. The external behaviour of an interactive component refers to the communication histories on the input and output channels. The component's implementation employs an internal state where inputs effect output and an update of the state. The black-box view is modelled by a stream processing function from input to output streams, the glass-box view by a state transition machine. We present a formal method how to implement a stream processing function with several arguments by a state transition machine in a correctness preserving way. The transformation involves two important steps, called differentiation and history abstraction. The differentiation localizes the effect of a single input on one of the input channels wrt. the previous input histories. The history abstraction introduces states as congruence classes of input histories. We extend our previous results from interactive components with one input channel to components with several input channels. The generalization employs a 'diamond property' for states and outputs which ensures the confluence of the resulting state transition system.

Keywords: Interactive component, communication history, stream processing function, state transition machine, history abstraction.

1 Introduction

An interactive system [23] consists of a network of components that communicate asynchronously via unidirectional channels. In general, the components possess multiple input channels and multiple output channels. In the setting of untimed systems, the communication history of a channel is modelled by a sequence of messages, called a stream. The input/output behaviour of a deterministic component is described by a stream processing function [13] mapping a tuple of input histories to a tuple of output histories.

During the design of an interactive component [5], the software engineer employs different points of view. On the specification level, the component is considered as a 'black box' whose behaviour is determined by the relation between input and output histories. The external view describes the service provided by the component when cooperating with other components or with the environment. On the implementation level, the component is considered as a 'glass box' described by a state transition machine with input and output. Inputs are processed message by message by successively updating the state and generating the output streams in an incremental way.

A crucial design step amounts to transforming the specified input/output behaviour of an interactive component into a state-based implementation. In our approach, we introduce machine states as abstractions of the input histories. A state stores information about the component's past input histories that determines the future output histories when receiving further input. In general, there are different abstractions of the input histories which lead to state spaces of different granularity [8].

We present a formal method how to implement a multiary stream processing function by a state transition machine in a correctness preserving way. The transformation involves two important steps, viz. *differentiation* and *history abstraction*. The differentiation localizes the effect of a single input on a particular input channel wrt. the previous input histories. The history abstraction introduces states as congruence classes of input histories. The resulting state transition machine is well defined, if we impose two requirements, viz. transition closedness and output compatibility. When receiving further input on an arbitrary input channel, the history abstraction must be compatible with the state transitions and with the generation of the output.

This paper extends our previous results [9] from interactive components with one input channel to components with several input and several output channels [21]. In the setting of asynchronous systems, the generalization from one to multiple input channels leads to state transition machines with interleaved inputs. We impose a 'diamond property' for states and outputs to ensure the confluence of the state transition system and the determinism of the input/output behaviour. Under these constraints, the produced output streams are independent of the relative order in which the messages arrived on the different input channels.

The paper is organized as follows. In Section 2 we summarize the basic notions for the functional description of interactive components with communication histories. Section 3 introduces state transition machines with multiple inputs and outputs. In Section 4 we present a universal approach for implementing an interactive component with a canonical state transition machine. Section 5 describes the systematic construction of coarser state transition machines with history abstractions. In Section 6 we apply the formal method to a requests driven sender. The conclusion surveys the approach and outlines future work.

2 Streams and Stream Processing Functions

In this section we briefly summarize the basic notions about streams and stream processing functions to render the paper self-contained. The reader is referred to [20] for a survey and to [21] for a comprehensive treatment.

2.1 Streams as Communication Histories

Streams model the communication history of a single directed channel by the temporal succession of messages transmitted. Streams abstract from time and record only the sequence of messages.

Given a non-empty set $\mathcal{A}$ of messages, a *(communication) history*, for short a *stream* $X = \langle x_1, \ldots, x_k \rangle$ of *length* $|X| = k \geq 0$ is a finite sequence of elements $x_i \in \mathcal{A}$ for $i \in [1, k]$. $\mathcal{A}^\star$ denotes the set of all streams, $\mathcal{A}^+$ the set of all non-empty streams over $\mathcal{A}$.

The *concatenation* $X \,\&\, Y$ of a stream $X = \langle x_1, \ldots, x_k \rangle$ with a stream $Y = \langle y_1, \ldots, y_l \rangle$ over the same set yields the stream $\langle x_1, \ldots, x_k, y_1, \ldots, y_l \rangle$ of length $k + l$. The concatenation $\langle x \rangle \,\&\, X$ (resp. $X \,\&\, \langle x \rangle$) appending a single element x at the front (resp. rear) of a stream X is denoted by $x \triangleleft X$ (resp. $X \triangleright x$) . The *subtraction* $Y \ominus X = R$ of the initial segment X from its extension $Y = X \,\&\, R$ yields the final segment R .

The *take* and *drop* operations split a stream into its initial part of a given length $n \geq 0$ and the remaining part. If $|X| = n$, then $\mathit{take}(n)(X \,\&\, Y) = X$ and $\mathit{drop}(n)(X \,\&\, Y) = Y$. If $|X| < n$ holds, then $\mathit{take}(n)(X) = X$ and $\mathit{drop}(n)(X) = \langle\rangle$.

Operational progress is modelled by the prefix relation on streams — the longer stream forms an extension of the shorter history. A stream $X \in \mathcal{A}^\star$ is called a *prefix* of a stream $Y \in \mathcal{A}^\star$, denoted $X \sqsubseteq Y$, iff a stream $R \in \mathcal{A}^\star$ exists with $X \,\&\, R = Y$. The structure $(\mathcal{A}^\star, \sqsubseteq, \langle\rangle)$ forms a partial order with the empty stream as the least element.

Components receive input on several input channels and deliver output to several output channels. The communication on independent channels is modelled by *tuples of streams.* A tuple $\overrightarrow{X} = (X_1, \ldots, X_m)$ of streams $X_i \in \mathcal{A}_i^\star$ with $i \in [1, m]$ and $m \geq 1$ is written using vector notation. We extend all operations on and relations between streams in a componentwise way to tuples of streams. In particular, $\overrightarrow{\&}$ denotes the concatenation of tuples of streams, and $\overrightarrow{\sqsubseteq}$ the prefix relation on tuples of streams.

For a channel index $i \in [1, m]$, the operation $\triangleleft_i$ (resp. $\triangleright_i$) attaches an element to the front (resp. rear) of the i-th stream of a tuple of streams: $x \triangleleft_i (X_1, \ldots, X_m) = (X_1, \ldots, X_{i-1}, x \triangleleft X_i, X_{i+1}, \ldots, X_m)$.

We use small letters to denote single messages of streams, capital letters to denote streams and vectors on capital letters to denote tuples of streams.

2.2 Components as Stream Processing Functions

A component repeatedly processes messages from its input channels and emits messages to its output channels. A deterministic component can be modelled by a function mapping a tuple of input histories to a tuple of output histories.

A *stream processing function*, also called a *history function* $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ maps $m \geq 1$ input streams to $n \geq 1$ output streams. The types $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ of the input streams and the types $(\mathcal{B}_1, \ldots, \mathcal{B}_n)$ of the output streams determine the *(syntactic) interface* of the component. A stream processing function describes the *(input/output) behaviour* of an interactive component.

We require that a stream processing function is *monotonic* with respect to the prefix order:

$$f(\overrightarrow{X}) \ \overrightarrow{\sqsubseteq} \ f(\overrightarrow{X} \ \overrightarrow{\&} \ \overrightarrow{Y}) \tag{1}$$

Monotonicity formalizes the requirement that an interactive component cannot change the previous output histories in reaction to additional input, but only prolong them.

A stream processing function $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ can be split into its *spontaneous output* $f(\langle\rangle, \ldots, \langle\rangle)$ emitted before input is processed, and the *reactive behaviour* $react(f) : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ in response to input:

$$react(f)(\overrightarrow{X}) = f(\overrightarrow{X}) \ \overrightarrow{\ominus} \ f(\langle\rangle, \ldots, \langle\rangle) \tag{2}$$

2.3 Differentiation

While stream processing functions summarize the overall behaviour on entire communication histories, the implementation processes the input streams message by message. The *differentiation* reveals the causal relationship between a single input on one of the input channels and the corresponding segments of the output streams depending on the previously processed input histories, cf. Fig. 1.

Definition 2.1. *The differentiation* $\varepsilon_i : [\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star] \to [\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star] \times \mathcal{A}_i \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ *of a stream processing function with respect to the i-th input channel* ($i \in [1, m]$) *is defined by*

$$\varepsilon_i(f)(\overrightarrow{X}, x) = f(\overrightarrow{X} \triangleright_i x) \ \overrightarrow{\ominus} \ f(\overrightarrow{X}) \ . \tag{3}$$

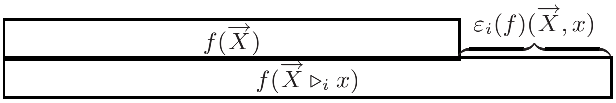

$f(\overrightarrow{X})$
$\varepsilon_i(f)(\overrightarrow{X}, x)$
$f(\overrightarrow{X} \triangleright_i x)$

Fig. 1. Differentiation of a stream processing function $f : \mathcal{A}_1^\star \times \ldots \times A_m^\star \to \mathcal{B}_1^\star$ with respect to the i-th input channel

The differentiation determines the reactive part of a stream processing function.

Proposition 2.2. *The differentiations of two stream processing functions* $f, g : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ *with the same interface agree iff their reactive parts agree:*

$$\left(\bigwedge_{i=1}^{m} \varepsilon_i(f) = \varepsilon_i(g) \right) \iff react(f) = react(g) \tag{4}$$

The differentiation of stream processing functions enjoys many useful properties. The implementation by state transition machines widely exploits the following property.

Proposition 2.3. *Subsequent differentiations of a stream processing function* $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ *with respect to different input channels* $i \neq j \in [1, m]$ *commute in the following way:*

$$\varepsilon_i(f)(\overrightarrow{X}, x_i) \mathbin{\overrightarrow{\&}} \varepsilon_j(f)(\overrightarrow{X} \triangleright_i x_i, x_j) = \varepsilon_j(f)(\overrightarrow{X}, x_j) \mathbin{\overrightarrow{\&}} \varepsilon_i(f)(\overrightarrow{X} \triangleright_j x_j, x_i) \quad (5)$$

3 State Transition Machines with Input and Output

The operational semantics of systems is often formalized by labelled transition systems [24] which specify a labelled transition relation between states. When modelling interactive components, state transitions are triggered by arriving inputs on the different input channels.

3.1 Architecture

A state transition machine processes an input at one of the input channels by an update of its internal state while emitting data segments to the different output channels.

Definition 3.1. *A* state transition machine with input and output*, for short a* state transition machine

$$M = (\mathcal{Q};\, \mathcal{A}_1, \ldots, \mathcal{A}_m;\, \mathcal{B}_1, \ldots, \mathcal{B}_n;\, next_1, \ldots, next_m;\, out_1, \ldots, out_m;\, q_0) \quad (6)$$

consists of
- *a non-empty set* $\mathcal{Q}$ *of* states*,*
- $m \geq 1$ *non-empty sets* $\mathcal{A}_i$ *of* input data $(i \in [1, m])$,
- $n \geq 1$ *non-empty sets* $\mathcal{B}_j$ *of* output data $(j \in [1, n])$,
- m *(one-step)* state transition functions $next_i : \mathcal{Q} \times \mathcal{A}_i \to \mathcal{Q}$ $(i \in [1, m])$,
- m *(one-step)* output functions $out_i : \mathcal{Q} \times \mathcal{A}_i \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ $(i \in [1, m])$,
- *an* initial state $q_0 \in \mathcal{Q}$.

The types $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ *and* $(\mathcal{B}_1, \ldots, \mathcal{B}_n)$ *determine the* interface *of the state transition machine.*

The transition functions have to compensate the order of the consumption of messages from different input channels. For all input channels $i \neq j \in [1, m]$ *we assume the following* diamond property*:*

$$next_j(next_i(q, x_i), x_j) = next_i(next_j(q, x_j), x_i) \quad (7)$$

$$out_i(q, x_i) \mathbin{\overrightarrow{\&}} out_j(next_i(q, x_i), x_j) = out_j(q, x_j) \mathbin{\overrightarrow{\&}} out_i(next_j(q, x_j), x_i) \quad (8)$$

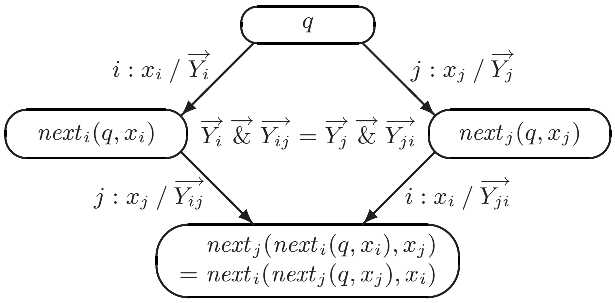

q
$i : x_i \ / \ \overrightarrow{Y_i}$
$j : x_j \ / \ \overrightarrow{Y_j}$
$next_i(q, x_i)$
$\overrightarrow{Y_i} \ \overrightarrow{\&} \ \overrightarrow{Y_{ij}} = \overrightarrow{Y_j} \ \overrightarrow{\&} \ \overrightarrow{Y_{ji}}$
$next_j(q, x_j)$
$j : x_j \ / \ \overrightarrow{Y_{ij}}$
$i : x_i \ / \ \overrightarrow{Y_{ji}}$
$next_j(next_i(q, x_i), x_j)$
$= next_i(next_j(q, x_j), x_i)$

Fig. 2. Diamond property with $\overrightarrow{Y_i} = out_i(q, x_i)$, $\overrightarrow{Y_{ij}} = out_j(next_i(q, x_i), x_j)$, $\overrightarrow{Y_j} = out_j(q, x_j)$, and $\overrightarrow{Y_{ji}} = out_i(next_j(q, x_j), x_i)$

The one-step transition functions associate with the current state and input on one of the input channels a unique successor state and a tuple of output sequences. The state transition machine processes in a nondeterministic style, if there is input available on two or more input channels, cf. Fig. 2. Condition (7) ensures that the one-step state transition functions can be integrated into one overall multi-step state transition function processing tuples of input streams. Condition (8) guarantees that the one-step output functions processing input from different input channels can consistently be chained into one multi-step output function.

Definition 3.2. *The* multi-step state transition function $next^\star : \mathcal{Q} \rightarrow [\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \rightarrow \mathcal{Q}]$ *yields the unique final state after processing a tuple of input streams:*

$$next^\star(q)(\langle\rangle, \ldots, \langle\rangle) = q \tag{9}$$

$$next^\star(q)(x \triangleleft_i \overrightarrow{X}) = next^\star(next_i(q, x))(\overrightarrow{X}) \tag{10}$$

In each state $q \in \mathcal{Q}$, the multi-step state transition function cooperates with concatenation:

$$next^\star(q)(\overrightarrow{X} \ \overrightarrow{\&} \ \overrightarrow{Y}) = next^\star(next^\star(q)(\overrightarrow{X}))(\overrightarrow{Y}) \tag{11}$$

Definition 3.3. *The* multi-step output function $out^\star : \mathcal{Q} \rightarrow [\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \rightarrow \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star]$ *accumulates the unique tuple of output streams generated by a tuple of input streams:*

$$out^\star(q)(\langle\rangle, \ldots, \langle\rangle) = (\langle\rangle, \ldots, \langle\rangle) \tag{12}$$

$$out^\star(q)(x \triangleleft_i \overrightarrow{X}) = out_i(q, x) \ \overrightarrow{\&} \ out^\star(next_i(q, x))(\overrightarrow{X}) \tag{13}$$

The multi-step output function describes the (input/output) behaviour *of the state transition machine.*

In each state $q \in \mathcal{Q}$, the multi-step output function $out^\star(q)$ forms a prefix monotonic stream processing function:

$$out^\star(q)(\overrightarrow{X} \mathbin{\overrightarrow{\&}} \overrightarrow{Y}) = out^\star(q)(\overrightarrow{X}) \mathbin{\overrightarrow{\&}} out^\star(next^\star(q)(\overrightarrow{X}))(\overrightarrow{Y}) \tag{14}$$

Starting in any state, the multi-step output function offers a history-based view of a state transition machine which abstracts from the interleaving of the transitions when processing single messages from different input channels.

3.2 Output Equivalence

We aim at transforming a state transition machine into a more compact one with a reduced number of states without changing the machine's input/output behaviour. To this end, we are interested in states which induce the same multi-step output function.

Definition 3.4. *Two states $p, q \in \mathcal{Q}$ of a state transition machine M are called* output equivalent*, denoted $p \approx q$, iff they generate the same tuple of output streams for all tuples of input streams: $out^\star(p) = out^\star(q)$*

An observer cannot distinguish output equivalent states by observing the machine's input/output behaviour. Successor states of output equivalent states are also output equivalent.

3.3 State Homomorphisms

A state homomorphism reduces the number of states by identifying subsets of output equivalent states [7].

Definition 3.5. *A* state homomorphism *$hom : \mathcal{Q} \to \mathcal{Q}'$ from a state transition machine $M = (\mathcal{Q}; \mathcal{A}_1, \ldots, \mathcal{A}_m; \mathcal{B}_1, \ldots, \mathcal{B}_n; next_1, \ldots, next_m; out_1, \ldots, out_m; q_0)$ to a state transition machine $M' = (\mathcal{Q}'; \mathcal{A}_1, \ldots, \mathcal{A}_m; \mathcal{B}_1, \ldots, \mathcal{B}_n; next'_1, \ldots, next'_m; out'_1, \ldots, out'_m; q'_0)$ with the same interface is compatible with the operations and constants of the machines ($i \in [1, m]$):*

$$hom(next_i(q, x)) = next'_i(hom(q), x) \tag{15}$$

$$out_i(q, x) = out'_i(hom(q), x) \tag{16}$$

$$hom(q_0) = q'_0 \tag{17}$$

A state homomorphism is compatible with the multi-step transition functions:

$$hom(next^\star(q)(\overrightarrow{X})) = (next')^\star(hom(q))(\overrightarrow{X}) \tag{18}$$

$$out^\star(q) = (out')^\star(hom(q)) \tag{19}$$

Consequently, if a state homomorphism exists between two state transition machines, then it is uniquely determined on the subset of reachable states by the multi-step state transition functions. Furthermore, states identified by a state homomorphism are output equivalent:

$$hom(p) = hom(q) \implies p \approx q \tag{20}$$

From the view point of universal algebra, state homomorphisms are indeed a specialization of Σ-homomorphisms to state transition machines regarded as multi-sorted algebras. Thus, well-known results from universal algebra [15,22] about Σ-homomorphisms carry over to state homomorphisms. For example, the composition of state homomorphisms yields a state homomorphism, the kernel of a state homomorphism forms a *state congruence*, and the quotient map $\phi_\sim(q) = [q]_\sim$ associating with each state its congruence class is a state homomorphism from M to the *quotient state transition machine* $M/_\sim$ with the same interface.

3.4 Related Models

There exist several computational devices which are closely related to our state transition machines with input and output processing multiple input streams and generating multiple output streams in a completely asynchronous way.

State transition machines with input and output extend *generalized sequential machines* [11], *Stream X-machines* [1], and *Mealy machines* (beyond further aspects) to multiple input and multiple output channels. In Harel's *statecharts* [12], *UML state diagrams* [17], and *mini-statecharts* [19] communication is conducted via a broadcast mechanism or via multicast communication, respectively.

In contrast to *port input/output automata* [14], in which output can be arbitrarily delayed by repeatedly always giving priority to further input actions, state transition machines establish a direct relation between input and the corresponding output.

State transition systems as defined in [3] are in particular used for the verification of safety and liveness properties [2,4]. Hence their states carry additional information: states are labelled with the previous and the future content of the channels and additional attributes.

For more thorough comparisons, the reader is referred to [9].

4 Canonical State Transition Machine

For every stream processing function, there is a canonical state transition machine recording the complete input histories in its states.

Definition 4.1. *The* canonical state transition machine $M[f] = (\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star; \mathcal{A}_1, \ldots, \mathcal{A}_m; \mathcal{B}_1, \ldots, \mathcal{B}_n; next_1, \ldots, next_m; out_1, \ldots, out_m; (\langle\rangle, \ldots, \langle\rangle))$ *of a stream processing function* $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ *is defined by*

$$next_i(\overrightarrow{X}, x) = \overrightarrow{X} \triangleright_i x \tag{21}$$

$$out_i(\overrightarrow{X}, x) = f(\overrightarrow{X} \triangleright_i x) \overrightarrow{\ominus} f(\overrightarrow{X}) \ . \tag{22}$$

The state transition function appends the current input from the i-th input channel to the state determined by the tuple of input histories. The output function corresponds to the differentiation of the stream processing function, cf. Definition 2.1. The constructed machine validates the diamond property (7) and (8).

In [6] the same basic idea is employed for a specification scheme for agents which process a sequence of input actions. With the same principle, [18] presents the construction of a total “spelling” automaton for a unary stream processing function.

The multi-step state transition function accumulates the input streams; the multi-step output function extends the output histories:

$$next^{\star}(\overrightarrow{X})(\overrightarrow{Y}) = \overrightarrow{X} \mathbin{\overrightarrow{\&}} \overrightarrow{Y} \tag{23}$$

$$out^{\star}(\overrightarrow{X})(\overrightarrow{Y}) = f(\overrightarrow{X} \mathbin{\overrightarrow{\&}} \overrightarrow{Y}) \mathbin{\overrightarrow{\ominus}} f(\overrightarrow{X}) \tag{24}$$

Equation (24) shows the correctness of the canonical state transition machine.

Proposition 4.2. *The canonical state transition machine* $M[f]$ correctly implements *the stream processing function* $f : \mathcal{A}_1^{\star} \times \ldots \times \mathcal{A}_m^{\star} \to \mathcal{B}_1^{\star} \times \ldots \times \mathcal{B}_n^{\star}$ *:*

$$react(f) = out^{\star}(\langle\rangle, \ldots, \langle\rangle) \tag{25}$$

State transition machines provide their first output only in reaction to some input, whereas stream processing functions can provide spontaneous output before receiving input.

Knowing that input streams mimic the states of the canonical state transition machine, we transfer the term ‘output equivalence’ to tuples of input streams and construct coarser state transition machines with state homomorphisms.

5 State Introduction with History Abstractions

History abstractions identify input histories which generate the same output streams for future input histories. History abstractions constitute a key issue for the systematic construction of state transition machines implementing a given stream processing function.

The following proposition names the unique state homomorphism from a canonical state transition machine to another state transition machine realizing the same stream processing function.

Proposition 5.1. *Let* $M = (\mathcal{Q}; \mathcal{A}_1, \ldots, \mathcal{A}_m; \mathcal{B}_1, \ldots, \mathcal{B}_n; next_1, \ldots, next_m; out_1, \ldots, out_m; q_0)$ *be any state transition machine implementing a stream processing function* $f : \mathcal{A}_1^{\star} \times \ldots \times \mathcal{A}_m^{\star} \to \mathcal{B}_1^{\star} \times \ldots \times \mathcal{B}_n^{\star}$ *. Then the multi-step state transition function* $next^{\star}(q_0) : \mathcal{A}_1^{\star} \times \ldots \times \mathcal{A}_m^{\star} \to \mathcal{Q}$ *forms the unique state homomorphism from the canonical state transition machine* $M[f]$ *to* M *.*

The canonical state transition machine serves as a universal starting point for constructing coarser state transition machines with the same behaviour.

Definition 5.2. *A function* $abstr : \mathcal{A}_1^{\star} \times \ldots \times \mathcal{A}_m^{\star} \to \mathcal{Q}$ *is called a* history abstraction *for a stream processing function* $f : \mathcal{A}_1^{\star} \times \ldots \times \mathcal{A}_m^{\star} \to \mathcal{B}_1^{\star} \times \ldots \times \mathcal{B}_n^{\star}$ *iff it is* transition closed *(26) and* output compatible *(27):*

$$abstr(\overrightarrow{X}) = abstr(\overrightarrow{Y}) \implies abstr(\overrightarrow{X} \triangleright_i x) = abstr(\overrightarrow{Y} \triangleright_i x) \tag{26}$$

$$abstr(\overrightarrow{X}) = abstr(\overrightarrow{Y}) \implies \varepsilon_i(f)(\overrightarrow{X}, x) = \varepsilon_i(f)(\overrightarrow{Y}, x) \tag{27}$$

A history abstraction identifies all the prolongations of a tuple of identified streams:

$$abstr(\overrightarrow{X}) = abstr(\overrightarrow{Y}) \implies abstr(\overrightarrow{X} \mathbin{\&} \overrightarrow{Z}) = abstr(\overrightarrow{Y} \mathbin{\&} \overrightarrow{Z}) \tag{28}$$

We summarize the formal basis for the envisaged method in the following

Theorem 5.3. *Let* $abstr : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{Q}$ *be a history abstraction for the stream processing function* $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ *. Then the state transition machine* $M[f, abstr] = (\mathcal{Q}; \mathcal{A}_1, \ldots, \mathcal{A}_m; \mathcal{B}_1, \ldots, \mathcal{B}_n; next_1, \ldots, next_m; out_1, \ldots, out_m; abstr(\langle\rangle, \ldots, \langle\rangle))$ *with transition functions*

$$next_i(abstr(\overrightarrow{X}), x) = abstr(\overrightarrow{X} \triangleright_i x) \tag{29}$$

$$out_i(abstr(\overrightarrow{X}), x) = \varepsilon_i(f)(\overrightarrow{X}, x) \tag{30}$$

correctly implements f *through the history abstraction* $abstr$ *.*

The constructed machine validates the diamond property (7) and (8). The set $abstr(\mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star)$ constitutes the set of states reachable from the initial state. Transitions originating from unreachable states are not relevant.

The theorem provides the foundations for a formal method how to construct a state transition machine which implements a given stream processing function. The crucial design decision concentrates on the choice of a suitable history abstraction and an associated state space. This decision determines the constituents of the resulting state transition machine apart from the definition of the transition functions on the subset of unreachable states.

The canonical state transition machine $M[f, id]$ obtained with the identity as a history abstraction is a finest state transition machine. Choosing the function $abstr(\overrightarrow{X}) = [\overrightarrow{X}]_\approx$ mapping each tuple $\overrightarrow{X}$ of input streams to its output equivalence class results in a state transition machine $M[f, abstr]$ with the coarsest set of states possible [21].

Frequently used history abstractions comprise, for example, constant functions, filter functions, the length function, truncation functions at the front or at the rear of the streams, set and multiset abstractions, and reductions of the input histories. The abstraction functions can be combined in a suitable way into further abstraction functions coping with more input streams.

6 Application: Requests Driven Sender

As an application, we construct a state transition machine for a request driven sender. The component receives data items and numbers of requests on two different channels and forwards the requested number of data items, as far as possible, to a receiver.

6.1 Specification

A *requests driven sender* is a communicating component with two input channels and one output channel. The component receives numbers as requests on its

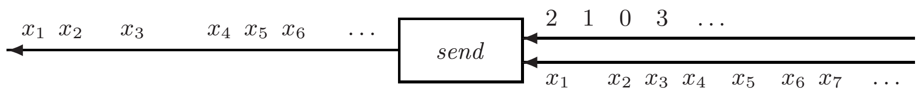

x_1 x_2 x_3 x_4 x_5 x_6 ...
send
2 1 0 3 ...
x_1 x_2 x_3 x_4 x_5 x_6 x_7 ...

Fig. 3. Input/output behaviour of a sender component

requests channel and forwards the respective number of data items from its *data channel* to its output channel, compare Fig. 3.

The component's *interface* is determined by the natural numbers $\mathbb{N}$ for request numbers and the non-empty type $\mathcal{D}$ of data. The component's *input/output behaviour* is described by a function

$$send : \mathbb{N}^\star \times \mathcal{D}^\star \to \mathcal{D}^\star$$

that maps a pair of input histories to an output history:

$$send(N, D) = take(sum(N))(D) \tag{31}$$

The auxiliary function $sum : \mathbb{N}^\star \to \mathbb{N}$ sums up the elements of a stream of natural numbers.

6.2 Differentiation

We differentiate the component's history function wrt. the requests channel

$$\varepsilon_1(send) : (\mathbb{N}^\star \times \mathcal{D}^\star) \times \mathbb{N} \to \mathcal{D}^\star$$

obtaining the following result:

$$sum(N) \geq |D| \implies \varepsilon_1(send)((N, D), n) = \langle\rangle \tag{32}$$
$$sum(N) = |D| \implies \varepsilon_1(send)((N, D \,\&\, E), n) = take(n)(E) \tag{33}$$

If the component collected more requests than data items were available on the data channel, then a further request produces no data on the output channel (32). Otherwise an additional request effects the output of the requested number of data items, as far as they exist (33).

The differentiation of the *send* function with respect to the data channel

$$\varepsilon_2(send) : (\mathbb{N}^\star \times \mathcal{D}^\star) \times \mathcal{D} \to \mathcal{D}^\star$$

leads to the following equation:

$$\varepsilon_2(send)((N, D), d) = \begin{cases} \langle\rangle & \text{if } |D| \geq sum(N) \\ \langle d\rangle & \text{if } |D| < sum(N) \end{cases} \tag{34}$$

A single data item can only pass the sender iff there exist pending requests (34).

The output only depends on the non-negative difference between the sum of the requests and the length of the previous data input and otherwise on the data items which have not been forwarded yet.

6.3 History Abstraction

The component's internal state records the sum of pending requests received on the requests channel or the data items that were received on the data channel, but were not sent on the output channel:

$$\mathcal{Q} = \mathit{Reqs}(\mathbb{N}) \cup \mathit{Buffer}(\mathcal{D}^+) \tag{35}$$

The *history abstraction*

$$\mathit{abstr} : \mathbb{N}^\star \times \mathcal{D}^\star \to \mathcal{Q}$$

retains the non-negative difference between the sum of the requests and the length of the previous data input and otherwise the data items which have not been forwarded yet:

$$\mathit{abstr}(N, D) = \begin{cases} \mathit{Reqs}(\mathit{sum}(N) - |D|) & \text{if } \mathit{sum}(N) \geq |D| \\ \mathit{Buffer}(\mathit{drop}(\mathit{sum}(N))(D)) & \text{if } \mathit{sum}(N) < |D| \end{cases} \tag{36}$$

The history abstraction is surjective: the state $\mathit{Reqs}(m)$ can be obtained from the pair $(\langle m \rangle, \langle\rangle)$ of input streams, the state $\mathit{Buffer}(D)$ from the pair $(\langle\rangle, D)$.

The 'state' $\mathit{Buffer}(\langle\rangle)$ is not needed because it would have the same effect on future output as the state $\mathit{Reqs}(0)$. The chosen history abstraction will lead to a coarsest state transition machine (up to isomorphism) for the sender component. Any function identifying further input histories would violate transition closedness (26) or output compatibility (27).

6.4 State Transition Machine

Given the history abstraction *abstr* from Section 6.3, the constituents of the state transition machine

$$M[\mathit{send}, \mathit{abstr}] = (\mathcal{Q}; \mathbb{N}, \mathcal{D}; \mathcal{D}; \mathit{next}_1, \mathit{next}_2; \mathit{out}_1, \mathit{out}_2; q_0)$$

can be derived from Equations (29) and (30) in Theorem 5.3 using the states $\mathit{abstr}(\langle m \rangle, \langle\rangle)$ and $\mathit{abstr}(\langle\rangle, D)$ as standard representations.

For the initial state we obtain $q_0 = \mathit{abstr}(\langle\rangle, \langle\rangle) = \mathit{Reqs}(0)$.

The state transition functions for input on the requests channel

$$\mathit{next}_1(\mathit{Reqs}(m), n) = \mathit{Reqs}(m + n) \tag{37}$$

$$\mathit{next}_1(\mathit{Buffer}(D), n) = \begin{cases} \mathit{Buffer}(\mathit{drop}(n)(D)) & \text{if } n < |D| \\ \mathit{Reqs}(n - |D|) & \text{if } n \geq |D| \end{cases} \tag{38}$$

and on the data channel

$$\mathit{next}_2(\mathit{Reqs}(m), d) = \begin{cases} \mathit{Reqs}(m - 1) & \text{if } m > 0 \\ \mathit{Buffer}(\langle d \rangle) & \text{if } m = 0 \end{cases} \tag{39}$$

$$\mathit{next}_2(\mathit{Buffer}(D), d) = \mathit{Buffer}(D \triangleright d) \tag{40}$$

are obtained with a few simple derivation steps.

Similarly, we develop the output functions for input on the requests channel

$$out_1(Reqs(m), n) = \langle\rangle \tag{41}$$
$$out_1(Buffer(D), n) = take(n)(D) \tag{42}$$

and on the data channel:

$$out_2(Reqs(m), d) = \begin{cases} \langle d\rangle & \text{if } m > 0 \\ \langle\rangle & \text{if } m = 0 \end{cases} \tag{43}$$
$$out_2(Buffer(D), d) = \langle\rangle \tag{44}$$

The state transition machine $M[send, abstr]$ implements the sender component in an iterative way processing input from the requests and from the data channels message by message. The state transition machine either stores the number of pending requests or the pending data items.

The resulting state transition machine is often described by a *state transition table*, compare Fig. 4. Each row of the state transition table defines a *transition rule* combining the results of the state transition functions $next_i$ and the output functions out_i . The component's state after a transition is denoted by a prime.

state		input		conditions	state′		output
		req	data				
$Reqs$	m	n			$Reqs$	$m+n$	$\langle\rangle$
$Reqs$	m		d	$m > 0$	$Reqs$	$m-1$	$\langle d\rangle$
$Reqs$	0		d		$Buffer$	$\langle d\rangle$	$\langle\rangle$
$Buffer$	D	n		$n \geq \lvert D\rvert$	$Reqs$	$n - \lvert D\rvert$	D
$Buffer$	D	n		$n < \lvert D\rvert$	$Buffer$	$drop(n)(D)$	$take(n)(D)$
$Buffer$	D		d		$Buffer$	$D \triangleright d$	$\langle\rangle$

Fig. 4. State transition table for the state transition machine $M[send, abstr]$

The formal method presented here has been applied to a variety of interactive components. We refer, for example, to transmission components [21], to synchronization components [21], multiplexers [10] or memory components [21].

7 Conclusion

We presented a formal method for transforming the communication oriented input/output behaviour of interactive components with multiple input channels and multiple output channels into a state-based implementation.

The state represents an abstraction of the input histories that records relevant information from the past which determines the component's behaviour in the future. The approach supports a smooth transition between the black-box and the glass-box views of an interactive component. The transformation contributes to a better understanding of state transition machines which tend to narrow the programmer's view to the local transitions, but do not exhibit the component's overall behaviour on an adequate level of abstraction.

As our particular contribution, we developed a simple theory how to implement interactive components with multiple input channels in a state-based way. The approach supports asynchronous components where input from different channels arrives in an independent way. The differentiations of multiary stream processing functions with respect to different input channels show a diamond property which ensures the confluence of the resulting state transition machine. Moreover, the output streams generated are independent of the particular computation path taken. We established sufficient criteria under which the resulting state transition machines compensate the interleaving of their input messages from different channels.

The sequential state transitions consume one input from one channel in one step. When we chain two sequential state transitions for inputs from different channels, we arrive at a 'parallel state transition' processing inputs from two different channels in one step. In this way, sequential state transition machines implicitly also describe parallel transitions processing two or more inputs from different channels in one step.

Future research will generalize the formal method to different types of timed streams [16]. For untimed systems, the history abstraction often splits into a control abstraction and a data abstraction. For timed systems, the history abstraction will additionally involve a timing abstraction to describe the processing schedule.

The history-oriented and the state-based description of software and hardware components allow complementary insights. Both formalisms show advantages and shortcomings with respect to compositionality, abstractness, verification, synthesis, and tool support. In long term, proven design methods must flexibly bridge the gap between functional behaviour and internal realization following sound refinement rules.

References

1. T. Bălănescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan. Communicating stream X-machines systems are no more than X-machines. *Journal of Universal Computer Science*, 5(9):494–507, 1999.
2. M. Breitling and J. Philipps. Step by step to histories. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST'2000)*, number 1816 in Lecture Notes in Computer Science, pages 11–25. Springer, 2000.
3. M. Broy. The specification of system components by state transition diagrams. Technical Report TUM-I9729, Technische Universität München, May 1997.
4. M. Broy. From states to histories: Relating state and history views onto systems. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *Series III: Computer and System Sciences*, pages 149–186. IOS Press, 2001.
5. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Monographs in Computer Science. Springer, 2001.
6. C. Dendorfer. *Methodik funktionaler Systementwicklung.* Herbert Utz Verlag Wissenschaft, München, 1996.

7. W. Dosch and A. Stümpel. From stream transformers to state transition machines with input and output. In N. Ishii, T. Mizuno, and R. Lee, editors, *Proceedings of the 2nd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'01)*, pages 231–238. International Association for Computer and Information Science (ACIS), 2001.
8. W. Dosch and A. Stümpel. History abstractions of a sequential memory component. In B. Gupta, editor, *Proceedings of the 19th International Conference on Computers and their Applications (CATA-2004)*, pages 241–247. International Society for Computers and their Applications (ISCA), 2004.
9. W. Dosch and A. Stümpel. Transforming stream processing functions into state transition machines. In W. Dosch, R. Lee, and C. Wu, editors, *Software Engineering Research and Applications (SERA 2004)*, number 3647 in Lecture Notes in Computer Science, pages 1–18. Springer, 2005.
10. W. Dosch and W. L. Yeung. High–level design of a ternary asynchronous multiplexer. In R. Hurley and W. Feng, editors, *Proceedings of the 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2005)*, pages 221–228. International Society for Computers and their Applications (ISCA), 2005.
11. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.
12. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North–Holland, 1974.
14. N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.
15. K. Meinke and J. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Background: Mathematical Structures*, volume 1 of *Handbook of Logic in Computer Science*, pages 189–411. Oxford University Press, 1992.
16. L. Motus, M. Meriste, and W. Dosch. Time-awareness and proactivity in models of interactive computation. ETAPS-Workshop on the Foundations of Interactive Computation (FInCo 2005). *Electronic Notes in Theoretical Computer Science*, 141(5):69–95, 2005.
17. Object Management Group (OMG). *OMG Unified Modeling Language Specification, 3. UML Notation Guide, Part 9: Statechart Diagrams*, Mar. 2003.
18. B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, 1996.
19. P. Scholz. Design of reactive systems and their distributed implementation with statecharts. PhD Thesis, TUM-I9821, Technische Universität München, Aug. 1998.
20. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
21. A. Stümpel. *Stream Based Design of Distributed Systems through Refinement.* Logos Verlag Berlin, 2003.
22. W. Wechler. *Universal Algebra for Computer Scientists.* Number 25 in EATCS Monographs on Theoretical Computer Science. Springer, 1992.
23. P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.
24. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 1–148. Oxford University Press, 1995.

Introducing Debugging Capabilities to Natural Semantics[⋆]

Alberto de la Encina, Luis Llana, and Fernando Rubio

Departamento de Sistemas Informáticos y Programación
Facultad de Informática, Universidad Complutense de Madrid
C/ Prof. José García Santesmases, 28040 Madrid, Spain
{albertoe,llana,fernando}@sip.ucm.es

Abstract. Reasoning about functional programs is simpler than reasoning about their imperative counterparts. However, finding bugs in lazy functional languages has been more complex until quite recently. The reason was that not much work was done on developing practical debuggers. Fortunately, several debuggers exist nowadays. One of the easiest to use Haskell debuggers is Hood, whose behavior is based on the concept of observation of intermediate data structures. However, although using Hood can be simple when observing some structures, it is known that it can be hard to understand how it works when dealing with complex situations.

In this paper, we formalize the behavior of the Hood debugger by extending Sestoft's natural semantics. Moreover, we also indicate how to derive an abstract machine including such debugging information. By doing so, we do not only provide a formal foundation, but we also provide an alternative method to implement debuggers. In fact, we have already implemented a prototype of the abstract machine commented in this paper.

Keywords: Functional programming, debugging, semantics.

1 Introduction

Classically, researchers working on lazy functional programming have devoted their efforts to show the elegance of their paradigm. In this sense, many relevant works show the clarity of the functional semantics. However, not so much effort was devoted in the past to deal with other more practical aspects of the paradigm, like the development of debuggers. Debugging lazy functional programs is an important and not trivial task. Although not much attention was paid to it in the past (see e.g. [23]), during the last years there have been several proposals for incorporating execution traces to lazy functional languages. For instance, we can highlight the work done with Hat [22,24], HsDebug [6], the declarative debuggers Freja [14,15] and Buddha[19], and specially the work done with the Haskell Object Observation Debugger (Hood) [7,20]. All of these works provide practical tools for debugging. However, their theoretical foundations are frequently missing.

[⋆] Work partially supported by the MCYT project TIC2003-07848-C02-01, the JCCLM project PAC-03-001, and the Marie Curie project MRTN-CT-2003-505121/TAROT.

All of the debuggers commented before are designed to be used with the language Haskell [18], the *de facto* standard in the lazy-evaluation functional programming community. The approaches followed in each of the previous debuggers are quite different, both from the point of view of the user of the system and from the implementation point of view. For instance, from an implementation point of view, most of them strongly depend on the compiler being used, while that is not the case in Hood. From the user point of view, Freja and Buddha are question-answer systems that directs the programmer to the cause of an incorrect value, while Hat allows the user to travel backwards from a value along the redex history leading to it. In this paper we will not concentrate on those differences (the interested reader can find a detailed comparison between Freja, Hat and Hood in [1], while [16] presents a common framework to describe all of them).

In this paper we will concentrate on how to provide a formal foundation for one of the debuggers commented before. Among all of the Haskell debuggers, Hood has an interesting advantage over the rest, as it is the only one that can be used with different Haskell compilers. The reason is that it is implemented as an independent library that can be used from any Haskell compiler, provided that the compiler implements some quite common extensions. Basically, it only needs to have access to a primitive allowing to perform an input/output action without being inside of the IO monad. Fortunately, most Haskell compilers include an `unsafePerformIO` primitive (or something similar to it). In fact, Hood can currently be used with the Glasgow Haskell Compiler, Hugs98, the Snowball Haskell compiler, and also with nhc98. Due to its portability, Hood has become one of the most used Haskell debuggers.

Hood works by using a relatively simple way. First, the programmer instruments the program marking the variables he wants to observe and, after finishing the execution of the program, the system produces a printing of their final value. Let us remark that *final value* does not necessarily mean normal form, but evaluation to the degree required by the computation. Unfortunately, it is sometimes tricky to understand what should be observed by using Hood in special situations. In fact, as the author recognizes in [7], the semantics of `observe` (the principal debugging combinator of Hood) should be clearly defined to help understanding these situations.

In this paper we propose a formalization of the Hood debugger allowing both to reason about it and to implement it in a different way. In order to do that, we will use as starting point the semantical foundations presented in [10,13,21,11]. More precisely, what we propose is an extension of Sestoft's natural semantics[1] [21] that incorporates new rules to deal with Hood observations. Moreover, we also comment how to derive an equivalent abstract machine. By doing so, we obtain two main benefits. First, the semantics of the observations are clearly defined. Second, we can reuse the work done in [3,4] to be able to actually implement a debugging system. In fact, we have already implemented a prototype of the abstract machine commented in the paper. Summarizing, we propose a cleaner and more modular approach to the trace problem in lazy

[1] Sestoft's semantics is an extension of the original natural semantics introduced by Launchbury in [13].

functional programming, allowing to easily provide both implementations and formal foundations for them.

The rest of the paper is structured as follows. In the next section we introduce the main characteristics of Hood. Then, in Section 3 we briefly review the main characteristics of Sestoft's semantics. Next, in Section 4 we show how to modify the original semantics to include debugging information equivalent to that obtained by Hood. Afterwards, in Section 5 we sketch how to introduce an abstract machine equivalent to our new semantics. Finally, in Section 6, in addition to presenting our conclusions and lines for future work, we briefly comment some details about our current implementation of the debugger.

2 An Introduction to Hood

In this section we show the basic ideas behind Hood. The interested reader is referred to [7,8] for more details about it.

When debugging programs written in an imperative language, the programmer can explore not only the final result of the computation, but also the intermediate values stored in the variables being used by the program. Moreover, it is simple to follow how the value of each variable changes along time. Unfortunately, this task is not that simple when dealing with lazy functional languages. However, Hood allows the programmer to observe something similar to it. In fact, Hood provides a way to observe any intermediate structure appearing in a program. Moreover, by using GHood [20] we can also observe the evolution in time of the evaluation of the structures under observation.

The core of Hood is the `observe` combinator. The type declaration of this combinator is: `observe :: String -> a -> a` From the evaluation point of view, `observe` only returns its second value. That is, `observe s a = a`. However, as a side effect, the value associated to `a` will be squirrelled away, using the label `s`, in a file that will be analyzed after the evaluation finishes. It is important to remark that `observe` returns its second parameter in a completely lazy, demand driven manner. That is, the evaluation degree of `a` is not modified by introducing the observation, in the same way that it is not modified when applying the identity function `id`. Thus, as the evaluation degree is not modified, Hood can deal with infinite structures.

It is important to remark that Hood does not only observe simple data types. In fact, it can observe anything appearing in a Haskell program. In particular, we can observe functions. For instance, `observe "sum" sum (4:2:5:[])` will observe the application of function `sum` to its parameter. Before executing it, "sum" will have no observation attached to it. Then, as the evaluation takes place, it will first be recorded that it was necessary to evaluate the first element of the list ("4"), then, the second, and so on. Finally, it will return the observation

```
-- sum
  { \ (4:2:5:[]) -> 11 }
```

Notice that what we observe can be read as *when the function receives as input the list 4:2:5:[], it returns as output the value 11.* The elements `4`, `2` and `5` appear explicitly because they were really demanded to evaluate the output. However, when observing something like `observe "length" length (4:2:5:[])` we will obtain the following observation:

```
-- length
   { \ (_:_:_:[]) -> 3 }
```

That is, we are observing a function that when it receives a list with three elements it returns the number 3 without evaluating the concrete elements appearing in the list. Note that it is only relevant the number of elements, but not the *concrete* elements. As it can be expected, higher-order functions can also be observed. This is done in a similar way as in the previous cases. For instance, in the following example we observe a higher-order function that produces as output an infinite list: `take 3 (observe "iterate" iterate (+2) 4)` The observation obtained is:

```
-- iterate
   { \ { \ 8 -> 10,  \ 6 -> 8,  \ 4 -> 6 }
       4  ->  4 : 6 : 8 : _ }
```

That is, it observes that `iterate` is a function that returns `4:6:8:_` when it receives as second parameter `4` and as first parameter a function `(+2)` that has been observed with three different input values: 4, 6 and 8. Note that only three elements of the infinite list where demanded. Thus, the rest of the infinite list remains unevaluated.

It is important to remark that Hood has to analyze who was the responsible of evaluating each component. That is, if we are observing a structure in a given environment, we are not interested in the parts of the structure that were evaluated due to other environments. For instance, if we are observing function `length` in the following example

```
  let xs = take 5 (1:2:3:4:5:6:7:[])
  in (observe "length" length xs) + (sum xs)
```

we will obtain the output

```
-- length
   { \ (_:_:_:_:_:[]) -> 5
   }
```

That is, even though all the elements of the list `xs` where actually computed (due to function `sum`), they were not needed at all to compute any application of the function `length`.

3 A Semantics for Lazy Evaluation

We begin by reviewing the language and semantics given by Sestoft [21] as an improvement to Launchbury's semantics. A well-known work from Launchbury [13] defines a big-step operational semantics for lazy evaluation. The only

$e \rightarrow x$	-- variable
$\mid\ \lambda x.e$	-- lambda abstraction
$\mid\ e\ x$	-- application
$\mid\ \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$	-- recursive let
$\mid\ C\ \overline{x_i}$	-- constructor application
$\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i}$	-- case expression

Fig. 1. Sestoft's normalized λ-calculus

machinery needed is an explicit heap where bindings are kept. A heap is considered to be a finite mapping from variables to expressions, i.e. duplicated bindings to the same variable are disallowed. The proposals of Launchbury and Sestoft share the language given in Figure 1 where $\overline{A_i}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities. It is a normalized λ-calculus, extended with recursive **let**, constructor applications and **case** expressions. Sestoft's normalization process forces constructor applications to be saturated and all applications to only have variables as arguments. Weak head normal forms are either lambda abstractions or constructions. Throughout this section, w will denote (weak head) normal forms.

Sestoft's semantic rules are given in Figure 2. There, a judgement $\Gamma : e \Downarrow \Delta : w$ denotes that expression e, with its free variables bound in heap Γ, reduces to normal form w and produces the final heap Δ. Let us remark that, if the configuration $\Gamma : e$ reduces to normal form, then Δ and w are unique. Thus, in the rest of the paper we will assume this fact to avoid introducing extra quantifiers in our formalizations. Let us also remark that the notation $\hat{e}$ in rule *Letrec* means the replacement of the variables x_i by the fresh pointers p_i. This is the only rule where new closures are created and added to the heap. We use the term *pointers* to refer to dynamically created free variables, bounded to expressions in the heap, and the term *variables* to refer to (lambda-bound, let-bound or case-bound) program variables. We consistently use $p, q, \ldots$ to denote pointers and $x, y, \ldots$ to denote program variables. The notation $\Gamma[p \mapsto e]$ means that $(p \mapsto e) \in \Gamma$, and $\Gamma \cup [p \mapsto e]$ represents the disjoint union of Γ and $(p \mapsto e)$.

4 Semantics with Debugging Features

4.1 Low Level Details of the Real Behavior of Hood

In order to better understand how we should define the rules of our semantics, it is convenient to start commenting some details of the implementation of Hood. When Hood is in action, it makes annotations that have this form: (*portId*, *parent*, *change*). The *portId* corresponds to a pointer to the place where the annotation is made: in the implementation it corresponds to a line number in the file of annotations. In order to be able to post-process the file, when a function or a constructor is evaluated, its arguments need to know the place where they were invoked. That is, we need to access to the *parent* of the arguments; formally, parent is a tuple (*observeParent*, *observePort*), where

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\Gamma : C\,\overline{p_i} \Downarrow \Gamma : C\,\overline{p_i} \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda x.e' \quad \Delta : e'[p/x] \Downarrow \Theta : w}{\Gamma : e\,p \Downarrow \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow \Delta \cup [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \Downarrow \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow \Delta : w} \text{ where } \overline{p_i} \text{ are fresh} \qquad \textit{Letrec}$$

$$\frac{\Gamma : e \Downarrow \Delta : C_k\ \overline{p_j} \quad \Delta : e_k[\overline{p_j/x_{kj}}] \Downarrow \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \Downarrow \Theta : w} \qquad \textit{Case}$$

Fig. 2. Sestoft's natural semantics

$e \to x$	-- variable
$\mid\ x^{@str}$	-- **observed variable**
$\mid\ \lambda x.e$	-- lambda abstraction
$\mid\ e\ x$	-- application
$\mid\ \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$	-- recursive let
$\mid\ C\ \overline{x_i}$	-- constructor application
$\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}$	-- case expression
$\mid\ p^{@(r,s)}$	-- **observed pointer (internal)**
$\mid\ \lambda^{@(r,s)}x.e$	-- **observed lambda abstraction (internal)**

Fig. 3. Sestoft's normalized λ-calculus extended

observeParent is the *portId* of the parent and the *observePort* is the position of the argument. Finally, parameter *change* corresponds to the type of observation carried out, and it can have one of the following forms:

– *Observe String* is generated when we enter in a closure annotated with the corresponding string. This kind of observation has no parent, actually it has the general parent, that corresponds with (0, 0). This is the first annotation generated when we start the evaluation of an annotated closure.
– *Enter* is generated when the evaluation of a closure starts.
– *Cons Int String* is generated when the evaluation arrives at a constructor. The integer appearing in the annotation is the arity of the closure, and the string is the name of the constructor. The children of this closure will be

annotated with ($parentPortId$, 0), ($parentPortId$, 1), $\ldots$($parentPortId$, $arity$). In this way, it is easy to reconstruct the evaluation tree where $parentPortId$ is the place where the annotation $Cons$ has been written.
– Fun is generated when the closure evaluation arrives at a lambda expression. In the observations of Hood, lambdas have only one argument and one result. When the evaluation finishes, the tree is analyzed to get the list of all arguments and results. The argument of the lambda is annotated with the following parent ($parentPortId$, 0) and the result of the lambda is annotated with ($parentPortId$, 1) where $parentPortId$ is the place where the annotation Fun has been written.

Therefore, in Hood it is not only possible to observe the normal forms of the closures, but also when the closures start the evaluation. Using this, it is easy to observe which closures are demanded by another one. These annotations are processed and are shown in a pretty way to the user.

4.2 Hood Semantics

Let us consider now how to introduce Hood-like observations in the semantics. Let us remind that Hood users can annotate their programs to mark which structures are to be observed. Thus, in our case we also have to be able to annotate any structure. Besides we need to write these annotations in a *file* in order to post-process it.

To achieve this, the judgments will have the form $\Gamma : e \looparrowright f \Downarrow \Delta : w \looparrowright f'$. As in the previous section, that means that expression e is evaluated in the heap Γ, we obtain as result the expression w and the new heap is Δ. The difference is that now we have added the file f where we write the annotations. The information in the file is added sequentially. Thus, we will write $f \circ \langle ann \rangle$ to indicate that we add the annotation ann at the end of the file f. The annotations that we will make will have the same form as the ones of Hood shown in the previous subsection: ($portId$, $parent$, $change$). In our case, the $portId$ component will make reference to a line in the file; typically it will be the line where a function or a pointer is observed. To handle this we will need the function $length\ f$ that returns the length of the file f. We will consider that the first line in the file is the 0 line.

We also have to be able to annotate any structure. This can be trivially done by allowing to annotate as *observable* any variable. Thus, we only need to slightly modify the language to include an extra construction as shown in Figure 3. The expression $x^{@str}$ is the equivalent to the Hood expression `observe str x`. Note that, according to the syntax, these observations cannot appear directly in *applications* or *constructor applications*. However, this is not a drawback, since they may appear in a *recursive let*. Once the language allows to include observations, we have to deal with them in the rules. Besides we need a new kind of normal form $\lambda^{@(r,s)}x.e$: an observed lambda expression; and a new kind of observed pointers $p^{@(r,s)}$: pointers that are observed and refer to their parents. It is important to note that the programmer is not allowed to write these kind of expressions, as they only appear as auxiliary expressions in the rules.

$$\frac{\Gamma : p^{@(length\ f,0)} \looparrowright f \circ \langle 0\, 0\ Observe\ str\rangle \ \Downarrow\ \Delta : w \looparrowright f'}{\Gamma : p^{@str} \looparrowright f \ \Downarrow\ \Delta : w \looparrowright f'} \qquad Var@S$$

$$\frac{\Gamma : p \looparrowright f \circ \langle r\ s\ Enter\rangle \ \Downarrow\ \Delta : C\ \overline{p_i}^{k} \looparrowright f'}{\Gamma : p^{@(r,s)} \looparrowright f \ \Downarrow\ \Delta \cup [\overline{q_i \mapsto p_i^{@(length\ f',i)}}] : C\ \overline{q_i} \looparrowright f' \circ \langle r\ s\ Cons\ k\ C\rangle}\ \overline{q_i}\text{ fresh} \qquad Var@C$$

$$\frac{\Gamma : p \looparrowright f \circ \langle r\ s\ Enter\rangle \ \Downarrow\ \Delta : \lambda x.e \looparrowright f'}{\Gamma : p^{@(r,s)} \looparrowright f \ \Downarrow\ \Delta : \lambda^{@(r,s)} x.e \looparrowright f'} \qquad Var@F$$

$$\Gamma : \lambda^{@(r,s)} x.e \looparrowright f \ \Downarrow\ \Gamma : \lambda^{@(r,s)} x.e \looparrowright f \qquad Lam@$$

$$\frac{\begin{array}{c}\Gamma : e \looparrowright f \ \Downarrow\ \Delta : \lambda^{@(r,s)} x.e' \looparrowright f' \\ \Delta \cup \left[\begin{array}{l} q \mapsto e'[q'/x], \\ q' \mapsto p^{@(length\ f',0)} \end{array}\right] : q^{@(length\ f',1)} \looparrowright f' \circ \langle r\ s\ Fun\rangle \ \Downarrow\ \Theta : w \looparrowright f'' \\ \text{where } q,\ q' \text{ fresh}\end{array}}{\Gamma : e\ p \looparrowright f \ \Downarrow\ \Theta : w \looparrowright f''} \qquad App@$$

Fig. 4. Hood's natural semantics

The rules in the original Sestoft's natural semantics (Figure 2) do not deal with observations so they are rewritten with the natural modification to include the annotation file. For instance our new *Case* rule is

$$\frac{\Gamma : e \looparrowright f \ \Downarrow\ \Delta : C_k\ \overline{p_j} \looparrowright f' \quad \Delta : e_k[\overline{p_j/x_{kj}}] \looparrowright f' \ \Downarrow\ \Theta : w \looparrowright f''}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \looparrowright f \ \Downarrow\ \Theta : w \looparrowright f''}$$

The rest of the rules in Figure 2 should be modified in a similar way. However, in addition to rewriting these rules, it is also necessary to write completely new rules to deal with the new constructions. The new rules we add to the system are those shown in Figure 4. Let us briefly describe their meaning:

– Rule *Var*@*S*. When we have to evaluate a closure annotated with the string *str*, we have to generate an annotation in the file $\langle 0\, 0\ Observe\ str\rangle$ and we have to continue to evaluate that closure but with an annotation that indicates its parent, in this case $p^{@(n,0)}$ ($n = length\ f$ is the length of the file at that point of the evaluation).
– Rule *Var*@*C*. When evaluating an expression such as $p^{@(r,s)}$, a new annotation $\langle r\ s\ Enter\rangle$ is generated indicating that we enter to evaluate that closure. Then $p^{@(r,s)}$ evaluates to a constructor, so the observation $\langle r\ s\ Cons\ k\ C\rangle$ is generated. This indicates that the closure whose parent is (r, s) has been reduced to the constructor C (whose arity is k). New clousures pointing to each argument of that constructor are generated. These closures are annotated to indicate that they are being observed. Moreover, in this annotation we must indicate its position in the constructor an that its parent is in the corresponding line in the file.

- Rule *Var@F*. Now we have to evaluate $p^{@(r,s)}$. As in the previous case, we generate the annotation $\langle r\ s\ \mathit{Enter}\rangle$, but in this case p reduces to a function. Now we have to continue to evaluate a new kind of normal form $\lambda^{@(r,s)}x.e$. A lambda whose parent is (r,s) and that it is being observed.
- Rule *Lam@* establishes that $\lambda^{@(r,s)}x.e$ is actually a normal form.
- Rule *App@* is the fundamental part of the new semantics. We are evaluating the application of an observed function. First, we generate the annotation in the file indicating that we are applying an observed function (note that *length* f' is the line where the annotation is made). Then we mark its argument as observable, and we use (*length* f', 0) as their parent. In order to observe the result we create a new observed closure whose parent is (*length* f', 1). The ports are different to remember that one is the argument and the other is the result of the lambda.

  Note that it is not necessary to specify the application to an observed pointer $e\ p^{@(r,s)}$. The reason is that, in the syntax, we have restricted the places where an observed variable may appear, and in the rules we never substitute a variable by an observed pointer.

4.3 Correctness and Equivalences

One important thing we must prove is that the observation marks do not change the meaning of an expression. That is, if we evaluate a marked expression and the equivalent one without marks we should obtain the same normal form. Let us remark that this property must be satisfied because Hood observations do not modify the normal computation of Haskell programs.

The first difference of our semantics with respect to the original one consists in the observation marks. Thus, in order to compare them we need to provide a function to *remove* the observations. Thus, we define the following simple function that transforms any Sestoft's expression with observations, that we call Sestoft$^{@}$, into an expression without observations.

Definition 1. *We define the function that erase the observations* $\mathbb{R}$: Sestoft$^{@}$ $\rightarrow$ Sestoft. *It is recursively defined, and all cases are trivial but the case of observed expressions:*

$$\begin{aligned}
\mathbb{R}\ x^{@str} &\overset{def}{=} x\\
\mathbb{R}\ p^{@(r,s)} &\overset{def}{=} p\\
\mathbb{R}\ \lambda^{@(r,s)}x.e &\overset{def}{=} \lambda x.\mathbb{R}\ e
\end{aligned}$$

This function is extended to work with heaps, and configurations. Basically, $\mathbb{R}\ \Gamma$ *corresponds with* $\{p \mapsto \mathbb{R}\ e \mid (p \mapsto e) \in \Gamma\}$ *and* $\mathbb{R}\ (\Gamma : e) = \mathbb{R}\ \Gamma : \mathbb{R}\ e$.

But the most difficult problem is that in our rules we introduce new pointers and the expressions appearing in the rules contain pointers. Thus, we have to prove that the expressions appearing in both formalisms are *essentially* the same. The pointers are kept in a heap; our new rules *Var@C* and *App@* add new pointers to the heap. These pointers points to the original ones, but they are marked remembering that they are under observation. We would like to define $\Gamma : w \sqsubseteq \Gamma' : w'$ if w in Γ has the same value as w' in Γ', that is, if we obtain the same expressions by following the pointers. We can do that as follows: if p is a pointer in w and $(p \mapsto e) \in \Gamma$, let us substitute the occurrences of p in w by e. By doing so, we obtain new expressions that may have pointers; in that case, we iterate the process. Analogously, we perform the same process to deal with w'. If both processes end then we look at the final expressions: if both expressions are the same we can say that w and w' has the same value. The problem appears when one of the processes does not end. In that case we have to take the *limit* of both sequences: w and w' have the same value if one sequence is a subsequence of the other.

Definition 2. *Let e be an expression and Γ be a heap.*

– *We denote by $rp\ e$ the substitution of all pointers in e by the symbol $\perp$.*
– *We denote $\Gamma\ e$ as the application of Γ to the expression e. It is recursively defined, and all cases are trivial but the case of a pointer:*

$$\begin{array}{ll} \Gamma\ p \overset{def}{=} rp\ e & \textit{if } (p \mapsto e) \in \Gamma \\ \Gamma\ p \overset{def}{=} \perp & \textit{if } (p \mapsto e) \notin \Gamma \end{array}$$

Definition 3. *Let e, e' be expressions and Γ, Γ' be heaps. Let us consider the possibly infinite sequences*

$$s = [rp\ e, \Gamma\ e, \Gamma^2\ e, \Gamma^3\ e, \ldots] \qquad \textit{and} \qquad s' = [rp\ e', \Gamma'\ e', \Gamma'^2\ e', \Gamma'^3\ e', \ldots]$$

We say that:

– $\Gamma : e \sqsubseteq \Gamma' : e'$ *if*
  - $rp\ e = rp\ e'$
  - $\forall i\, \exists j \geq i,\ rp\ \Gamma^i\ e = rp\ \Gamma'^j\ e'$
  - $\forall j,\ rp\ \Gamma'^j\ e' \neq rp\ \Gamma'^{j+1}\ e' \Rightarrow \exists i \leq j,\ rp\ \Gamma^i\ e = rp\ \Gamma'^j\ e'$
– $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' : e'$ *if* $\Gamma : e \sqsubseteq \mathbb{R}\ (\Gamma' : e')$

According to the previous definition, if $\Gamma : e \sqsubseteq \Gamma' : e'$ we have that e and e' are essentially the same, and the only differences may appear in the pointers. First we require that $rp\ e = rp\ e'$: if e is a lambda expression, an application, a recursive let, a constructor or a case expression, so must be e', and vice versa; the constructors and variables appearing at the top level must be the same. We do not require that the pointers are the same or that they point to the same expressions; what we require is that whenever there is a pointer in e, $(p \mapsto e_1) \in \Gamma$, then there must be a sequence of pointers $[q_1 \mapsto q_2, \ldots q_n \mapsto e_n] \subseteq \Gamma'$ such that q_1 appears in e', and if we apply all the corresponding substitutions in e and e' and then remove the pointers, we obtain the same expression.

Now we need to prove the equivalence between the evaluation of a marked expression and the corresponding one without marks. The following theorem proves that:

Theorem 1. *For all $e \in Sestoft$ and all $e^{@} \in Sestoft^{@}$ such that $e = \mathbb{R}\ e^{@}$ then $\{\,\} : e^{@} \looparrowright \langle\,\rangle \Downarrow \Delta^{@} : w^{@} \looparrowright f$ iff $\{\,\} : e \Downarrow \Delta : w$ and $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$*

In order to prove this theorem we need to take into account some considerations. We have to substitute the rule *Var* with this one:

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma[p \mapsto e] : p \Downarrow \Delta \diamond [p \mapsto w] : w} \qquad Var'$$

The equivalence between the evaluation is maintained. The only differences with the original one is that now we do not remove the closure $(p \mapsto e)$, that is under evaluation, to evaluate the expression e. Besides, in this case $\Delta \diamond [p \mapsto w]$ means to update in Δ the expression corresponding with the pointer p with the expression w. It is very easy to prove the equivalences between the rules. In the evaluation of $\Gamma : e$ the closure $(p \mapsto e)$ has not been used, otherwise we would have entered in a "black hole", as we need p to evaluate p. In that case, evaluation would not have finished.

Proposition 1. *$\Gamma : e \Downarrow \Delta : w$ iff $\Gamma : e \Downarrow \Delta : w$ with the rule Var'.*

From now on, in the rest of this section we will consider that we use the rule Var' instead of rule *Var*. We also need to observe some properties that are invariant during the evaluation.

Definition 4. *$\Gamma : e$ is a* good *configuration if all the reachable pointers from e are bound in the heap.*

Proposition 2. *Let $\Gamma : e$ be a good configuration. If $\Gamma : e \Downarrow \Delta : w$ then $\Delta : w$ and $\Delta : e$ are good configurations.*

Finally we prove a proposition that is more general than the original Theorem 1.

Proposition 3. *Let be $e, e' \in Sestoft$, all $e^@, e'^@ \in Sestoft^@$, $\Gamma : e$, $\Gamma^@ : e^@$, $\Gamma : e'$ and $\Gamma^@ : e'^@$ good configurations, such that $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^@ : e^@$ and $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^@ : e'^@$ then:*
$\Gamma^@ : e^@ \looparrowright f \Downarrow \Delta^@ : w^@ \looparrowright f'$ iff $\Gamma : e \Downarrow \Delta : w$, $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^@ : w^@$ and $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^@ : e'^@$

Proof. The proof is made by rule induction. In order to make the proof easier to read, we will drop the observation file from the rules since it does not participate in the evaluation of the expressions. This file is only a side effect of the evaluation. Notice that, if configurations are good, the last case considered in Definition 2 ($(p \mapsto e) \notin \Gamma$) cannot occur. However, the definition must take the case into account for the sake of completeness.

As a corollary of the previous proposition, we have that Theorem 1 holds. Due to lack of space, we do not show the details of the proofs. However, the interested reader can find all the details of the proofs in the extended version of this paper [2].

5 Towards Obtaining an Abstract Machine

Once we have extended the natural semantics to deal with observations, the next step is to define an abstract machine equivalent to our new semantics. By doing so, we will be able to derive an implementation, and we will also be able to proof the correctness of such implementation.

Sestoft introduced in [21] several abstract machines in sequence, respectively called *Mark-1*, *Mark-2* and *Mark-3*. We will use the machine *Mark-2* for deriving the new rules for our semantics because it is close enough to reality and it does not have many low level details.

The main theorem proved by Sestoft, is that successful derivations of the machine are exactly the same as those of the semantics. The reason why the environments are needed is because control expressions, lambda expressions and alternatives keep their original variables and in execution we need to know their associated pointers. Basically, the machine consists in a flattening of the semantic tree.

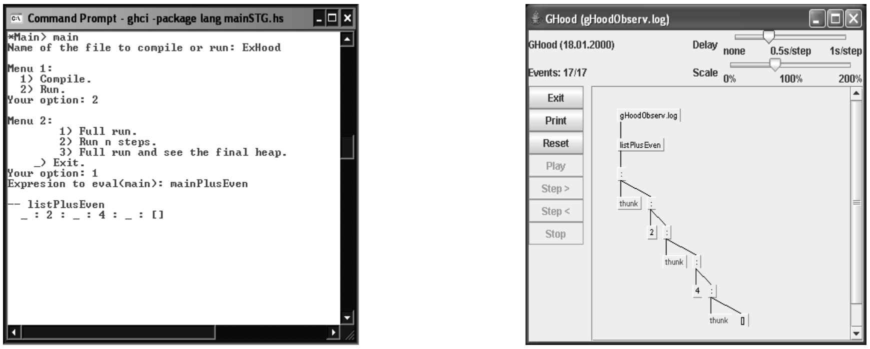
Following the same ideas we have derived new rules for the new semantic rules. These new rules are presented in the extended version of this paper. To include observations in our machine we need to add some modifications to the original machine. First we need a new column of side effects for the rules containing the observations. We would have to rewrite all the rules appearing in the original machine to add a side effect column. Second, we need to add a new type of objects in the stack, where this kind of objects $@(r, s)$ corresponds to pending observations.

Following the same approach as Sestoft, it is easy to prove that the semantic rules and the machine rules are equivalent. The details of the proof can be found in the extended version of this paper [2]. The proof is based on the concept of balanced computations. A *balanced trace* is a computation where the initial and final stack are the same, and in which every intermediate stack extends the initial one.

6 Conclusions and Future Work

In this paper we have provided a formal semantics for the Hood debugger. In particular, we have described how to embed Hood inside Sestoft's natural semantics. The main aim of our work is to clarify the formal foundations of the debugger, but we also provide an alternative method for implementing it. Let us also remark that the approach we have followed to codify Hood inside the natural semantics can also be used to provide a formal foundation for any other Haskell debugger. In this sense, it could be used as a common framework for describing (and also implementing) all of them.

In order to obtain an implementation of our modified semantics, we have used the abstract machine commented in the previous section. To derive an implementation of it, we have reused the work done in [4]. We would like to point out that in our programming environment (see Figure 5) we do not only generate plain text as observations. In fact, the user can choose between observing the output of the observations by using plain text (with the same style as in Hood) or by using the GHood[20] graphical environment. By using GHood, we do not only observe the degree of evaluation of the observed structures, but also the order of evaluation of it (see Figure 5 right).



```
*Main> main
Name of the file to compile or run: ExHood

Menu 1:
  1) Compile.
  2) Run.
Your option: 2

Menu 2:
        1) Full run.
        2) Run n steps.
        3) Full run and see the final heap.
    _) Exit.
Your option: 1
Expresion to eval(main): mainPlusEven

-- listPlusEven
  _ : 2 : _ : 4 : _ : []
```

Fig. 5. Debugging environment: Produces text output (left) or graphical output (right)

We are currently working on an extension of our semantics to deal with debugging issues in parallel dialects of Haskell. As a first case study, now we are working with the language Eden[12,17]. In fact, in [5] we have already implemented a parallel extension of Hood to deal with Eden programs. Our extension includes a parallelization of the basic Hood library and also a set of tools to allow checking the amount of speculative work that the parallel programs are performing. However we have not provided yet a semantics for our parallel observations. In this sense, we plan to extend the work done in the present paper to deal with parallel semantics. In order to do that, the best choice is to try to embed the debugging method inside the Jauja language[9], a very simple parallel functional language whose semantics are clearly defined in terms of an extension of Launchbury's natural semantics.

Acknowledgments

The authors would like to thank the anonymous referees for valuable comments on a previous version of this paper.

References

1. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages (IFL'00)*, LNCS 2011, pages 176–193. Springer-Verlag, 2001.
2. A. Encina, L. Llana, and F. Rubio. Introducing debugging capabilities to natural semantics (extended version). `http://dalila.sip.ucm.es/~albertoe/publics/psiExtd.ps`, 2006.
3. A. Encina and R. Peña. Proving the correctness of the STG machine. In *Implementation of Functional Languages (IFL'01)*, LNCS 2312, pages 88–104. Springer-Verlag, 2001.
4. A. Encina and R. Peña. Formally deriving an STG machine. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 102–112. ACM, 2003.

5. A. Encina, I. Rodríguez, and F. Rubio. Testing speculative work in a lazy/eager parallel functional language. In *Languages and Compilers for Parallel Computing (LCPC'05)*, LNCS. Springer-Verlag, 2005.
6. R. Ennals and S. Peyton Jones. HsDebug: Debugging lazy programs by not being lazy. In *Proceedings of the 7th Haskell Workshop*, pages 84–87. ACM, 2003.
7. A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*. Technical Report of the University of Nottingham, 2000.
8. A. Gill. Hood homepage. `http://www.haskell.org/hood`, 2006.
9. M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, LNCS 1058, pages 303–321. Springer-Verlag, 2003.
10. N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Symposium on Principles of Programming Languages (POPL'86)*, pages 296–306. ACM, 1986.
11. N. Jones and M. Rosendahl. Higher order minimal function graphs. *Journal of Functional and Logic Programming*, 2, Feb. 1997.
12. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *Implementation of Functional Languages (IFL'00)*, LNCS 2011, pages 71–88. Springer-Verlag, 2001.
13. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. Conference on Principles of Programming Languages, POPL'93*. ACM, 1993.
14. H. Nilsson. Declarative debugging for lazy functional languages. PhD thesis, Depat. of Computer and Information Science, Linköping University, Sweden, 1998.
15. H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671,2001.
16. C. Pareja, R. Peña, F. Rubio, and C. Segura. Adding traces to a lazy functional evaluator. In *Eurocast 2001*, LNCS 2178, pages 627–641. Springer-Verlag, 2001.
17. R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *Principles and Practice of Declarative Programming (PPDP'01)*, pages 187–198. ACM, 2001.
18. S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org`, February 1999.
19. B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell 98. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 230–240. ACM, 2003.
20. C. Reinke. GHood — graphical visualization and animation of Haskell object observations. In *Proceedings of the 5th Haskell Workshop*, volume 59 of *ENTCS*. Elsevier Science, 2001.
21. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.
22. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Programming Languages, Implementations, Logics and Programs (PLILP'97)*, LNCS 1292, pages 291–308. Springer-Verlag, 1997.
23. P. Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, August 1998. Functional Programming Column.
24. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multipleview tracing for Haskell: a new Hat. In *Proc. of the 5th Haskell Workshop*, pages 151–170, 2001.

Solution Strategies for Multi-domain Constraint Logic Programs

Stephan Frank, Petra Hofstedt, Peter Pepper, and Dirk Reckmann

Berlin University of Technology, Germany
{sfrank,ph,pepper,reckmann}@cs.tu-berlin.de

Abstract. We integrated a logic programming language into Meta-S, a flexible and extendable constraint solver cooperation system, by treating the language evaluation mechanism resolution as constraint solver. This new approach easily yields a CLP language with support for solver cooperation that fits nicely into our cooperation framework.

Applying the strategy definition framework of Meta-S we define classical search strategies and more sophisticated ones and discuss their effects on an efficient evaluation of multi-domain constraint logic programs by illustrating examples.

1 Introduction

To allow an efficient processing, in constraint programming the constraint solving algorithms are limited to restricted domains, e.g. linear or non-linear arithmetics, boolean constraints, finite domain constraints etc. However, many interesting problems are intuitively expressed as multi-domain descriptions. In multi-domain constraint problems every constraint may come from a different constraint domain and as well can be built itself using symbols of different domains. The cost and time for developing new constraint solvers that are able to handle such problems are significant. A different approach that has been researched actively during recent years is the collaboration of a number of individual (preexisting) constraint solvers managed by some meta mechanism which enables the cooperative solution of multi-domain constraint problems.

Meta-S [3] is a solver cooperation system embodying this approach. It allows the integration of arbitrary black box constraint solvers and provides the user with a flexible strategy definition framework. In [5,6] we theoretically discussed the idea of considering declarative programs (logic and functional logic ones) together with the associated language evaluation mechanism as constraint solvers and the integration of these solvers into our system. This yields multi-domain constraint programming languages and allows the definition of new evaluation strategies for them.

The present paper elaborates on this approach in practice. It is structured as follows: Section 2 briefly sketches our approach on solver cooperation. The integration of a logic language into our framework is explained in Sect. 3. Using the strategy definition framework of Meta-S we define classical search strategies

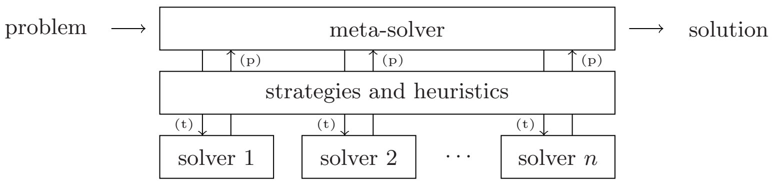


Fig. 1. Structure of Meta-S

as well as new ones in Sect. 4. Section 5 is dedicated to the evaluation and interpretation of experiments using these strategies. Finally, we draw a conclusion and discuss related work.

2 Constraint Solver Cooperation

Formally, a *constraint solver CS* consists of a solving algorithm associated to a constraint store. The solving algorithm is usually able to handle constraints of a particular constraint domain, e.g. linear constraints or finite domain constraints. The constraint store is a set of constraints which is initially empty, more precisely it contains the constraint true only. By constraint propagation the constraint solver adds constraints to its store. At this the set of possible solutions for the constraint conjunction of the store is successively narrowed, while ensuring its satisfiability. If the solver detects an inconsistency then the constraint for propagation is rejected.

Our solver cooperation system Meta-S (for a detailed description see [3]) enables the cooperative solving of mixed-domain constraint problems using several specialized solvers, none of which would be able to handle the problem on its own. Such solvers can be traditional solvers as mentioned above, but as well language evaluation mechanisms, like reduction or resolution as we will see in the following.

Fig. 1 shows the structure of Meta-S. A coordination mechanism – the *meta-solver* – treats the different applied *solvers as black boxes.* It receives the problem to solve in form of mixed-domain constraints, analyzes them and splits them up into single-domain ones processable by the individual solvers. These problem constraints are held in a *global pool,* which is managed by the meta-solver. Constraints are taken from this pool and they are propagated to the individual solvers. These collect the constraints in their *constraint stores* ensuring at this their satisfiability. In return the solvers are requested to provide newly gained information (i.e. constraints) back to the meta-solver to be added to the pool and propagated to other solvers during the following computation. This communication is handled via the *solver interface* functions *tell* and *project.* These are based on typical solver functions, which for most preexisting solvers already exist or can be implemented by very little glue code, hence allowing a simple integration into our system. Their formal definitions are given e.g. in [6].

– The function *tell* (denoted by (t) in the figure) for *constraint propagation* allows an individual solver to add constraints taken from the global pool to its store narrowing the solution set of the store's constraints.
– The *projection* function *project* (denoted by (p)) provides constraints implied by a solvers store for information interchange between the solvers. They are added to the pool of the solver cooperation system to be propagated later to another solver.

The meta-solver repeats the overall *tell–projection* cycle until a failure occurs or the global pool is emptied. Problem solutions can then be retrieved by projection.

In [4] we give requirements which allow the integration of solvers into the framework and, at the same time, ensure a well-defined behaviour of the overall system. Furthermore, we discuss termination, confluency, soundness and completeness. The soundness and completeness results do not depend on the particular cooperation strategy of the solvers. This was an important precondition for the versatility of our solver cooperation system.

Meta-S provides a flexible *framework for strategy definitions* (including a strategy definition language) such that the user can formulate choice heuristics for constraints with respect to their structure and their domains, prescribe the order of propagation and projection, use constraint rewriting facilities and so forth.

We discuss the impact of the cooperation strategy in connection with the structure of the constraint problem at hand on the effort of the solving process in Sect.5.

3 Considering a Logic Language as Constraint Solver

In [5] we presented a generalized method for the integration of arbitrary declarative languages and multi-domain constraints, we discussed this approach theoretically in detail in [6]. The idea is to consider declarative programs together with the associated language evaluation mechanism as constraint solver and to integrate this solver into our solver cooperation system.

It is widely accepted that logic programming can be interpreted as constraint programming over the Herbrand universe. Thus, by considering a *logic language as constraint solver* $CS_{\mathcal{L}}$, we can identify the goals according to a given logic program as the constraints handled by our new solver. Resolution as the language evaluation mechanism provides the constraint solving ability.

Besides this, the overall framework requires an equality relation for every solver such that the set of constraints of our logic language solver must include furthermore equality constraints between terms. For an actual integration of a language with constraints it is of course necessary to extend the language itself by constraints of other domains. In case of a logic language this yields the typical CLP syntax (cf. e.g. [7]).

Furthermore one needs to extend the existing language evaluation mechanism with respect to this syntactical extension, i.e. treating the constraints of other domains. This approach finally yields a CLP like language, where the constraints of other domains are collected by $CS_{\mathcal{L}}$ to be treated later by corresponding solvers within the Meta-S framework.

Let us consider the logic language solver $CS_{\mathcal{L}}$ in detail. For its integration into Meta-S we just need to define its *interface functions tell* and *project*.

Since it simplifies our presentation considerably, we write substitutions as special equations, e.g. $\sigma = \{x_1 = t_1, \ldots, x_m = t_m\}$ with variables x_i and terms t_i. Furthermore we make use of the usual notions, like the application $\sigma(e)$ of a substitution σ to a term e or the parallel composition $(\sigma \uparrow \phi) = mgu(\phi, \sigma)$ of idempotent substitutions, where mgu denotes the most general unifier (cf. [6]).

tell. Resolution steps on goals correspond to constraint propagations. The thereby computed substitutions are collected in the constraint store $C_{\mathcal{L}}$ of $CS_{\mathcal{L}}$. Consider Fig.2. Our presentation of substitutions allows to consider the constraint store $C_{\mathcal{L}}$ of the solver $CS_{\mathcal{L}}$ directly as a substitution, which is written in the form of equations and thus can be treated like constraints.

Case 1a represents a successful resolution step. The result is a tuple of three values: First, the value *true* indicates success. The second value is a conjunction of constraints representing the constraint store after propagation (which remains unchanged). The third value is a disjunction of the instantiated bodies of the matching rules together with the computed substitutions σ_p which are returned back to the constraint pool for following propagations. If there is no applicable rule for a resolution step then the goal is rejected, indicating a contradiction by the first value $false$ (Case 1b).

Substitutions computed by resolution steps are given back to the pool (as equality constraints). A substitution $\phi = \{Y = t\}$ is propagated to $C_{\mathcal{L}}$ by parallel composition $(\phi \uparrow C_{\mathcal{L}})$ which may be successful (Case 2a) or failing (Case 2b).

This way the system performs a sequence of resolution steps and forwards the thereby computed substitutions via the pool to the store $C_{\mathcal{L}}$ to be collected.

project. Projection is used for information interchange between the solvers which are integrated into the cooperation in Meta-S, here between $CS_{\mathcal{L}}$ and CS_{ν}, $\nu \in L$, where L denotes the solver indices.

The store of $CS_{\mathcal{L}}$ contains substitutions (received by resolution steps). Projection wrt. the domain of another solver CS_{ν}, provides an implication of the store $C_{\mathcal{L}}$, and thus simply generates equality constraints for the variables common to both solvers (cf. Fig.2).

Example 1. Consider the cooperation of our logic language solver $CS_{\mathcal{L}}$ within a framework of additional solvers, e.g. including an arithmetic solver $CS_{\mathcal{A}}$. A program describing electrical circuit problems may contain a rule for the sequential composition of resistors ($=_{\mathcal{A}}$ is the equality provided by $CS_{\mathcal{A}}$):

`r(seq(R1,R2),R) :- X + Y` $=_{\mathcal{A}}$ `R, r(R1,X), r(R2,Y).`

Let the pool hold the constraint $c =$ `r(seq(simple(A),simple(B)),600)` asking for a sequential composition of 600Ω of two simple resistors `A` and `B`.

tell: Let P be a constraint logic program, let $C_{\mathcal{L}} = \phi$ be the current store of $CS_{\mathcal{L}}$.

1. Let $R = p(t_1, \ldots, t_m)$ be the constraint (goal) which is to be propagated. Let $\hat{R} = \phi(R)$. We use the following notion: A rule $p = (Q_p \texttt{ :- } rhs_p)$ *applies to* $\hat{R}$, if there is a unifier $\sigma_p = mgu(\hat{R}, Q_p)$.
   (a) If the set $P_R \subseteq P$ of applicable rules is nonempty, then
   $tell(R, C_{\mathcal{L}}) = (true,\ C_{\mathcal{L}},\ \bigvee_{p \in P_R}(\sigma_p \wedge \sigma_p(rhs_p)))$.
   (b) If there is no applicable rule in P, then
   $tell(R, C_{\mathcal{L}}) = (false, C_{\mathcal{L}}, false)$.
2. Let $c = (Y = t)$ be the constraint (i.e. an equality constraint resp. substitution) which is to be propagated.
   (a) If $(\{Y = t\} \uparrow C_{\mathcal{L}}) \neq \emptyset$, then $tell(c, C_{\mathcal{L}}) = (true,\ \{Y = t\} \uparrow C_{\mathcal{L}}\ , true)$.
   (b) If $(\{Y = t\} \uparrow C_{\mathcal{L}}) = \emptyset$, then $tell(c, C_{\mathcal{L}}) = (false, C_{\mathcal{L}}, false)$.

---

project$_\nu$: The projection of a store $C_{\mathcal{L}} = \phi$ wrt. another solvers domain ν, $\nu \in L$, and a set of variables $X \subseteq X_{\mathcal{L}} \cap X_\nu$ makes the substitutions for $\texttt{x} \in X$ explicit:

$$project_\nu(X, \phi) = \begin{cases} \phi|_X & \text{if } \phi \neq \emptyset \\ true & \text{otherwise.} \end{cases}$$

Fig. 2. Interface functions *tell* and *project* of $CS_{\mathcal{L}}$

Let the store $C_{\mathcal{L}}$ of $CS_{\mathcal{L}}$ contain the equality constraint resp. substitution $\sigma = \{\texttt{B = 200}\}$ (computed during precedent resolution steps).

The *propagation of c wrt. the store* $C_{\mathcal{L}}$ by *tell* is done by performing a resolution step on $\sigma(c)$ (with most general unifier ϕ):

$tell(c, \sigma = C_{\mathcal{L}}) = (true, C_{\mathcal{L}}, \phi \wedge c')$ with
$\sigma(c) \rightsquigarrow_\phi c' = (\texttt{X + Y} =_{\mathcal{A}} \texttt{600} \wedge \texttt{r(simple(A),X)} \wedge \texttt{r(simple(200),Y)})$.

The constraint store $C_{\mathcal{L}}$ does not change. The constraint c is deleted from the pool and replaced by c' and ϕ derived from the resolution. Thus, in the next step an equality from ϕ or as well a constraint from c' could be chosen for propagation, e.g. $\texttt{X + Y} =_{\mathcal{A}} \texttt{600}$ for the arithmetic solver $CS_{\mathcal{A}}$ or `r(simple(200),Y)` (representing a logic goal) for a further resolution step. This choice depends on the cooperation strategy (cf. Sect.4).

Projection of the logic language store wrt. a set of variables means providing the collected substitution as equality constraints to other solvers. E.g. we project $C_{\mathcal{L}}$ wrt. `B` and the arithmetic solver: $project_{\mathcal{A}}(\{\texttt{B}\}, C_{\mathcal{L}}) = (\texttt{B} =_{\mathcal{A}} \texttt{200})$.

According to the described method, we plugged a new logic language solver, called CLL, into Meta-S. It quickly turned out that this method does not only allow the *integration of logic languages*[1] *with multi-domain constraint solving* but besides this the generation and experimentation with very different language evaluation strategies in an easy way. We consider this in the rest of the paper.

[1] and, in general, arbitrary declarative languages.

4 Solver Cooperation Strategies

We integrated a logic language solver CLL, an arithmetic solver and a finite domain constraint solver into our solver cooperation system Meta-S and analysed evaluation strategies for several constraint logic programs with different characteristics. In this section we discuss the strategies before presenting benchmark results for the program examples in Section 5.

We considered variations of the classical evaluation strategies *depth/breadth first search* for logic languages and two more advanced strategies, i.e. *lazy cloning* and *heuristic search.* Additionally, we examined *residuating* rules. Other search strategies like branch-and-bound, best-first, etc. can be easily integrated into our framework. However, we chose the above strategies to compare the typical Prolog evaluation with more sophisticated strategies, which already have been proven to be advantageous in the context of constraint solver cooperation without language integration [3]. This comparison is instructive, because the differences between language evaluation and constraint solving influence the choice of an appropriate strategy for the problem at hand.

The solving process consists of a sequence of projection and propagation phases. In each of these phases, disjunctions may be returned by the constraint solvers, which evoke alternative branches in the search tree and, thus, make a cloning of the current system status (i.e. the pool and all stores) necessary. To cushion the exponential behaviour, such cloning steps should be avoided as much as possible. Therefore, our strategies perform *weak projections* which are only allowed to produce constraint conjunctions until a fixed point is reached. Then they switch to *strong projection* and allow the creation of disjunctions. E.g. for a store which allows only certain discrete values for a variable X, strong projection may express this by enumerating all alternatives, like $X = 1 \vee X = 3 \vee X = 7$, whereas weak projection results in a less precise conjunction $1 \leq X \wedge X \leq 7$.

Depth/Breadth First Search (dfs and bfs). These strategies are variants of the classical ones: A propagation phase performing depth/breadth first search until the pool is emptied alternates with a projection phase, where all solvers are projected and the results are added into the constraint pool.

For an illustration of the behaviour of these strategies see Fig. 3a. To depict a configuration, we use tables showing the meta solvers pool in the first row, and the individual solvers' stores in the second row, separated by vertical bars. Assuming a pool containing the conjunction $A \wedge B$ and a store containing the constraint S (further stores are not considered in this example), the next constraint propagated is A. Suppose that only a part of this constraint can be handled by the solver, i.e. A_3, and the disjunction $A_1 \vee A_2$ is put back into the pool. To ensure correctness, A must be equivalent to $(A_1 \vee A_2) \wedge A_3$. In the next step the configuration is cloned. To find all solutions of the constraint problem both resulting configurations must be processed further. This can be done by first processing the left one and after that the right one, which results in depth first search, or by interleaving steps with both configurations, which results in breadth first search.

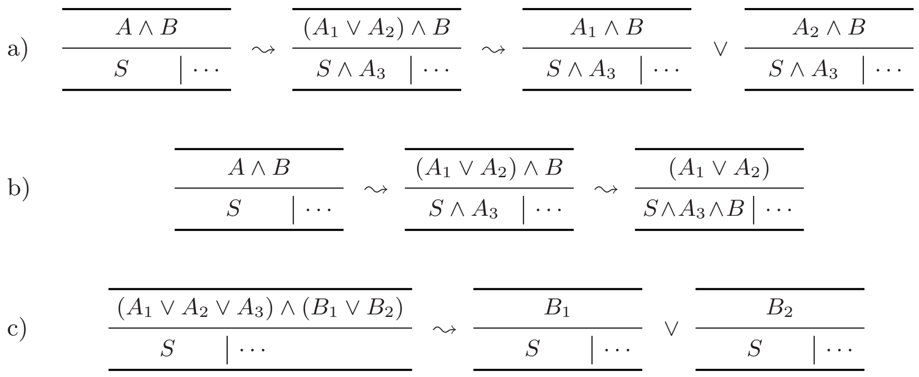

a)
$A \wedge B$ / S | ⋯ ⇝ $(A_1 \vee A_2) \wedge B$ / $S \wedge A_3$ | ⋯ ⇝ $A_1 \wedge B$ / $S \wedge A_3$ | ⋯ ∨ $A_2 \wedge B$ / $S \wedge A_3$ | ⋯
b)
$A \wedge B$ / S | ⋯ ⇝ $(A_1 \vee A_2) \wedge B$ / $S \wedge A_3$ | ⋯ ⇝ $(A_1 \vee A_2)$ / $S \wedge A_3 \wedge B$ | ⋯
c)
$(A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2)$ / S | ⋯ ⇝ B_1 / S | ⋯ ∨ B_2 / S | ⋯

Fig. 3. Strategies: a) Depth/breadth first search b) Lazy cloning c) Heuristic search

Thereby, it is advantageous wrt. efficiency to abort the projection phase as soon as a first constraint disjunction is returned by any individual solver, and then to reenter the propagation phase. This delays (often costly) projections of stores which could be further restricted by propagating a certain part of the newly gained disjunction before.

Since depth first search and breadth first search traverse the same search tree, the measurements of computation steps in Sect.5 correlate; a small management overhead of breadth first search becomes obvious in the runtime measurements.

Lazy Cloning (lazy). The lazy cloning strategy described in [3] tries to increase performance by delaying cloning of the system state as much as possible. This approach is related to the Andorra principle as described in [16,2]. Whenever a disjunction appears, the current configuration is not cloned at once, instead the new disjunction is pushed into a queue and all other pending constraints are propagated first. When the queue of pending constraints is empty, a disjunction is dequeued and the configuration gets cloned.

The idea behind this strategy is to propagate common constraints only once. Otherwise, i.e. if configurations were cloned as soon as disjunctions appear, each clone would need to propagate the pending constraints.

Fig.3b shows an example using the lazy cloning strategy. The first step is the same as for depth first and breadth first search (cf. Fig.3a), but the next step differs. The cloning is delayed, and the constraint B is propagated first. In this example, the entire constraint B is assumed to be put into the store, and no disjunction is put back into the pool. Obviously, B is propagated only once, while it had to be propagated twice in Fig.3a.

Like the classical strategies, lazy cloning avoids projection as long as possible, which means propagating conjunctions first, then breaking disjunctions and switching to projection only when the constraint pool is completely empty. However, this avoidance of projection has a tradeoff: It implies a higher number of clones again, since a projection might disclose inconsistencies between two solvers *before* a queued disjunction is split into several clones.

As already discussed in [3] lazy cloning interferes with the weak/strong-projection schema which avoids disjunctions as long as possible anyway. Therefore situations where lazy is beneficial could be rare. However, it turns out, that lazy cloning is interesting in our language integration nevertheless, because the CLL solver (in contrast to most other solvers) now returns disjunctions not during projection but as a result of propagation, which is not subject to the strong and weak differentiation.

Heuristic Search (heuristic). The heuristic search strategy is related to the fail first principle, proposed in [1], which tries to minimize the expanded search tree size and proved to be beneficial in solving constraint satisfaction problems. There, the idea is to choose the variable with the smallest domain size for value assignment. This choice is the one most likely to fail and hence will cut away failing branches very fast. Furthermore, variables with large domains may get restricted further such that unnecessary search can be avoided.

The heuristic strategy uses this selection heuristic for choosing between several queued disjunctions while performing lazy cloning. We dequeue a disjunction with the least number of constraint conjunctions. As a further optimization we throw away all other disjunctions. To ensure that no knowledge is lost, all solvers are marked in order to project them later again. For illustration see Fig. 3c. Here, the disjunction $(A_1 \vee A_2 \vee A_3)$ is removed, but can be reproduced later since it originates from a projection. Of course, this schema can be used only when propagation does not return disjunctions because they cannot be reproduced in this way. Thus, in general the heuristic strategy cannot be used in cooperations with the CLL solver, but in ordinary solver cooperations (cf. [3]).

Residuation. Residuation is a method to cope with indeterminism caused by insufficiently bound variables, used in concurrent constraint languages. The concept of residuation, as defined for logic languages in [15], divides all rules into two groups. *Residuating rules* may be applied only when their application is deterministic, i.e. only one rule matches the goal that is subject to resolution. *Generating rules* do not need to be deterministic, but may only be used when no residuating rule is applicable. CLL supports both residuating and generating rules.

5 Evaluation

We present benchmark results of five illustrating examples (cf. Table 1) to investigate the effects of different strategies for multi-domain constraint logic problems with varying extent of logic predicates and multi-domain constraints. In all these examples we search for all solutions, instead of for example only the first solution.

The table captures the numbers of clones generated during the computation, of propagations and of projections. The propagations cover resolution steps and the propagation of equality constraints resulting from substitutions computed during resolution, both for CLL, as well as constraint propagation steps for other

Table 1. Experimental results

example	strategy	clones	prop.	resol.'s	w.proj.	s.proj.	time in s
`dag`	bfs/dfs	6143	24575	14335	24576	24576	21.74/19.02
	lazy	6143	22528	10241	24576	24576	18.05
`cgc`	bfs/dfs	9540	53426	12402	3027	3027	16.59/11.12
	lazy	9540	19082	12404	3027	3027	8.43
`hamming`	lazy	148	2141	895	266	266	0.85
`smm`	lazy	135	6550	19	456	148	1.38
	bfs/dfs	3	3333	19	902	105	1.34/1.32
`nl-csp`	lazy	35352	109852	0	285	192	55.33
	bfs/dfs	493	118751	0	34758	2614	49.38/47.82
	heuristic	345	109121	0	31485	4944	45.76

incorporated solvers. For a better discussion, the number of resolution steps is, however, pointed out in an extra column again. Projections are divided into weak and strong projections. Run time measurements are given merely as further additional information. Our solvers are proof-of-concept implementations and, their performance is, of course, not comparable to advanced solvers of modern systems like ECL^{i}PS^e PROLOG. Nevertheless, transferring our strategies to such systems might improve their efficiency even further.

A Directed Acyclic Graph (`dag`). Our first example concentrates on logic language evaluation. It does not contain additional constraints of other domains. The program describes a directed acyclic graph (*dag*) which consists of 13 nodes, labelled from a to m. There is an edge between two nodes if and only if the label of the first node appears before the label of the second node in the alphabet. Thus, we have twelve edges starting from node a, eleven edges starting from b and so on. The 78 edges are described by 78 logic CLL facts. Furthermore, the program contains the rules shown in Fig. 4 for traversal and path finding. Given this program, we search for all paths from node a to node m.

For better understanding of the benchmark results, let us examine the evaluation of a goal `path(N, M, X)` in detail. Whenever such a goal is propagated, a disjunction with two alternatives is created, one alternative for each rule definition. The evaluation of the first rule definition is the same for every search strategy, and yields one further resolution step for the goal `edge(N, M)`, either with result success if this edge is in the graph, or a failure otherwise.

```
path(N1 N2 [N1, N2]) :-
   edge(N1, N2).

path(N1, N2, [N1 | REST]) :-
   edge(N1, X),
   path(X, N2, REST).
```

Fig. 4. Rule definition for paths in directed graphs

The evaluation of the second rule is more interesting: We gain a conjunction of two goals, `edge(N, X)` and `path(X, M, REST)`. The propagation of the goal `edge(N,X)` results in a disjunction with as many elements as there are edges starting from `N`. At this point the strategy lazy differs from depth first and breadth first search. While the strategies dfs and bfs break the disjunction at once, creating clones of the meta solver configuration and propagating the `path` goal after that once for every clone, lazy cloning propagates the `path` goal *first*, and clones the configuration afterwards. Thus the `lazy` strategy saves a number of resolution steps and propagations (as obvious from Table 1). At this, the number of propagations is the sum of the resolution steps and the propagation of equality constraints resulting from substitutions computed while unifying goals with rule heads.

Since all strategies create (sooner or later) all the clones for the possible edges, the number of clones does not differ between the strategies.

Even if only one solver (the CLL solver) is used in this example, a projection of the solver on each variable is necessary before successful termination, since solvers in the Meta-S framework are allowed to delay costly operations until the projection phase, and hence unsatisfiability of a problem may be noticed during projection (instead of during propagation as usually).

The wanted effect of the lazy strategy first to propagate pending constraints of a conjunction before cloning stores (due to disjunctions produced by propagations), yields an explicitly smaller number of resolution steps (and thus propagations) here, and explains why lazy search is the fastest strategy in this scenario.

Path Costs in a Complete Graph (`cgc`). In example `cgc` a complete graph (i.e. a graph with edges between every pair of nodes, including the reflexive cases) with nodes labelled a to j is traversed.

Again, the graph is described by logic facts, but this time each edge is annotated with random costs between 1 and 100. We are looking for all paths from a to b with the additional constraint that for the accumulated costs C of the whole path $50 \leq C \leq 80$ holds. With our particular cost annotations, 93 solutions exist.

The structure of this problem is similar to our last example, as well as the results. The only remarkable difference is the number of resolution steps, being nearly the same now for all strategies, although the number of overall propagations *is* lower for the lazy strategy.

On the one hand, lazy search saves some resolution steps because goals are propagated before cloning a configuration. Concurrently, a corresponding number of goals is propagated in the leaf nodes of the search tree which propagation is avoided by breadth first search and depth first search: In depth/breadth first search, the propagation of the arithmetic constraints in Lines 3-5, Fig. 5, encounters a contradiction, thus, the resolution of the last recursive goal (in Line 6) is avoided. In lazy search, the arithmetic constraints cannot fail at once, because the disjunction of the edges in Line 2 has not been split, and thus the concrete costs of the edge currently considered are not known yet. So the goal in Line 6 is propagated, after that the disjunction is split, and afterwards the arithmetic constraints fail.

This "stepping over a disjunction", what looks like a handicap for the lazy strategies at first, can turn into an advantage, too. Consider a modification of the rule from Fig. 5 such that the recursive goal in Line 6 is placed *before* the arithmetic constraints. In this case, depth first and breadth first search will never terminate, but the lazy strategy "steps over" the disjunction of both logic goals and propagates the arithmetic constraints first. Hence, only lazy search terminates in this case (and has exactly the same benchmark results as cgc in Table 1).

In the original formulation of cgc, although the number of resolution steps does not evoke any difference for the performance of the solving process, the number of propagations is significant smaller for lazy search due to the arithmetic constraints being propagated before the cloning operation. So the lazy cloning strategy is the fastest, again.

The Hamming Numbers (hamming). This example was chosen to show that Meta-S allows the combination of *residuating* and *nonresiduating* rules. A residuating CLL rule computes (in cooperation with an arithmetic solver) multiples of 2, 3 and 5; its results are requested by nonresiduating CLL rules and merged into a common list of hamming numbers. Table 1 shows the results of the computation of the first 20 hamming numbers.

In the table we only give the results for the lazy strategy. The other strategies behave similar because of the highly deterministic nature of the rule choice here.

Send-More-Money (smm). While it is well known that this problem can be solved by a usual finite domain solver providing arithmetics, we nevertheless found it appropriate as a short and simple example to illustrate general strategy effects for a solver cooperation of different solvers and domains.

The problem is constructed by a CLL predicate describing the composition of letters to words and constructing the respective equality constraint, e.g. S*1000+E*100+N*10+D=SEND. This is done in a few deterministic evaluation steps, where only one rule definition at a time matches the goals to solve. Thus, the influence of language evaluation is surpassed by constraint solving. Besides the CLL solver two other solvers cooperate in this example. A linear arithmetic solver is supported by a finite domain solver. These two solvers rely on information exchange by projection, and hence the overall effort is greatly affected by the timing when switching from propagation to projection.

```
1  path(N1, N2, [N1 | REST], COST) :-
2     edge(N1, X, EDGE_COST),
3     EDGE_COST > 0,
4     REST_COST > 0,
5     COST = EDGE_COST + REST_COST,
6     path(X, N2, REST, REST_COST).
```

Fig. 5. Rule definition for paths with edge costs in directed graphs

The tradeoff of projection avoidance for the lazy strategy which results in a higher cloning rate (as discussed in Sect.4) becomes obvious and hindering here. This shows, that mainly problems with an larger portion (and allowing alternative steps) of language evaluation than `smm` profit from lazy cloning.

It is well known that the variable order for projection makes an impact on the effort needed to find solutions for this kind of problems. The influences of the portrayed search strategies, however, have been proved to be representative.

A Nonlinear Constraint Satisfaction Problem (`nl-csp`). Another crypto-arithmetic puzzle: We want to replace different letters by different digits, s.t. `ABC * DEF = GHIJ` holds. This problem is given without any CLL rules.

The effect of an increased cloning rate for the lazy strategy gets enormous here. However, since the CLL solver is not used in this example, we may use the heuristic strategy, and the reordering of disjunctions is decisive in this case. This result corresponds to our observations in [3], where the heuristic strategy proved to be very efficient in many cases.

Overall Evaluation. Our examples show that in problems with a large extent of logic language evaluation, the strategy lazy gives the best results. This strategy reorders goals in favour of deterministic ones and thus prevents a lot of propagations. An evaluation using the lazy strategy may even terminate in circumstances where depth/breadth first search produce infinite resolution derivations. In problems predominated by constraint solving most disjunctions are already avoided by the weak/strong projection schema. Hence the overhead of the lazy strategy outweighs its advantages here and depth/breath first search are more appropriate. Problems that can be easily described without CLL constraints may benefit from the heuristic strategy, which gives almost optimal results in many situations.

6 Conclusion and Related Work

We presented the integration of the logic language CLL into the solver cooperation framework Meta-S and the usage of its strategy definition framework to define very different evaluation strategies for the CLP language gained by this approach. We discussed implications and recommendations concerning the appropriateness of different cooperation strategies for the evaluation of multi-domain constraint logic programs of different form.

The development and application of solver cooperation systems has become an interesting research topic during the last years.

Besides systems with fixed solvers and fixed cooperation strategies, e.g. [13], there are systems which allow a more flexible strategy handling (e.g. BALI [10] or CFLP [8]) up to the definition of problem specific solver cooperation strategies and the integration of new solvers, like the approach in [11] or our system Meta-S.

Flexible solver cooperation systems enable us to take advantage of existing solvers (implementing solvers is a time consuming and tedious work) and to design appropriate solving mechanisms customized for particular application problems in a simple way.

Meta-S differs from other systems wrt. its choices of languages and language constructs for the strategy definition. Meta-S provides a set of typical predefined constructs but finally offers the user all normal Common Lisp constructs as well as abstraction features like macros. Meta-S already provides a set of predefined typical strategies.

Furthermore, our system explicitly distinguishes between propagation and projection in contrast to other approaches which do not separate these steps. However, this separation turned out to be useful and certainly important for defining efficient strategies, as can be seen in the `nl-csp` problem, where the avoidance of projections during depth first or breadth first search results in an approximately 15% faster solving process.

In [12] Prosser considers search in constraint satisfaction algorithms, e.g. backmarking, backjumping, forward checking and hybrids of them. These strategies cannot be applied directly in our architecture since it combines black box solvers which may work on miscellaneous domains. However, the strategy which we named dfs here, is in fact closely related to forward checking which results from interlocking depth first search and constraint propagation/satisfiability checking. Differences issue from solver cooperation features like the explicit separation of the constraint treatment into a propagation and a projection phase. Our lazy strategy delays disjunction splitting and, thus, allows to consider (and to cut) a number of subtrees of the original search tree at once. Even though very far, it is related to the backmarking strategy of [12] which is based on a completely different concept.

Another concept for solver cooperation are computation spaces, described in detail in [14]. Computation spaces are the underlying mechanism of the constraint solving features of the multi paradigm programming language Oz. Here, a constraint store contains only *basic* constraints that are simple enough to be understood by all participating solvers of one domain. These basic constraints are manipulated by propagators that are responsible for the satisfiability of more complex constraints. However, this system relies on all solvers sharing the same store format, and hence is not satisfying for the main goal of Meta-S, i.e. cooperation of black box solvers independent of their implementation. On the other hand, the sharing of a common store format allows a tighter communication and in many cases a faster solving process.

Our approach of considering programming languages themselves as constraint solvers and integrating them into a system of cooperating solvers [5,6] is new. Meta-S allows this integration, further distinguishing our system from other solver cooperation approaches which usually provide a fixed host language (often a logic one). Using our cooperation approach, we achieve a covering of the well known scheme CLP(X) and as well CFLP(X) [9] for constraint functional logic programming.

References

1. J. Bitner and E. M. Reingold. Backtrack Programming Techniques. *Communications of the ACM (CACM)*, 18:651–655, 1975.
2. V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 26 (7) of *SIGPLAN Notices*, pages 83–93. ACM Press, 1991.
3. St. Frank, P. Hofstedt, and P.R. Mai. Meta-S: A Strategy-oriented Meta-Solver Framework. In *Proceedings of the 16th International Florida Artificial Intelligence Research Symposium Conference (FLAIRS)*. The AAAI Press, May 2003.
4. P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Dresden University of Technology, 2001.
5. P. Hofstedt. A general Approach for Building Constraint Languages. In *Advances in Artificial Intelligence*, volume 2557 of *LNCS*, pages 431–442. Springer, 2002.
6. P. Hofstedt and P. Pepper. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming (TPLP). Special Issue on Multiparadigm Languages and Constraint Programming*, 2006. to appear.
7. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19&20:503–581, 1994.
8. N. Kobayashi, M. Marin, and T. Ida. Collaborative Constraint Functional Logic Programming System in an Open Environment. *IEICE Transactions on Information and Systems*, E86-D(1):63–70, 2003.
9. F.-J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming – ALP'92*, volume 632 of *LNCS*. Springer, 1992.
10. E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy. INRIA, 1996.
11. B. Pajot and E. Monfroy. Separating search and strategy in solver cooperations. In *Perspectives of Systems Informatics – PSI 2003*, volume 2890 of *LNCS*. Springer, 2003.
12. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.
13. M. Rueher. An Architecture for Cooperating Constraint Solvers on Reals. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer, 1995.
14. Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer, 2002.
15. Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Technical Report RR-91-13, DFKI, 1991.
16. D.H.D. Warren. The Andorra Principle. Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden, 1988.

Information-Flow Attacks Based on Limited Observations[⋆]

Damas P. Gruska

Institute of Informatics, Comenius University,
Mlynska dolina, 842 48 Bratislava, Slovakia
`gruska@fmph.uniba.sk`

Abstract. Two formal models for description of timing attacks are presented, studied and compared with other security concepts. The models are based on a timed process algebra and on a concept of observations which make visible only a part of a system behaviour. An intruder tries to deduce some private system activities from this partial information which contains also timing of actions. To obtain realistic security characterizations some limitations on observational power of the intruder are applied. It is assumed that the intruder has only limited time window to perform the attack or time of action occurrences can be measured only with a given limited precision.

Keywords: process algebras, timing attacks, information flow.

1 Introduction

Several formulations of a system security can be found in the literature. Many of them are based on a concept of non-interference (see [12]) which assumes the absence of any information flow between private and public systems activities. More precisely, systems are considered to be secure if from observations of their public activities no information about private activities can be deduced. This approach has found many reformulations for different formalisms, computational models and nature or "quality" of observations. They try to capture some important aspects of systems behaviour with respect to possible attacks against systems security, often they are tailored to some types of specific attacks. An overview of language-based approaches to information flow based security can be found in [21].

Timing attacks have a particular position among attacks against systems security. They represent a powerful tool for "breaking" "unbreakable" systems, algorithms, protocols, etc. For example, by carefully measuring the amount of time required to perform private key operations, attackers may be able to find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems (see [19]). This idea was developed in [6] where a timing attack against smart card implementation of RSA was conducted. In [18], a timing attack on the RC5

[⋆] Work supported by the grant VEGA 1/3105/06 and APVV-20-P04805.

block encryption algorithm, in [22] the one against the popular SSH protocol and in [7] the one against web privacy are described.

In the literature several papers on formalizations of timing attacks can be found. Papers [9], [10], [11] express the timing attacks in a framework of (timed) process algebras. In all these papers system actions are divided into private and public ones and it is required that there is not an interference between them. More precisely, in [9,10] it is required that on a level of system traces which do not contain internal actions one cannot distinguish between system which cannot perform private actions and system which can perform them but all of them are reduced to internal actions. In paper [11] a concept of public channels is elaborated. In the above mentioned papers also a slightly different approach to system security is presented — the system behaviour must be invariant with respect to composition with an attacker which can perform only private actions ([9], [10]) or with an attacker which can see only public communications ([11]).

In the presented approach actions are not divided to private and public ones on a system description level. Instead of this we work with a concept of observations. These are mappings on the set of actions which can hide some of actions (for example, internal actions, communications via encrypted channels, actions hidden by a firewall etc) but not elapsing of time. Since many of timing attacks described in the literature are based on observations of "internal" actions we work also with this information what is not the case of the above mentioned papers. Moreover we will study two (realistic) restrictions of an observational power of an intruder. First we will assume that the intruder has only a limited time window for observation, i.e. system to be attacked can be observed only for some (finite) time interval (we will call this *limited access attacks*). In the second case we will assume that the intruder can measure time of action occurrences only with some given precision (*limited precision attacks*). In this way we can consider timing attacks which could not be taken into account otherwise. Moreover, the resulting security properties are more adequate for real applications for which standard non-information flow security property is too restrictive.

The paper is organized as follows. In Section 2 we describe the timed process algebra which will be used as a basic formalism. In Section 3 we present the notion of non-information flow property in the case of unlimited, limited access and limited precision (timing) attacks for both passive and active cases. The presented formalism is compared with other security concepts described in the literature and it is shown that it is more general and stronger in the sense that it can describe attacks which are not captured by the other concepts.

2 Timed Process Algebra

In this section we introduce the Timed Process Algebra, TPA for short. It is based on Milner's CCS (see [20]) but the special time action t which expresses elapsing of (discrete) time is added. The presented language is a slight simplification of the Timed Security Process Algebra introduced in [9]. We omit the explicit idling operator ι used in tSPA and instead of this we use an alternative approach known in the literature and we allow implicit idling of processes. Hence processes can perform either "enforced idling" by performing t actions which are explicitly expressed in their descriptions or "voluntary idling". But in the both situations internal communications have priority to actions t in the case of the parallel operator. Moreover we do not divide actions into private and public ones as it is in tSPA. TPA differs also from the tCryptoSPA (see [11]) besides absence of value passing, by semantics of choice operator $+$ which in some cases abandons *time determinacy* which is strictly preserved in TPA.

To define the language TPA, we first assume a set of atomic action symbols A not containing symbols τ and t, and such that for every $a \in A$ there exists $\overline{a} \in A$ and $\overline{\overline{a}} = a$. We define $Act = A \cup \{\tau\}, Actt = Act \cup \{t\}$. We assume that $a, b, \ldots$ range over A, $u, v, \ldots$ range over Act, and $x, y \ldots$ range over $Actt$. The set of TPA terms over the signature Σ is defined by the following BNF notation:

$$P \ ::= \ X \ | \ op(P_1, P_2, \ldots P_n) \ | \ \mu X P$$

where $X \in Var$, Var is a set of process variables, $P, P_1, \ldots P_n$ are TPA terms, $\mu X-$ is the binding construct, $op \in \Sigma$. Assume the signature $\Sigma = \bigcup_{n \in \{0,1,2\}} \Sigma_n$, where

$$\begin{aligned} \Sigma_0 &= \{Nil\} \\ \Sigma_1 &= \{x. \mid x \in A \cup \{t\}\} \cup \{[S] \mid S \text{ is a relabeling function}\} \\ &\quad \cup \{\backslash M \mid M \subseteq A\} \\ \Sigma_2 &= \{|, +\} \end{aligned}$$

with the agreement to write unary action operators in prefix form, the unary operators $[S], \backslash M$ in postfix form, and the rest of operators in infix form. Relabeling functions, $S : Actt \to Actt$ are such that $\overline{S(a)} = S(\bar{a})$ for $a \in A, S(\tau) = \tau$ and $S(t) = t$. The set of CCS terms consists of TPA terms without t action. We will use an usual definition of opened and closed terms where μX is the only binding operator. Closed terms are called processes. Note that Nil will be often omitted from processes descriptions and hence, for example, instead of $a.b.Nil$ we will write just $a.b$.

We give a structural operational semantics of terms by means of labeled transition systems. The set of terms represents a set of states, labels are actions from $Actt$. The transition relation $\to$ is a subset of TPA $\times$ $Actt$ $\times$ TPA. We write $P \xrightarrow{x} P'$ instead of $(P, x, P') \in \to$ and $P \stackrel{x}{\not\to}$ if there is no P' such that $P \xrightarrow{x} P'$. The meaning of the expression $P \xrightarrow{x} P'$ is that the term P can evolve to P' by performing action x, by $P \xrightarrow{x}$ we will denote that there exists a term P' such that $P \xrightarrow{x} P'$. We define the transition relation as the least relation satisfying the following inference rules:

$$\frac{}{x.P \xrightarrow{x} P} A1 \qquad \frac{}{u.P \xrightarrow{t} u.P} A2$$

$$\frac{}{Nil \xrightarrow{t} Nil} A3 \qquad \frac{P \xrightarrow{u} P'}{P \mid Q \xrightarrow{u} P' \mid Q} Pa1$$

$$\frac{P \xrightarrow{u} P'}{Q \mid P \xrightarrow{u} Q \mid P'} \quad Pa2 \qquad \frac{P \xrightarrow{a} P', Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad Pa3$$

$$\frac{P \xrightarrow{t} P', Q \xrightarrow{t} Q', P \mid Q \not\xrightarrow{\tau}}{P \mid Q \xrightarrow{t} P' \mid Q'} \quad Pa4 \qquad \frac{P \xrightarrow{u} P'}{P + Q \xrightarrow{u} P'} \quad S1$$

$$\frac{P \xrightarrow{u} P'}{Q + P \xrightarrow{u} P'} \quad S2 \qquad \frac{P \xrightarrow{t} P', Q \xrightarrow{t} Q'}{P + Q \xrightarrow{t} P' + Q'} \quad S3$$

$$\frac{P \xrightarrow{x} P'}{P \setminus M \xrightarrow{x} P' \setminus M}, (x, \overline{x} \notin M) \; Res \qquad \frac{P[\mu XP/X] \xrightarrow{x} P'}{\mu XP \xrightarrow{x} P'} \; Rec$$

$$\frac{P \xrightarrow{x} P'}{P[S] \xrightarrow{S(x)} P'[S]} \quad Rl$$

Here we mention rules that are new with respect to CCS. Axioms $A2, A3$ allow arbitrary idling. Concurrent processes can idle only if there is no possibility of an internal communication ($Pa4$). A run of time is deterministic ($S3$). In the definition of the labeled transition system we have used negative premises (see $Pa4$). In general this may lead to problems, for example with consistency of the defined system. We avoid these dangers by making derivations of τ independent of derivations of t. For an explanation and details see [13]. Regarding behavioral relations we will work with the timed version of weak trace equivalence. Note that here we will use also a concept of observations which contain complete information which includes also τ actions and not just actions from A and t action as it is in [9]. For $s = x_1.x_2.\ldots.x_n, x_i \in Actt$ we write $P \xrightarrow{s}$ instead of $P \xrightarrow{x_1}\xrightarrow{x_2} \ldots \xrightarrow{x_n}$ and we say that s is a trace of P. The set of all traces of P will be denoted by $Tr(P)$. We will write $P \overset{x}{\Rightarrow} P'$ iff $P(\xrightarrow{\tau})^* \xrightarrow{x} (\xrightarrow{\tau})^* P'$ and $P \overset{s}{\Rightarrow}$ instead of $P \overset{x_1}{\Rightarrow}\overset{x_2}{\Rightarrow} \ldots \overset{x_n}{\Rightarrow}$. By ϵ we will denote the empty sequence of actions, by $Succ(P)$ we will denote the set of all successors of P and $Sort(P) = \{x | P \xrightarrow{s.x}$ for some $s \in Actt^\star\}$. If the set $Succ(P)$ is finite we say that P is finite state.

Definition 1. *The set of timed traces of a process P is defined as $Tr_t(P) = \{s \in (A \cup \{t\})^\star | \exists P'.P \overset{s}{\Rightarrow} P'\}$. Two process P and Q are weakly timed trace equivalent ($P \approx_w Q$) iff $Tr_t(P) = Tr_t(Q)$.*

3 Information Flow

In this section we will formalize a notion of timing attacks based on an information flow between invisible (private) and visible (public) activities of a system. At the beginning we assume that an attacker is just an eavesdropper who can see a (public) part of the system behaviour and who tries to deduce from this information some private information. In the case of timing attacks time of occurrences of observed events plays a crucial role i.e. timing of actions represents a fundamental information. First we will not put any restrictions on intruder's capability. Later we will model two restricted or limited intruders.

To formalize the attacks we do not divide actions to public and private ones on the level of process description as it is done for example in [11,5] but instead of this we use more general concept of observations. This concept was recently exploited in [2], [16] and [3] in a framework of Petri Nets, process algebras and transition systems, respectively, where a concept of opacity is defined with the help of the observations.

Definition 2. *An observation $\mathcal{O}$ is a mapping $\mathcal{O} : Actt \rightarrow Actt \cup \{\epsilon\}$ such that $\mathcal{O}(t) = t$ and for every $u \in Act, \mathcal{O}(u) \in \{u, \tau, \epsilon\}$.*

An observation expresses what can an observer - eavesdropper see from a system behaviour. It cannot rename actions but only hide them completely ($\mathcal{O}(u) = \epsilon$) or indicate just a performance of some action but its name cannot be observed ($\mathcal{O}(u) = \tau$). Observations can be naturally generalized to sequences of actions. Let $s = x_1.x_2.\ldots.x_n, x_i \in Actt$ then $\mathcal{O}(s) = \mathcal{O}(x_1).\mathcal{O}(x_2).\ldots.\mathcal{O}(x_n)$. Since the observation expresses what an observer can see we will alternatively use both terms (observation - observer) with the same meaning.

In general, systems respect the property of privacy if there is no leaking of private information, namely there is no *information flow* from the private level to the public level. This means that the secret behavior cannot influence the observable one, or, equivalently, no information on the observable behavior permits to infer information on the secret one. Moreover, in the case of timing attacks, timing of actions plays a crucial role. In the presented setting private actions are those that are hidden by the observation $\mathcal{O}$, i.e. such actions a that $\mathcal{O}(a) \in \{\tau, \epsilon\}$ and for public actions we have $\mathcal{O}(a) = a$ i.e the observer can see them. Now we are ready to define Non-Information Flow property (NIF) for TPA processes, but first some notations are needed. An occurrence of action x (or of sequence s') in a sequence of actions s we will indicate by $x \in s$ ($s' \in s$) i.e. $x \in s$ ($s' \in s$) iff $s = s_1.x.s_2$ ($s = s_1.s'.s_2$) for some $s_1, s_2 \in Actt^\star$ and for $\mathcal{S} \subseteq Actt$ we indicate $\mathcal{S} \cap s \neq \emptyset$ iff $x \in s$ for some $x \in \mathcal{S}$ otherwise we write $\mathcal{S} \cap s = \emptyset$. By $s_{|M}$ we will denote a sequence obtained from s by removing all elements not belonging to the set M.

Clearly, NIF property has to be parameterized by observation $\mathcal{O}$ and by set of private actions $\mathcal{S}$ which occurrence is of interest. In other words, process P has NIF property if from the observation of its behaviour (given by $\mathcal{O}$) it cannot be deduced that some of given private actions ($\mathcal{S}$) were performed. We expect a consistency between $\mathcal{O}$ and $\mathcal{S}$ in the sense that the observation does not see actions from $\mathcal{S}$. The formal definition follows.

Definition 3. *Let $\mathcal{O}$ be an observation and $\mathcal{S} \subseteq A$ such that $\mathcal{O}(a) \in \{\tau, \epsilon\}$ for $a \in \mathcal{S}$. We say that process P has $NIF^{\mathcal{S}}_{\mathcal{O}}$ property (we will denote this by $P \in NIF^{\mathcal{S}}_{\mathcal{O}}$) iff whenever $\mathcal{S} \cap s_1 \neq \emptyset$ for some $s_1 \in Tr(P)$ then there exists $s_2 \in Tr(P)$ such that $\mathcal{S} \cap s_2 = \emptyset$ and $\mathcal{O}(s_1) = \mathcal{O}(s_2)$.*

Informally, process P has $NIF^{\mathcal{S}}_{\mathcal{O}}$ property if the observer given by $\mathcal{O}$ (note that (s)he can always see timing of actions) cannot deduce that process P has performed a sequence of actions which includes some private (secrete) actions from S. In other words, $P \in NIF_{\mathcal{O}}^{\mathcal{S}}$ means that observer $\mathcal{O}$ cannot deduce anything about execution of actions from $\mathcal{S}$ and hence P is robust against attacks which try to deduce that some private action from $\mathcal{S}$ was performed. By $NIF_{\mathcal{O}}^{\mathcal{S}}$ we will denote also the set of processes which have $NIF_{\mathcal{O}}^{\mathcal{S}}$ property.

Example 1. Let $P = ((b.t.\bar{c} + a.\bar{c})|c) \setminus \{c\}$ and $\mathcal{O}(a) = \mathcal{O}(b) = \epsilon, \mathcal{O}(\tau) = \tau$. The observer given by $\mathcal{O}$ can detect occurrence of the action a but not b i.e. $P \in NIF_{\mathcal{O}}^{\{b\}}$but $P \notin NIF_{\mathcal{O}}^{\{a\}}$ since from observing just τ action (without any delay) it is clear that action a was performed. □

Now we compare NIF property with another security concept known in the literature, with Strong Nondeterministic Non-Interference, SNNI, for short (see [9]). We recall its definition . Suppose that all actions are divided in two groups, namely public (low level) actions L and private (high level) actions H i.e. $A = L \cup H, L \cap H = \emptyset$. Then process P has SNNI property if $P \setminus H$ behaves like P for which all high level actions are hidden for an observer. To express this hiding we introduce hiding operator $P/M, M \subseteq A$, for which if $P \xrightarrow{a} P'$ then $P/M \xrightarrow{a} P'/M$ whenever $a \notin M \cup \bar{M}$ and $P/M \xrightarrow{\tau} P'/M$ whenever $a \in M \cup \bar{M}$. Formal definition of SNNI follows.

Definition 4. *Let $P \in TPA$. Then $P \in SNNI$ iff $P \setminus H \approx_w P/H$.*

Relationship between $NIF_{\mathcal{O}}^{\mathcal{S}}$ and $SNNI$ express the following theorem (see [16]). Note that it is clear from this theorem that $SNNI$ property is just a special case of $NIF_{\mathcal{O}}^{\mathcal{S}}$ property.

Theorem 1. *$P \in SNNI$ iff $P \in NIF_{\mathcal{O}}^{H}$ for $\mathcal{O}(h) = \tau$, $h \in H$ and $\mathcal{O}(x) = x$, $x \notin H$.*

3.1 Passive Attacks with Limited Access

Now we will assume that an intruder can observe system behaviour only for a limited amount of time, what is more realistic than unlimited access to system to be attacked. First some notation is needed. Let $s \in Actt$, by time length we will mean the number of t actions occurring in s, and we will denote it by $|s|_t$. Now we can define non-information flow under the condition that an intruder can observe system behaviour for time n (this property will be denoted by $NIF_{\mathcal{O}_n}^{\mathcal{S}}$).

Definition 5. *Let $\mathcal{O}$ be an observation and $\mathcal{S} \subseteq A$ such that $\mathcal{O}(a) \in \{\tau, \epsilon\}$ for $a \in \mathcal{S}$. We say that process P has $NIF_{\mathcal{O}_n}^{\mathcal{S}}$ property (we will denote this by $P \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$) iff whenever $\mathcal{S} \cap s_1' \neq \emptyset$ for some $s_1' \in s_1, s_1 \in Tr(P)$ then there exists $s_2' \in s_2, s_2 \in Tr(P)$ such that $\mathcal{S} \cap s_2' = \emptyset$ and $|s_1'|_t = |s_2'|_t = n$ and it holds $\mathcal{O}(s_1') = \mathcal{O}(s_2')$.*

Example 2. Let $P = l_1.t.t.t.h.l_2 + l_1.t.t.l_2$ and $\mathcal{O}(h) = \epsilon, \mathcal{O}(l_i) = l_i$. It is easy to check that $P \in NIF_{\mathcal{O}_2}^{\{h\}}$ but $P \notin NIF_{\mathcal{O}}^{\{h\}}$. In other words, the “unlimited” observer given by $\mathcal{O}$ can detect occurrence of the action h but it cannot be performed if only time window of length 2 is at disposal. If a time window of length 3 (and more) is at disposal, then $P \notin NIF_{\mathcal{O}_3}^{\{h\}}$. □

The relationship between $NIF_{\mathcal{O}}^{\mathcal{S}}$ and $NIF_{\mathcal{O}_n}^{\mathcal{S}}$ states the following theorem.

Theorem 2. *$NIF_{\mathcal{O}}^{\mathcal{S}} \subset NIF_{\mathcal{O}_n}^{\mathcal{S}}$ for every n and $NIF_{\mathcal{O}_m}^{\mathcal{S}} \subset NIF_{\mathcal{O}_n}^{\mathcal{S}}$ for $m > n$.*

Proof. The main idea. Clearly $NIF_{\mathcal{O}_m}^{\mathcal{S}} \subseteq NIF_{\mathcal{O}_n}^{\mathcal{S}}$ for $m > n$. The rest follows from Example 2 and its generalization. □

In many cases it seems to be sufficient to check occurrence of only one private action instead of a bigger set, i.e. the cases $\mathcal{S} = \{a\}$ for some $a \in A$. In these cases an observer tries to deduce whether confident action a was or was not performed. But even in this simplest possible case the NIF properties are undecidable, but in general they are decidable for finite state processes.

Theorem 3. *$NIF_{\mathcal{O}_n}^{\{a\}}$ property is undecidable but $NIF_{\mathcal{O}_n}^{\mathcal{S}}$ is decidable for finite state processes if $\mathcal{O}(x) \neq \epsilon$ for every $x \in Actt$ and $n \geq 1$.*

Proof. The main idea. Let T_i is i-th Turing machine (according to some ordering). Let machine T accept a sequence a^i and also τ^i but this only in the case that T_i halts with the empty tape as an input. Let $\mathcal{O}(a) = \tau$. The rest of the proof follows from undecidability of the halting problem. Note that CCS process and so TPA process as well have power of Turing machines.

Regarding the second part of the theorem, we construct from a finite labeled transition system which corresponds to P a finite state automaton A with all sates treated as final. From this automaton we construct a new automaton A' in such a way that transitions labeled by actions which are seen as τ action are labeled by τ and again all sates treated as final. The rest of the proof follows from decidability properties for finite automata. □

Even if $NIF_{\mathcal{O}_n}^{\mathcal{S}}$ is decidable the corresponding algorithms are of exponential complexity. On way how to overcome this disadvantage is a bottom-up design of processes. Hence compositionality of $NIF_{\mathcal{O}_n}^{\mathcal{S}}$ plays an important role. We have the following property.

Theorem 4. (Compositionality) *Let $P, Q \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$. Then*

$x.P \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$ *if* $x \notin S \cup \{t\}$
$P + Q \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$
$P|Q \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$
$P[f] \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$ *for any* f *such that* $f(S) \subseteq S$
$P \setminus M \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$ *for any* $M, M \subseteq S$.

Proof. We will prove the first three cases which are the most interesting.

(1) Let $P \in NIF_{\mathcal{O}_n}^{\mathcal{S}}$ and $\mathcal{S} \cap s_1' \neq \emptyset$ for some $s_1' \in s_1, s_1 \in Tr(x.P)$. If $s_1 = x$ then since $x \notin \mathcal{S}$ the NIF condition holds. Hence let $s_1 = x.s_1''$, $s_1' \in s_1''$,

$s_1'' \in Tr(x.P)$. Since $P \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$ there exists $s_2' \in s_2, s_2 \in Tr(P)$ such that $\mathcal{S} \cap s_2' = \emptyset$ and $|s_1'|_t = |s_2'|_t = n$ and it holds $\mathcal{O}(s_1') = \mathcal{O}(s_2')$. Hence for $s_2, s_2 = x.s_2'$ we have $s_2 \in Tr(x.P)$ and hence $x.P \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$.

(2) Let $P, Q \in NIF^{\mathcal{S}}_{\mathcal{O}}$ and $\mathcal{S} \cap s_1' \neq \emptyset$ for some $s_1' \in s_1, s_1 \in Tr(P+Q)$. Without lost of generality we can assume that $s_1 \in Tr(P)$. Since $P \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$ there exists $s_2' \in s_2, s_2 \in Tr(P)$ such that $\mathcal{S} \cap s_2' = \emptyset$ and $|s_1'|_t = |s_2'|_t = n$ and it holds $\mathcal{O}(s_1') = \mathcal{O}(s_2')$. But since $s_2 \in Tr(P+Q)$ we have $P + Q \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$.

(3) Let $P, Q \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$ but $P|Q \notin NIF^{\mathcal{S}}_{\mathcal{O}_n}$. Let s_1 is the shortest trace of $P|Q$ such that $\mathcal{S} \cap s_1' \neq \emptyset$ for some $s_1' \in s_1$ and since $P|Q \notin NIF^{\mathcal{S}}_{\mathcal{O}_n}$ then for every trace $s_2' \in s_2, s_2 \in Tr(P|Q)$ such that $|s_1'|_t = |s_2'|_t = n$ and it holds $\mathcal{O}(s_1') = \mathcal{O}(s_2')$ it holds $\mathcal{S} \cap s_2' \neq \emptyset$. Since s_1 is the shortest trace clearly only its last element belong to $\mathcal{S}$. This element was performed either by P or by Q. By case analysis and structural induction we came to a contention with the assumption that $P, Q \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$. □

3.2 Passive Attacks with Limited Precision

Till know we have considered the situation when an intruder has only a limited access to a system to be attacked i.e. (s)he has only a limited time for which the system behaviour can be observed. Now we investigate a different situation. We assume that the intruder can observe the system behaviour only with limited time precision. Say, than the intruder has unprecise stop-watch at disposal when time of occurrence of actions is observed. This models situations when the system to be attacked is remote and interconnection network properties (mainly throughput) cannot be predicted. Now we define non-information flow for the case that the intruder can measure time with precision k.

Definition 6. *Let $\mathcal{O}$ be an observation and $\mathcal{S} \subseteq A$ such that $\mathcal{O}(a) \in \{\tau, \epsilon\}$ for $a \in \mathcal{S}$. We say that process P has $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ property (we will denote this by $P \in NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$) iff whenever $\mathcal{S} \cap s_1 \neq \emptyset$ for some $s_1 \in Tr(P)$ then there exists $s_2 \in Tr(P)$ such that $\mathcal{S} \cap s_2 = \emptyset$, $||s_1|_t - |s_2|_t| \leq k$ and it holds $\mathcal{O}(s_{1|Act}) = \mathcal{O}(s_{2|Act})$.*

Example 3. Let $P = l_1.t.t.t.h.l_2 + l_1.t.t.l_2$,$P' = l_1.t.t.t.t.h.l_2 + l_1.t.t.l_2$ and $\mathcal{O}(h) = \epsilon, \mathcal{O}(l_i) = l_i$. It is easy to check that $P \in NIF^{\{h\}}_{\mathcal{O}_{p1}}$ but $P' \notin NIF^{\{h\}}_{\mathcal{O}_{p1}}$. Note that $P, P' \in NIF^{\{h\}}_{\mathcal{O}_1}$. □

By generalization of this example we get the following relationships among $NIF^{\{h\}}_{\mathcal{O}_n}$ and $NIF^{\{h\}}_{\mathcal{O}_{pk}}$ properties.

Theorem 5. *$NIF^{\mathcal{S}}_{\mathcal{O}_n} \not\subseteq NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ and $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}} \not\subseteq NIF^{\mathcal{S}}_{\mathcal{O}_n}$.*

The relationship between $NIF^{\mathcal{S}}_{\mathcal{O}}$ and $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ states the following theorem.

Theorem 6. *$NIF^{\mathcal{S}}_{\mathcal{O}} \subset NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ for every k and $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}} \subset NIF^{\mathcal{S}}_{\mathcal{O}_{pl}}$ for $k < l$.*

Proof. The main idea. Clearly $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}} \subseteq NIF^{\mathcal{S}}_{\mathcal{O}_{pl}}$ for $k < l$. The rest follows from Example 2 and its generalization obtained by appropriate choice of an amount of t actions between actions l_1 and l_2. □

Combining Theorems 2 and 6 we get a hierarchy of NIF properties (see Fig. 1).

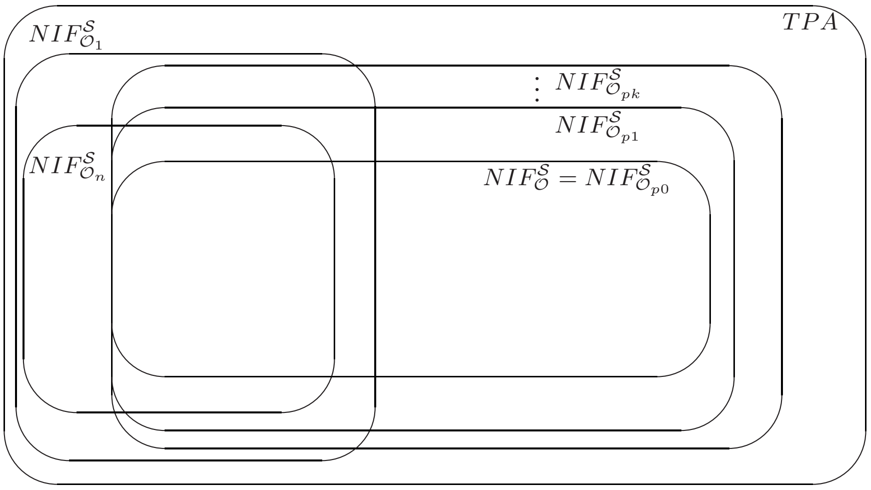

$NIF^{\mathcal{S}}_{\mathcal{O}_1}$
TPA
$\vdots$ $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$
$NIF^{\mathcal{S}}_{\mathcal{O}_{p1}}$
$NIF^{\mathcal{S}}_{\mathcal{O}_n}$
$NIF^{\mathcal{S}}_{\mathcal{O}} = NIF^{\mathcal{S}}_{\mathcal{O}_{p0}}$

Fig. 1. NIF Hierarchy

For the $NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ properties can be similar theorems as for $NIF^{\mathcal{S}}_{\mathcal{O}_n}$ (see Theorem 3 and 4) formulated. But for the lack of space, instead of this, we turn our attention from passive to active attacks.

3.3 Active Attacks

Up to now we have considered so called passive attacks. An intruder could only observe system behaviour. Now we will consider more powerful intruders which can employ some auxiliary processes to perform attacks. There is a natural restriction for such processes (see [8]), in the presented context this means that such the processes could perform only actions u for which $\mathcal{O}(u) = \epsilon$. We formulate the concept of active attacks (we will denote them by index a) in the framework of NIF property. A process which is considered to be safe also represents a safe context for the auxiliary private processes.

Definition 7. (Active NIF) *$P \in NIFa^{\mathcal{S}}_{\mathcal{O}_n}$ $(NIFa^{\mathcal{S}}_{\mathcal{O}_{pk}})$ iff $(P|A) \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$ $(NIF^{\mathcal{S}}_{\mathcal{O}_{pk}})$ for every $A, Sort(A) \subseteq \mathcal{S} \cup \{\tau, t\}$ and for every $x \in Sort(A), x \neq t$ it holds $\mathcal{O}(x) = \epsilon$.*

Active attacks are really more powerful than passive ones for both limited access and limited precision attacks.

Theorem 7. *$NIFa^{\mathcal{S}}_{\mathcal{O}_n} \subset NIF^{\mathcal{S}}_{\mathcal{O}_n}$ and $NIFa^{\mathcal{S}}_{\mathcal{O}_{pk}} \subset NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$.*

Proof. Sketch. Clearly $NIFa^{\mathcal{S}}_{\mathcal{O}_n} \subseteq NIF^{\mathcal{S}}_{\mathcal{O}_n}$ and $NIFa^{\mathcal{S}}_{\mathcal{O}_{pk}} \subseteq NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$. For the rest of the proof we construct processes P, A such that $P \in NIF^{\mathcal{S}}_{\mathcal{O}_n}$ but $(P|A) \notin NIF^{\mathcal{S}}_{\mathcal{O}_n}$ and $P \in NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$ but $(P|A) \notin NIF^{\mathcal{S}}_{\mathcal{O}_{pk}}$, respectively. For example we can consider processes $P = h_1.t^i.l + h_2.t^j.l$ and $A = t.\bar{h}_1$. By choosing appropriate values for i and j we get counterexamples which shows that both the inclusions are proper. □

The definition of active NIF properties contain two universal quantifications (over all possible intruders and over all possible traces). To avoid them we could exploit an idea of generalized unwinding introduced by Bossi, Focardi, Piazza and Rossi (see [4,1]) and in this way we can obtain decidability results for active NIF for finite state systems.

Note that also for $NIFa^{\mathcal{S}}_{\mathcal{O}_n}$ and $NIFa^{\mathcal{S}}_{\mathcal{O}_{pk}}$ similar properties as for $NIF^{\mathcal{S}}_{\mathcal{O}_n}$ (see Theorem 3 and 4) can be formulated.

4 Conclusions and Further Work

Timing attacks can "break" systems which are often considered to be "unbreakable". More precisely, the attacks usually do not break system algorithms themselves but rather their bad, from security point of view, implementations. For example, such implementations, due to different optimizations, could result in dependency between time of computation and data to be processed, and as a consequence systems might become open to timing attacks. An attacker can deduce from time information also some information about private data, despite the fact that safe algorithms were used.

In real applications an intruder very often has not full and complete access to systems to be attacked. In this case non-information flow property as it is known in the literature is too restrictive. There are systems which exhibit some information flow but only in case of an "ideal" condition for the intruder, i.e. when the intruder has unlimited access to system and when time of action occurrences can be measured with absolute precision. In both these cases the standard non-information flow property is rather strong and for many applications too restrictive.

In this paper we have presented two formal models which model two different types of intruders. The first one has access to a system to be attacked only within some time window, i.e. (s)he can see its behaviour only during some time interval. The second one can measure time of actions occurrences only with some given precision. The presented formalisms are studied and compared with other concepts described in the literature and it is shown that they are more general and stronger in the sense that they can describe attacks which are not captured by the other concepts. With the help of presented models we can check systems with respect to more adequate security requirements. In this paper we have studied these requirements and we have obtained some decidability and undecidability results for them.

We see our work as a first step towards an analysis of timing attacks. Further study will concern on more efficient decision algorithms, modeling of more elaborated active time attacks where an attacker can implement some less restricted processes to the system to be attacked (for example in the style of Trojan horse) to deduce some private activities. To have better described system activities (particularly to be able to perform traffic analysis), we consider to use formalism which can express also some network properties in style of [14,17]. This approach was used in [15] to study Bisimulation-based Non-deducibility on Composition which is an (stronger) alternative to SNNI. Since many of timing attacks are based on statistic behaviour it seems to be reasonable to exploit also some features of probabilistic process algebras.

References

1. Bossi A., D. Macedonio, C. Piazza and S. Rossi. Information Flow in Secure Contexts. Journal of Computer Security, Volume 13, Number 3, 2005
2. Bryans J., M. Koutny and P. Ryan: Modelling non-deducibility using Petri Nets. Proc. of the 2nd International Workshop on Security Issues with Petri Nets and other Computational Models, 2004.
3. Bryans J., M. Koutny, L. Mazare and P. Ryan: Opacity Generalised to Transition Systems. CS-TR-868, University of Newcastle upon Tyne, 2004.
4. Bossi A., R. Focardi, C. Piazza and S. Rossi. Refinement Operators and Information Flow Security. Proc. of SEFM'03, IEEE Computer Society Press, 2003.
5. Busi N. and R. Gorrieri: Positive Non-interference in Elementary and Trace Nets. Proc. of Application and Theory of Petri Nets 2004, LNCS 3099, Springer, Berlin, 2004.
6. Dhem J.-F., F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater and J.-L. Willems: A practical implementation of the timing attack. Proc. of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998), LNCS 1820, Springer, Berlin, 1998.
7. Felten, E.W., and M.A. Schneider: Timing attacks on web privacy. Proc. of the 7^{th} ACM Conference on Computer and Communications Security, 2000.
8. Focardi, R. and R. Gorrieri: Classification of security properties. Part I: Information Flow. Proc. of Foundations of Security Analysis and Design, LNCS 2171, Springer, Berlin, 2001.
9. Focardi, R., R. Gorrieri, and F. Martinelli: Information flow analysis in a discrete-time process algebra. Proc. of the 13^{th} Computer Security Foundation Workshop, IEEE Computer Society Press, 2000.
10. Focardi, R., R. Gorrieri, and F. Martinelli: Real-Time information flow analysis. IEEE Journal on Selected Areas in Communications 21 (2003).
11. Gorrieri R. and F. Martinelli: A simple framework for real-time cryptographic protocol analysis with compositional proof rules. Science of Computer Programing, Volume 50, Issue 1-3, 2004.
12. Goguen J.A. and J. Meseguer: Security Policies and Security Models. Proc. of the IEEE Symposium on Security and Privacy, 1982.
13. Groote, J. F.: "Transition Systems Specification with Negative Premises". Proc. of *CONCUR'90*, Springer Verlag, Berlin, LNCS 458, 1990.
14. Gruska D.P. and A. Maggiolo-Schettini: Process algebra for network communication. Fundamenta Informaticae 45(2001).
15. Gruska, D., Maggiolo-Schettini, A.: Nested Timing Attacks, Proc. of FAST 2003, 2003.

16. Gruska D.P.: Information Flow in Timing Attacks. Proc. of CS&P'04, 2004.
17. Gruska D.P.: Network Information Flow, Fundamenta Informaticae 72 (2006).
18. Handschuh H. and Howard M. Heys: A timing attack on RC5. Proc. of the Selected Areas in Cryptography, LNCS 1556, Springer, Berlin, 1999.
19. Kocher P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. Proc. of the Advances in Cryptology - CRYPTO'96, LNCS 1109, Springer, Berlin, 1996.
20. Milner, R.: *Communication and concurrency.* Prentice-Hall International, New York,1989.
21. Sabelfeld A. and A.C. Myers: Language-Based Information Flow Security. IEEE Journal on Selected Areas in Communication, 21(1), 2003.
22. Song. D., D. Wagner, and X. Tian: Timing analysis of Keystrokes and SSH timing attacks. Proc. of the 10th USENIX Security Symposium, 2001.

Verifying Generalized Soundness of Workflow Nets

Kees van Hee, Olivia Oanea*, Natalia Sidorova, and Marc Voorhoeve

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee, o.i.oanea, n.sidorova, m.voorhoeve}@tue.nl

Abstract. We improve the decision procedure from [10] for the problem of generalized soundness of workflow nets. A workflow net is generalized sound iff every marking reachable from an initial marking with k tokens on the initial place terminates properly, i.e. it can reach a marking with k tokens on the final place, for an arbitrary natural number k. Our new decision procedure not only reports whether the net is sound or not, but also returns a counterexample in case the workflow net is not generalized sound. We report on experimental results obtained with the prototype we made and explain how the procedure can be used for the compositional verification of large workflows.

Keywords: Petri nets; workflows; verification; soundness.

1 Introduction

Petri nets are intensively used in workflow modeling [1,2]. Workflow management systems are modeled by *workflow nets* (WF-nets), i.e. Petri nets with one initial and one final place and every place or transition being on a directed path from the initial to the final place. The execution of a case is represented as a firing sequence that starts from the initial marking consisting of a single token on the initial place. The token on the final place with no garbage (tokens) left on the other places indicates the *proper termination* of the case execution. A model is called *sound* iff every reachable marking can terminate properly.

In [9] we showed that the traditional notion of soundness from [1] is not compositional, and moreover, it does not allow for handling of multiple cases in the WF-net. We introduced there a notion of *generalized soundness* that amounts to the proper termination of all markings obtained from markings with multiple tokens on the initial place, which corresponds to the processing of batches of cases in the WF-net. We proved that generalized soundness is compositional w.r.t. refinement, which allows the verification of soundness in a compositional way.

* Supported by the NWO Open Competitie project "Modeling and Verification of Business Processes" (MoveBP), project number 612.000.315.

The generalized soundness problem is decidable and [10] gives a decision procedure for it. The problem of generalized soundness is reduced to two checks. First, some linear equations for the incidence matrix are checked. Secondly, the proper termination of a finite set of markings is checked. This finite set of markings is computed from an over-approximation of the set of reachable markings that has a regular algebraic structure. In practice, this set turns out to be very large, which seriously limits the applicability of the decision procedure from [10].

In this paper we show that the check of generalized soundness can be reduced to checking proper termination for a much smaller set of markings, namely minimal markings of the set from [10]. We describe a new decision procedure for soundness. Additionally, our procedure produces a counterexample in case a WF-net turns out to be unsound, showing a reachable marking that cannot terminate properly and a trace leading to it.

We implemented our decision procedure in a prototype tool and performed a series of experiments with it. The experimental results confirmed that the new procedure is considerably more effective than the old one. When applied together with standard reduction techniques in a compositional way, it allows us to check soundness of real-life examples.

Related work. For some subclasses of workflow nets (e.g. well-handled, free-choice, extended free-choice, asymmetric choice where every siphon includes at least one trap, extended non-self controlling workflow nets), 1-soundness implies generalized soundness (see [13]). A different decision procedure for generalized soundness was presented in [17], where the generalized soundness problem is reduced to the home marking problem in an extension of the workflow net, which is an *unbounded* net. The home marking problem is shown to be decidable in [7] by reducing it to the reachability problem for a finite set of markings. Checking generalized soundness in this way can however hardly be done in practice, since the complexity of the reachability problem for unbounded nets is still an open question, and the procedure for checking reachability, though known from 1981 [11], has never been implemented due to its complexity (the known algorithms require non-primitive recursive space) [15]. In our procedure we also check reachability for a finite set of markings, but reachability is checked on *bounded* nets only.

The paper is structured as follows. Section 2 introduces basic notions. Section 3 presents the new decision procedure, and Section 4 provides details about the implementation and experimental results. Section 5 covers conclusions and directions for future work.

2 Preliminaries

We denote the set of natural numbers by $\mathbb{N}$, the set of non-zero natural numbers by $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$, the set of integers by $\mathbb{Z}$, the set of rational numbers by $\mathbb{Q}$ and the set of non-negative rational numbers by $\mathbb{Q}^+$. We denote the set of all finite words over a finite set S by S^*. The empty word is denoted by ϵ.

A *Petri net* is a tuple $N = (P, T, F^+, F^-)$, where

- P and T are two disjoint non-empty finite sets of *places* and *transitions* respectively;
- F^+ and F^- are mappings $(P \times T) \to \mathbb{N}$ that are *flow functions* from transitions to places and from places to transitions respectively.

$F = F^+ - F^-$ is the *incidence matrix* of net N.

We denote the set of *output transitions* of a place p by $p^\bullet$, i.e. $p^\bullet \stackrel{\text{def}}{=} \{t \mid F^+(p,t) > 0, t \in T\}$, and the set of output transitions of $Q \subseteq P$ by $Q^\bullet$, i.e. $Q^\bullet \stackrel{\text{def}}{=} \bigcup_{p \in Q} p^\bullet$. Similarly, ${}^\bullet p \stackrel{\text{def}}{=} \{t \mid F^-(p,t) > 0, t \in T\}$ denotes the set of *input transitions* of a place p and ${}^\bullet Q \stackrel{\text{def}}{=} \bigcup_{p \in Q} {}^\bullet p$ the set of input transitions of $Q \subseteq P$. A place p with ${}^\bullet p = \emptyset$ is called a *source place* and a place q with $q^\bullet = \emptyset$ is called a *sink place*.

Markings, the states (configurations) of a net, represent the distribution of tokens in the places are interpreted as vectors $m \colon P \to \mathbb{N}$. We denote by $\bar{0}$ the zero marking (vector) of an arbitrary (defined by the context) dimension and by $\bar{p}$, for some $p \in P$, the vector such that $\bar{p}(p) = 1$ and $\bar{p}(p') = 0$ for all $p' \in P$ such that $p' \neq p$. A *marked net* is a tuple (N, m), where N is a net and m is a marking.

A transition $t \in T$ is *enabled* in a marking m if $F^-(p,t) \leq m(p)$, for all $p \in P$. If t is enabled in a marking m (denoted by $m \xrightarrow{t}$), t may *fire* yielding a new marking m', denoted by $m \xrightarrow{t} m'$, where $m'(p) = m(p) - F^-(p,t) + F^+(p,t)$, for all $p \in P$. We extend this homomorphically to the firing sequences $\sigma \in T^*$, denoted by $m \xrightarrow{\sigma} m'$. We say that m' is reachable from m and write $m \xrightarrow{*} m'$ when there exists $\sigma \in T^*$ such that $m \xrightarrow{\sigma} m'$. We denote the set of all markings reachable from m by $\mathcal{R}(m)$. Similarly, $\mathcal{S}(m)$ denotes the set of markings that can reach m. A marked net (N, m_0) is *bounded* iff there exists $n \in \mathbb{N}$ such that for all $m \in \mathcal{R}(m_0)$, $m(p) < n$ for all $p \in P$. A marked net (N, m_0) is *t-live* iff for all markings $m \in \mathcal{R}(m_0)$ there exists a marking m' such that $m \xrightarrow{*} m'$ and $m \xrightarrow{t}$.

Let σ be a sequence of transitions. The *Parikh vector* $\overrightarrow{\sigma}$ maps every transition t of σ to the number of occurrences of t in σ. Let $m \xrightarrow{\sigma} m'$. Then the *marking equation* [8] holds: $m' = m + F \cdot \overrightarrow{\sigma}$. Note that the reverse is not true: not every marking m' that can be represented as $m + F \cdot x$, for some $x \in \mathbb{N}^T$, is reachable from the marking m.

A subset of places Q is called a *trap* if $Q^\bullet \subseteq {}^\bullet Q$. A subset $Q \subseteq P$ is called a *siphon* if ${}^\bullet Q \subseteq Q^\bullet$. A trap or a siphon is called *proper* iff it is nonempty. Traps have the property that once marked they remain marked, whereas unmarked siphons remain unmarked whatever transition sequence occurs [6].

A place invariant is a row vector $I \colon P \to \mathbb{Q}$ such that $I \cdot F = 0$. We denote a matrix that consists of basis place invariants as rows by $\mathcal{I}$. We say that markings m and m' *agree on a place invariant* I if $I \cdot m = I \cdot m'$ (see [8]). The main property of place invariants is that any two markings m, m' such that $m \xrightarrow{*} m'$ agree on all place invariants, i.e. $\mathcal{I} \cdot m = \mathcal{I} \cdot m'$.

Batch Workflow Nets. Workflow nets are used to model the processing of tasks in workflow processes. The initial and final nodes indicate respectively the initial and final states of cases flowing through the process.

Definition 1. *A Petri net N is a* Workflow net (WF-net) *iff:*

1. *N has two special places: i and f. The initial place i is a source place, i.e. ${}^\bullet i = \emptyset$, and the final place f is a sink place, i.e. $f^\bullet = \emptyset$.*
2. *For any node $n \in (P \cup T)$ there exists a path from i to n and a path from n to f (*path property *of WF-nets.)*

In [10], we introduced two structural correctness criteria for WF-nets based on siphons and traps:

non-redundancy. Every place can be marked and every transition can fire, provided there are enough tokens in the initial place.
non-persistency. All places can become empty again.

As proven in [10], non-redundancy and non-persistency are behavioral properties which imply restrictions on the structure of the net: all proper siphons of the net should contain i and all proper traps should contain f.

Following [10], we define a class of nets called batch workflow nets (BWF-nets). Actually, BWF-nets are WF-nets without redundant and persistent places, i.e. workflow nets that satisfy minimal correctness requirements.

Definition 2. *A* Batch Workflow net (BWF-net) *N is a Petri net having the following properties:*

1. *N has a single source place i and a single sink place f.*
2. *Every transition of N has at least one input and one output place.*
3. *Every proper siphon of N contains i.*
4. *Every proper trap of N contains f.*

Workflow nets were originally used to model the execution of one case. In [10], we defined a generalized notion of soundness for modeling the execution of batches of cases in WF-nets.

Definition 3. *A WF-net N is called k-*sound *for some $k \in \mathbb{N}$ iff*

$$\mathcal{R}(k \cdot \bar{i}) \subseteq \mathcal{S}(k \cdot \bar{f}).$$

A WF-net N is called generalized sound *iff*

$$\forall k \in \mathbb{N} \colon \mathcal{R}(k \cdot \bar{i}) \subseteq \mathcal{S}(k \cdot \bar{f}).$$

For the sake of brevity, we omit the word "generalized" in the rest of the paper. In [10], it has been shown that a WF-net N is sound iff a certain derived BWF-net N' is sound. The derivation is straightforward and only uses structural analysis of the net.

We assume that the reader is familiar with the basics of convexity theory (see e.g. [16]). A *convex polyhedral cone* $\mathcal{H}$ over $\mathbb{Q}^m$ can be defined by its finite set of generators $E \subseteq \mathbb{Q}^m$, i.e. $\mathcal{H} = \{\Sigma_{e \in E} \lambda_e \cdot e \mid \lambda_e \in \mathbb{Q}^+\}$. A generator e is called *trivial* if $e = \bar{j}$, for some $1 \leq j \leq m$.

3 Decision Procedure of Soundness for BWF-Nets

In this section, we describe our decision procedure for checking generalized soundness of BWF-nets. Our decision procedure improves the one from [10] since we check proper termination for a much smaller set of markings. We give an algorithm for computing this set of markings and enhance the procedure with a backward reachability algorithm that checks whether these markings are backward reachable from some final marking. If not, our procedure returns a counterexample.

We start by briefly discussing the decision procedure from [10]. We first give some necessary conditions of soundness:

Lemma 4. [10] *Let N be a sound BWF-net. Then,*

1. $\mathcal{I} \cdot \bar{i} = \mathcal{I} \cdot \bar{f}$ *(i and f agree on the basis place invariants);*
2. $\mathcal{I} \cdot x = \bar{0}$ *for* $x \in (\mathbb{Q}^+)^P$ *iff* $x = \bar{0}$.

The conditions of Lemma 4 can be easily checked by standard algebraic techniques. Further on, we we consider only nets that meet these two conditions.

The set of all markings reachable from some initial marking of N is given by $\mathcal{R} = \bigcup_{k \in \mathbb{N}} \mathcal{R}(k \cdot \bar{i})$. Due to the marking equation, $\mathcal{R}(k \cdot \bar{i})$ is a subset of $\mathcal{G}_k = \{k \cdot \bar{i} + F \cdot v \mid v \in \mathbb{Z}^T\} \cap \mathbb{N}^P$. Note that the reverse is not true.

Let $m \in \mathcal{G}_k$, for some $k \in \mathbb{N}$. Then $\mathcal{I} \cdot m = \mathcal{I} \cdot (k \cdot \bar{i})$. Since condition 2 of Lemma 4 holds, $\mathcal{G}_k \cap \mathcal{G}_\ell = \emptyset$ for all $k, \ell \in \mathbb{N}$, with $k \neq l$, and we can define the *i-weight* function $w(m)$ for m as k. Now consider the set $\mathcal{G} = \bigcup_{k \in \mathbb{N}} \mathcal{G}_k$, i.e. $\mathcal{G} = \{k \cdot \bar{i} + F \cdot v \mid k \in \mathbb{N} \wedge v \in \mathbb{Z}^T\} \cap \mathbb{N}^P$. We will say that a marking $m \in \mathcal{G}$ *terminates properly* if $m \xrightarrow{*} w(m) \cdot \bar{f}$.

Lemma 5. [10] *Let $m_1, m_2 \in \mathcal{G}$ be markings that terminate properly and $m = \lambda_1 \cdot m_1 + \lambda_2 \cdot m_2$ for some $\lambda_1, \lambda_2 \in \mathbb{N}$. Then $m \in \mathcal{G}$ and it terminates properly.*

Theorem 6. [10] *Let N be a BWF-net. Then N is sound iff for any $m \in \mathcal{G}$, $m \xrightarrow{*} w(m) \cdot \bar{f}$.*

$\mathcal{G}$ is an infinite set, but unlike $\mathcal{R}$ it has a regular algebraic structure, which allows to reduce the check of proper termination to a check on a finite set of markings.

The following lemma is proved by using convexity analysis [16], notably the Farkas-Minkowski-Weyl theorem.

Lemma 7. [10] *Let $\mathcal{H} \overset{def}{=} \{a \cdot \bar{i} + F \cdot v \mid a \in \mathbb{Q}^+ \wedge v \in \mathbb{Q}^T\} \cap (\mathbb{Q}^+)^P$. Then there exist a finite set $E_\mathcal{G} \subseteq \mathcal{G}$ of generators of $\mathcal{H}$, i.e. $\mathcal{H} = \{\Sigma_{e \in E_\mathcal{G}} \lambda_e \cdot e \mid \lambda_e \in \mathbb{Q}^+\}$.*

The soundness check can now be reduced to the check of proper termination for a finite set of markings:

Theorem 8. [10] *Let N be a BWF-net such that the conditions of Lemma 4 hold and let $\Gamma \overset{def}{=} \{\sum_{e \in E_\mathcal{G}} \alpha_e \cdot e \mid 0 \leq \alpha_e \leq 1 \wedge e \in E_\mathcal{G}\} \cap \mathcal{G}$, where $E_\mathcal{G} \subseteq \mathcal{G}$ is a finite set of generators. Then N is sound iff all markings in Γ terminate properly.*

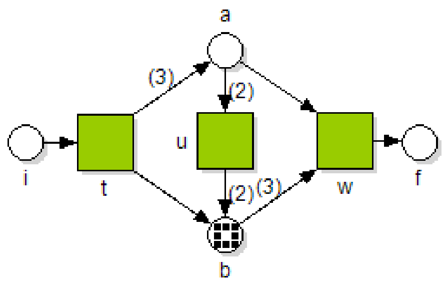


Fig. 1. Example of a BWF-net

In fact, Γ represents the set of integer points of the bounded convex polyhedral cone (also called polytope) having the set $E_{\mathcal{G}}$ as generators.

The decision procedure from [10] comprises the following steps:

1. Find an invariant matrix $\mathcal{I}$ and check whether $\mathcal{I} \cdot \bar{i} = \mathcal{I} \cdot \bar{f}$ and whether $\mathcal{I} \cdot x = \bar{0}$ has only the trivial solution on $\mathbb{N}^P$;
2. Find a set $E_{\mathcal{G}} \subset \mathcal{G}$ of generators of $\mathcal{H}$;
3. Compute the set of markings Γ;
4. Check for all markings $m \in \Gamma$ that $m \xrightarrow{*} w(m) \cdot \bar{f}$.

Example 9. We illustrate the main steps of the algorithm on the BWF-net in Figure 1. First we compute $\mathcal{I} = (4,1,1,4)$. The first two conditions are satisfied: $(4,1,1,4) \cdot \bar{i} = (4,1,1,4) \cdot \bar{f}$ and $(4,1,1,4) \cdot x = \bar{0}$ implies $x = \bar{0}$. Further we compute $\mathcal{H} = \{a \cdot \bar{i} + F \cdot v | a \in \mathbb{Q}^+, v \in \mathbb{Q}^T\} \cap (\mathbb{Q}^+)^P = (A+B) \cap C$, where A, B and C are polyhedra having the generators $\{\bar{i}\}$, $\{\pm(3\cdot\bar{a}+\bar{b}-\bar{i}), \pm(\bar{a}+\bar{b}), \pm(\bar{i}-\bar{a}-3\cdot\bar{b})\}$ and $\{\bar{i}, \bar{f}, \bar{a}, \bar{b}\}$, respectively. Next we compute the generators of the polytope: $E_{\mathcal{G}} = \{\bar{i}, \bar{f}, 8 \cdot \bar{a}, 8 \cdot \bar{b}\}$. The markings of Γ have the following form:

$$\Gamma = \{m' | (m' = \sum_{m \in E_{\mathcal{G}} \cup \{3\cdot\bar{a}+\bar{b}, \bar{a}+3\cdot\bar{b}\}; \alpha_m \in \mathbb{N}} \alpha_m \cdot m) \wedge (\bar{0} \leq m' \leq \sum_{e \in E_{\mathcal{G}}} e)\}$$

The size of Γ is very large compared to the size of the net: $|\Gamma| = 44$. Furthermore in order to check whether all markings of Γ terminate properly, we need to build the backward reachability sets $\mathcal{S}(k \cdot \bar{i})$ for $0 \leq k \leq \max_{m \in \Gamma} w(m) = 6$ and check whether they include all markings of Γ. We observe that $8 \cdot \bar{b} \notin \mathcal{S}(2 \cdot \bar{f})$ and therefore the net is not sound.

Steps $1-2$ are not computationally costly. The set of markings Γ turns out to be very large in practice, and Step 3 and 4 are thus very expensive for real-life examples. We shall reduce the check of proper terminations of markings from Γ to a check of a smaller set of markings by replacing the last two steps with the following steps:

3'. Compute the set Υ of *minimal markings* of $\mathcal{G}^+ \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}^+} \mathcal{G}(k \cdot \bar{i})$, i.e.

$$\Upsilon \stackrel{\text{def}}{=} \{m \mid \forall m' \in \mathcal{G}^+ : m' \leq m \Rightarrow m' = m\}.$$

4'. Check that for all markings $m \in \Upsilon$, $m \xrightarrow{*} w(m) \cdot \bar{f}$. In case this does not hold, give a counterexample, i.e. a trace σ such that $w(m') \cdot \bar{i} \xrightarrow{\sigma} m' \not\xrightarrow{*} w(m') \cdot \bar{f}$, for some m'.

To show that Υ can be used instead of Γ, we first prove an auxiliary lemma.

Lemma 10. *Let N be a BWF-net, $m_1 \in \mathcal{G}_{k_1}$, $m_2 \in \mathcal{G}_{k_2}$, for some $k_1, k_2 \in \mathbb{N}$, and $m_2 > m_1$. Then $k_2 > k_1$.*

Proof. Since $m_2 \in \mathcal{G}_{k_2}$ and $m_1 \in \mathcal{G}_{k_1}$, $m_1 = k_1 \cdot \bar{i} + F \cdot v$ and $m_2 = k_2 \cdot \bar{i} + F \cdot v_2$ for some v_1 and v_2. Hence, $\mathcal{I} \cdot m_1 = \mathcal{I} \cdot k_1 \cdot \bar{i}$ and $\mathcal{I} \cdot m_2 = \mathcal{I} \cdot k_2 \cdot \bar{i}$ and by linearity $\mathcal{I}(m_2 - m_1 + (k_1 - k_2)\bar{i}) = \bar{0}$. Since condition 2 of Lemma 4 holds for N, $m_2 - m_1 + (k_1 - k_2) \cdot \bar{i} \notin (\mathbb{Q}^+)^P \setminus \{\bar{0}\}$. Since $m_2 - m_1$ and $\bar{i}$ are in $(\mathbb{Q}^+)^P$, we deduce that $k_1 - k_2 < 0$. □

The set of markings Υ has the following properties:

– Let $E_{\mathcal{G}} \subseteq \mathcal{G}$ be a set of minimal generators of $\mathcal{H}$ in $\mathcal{G}$ (i.e. for any $e \in E_{\mathcal{G}}$ and $e' \in \mathcal{G}$, $e' \leq e$ implies $e = e'$). Then $E_{\mathcal{G}} \subseteq \Upsilon$. Note that in particular $\bar{i}$, $\bar{f} \in E_{\mathcal{G}} \subseteq \Upsilon$.
– $\mathcal{G}_1 \subseteq \Upsilon$. Suppose that there is an $m \in \mathcal{G}_1$ such that $m \notin \Upsilon$. Then there is $m' \in \Upsilon$ such that $m' < m$. By Lemma 10, $w(m') < w(m) = 1$, contradiction.

We now formulate our theorem.

Theorem 11. *Let N be a BWF-net such that $\mathcal{I} \cdot \bar{i} = \mathcal{I} \cdot \bar{f}$, $\mathcal{I} \cdot x = \bar{0}$ has only the trivial solution in $(\mathbb{Q}^+)^P$, $\mathcal{G}^+ \overset{def}{=} \{k \cdot \bar{i} + F \cdot v \mid k \in \mathbb{N}^+ \wedge v \in \mathbb{Z}^T\} \cap \mathbb{N}^P$, $\mathcal{H} = \{a \cdot \bar{i} + F \cdot v \mid a \in \mathbb{Q}^+, v \in \mathbb{Q}^T\} \cap (\mathbb{Q}^+)^P$, $E_{\mathcal{G}} \subseteq \mathcal{G}^+$ be a set of minimal generators of $\mathcal{H}$ in $\mathcal{G}^+$, $\Gamma = \{\sum_{e \in E_{\mathcal{G}}} \alpha_e \cdot e \mid 0 \leq \alpha_e \leq 1 \wedge e \in E_{\mathcal{G}}\} \cap \mathcal{G}$, and Υ be the set of minimal markings of $\mathcal{G}^+$. Then:*

1. *N is sound iff for any marking $m \in \Upsilon$, $m \xrightarrow{*} w(m) \cdot \bar{f}$.*
2. *Each marking $m \in \Upsilon$ satisfies $m < M$, where $M(p) = \max_{e \in E_{\mathcal{G}}} e(p)$, for every $p \in P$.*
3. *$\Upsilon \subset \Gamma$.*

Proof. (1) ($\Rightarrow$) Since N is sound, all markings of $\mathcal{G}$ terminate properly (by Theorem 6). Since $\Upsilon \subseteq \mathcal{G}$, all markings of Υ terminate properly.
($\Leftarrow$) Let $m \xrightarrow{*} w(m) \cdot \bar{f}$ for every marking m from Υ. We will prove that $m \xrightarrow{*} w(m) \cdot \bar{f}$ for every marking m from Γ, which implies then that N is sound (by Theorem 8).

Let $m \in \Gamma$. We have two cases: $m \in \Upsilon$ and $m \in \Gamma \setminus \Upsilon$. If $m \in \Upsilon$, then $m \xrightarrow{*} w(m) \cdot \bar{f}$. If $m \in (\Gamma \setminus \Upsilon)$, $m > \Delta^0$ for some $\Delta^0 \in \Upsilon$ and by Lemma 10, $w(m) > w(\Delta^0)$, which also implies that $(m - \Delta^0) \in \mathcal{G}^+$.

Set $m^0 = m - \Delta^0$. In case $m^0 \in \Upsilon$, $m^0 \xrightarrow{*} w(m^0) \cdot \bar{f}$. By Lemma 5, since $\Delta^0 \xrightarrow{*} w(\Delta^0) \cdot \bar{f}$, $m \xrightarrow{*} w(m) \cdot \bar{f}$. In case $m^0 \notin \Upsilon$, m^0 can be further written as $m^0 = \Delta^1 + m^1$, where $\Delta^1 \in \Upsilon$ and $m^1 \in \mathcal{G}^+$.

We continue until we reach an $m^{l-1} = m^l + \Delta^l$ with $\Delta^l \in \Upsilon$ and $m^l \in \Upsilon$. Note that the process is finite since $0 < m^{i+1} < m^i$, for $0 \leq i \leq l$. Therefore $m = \sum_{i=0}^{l} \Delta^i + m^l$, where $m^l \in \Upsilon$ and $\Delta^i \in \Upsilon$ for all $i = 0 \ldots l$. Since the markings of Υ terminate properly, we can apply Lemma 5 to $\sum_{i=0}^{l} \Delta^l + m^0$. As a result, $m \xrightarrow{*} w(m) \cdot \bar{f}$.
(2) Suppose that there is a marking $m \in \Upsilon$ such that $m \geq M$. Since $M \geq e$ for every generator $e \in E_{\mathcal{G}}$, we have $\forall e \in E_{\mathcal{G}}\colon m \geq e$. That means that m and e are comparable, which contradicts the hypothesis.
(3) $\Upsilon \subseteq \Gamma$ follows trivially from (2) and the definition of Γ. Furthermore, $\bar{M} = \sum_{e \in E_{\mathcal{G}}} e \in \Gamma$. However $\bar{M} > M$ and from (2), we have that $\bar{M} \notin \Upsilon$, hence $\Upsilon \subset \Gamma$. □

Now we can describe the implementation the steps $2, 3', 4'$.

Computing the generators of the convex polyhedral cone $\mathcal{H}$. $\mathcal{H}$ is given as the intersection of two polyhedra: A with the set of generators $\{\bar{i}\} \cup \{\pm F(t) \mid t \in T\}$ (column vectors of the matrices F and $-F$) and B with the set of generators $\{\bar{p} \mid p \in P\}$ (trivial generators). Let E be a (minimal) set of generators of the convex polyhedral cone $\mathcal{H} = \{a \cdot \bar{i} + F \cdot v \mid a \in \mathbb{Q}^+, v \in \mathbb{Q}^T\} \cap (\mathbb{Q}^+)^P$. All generators of $\mathcal{H}$ can be represented as $a \cdot \bar{i} + F \cdot v$, where $a \in \mathbb{Q}$ and $v \in \mathbb{Q}^T$ can be found by solving linear equations. In order to find the set of generators that are in $\mathcal{G}$ ($E_{\mathcal{G}}$), the generators of $\mathcal{H}$ need to be rescaled. The rescaling factor of each generator is the lcm of the denominators of a and v_t, for all $t \in T$ divided by the gcd of the numerators of them. $\bar{i}$ and $\bar{f}$ are generators of $\mathcal{H}$ with rescaling factor 1.

Computing Υ. The next step is to find Υ — the set of minimal markings of $\mathcal{G}$. Note that the markings of Υ are smaller than the marking M whose components are the maxima of the respective components of the rescaled generators (statement 2 of Theorem 11).

We compute Υ by an optimized enumeration of all vectors m from $\mathbb{N}^P$ which are smaller than M and checking whether $m = k \cdot \bar{i} + F \cdot v$ has a solution in $\mathbb{N}^+$, i.e. whether $m \in \mathcal{G}$. The optimization is due to avoiding the consideration of markings which are greater than some markings already added to Υ.

Checking proper termination for markings of Υ. We need to check that $m \xrightarrow{*} w(m) \cdot \bar{f}$ for all $m \in \Upsilon$. Since condition 2 of Lemma 4 holds, we conclude that $\mathcal{S}(k \cdot \bar{f})$ is a finite set for any k. Therefore we employ a backward reachability algorithm to check proper termination of markings in Υ. Let J be the (finite) set of weights of markings from Υ. The backward reachability algorithm constructs for each i-weight $j \in J$, starting from weight 1, the backward reachability set B_j. We start from the marking $j \cdot \bar{f}$ and continue by adding the markings $\{m - F_t \mid m \in B_j \wedge m - F_t^+ \geq \bar{0} \wedge t \in T\}$, where F_t is column of F corresponding to transition t, until B_j contains all markings from Υ_j or we reach the fixpoint $\mathcal{S}(j \cdot \bar{f})$. In the first case all markings of Υ_j terminate properly; as a

Algorithm 1. Backward reachability check

```
Input: $N = (P, T, F)$, $\Upsilon$, $J = \{w(m) \mid m \in \Upsilon\}$
Output: "the BWF-net is sound" or "the BWF-net is not sound, $m, k$" where
        $m \in \mathcal{G}_k$ and $m \not\xrightarrow{*} k \cdot \bar{f}$.
for $j \in J$ do
    $B_j = \{j \cdot \bar{f}\}$;
    repeat
        $B_j = B_j \cup \{m - F_t \mid \forall p \in P : m(p) \geq F(p,t) \wedge m \in B_j \wedge t \in T\}$
    until a fixpoint is reached or $\Upsilon_j \subseteq B_j$ ;
    if $\Upsilon_j \not\subseteq B_j$ then
        pick $m \in \Upsilon_j \setminus B_j$;
        return ("the BWF-net is not sound", $m, j$)
    end
end
return ("the BWF-net is sound")
```

result the BWF-net is sound. In the latter case the markings in Υ_j do not terminate properly; therefore the net is not sound. Note that the backward reachability sets B_j are distinct (since $\mathcal{G}_k \cap \mathcal{G}_\ell = \emptyset$ for any $k \neq \ell$).

This check results either in verdict "sound" (if all markings from Υ terminate properly), or "unsound" together with some marking that does not terminate properly in the contrary case.

Finding a counterexample. Let m be a marking from Υ_j returned by the check above as non-properly terminating. Like all markings from Υ_j, m does not necessarily belong to $\mathcal{R}(j \cdot \bar{i})$. To give a counterexample, we search through $\mathcal{R}(k \cdot \bar{i})$ $(k \geq w(m))$ to find a marking m' reachable from $w(m') \cdot \bar{i}$ and not terminating properly and show a trace σ such that $w(m') \cdot \bar{i} \xrightarrow{\sigma} m'$.

Example 9 continued. We compute Υ for the example from Figure 1:

$$\Upsilon = \{\bar{i}, \bar{f}, 8 \cdot \bar{a}, 8 \cdot \bar{b}, \bar{a} + 3 \cdot \bar{b}, 3 \cdot \bar{a} + \bar{b}\}$$

Note that $|\Upsilon| = 6$, while $|\Gamma| = 44$. Moreover, the maximal i-weight of the markings of Υ is a lot smaller than that of the markings of Γ: $\max_{m \in \Upsilon} w(m) = 2 < \max_{m \in \Gamma} w(m) = 6$. Hence, we need to compute only $\mathcal{S}(\bar{f})$ and $\mathcal{S}(2 \cdot \bar{f})$ instead of $\mathcal{S}(k \cdot \bar{f})$ for $k = 1 \ldots 6$. We find a counterexample $8 \cdot \bar{b} \in \mathcal{R}(2 \cdot \bar{i})$: $2 \cdot \bar{i} \xrightarrow{tt} 6 \cdot \bar{a} + 2 \cdot \bar{b} \xrightarrow{uuu} 8 \cdot \bar{b}$ and conclude that the net is not sound. Figure 1 shows the dead marking.

Example 12. Figure 2 shows a Petri net which is 1-sound, but not 2-sound. In this case $\Upsilon = \Upsilon_1 = \{\bar{i}, \bar{f}, \bar{a}, \bar{b}, \bar{c}\} = E_{\mathcal{G}}$. Using the backward reachability algorithm, we find that the net is not sound and $\bar{b} \in \Upsilon_1$ such that $\bar{b} \not\xrightarrow{*} \bar{f}$. However, $\bar{b} \notin \mathcal{R}(\bar{i})$. We find $2 \cdot \bar{b} + \bar{f} > \bar{b}$ such that $2 \cdot \bar{i} \xrightarrow{tvy} 2 \cdot \bar{b} + \bar{f} \not\xrightarrow{*} 2 \cdot \bar{f}$.

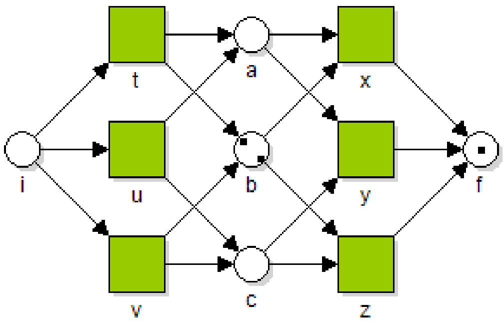


Fig. 2. A net for which $\Upsilon = \Upsilon_1 \not\subseteq \mathcal{R}(\bar{i})$

4 Practical Application of the Decision Procedure

In this section, we we give some details on the implementation of the procedure and experimental results and discuss how to check soundness for large nets compositionally and by using reduction techniques.

Implementation and experimental results. The decision procedure described in Section 3 has been implemented in a prototype tool. The tool uses the Yasper editor [14] for input of batch workflow nets and gives as output the conclusion on soundness and a counterexample in case the net is not sound. The prototype is written in C++ and uses the Parma Polyhedra Library [3,4] for the computation of the minimal set of generators of the convex polyhedral cone $\mathcal{H}$.

The complexity of the procedure is dominated by the complexity of the reachability problem (which is still not known, all known algorithms are non-primitive recursive); however, for BWF-nets modelling real-life business processes the performance turned out to be acceptable. We have run our prototype on a series of examples. The nets were first reduced with the standard reduction rules from [12], which preserve soundness. Table 1 shows the experimental results comparing the size of Γ with the size of Υ. In most of the experiments Υ turned out to be equal to the set of rescaled generators. Our experiments showed that our tool can handle models of business processes of realistic size in reasonable time; a typical case: for a (reduced) BWF-net with $|P| = 18$ and $|T| = 22$, our algorithm checks soundness within 8 seconds.

Using reduction rules to verify soundness. We can apply our procedure in combination with reduction techniques that preserve soundness in order to reduce the size of the net for which we are checking soundness.

We start with introducing the notion of *k-closure of a BWF-net*, which is the strongly connected net obtained by adding a transition whose only input place is the final place, the only output place is the initial place, and the weights of the arcs equal k.

Table 1. Experimental results

Net	Soundness	$\|P\|$	$\|T\|$	$\|\Gamma\|/\|\Upsilon\|$	$\max_{m\in\Gamma} w(m)$	$\max_{m\in\Upsilon} w(m)$	$\|\Upsilon\|$	Time(ms)
1	sound	23	27	19	75	1	75 $(= \|E_{\mathcal{G}}\|)$	19909
2	sound	18	22	11	70	1	70 $(= \|E_{\mathcal{G}}\|)$	8005
3	sound	12	12	46	14	1	14 $(= \|E_{\mathcal{G}}\|)$	131
4	sound	9	10	57	9	1	9 $(= \|E_{\mathcal{G}}\|)$	16
5	sound	9	9	18	10	1	10 $(= \|E_{\mathcal{G}}\|)$	26
6	sound	7	8	18	8	2	7 $(= \|E_{\mathcal{G}}\|)$	9
7	sound	9	6	8	11	1	11 $(= \|E_{\mathcal{G}}\|)$	48
8	sound	6	6	6	6	1	6 $(= \|E_{\mathcal{G}}\|)$	9
9	sound	7	5	5	6	1	6 $(= \|E_{\mathcal{G}}\|)$	5
10	not 2-sound	5	6	6	5	1	5 $(= \|E_{\mathcal{G}}\|)$	5
11	sound	5	5	8	5	1	5	7
12	not 2-sound	4	3	7	6	2	6	8

Definition 13. *The* k-closure of a BWF-net $N = (P, T, F^+, F^-)$ *is a net* $(P, T \cup \{\bar{t}\}, \bar{F}^+, \bar{F}^-)$*, where* $\bar{F}^-(i,\bar{t}) = \bar{F}^+(f,\bar{t}) = k$*,* $\bar{F}^+(i,\bar{t}) = \bar{F}^-(f,\bar{t}) = 0$*,* $\bar{F}^+(p,t) = F^+(p,t)$ *and* $\bar{F}^-(p,t) = F^-(p,t)$ *for all* $(p,t) \in P \times T$*.*

Lemma 14. *The* k*-closure of a BWF-net* N *is bounded and* $\bar{t}$*-live iff* N *is* k*-sound.*

Proof. ($\Rightarrow$) Since the closure of N is $\bar{t}$-live, for all $m \in \mathcal{R}(k \cdot \bar{i})$, there exists an m' such that $m \xrightarrow{*} m' \xrightarrow{\bar{t}} m''$. Boundedness of N implies $m' = k \cdot \bar{f}$ and $m'' = k \cdot \bar{i}$. Thus, N is sound.

($\Leftarrow$) Suppose the closure of N is unbounded. Then there exists $m \in \mathcal{R}(k \cdot \bar{i})$ such that $m \xrightarrow{*} m'$ and $m < m'$. Since N is k-sound, $m \xrightarrow{*} k \cdot \bar{f}$ and $m' \xrightarrow{*} k \cdot \bar{f} + m - m'$, which contradicts soundness of N. Hence N is bounded. By k-soundness of N, for all $m \in \mathcal{R}(k \cdot \bar{i})$, $m \xrightarrow{*} k \cdot \bar{f}$. Hence, $m \xrightarrow{*} k \cdot \bar{f} \xrightarrow{\bar{t}} k \cdot \bar{i}$. Thus the closure of N is $\bar{t}$-live. □

Thus, natural candidates for preserving soundness are rules that preserve $\bar{t}$-liveness and boundedness of the closure of the net in both directions, i.e. the closure of the BWF-net is $\bar{t}$-live and bounded iff the reduced closure of the BWF-net is $\bar{t}$-live and bounded. Such rules have been intensively investigated; among them, we recall the place substitution rule and the identical transitions rule of Berthelot [5] and the reduction rules Murata [12] (fusion of series places/transitions, fusion of parallel places/transitions, elimination of self loop transitions).

Let $\mathfrak{R}$ be a set of transformation rules between two k-closures of a BWF-net which preserve boundedness and $\bar{t}$-liveness in both directions (we also assume that $\bar{t}$, i and f are not reduced). Note that since the only initially marked place is i, the transformations from $\mathfrak{R}$ are applied to unmarked places only.

Soundness is preserved by applying rules from $\mathfrak{R}$ to the closure a BWF-net:

Theorem 15. *A BWF-net is sound iff the BWF-net obtained by applying reductions from the set* $\mathfrak{R}$ *is sound.*

Proof. By Lemma 14 soundness of a BWF-net is equivalent to the boundedness and $\bar{t}$-liveness of the k-closure of the BWF-net, for all $k \in \mathbb{N}$. The latter is equivalent to the boundedness and $\bar{t}$-liveness of the k-closure of the reduced BWF-net, for all $k \in \mathbb{N}$. By applying again Lemma 14, this is equivalent to the soundness of the reduced BWF-net. □

Compositional verification of soundness. In practice it is often needed to verify soundness of large workflow nets that cannot be handled by current verification tools. Therefore, a more efficient approach is needed to handle these cases. Applying simple reduction rules that preserve soundness, like the ones from [12], facilitates the task a lot. The reduced net can then be checked using a compositional approach:

1. Identify BWF-subnets in the original workflow by using classical graph techniques (e.g. by detecting strongly connected components).
2. Check whether the found BWF-subnets are generalized sound using the procedure described.
3. Reduce every sound BWF-subnet to one place and repeat the procedure iteratively, till the soundness of the whole net is determined.

Correctness of the reduction part of Step 3 is justified by Theorem 6 from [9].

5 Conclusion and Future Work

In this paper, we have presented an improved procedure for deciding generalized soundness of BWF-nets. We showed that the problem reduces to checking proper termination for a set of *minimal markings* from the set found in [10], which significantly reduces the number of markings for which proper termination has to be checked. Further, we described a backwards reachability algorithm for checking proper termination for the found set of markings.

As discussed in Section 4, soundness of workflow nets can be checked in a compositional way. In addition to that, our soundness check can be used for compositional verification of Petri net properties. By adapting the proof of Theorem 6 from [9], it is easy to prove that if a Petri net has a subnet which is a generalized sound net whose transitions are labelled by invisible labels, the net obtained by reducing this subnet to one place is branching bisimilar to the original net. For future work, we are interested in the verification of temporal logic properties of Petri nets (not necessarily WF-nets) with using such a reduction technique.

The idea can also be applied to build sound by construction nets in a hierarchical way similarly to Vogler's refinement by modules [18,19].

References

1. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

3. R. Bagnara, P. Hill, and E. Zaffanela. *The Parma Polyhedra Library users manual.* Department of Mathematics, University of Parma,Italy. `www.cs.unipr.it/ppl/Documentation`.
4. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *SAS*, volume 2477 of *LNCS*, pages 213–229, 2002.
5. G. Berthelot. *Verification de Reseaux de Petri.* PhD thesis, Universite Pierre et Marie Curie (Paris), 1978.
6. F. Commoner. *Deadlocks in Petri Nets.* Applied Data Research, Inc., Wakefield, Massachusetts, Report CA-7206-2311, 1972.
7. D. de Frutos Escrig and C. Johnen. Decidability of home space property. Technical report, Univ. de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique Report LRI–503, July 1989.
8. J. Desel and J. Esparza. *Free Choice Petri nets.*, volume 40 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1995.
9. K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In *Proc. of ICATPN'2003*, volume 2679 of *LNCS*, pages 337–356, 2003.
10. K. van Hee, N. Sidorova, and M. Voorhoeve. Generalized soundness of workflow nets is decidable. In *Proc. of ICATPN'2004*, volume 3099 of *LNCS*, pages 197–216, 2004.
11. E. W. Mayr. An algorithm for the general Petri net reachability problem. In *Conference Proceedings of the 13th Annual ACM Symposium on Theory of Computation, STOC'1981*, pages 238–246. ACM, 1981.
12. T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 7(4), pages 541–580, 1989.
13. L. Ping, H. Hao, and L. Jian. On 1-soundness and soundness of workflow nets. In *Third Workshop on Modelling of Objects, Components, and Agents Aarhus, Denmark, October 11-13, 2004*, pages 21–36, 2004.
14. R. Post. YASPER Petri net editor. Department of Mathematics and Computer Science, Technical University Eindhoven, The Netherlands. `www.yasper.org`.
15. C. Reutenauer. *The mathematics of Petri nets.* Prentice-Hall, Inc., 1990.
16. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience series in discrete mathematics. John Wiley & Sons, 1986.
17. F. Tiplea and D. Marinescu. Structural soundness for workflow nets is decidable. *Information Processing Letters*, 96(2):54–58, 2005.
18. W. Vogler. Behaviour preserving refinement of Petri nets. In *WG*, volume 246 of *LNCS*, pages 82–93. Springer, 1986.
19. W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *LNCS.* Springer-Verlag, 1992.

Specifying Monogenetic Specializers by Means of a Relation Between Source and Residual Programs[⋆]

Andrei Klimov

Keldysh Institute for Applied Mathematics,
Russian Academy of Sciences,
4 Miusskaya sq., Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. A specification of a class of specializers for a tiny functional language in form of natural semantics inference rules is presented. The specification defines a relation between source and residual programs with respect to an initial configuration (a set of input data). The specification expresses the idea of *what* is to be a specialized program, avoiding where possible the details of *how* a specializer builds it. In particular, it abstracts from the difference between on-line and off-line specialization.

The class of specializers specified here is limited to *monogenetic* specializers, which includes specializers based upon partial evaluation as well as restricted supercompilation. The specification captures such supercompilation notions as *configuration*, *driving*, *generalization of a configuration*, and a simple case of *splitting a configuration*.

The proposed specification is an *intensional* definition of equivalence between source and residual programs. It provides a shorter cut for proving the correctness and other properties of specializers than usual reduction to the *extensional* equivalence of programs does.

Keywords: specialization, partial evaluation, supercompilation, driving, specification, natural semantics, correctness, program equivalence.

1 Introduction

Program specialization is an equivalence transformation. A specializer *spec* maps a source program p to a residual program q, which is equivalent to p on a given subset D of the domain of the program p: $q = spec(p, D)$. The equivalence of the source and residual programs is understood *extensionally*, that is, nonconstructively: $p \approx_D q$ if for all $d \in D : p(d) = q(d)$ or both $p(d)$ and $q(d)$ do not terminate. In this paper we define a constructive, *intensional* relation of specialization, that is, a relation of equivalence of source and residual programs, which many specialization methods satisfy, including partial evaluation [4], and

[⋆] Supported by Russian Foundation for Basic Research grant No. 06-01-00574.

$k \in Atom$	atomic data
$x \in Data$	ground data
$y \in CData$	configuration data
$a \in Arg$	source arguments
$b \in Prim$	source primitives
$s \in Term$	source program terms
$r \in Term$	residual program terms
$v \in Var$	source program variables
$u \in Var$	residual program variables and configuration variables
$f \in FName$	function names
$p \in Prog$	source programs
$q \in Prog$	residual programs
$c \in Contr$	contractions
$b, s \in MConf$	(monogenetic) configurations

$$\begin{aligned}
k &::= \text{True} \mid \text{False} \mid \text{Nil} \mid \dots \\
x &::= k \mid \text{Cons}\ x\ x \\
y &::= k \mid \text{Cons}\ y\ y \mid u \\
a &::= y \mid \text{Cons}\ a\ a \mid v \\
b &::= a \mid \textbf{fst}\ v \mid \textbf{snd}\ v \\
&\quad\ \mid \textbf{cons}?\ v \mid \textbf{equ}?\ v\ a \\
s &::= b \\
&\quad\ \mid \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \\
&\quad\ \mid \textbf{let}\ v = s_1\ \textbf{in}\ s_2 \\
&\quad\ \mid \textbf{call}\ f\ \{v \mapsto a, \dots\}
\end{aligned}$$

$$\begin{aligned}
Prog &= FName \to Term \\
Args &= Var \to Arg \\
Contr &= Var \to CData \\
MConf &= Term
\end{aligned}$$

Fig. 1. Object language syntax and semantic domains

restricted (monogenetic) supercompilation [11,12]. The relation is defined by inference rules in style of natural semantics [5], and serves as a specification of a class of specializers.

For brevity's sake, in this paper we limit ourselves to *monogenetic* specialization [7] where any program point in the residual program is produced from a single program point of the source program. Generalization of the specification to the *polygenetic* case where a residual program point is produced from one or several source program points, is rather straightforward and will be presented elsewhere.

The specification is built upon the ideas of supercompilation. The inference rules model at an abstract level the operational behavior of supercompilers.

2 Basic Notions

2.1 Object Language and Semantic Domains

Figure 1 contains the definition of the abstract syntax of the object language together with semantic domains for interpretation and specialization. It is a tiny first-order functional language with call-by-value semantics. It has conventional control constructs **if-then-else**, **let-in**, **call**, adjusted a bit to make the inference rules simpler. Figure 2 shows an example of a program and an initial configuration for specialization.

Data. A data domain *Data* is a *constructor-based* domain recursively defined from a set of atoms *Atom* by applying a binary constructor Cons. The set *Atom* contains at least True, False, and Nil.

Any constructor-based domain has the nice property that it can be easily extended to meta-data without the need for encoding. In particular, a constant

```
rev  v1            =  loop v1 []                    — a program in Haskell
loop [] v2         =  v2
loop (v4 : v5) v2  =  loop v5 (v4 : v2)
```

```
p  = { rev   ↦  call loop {v1 ↦ v1, v2 ↦ Nil},         — the same program in
       loop  ↦  let v3 = cons? v1 in                       the object language
                if v3
                then let v4 = fst v1 in
                     let v5 = snd v1 in
                     call loop {v1 ↦ v5, v2 ↦ Cons v4 v2}
                else v2
     }

s0 = call rev {v1 ↦ Cons A (Cons u1 (Cons B u2))}     — an initial configuration
```

Fig. 2. An example of a program p in the object language and an initial configuration s_0

in program code coincides with the value it represents. That is, $\mathit{Data} \subset \mathit{Term}$, where *Term* is a domain of program terms.

Configuration data. The second extension of *Data* originates from the need to constructively represent sets of data values, sets of program states, in specializers. The basic method to represent sets is to embed free variables into data. For a constructor-based domain, position of any constructor argument may be replaced with a variable, $u \in \mathit{Var}$, which is referred to as a *configuration variable*. That is, $\mathit{Data} \subset \mathit{CData}$, where *CData* is the domain of *configuration values*.

A characteristic feature of supercompilation, which is preserved in our specification, is that configuration variables become residual program variables.

Primitives. Data values are analyzed by primitive predicates **equ**? v a (are two values equal?) and **cons**? v (is the value of a variable v a term of form Cons y_1 y_2?), which return atoms True or False, and selectors **fst** v and **snd** v, which require the value of v to be a Cons term and return its first and second argument respectively.

For the sake of specification simplicity, the arguments of primitives are syntactically restricted to variables v or arguments a, which do not contain terms of general form. We also impose two restrictions on the usage of selectors **fst** v and **snd** v: they must occur, firstly, only in a **let** binding, *e.g.*, **let** $v_1 =$ **fst** v_2 **in** s, and, secondly, only after a check of form **cons**? v on the positive branch to avoid dealing with exceptions in the specification.

Control. Control terms **if-then-else** and **let-in** are the regular conditional term and **let** binding respectively; only notice the restriction of the conditional to variable v for simplicity's sake. Additionally, we require the variable v to be bound to a conditional primitive **equ?** or **cons?** by an enclosing **let** term.

A function call, which usually looks like $f(a_1, \ldots, a_n)$, is coded in our language as **call** $f\,\{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}$, where $v_1, \ldots v_n$ are the local variable names occurring in the term that the name f is bound to in a program.

A program is a finite mapping of function names to program terms.

Notation. Vars(t) denotes the set of variables occurring in term t. Dom(m) and Rng(m) denote the domain and range of mapping m.

2.2 Configuration

While an interpreter runs a program on a particular data, a specializer runs a source program on a set of data. A representation of a program state in interpretation and that of a set of states in specialization is referred to as a *configuration*. We follow the general rule to construct the notion of the specializer configuration from that of an interpreter configuration that reads as follows: add configuration variables to the data domain, and allow the variables to occur anywhere where an ordinary ground value can occur. A configuration represents the set that is obtained by replacing all configuration variables with all possible values.

For the purpose of the definition of monogenetic specialization, a *configuration* is a source program term, in which program variables are replaced with their values.[1]

2.3 Substitution

To avoid the ambiguity of traditional postfix notation for substitution $t\theta$ when used in inference rules, we lift up the substitution symbol θ and use a kind of power notation t^θ.

Thus, t^θ denotes the replacement of all occurrences of variables $v \in \text{Dom}(\theta)$ in t with their values from a binding θ. Notation $t^{\eta\theta}$ means sequential application of substitutions η and θ to t in that order. When the substitution argument is unclear, it is over-lined, *e.g.*, $a\ \overline{b\ c}^{\theta} d$.

The bindings listed in Fig. 1 are used as substitutions as follows:

– f^p gets the term bound to a function name f in a program p;
– $s^a = f^{pa}$ builds a configuration from a program term $s = f^a$ and the argument binding a of a configuration **call** $f\ a$;
– s^c *contracts* a configuration s by replacing configuration variable u with the configuration value y bound to u by a *contraction* $c = \{u \mapsto y\}$.

[1] An alternative is to keep program terms untouched and to represent the configuration as a pair consisting of a program term and an environment that binds program variables to their values. Although this representation is more common in implementations of interpreters and specializers, we prefer to substitute the environment into the term for conciseness of inference rules.

2.4 Contraction

After evaluation of a conditional, the current configuration divides into two subconfigurations, the initial configurations of the positive and negative branches. In our specification the subconfigurations represent the subsets precisely, that is, driving is *perfect* [2].

There are two Boolean primitives, **equ**? $v\ a$ and **cons**? v, in the object language. After substitution of configuration values into the arguments of the primitives, they ultimately reduce (by rules in Fig. 5) to the following checks on configuration variables that produce branching in residual code: **equ**? $u\ k$, **equ**? $u\ u'$, and **cons**? u, where k is an atom, u and u' configuration variables.

For each of the primitives, the set of values of u that goes to the positive branch can be represented by a respective substitution $\{u \mapsto k\}$, $\{u \mapsto u'\}$, and $\{u \mapsto \text{Cons } u_1\ u_2\}$, where u_1 and u_2 are new configuration variables. Such a substitution is referred to as a *contraction*. Being applied to a configuration, it produces a configuration representing a subset of the original one.

For uniformity's sake, we represent "negative" information by substitutions as well. To achieve this, we assume the representation of a configuration variable contains a *negative set*: a set of "negative entities" the variable must be *unequal to*. The negative entities are atoms, configuration variables, and a word Cons representing inequality to all terms of form Cons $y_1\ y_2$.

We denote the operation to add an entity n to the negative set of a configuration variable u by u^{-n}. Thus, the following substitutions are *negative contractions*: $\{u \mapsto u^{-k}\}$, $\{u \mapsto u^{-u'}\}$, and $\{u \mapsto u^{-\text{Cons}}\}$.

3 Specification of Semantics and Specialization

Figure 3 sums up the judgments used in the specification. Figures 4–6 contain the inference rules of the specification. The rules marked by an asterisk define the interpretation semantics of the object language. The unmarked rules extend it to specialization.

3.1 Interpretation of Primitives

A judgment of form $b \to y$ asserts that interpretation of a primitive configuration b (*i.e.*, a source program primitive, in which program variables have been replaced with their values) produces a value y.

The interpretation rules also define *transient driving* that is the basic case of driving where configuration variables do not prevent a specializer from unambiguously performing a step. In this case, b and y may contain configuration variables.

The unmarked rules in Fig. 4 and 5 that infer judgments of form $b \to y$, define the cases of transient driving that are not covered by the interpretation rules.

$b \to y$	**Interpretation** (or transient driving) of a primitive b produces a (configuration) value y.
$b \prec \mathcal{T}(c_1, c_2)$	**Driving with branching**: Driving of a primitive b produces a branching represented by a residual conditional term $\mathcal{T}(\_, \_)$ with two free positions for positive and negative branches. In the right-hand side $\mathcal{T}(c_1, c_2)$ these positions are occupied by contractions c_1 and c_2. The contractions being substituted to the configuration before the branching produce the initial configurations of the branches.
$m \vdash p : s \Rightarrow q : r$	**Monogenetic specialization**: A residual program q with an initial term r is a specialization of a source program p with an initial term s with respect to a mapping m of residual function names to configurations.
$\{\} \vdash p : s \Rightarrow \{\} : r$	**Interpretation**: In the case where the residual program is empty and hence $m = \{\}$ and $q = \{\}$ the previous judgement denotes interpretation.

Fig. 3. Judgments

I-VALUE*	$y \ \to\ y$	
I-FST*	$\textbf{fst}\ (\text{Cons}\ y_1\ y_2) \ \to\ y_1$	
I-SND*	$\textbf{snd}\ (\text{Cons}\ y_1\ y_2) \ \to\ y_2$	
I-CONS-T*	$\textbf{cons}?\ (\text{Cons}\ y_1\ y_2) \ \to\ \text{True}$	
I-CONS-F*	$\textbf{cons}?\ k \ \to\ \text{False}$	
I-EQ-T*	$\textbf{equ}?\ y\ y \ \to\ \text{True}$	
I-EQ-F*	$\textbf{equ}?\ x_1\ x_2 \ \to\ \text{False}$	if $x_1 \neq x_2$
D-EQ-CK	$\textbf{equ}?\ (\text{Cons}\ y_1\ y_2)\ k \ \to\ \text{False}$	
D-EQ-CC	$\dfrac{\textbf{equ}?\ y_{1i}\ y_{2i} \ \to\ \text{False}}{\textbf{equ}?\ (\text{Cons}\ y_{11}\ y_{12})\ (\text{Cons}\ y_{21}\ y_{22}) \ \to\ \text{False}}$	$i \in \{1, 2\}$

Fig. 4. Interpretation and transient driving of primitives

D-CONS-F	$\textbf{cons?}\ u \ \rightarrow\ \text{False}$	$u = u^{-\text{Cons}}$
D-CONS	$\begin{array}{ll} \textbf{cons?}\ u \ \prec & \textbf{let}\ u_0 = \textbf{cons?}\ u\ \textbf{in} \\ & \textbf{if}\ u_0 \\ & \textbf{then let}\ u_1 = \textbf{fst}\ u\ \textbf{in} \\ & \qquad \textbf{let}\ u_2 = \textbf{snd}\ u\ \textbf{in} \\ & \qquad \{u \mapsto \text{Cons}\ u_1\ u_2\} \\ & \textbf{else}\ \ \{u \mapsto u^{-\text{Cons}}\} \end{array}$	$u \neq u^{-\text{Cons}}$ $u_0,\ u_1,\ u_2$ new
D-EQ-COM	$\dfrac{\textbf{equ?}\ u\ y \ \rightarrow\ \mathcal{T}}{\textbf{equ?}\ y\ u \ \rightarrow\ \mathcal{T}}$	
D-EQ-UKF	$\textbf{equ?}\ u\ k \ \rightarrow\ \text{False}$	$u = u^{-k}$
D-EQ-UK	$\begin{array}{ll} \textbf{equ?}\ u\ k \ \prec & \textbf{let}\ u_0 = \textbf{equ?}\ u\ k\ \textbf{in} \\ & \textbf{if}\ u_0 \\ & \textbf{then}\ \{u \mapsto k\} \\ & \textbf{else}\ \ \{u \mapsto u^{-k}\} \end{array}$	$u \neq u^{-k}$ u_0 new
D-EQ-UUF	$\textbf{equ?}\ u_1\ u_2 \ \rightarrow\ \text{False}$	$u_1 = u_1^{-u_2}$
D-EQ-UU	$\begin{array}{ll} \textbf{equ?}\ u_1\ u_2 \ \prec & \textbf{let}\ u_0 = \textbf{equ?}\ u_1\ u_2\ \textbf{in} \\ & \textbf{if}\ u_0 \\ & \textbf{then}\ \{u_1 \mapsto u_2\} \\ & \textbf{else}\ \ \{u_1 \mapsto u_1^{-u_2},\ u_2 \mapsto u_2^{-u_1}\} \end{array}$	$u_1 \neq u_1^{-u_2}$ u_0 new
D-EQ-UCF	$\textbf{equ?}\ u\ (\text{Cons}\ y_1\ y_2) \ \rightarrow\ \text{False}$	$u = u^{-\text{Cons}}$ or $u \in \text{Vars}(y_1)$ or $u \in \text{Vars}(y_2)$
D-EQ-UC	$\dfrac{\textbf{cons?}\ u \ \prec\ \mathcal{T}}{\textbf{equ?}\ u\ (\text{Cons}\ y_1\ y_2) \ \prec\ \mathcal{T}}$	$u \neq u^{-\text{Cons}}$ $u \notin \text{Vars}(y_1)$ $u \notin \text{Vars}(y_2)$

Fig. 5. Driving of primitives

3.2 Branching

A judgment of form $b \prec \mathcal{T}(c_1, c_2)$ means *driving* of a primitive b produces a branching in residual code represented by a conditional term $\mathcal{T}(\_, \_)$ with two free positions for positive and negative branches and two contractions c_1 and c_2.

The contractions c_1 and c_2, being applied as substitutions to the configuration b, divide it into two subconfigurations, which are initial configurations for positive and negative branches respectively.

For the sake of notation simplicity, the contractions c_1 and c_2 are placed in $\mathcal{T}(\_, \_)$ in the positions where the residual terms for the positive and negative branches will occur in the final residual code. Notice the abstract syntax in Fig. 1 does not take account of such use case of terms and substitutions, since it does not concern program code and is used only locally in judgments of the form $b \prec \mathcal{T}(c_1, c_2)$. Theoretically, this judgment can be excluded at the expense of multiple instantiations of rule MS-FORK where it is used.

The following cases of $\mathcal{T}(c_1, c_2)$ are used in the specification:

```
1. if equ? $u$ $k$ then $\{u \mapsto k\}$ else $\{u \mapsto u^{-k}\}$
2. if equ? $u_1$ $u_2$ then $\{u_1 \mapsto u_2\}$ else $\{u_1 \mapsto u_1^{-u_2}\}$
3. let $u_0$ = equ? $u_1$ $u_2$ in
   if $u_0$
   then let $u_1$ = fst $u$ in
          let $u_2$ = snd $u$ in
          $\{u \mapsto \text{Cons}\ u_1\ u_2\}$
   else  $\{u \mapsto u^{-\text{Cons}}\}$
```

In each of the cases term $\mathcal{T}(\_, \_)$ meets the following property: the value of $\mathcal{T}(x_1, x_2)$ is either x_1 for the configuration obtained by contraction c_1, or x_2 for the configuration obtained by contraction c_2. The correctness of rule MS-FORK relies on this property.

3.3 Specialization

Judgments of form $m \vdash p : s \Rightarrow q : r$ inferred by rules in Fig. 6 assert that a residual program q with an initial term r is a specialization of a source program p with an initial term s provided a mapping m establishes a correct correspondence between source and residual programs p and q. The semantics of the judgment implies that for all values of configuration variables in the configuration s and in the term r, the evaluation of the source program p starting from the configuration s and the evaluation of the term r with the residual program q gives the same result.

Mapping of residual functions to configurations. The initial state of each residual function f is equivalent to some configuration expressed in terms of the source program. The mapping $m : \text{Dom}(q) \to \mathit{MConf}$ keeps this correspondence. It must be consistent with programs p and q. Only judgments with a consistent mapping m define correct specialization. The formal definition of the consistency is as follows.

Definition 1. *A mapping $m : \text{Dom}(q) \to \mathit{MConf}$ of residual function names to configurations is* consistent with source and residual programs *p and q if for every residual function name $f \in \text{Dom}(q)$ the following judgment is deducible:*

$$m \vdash p : f^m \Rightarrow q : f^q.$$

MS-PRIM* $$\frac{b \;\rightarrow\; y}{m \vdash p : b \;\Rightarrow\; q : y}$$

MS-FORK $$\frac{\begin{array}{r} b \;\prec\; \mathcal{T}(c_1, c_2) \\ m \vdash p : \overline{\textbf{let } v = b \textbf{ in } s}^{c_1} \Rightarrow\; q : r_1 \\ m \vdash p : \overline{\textbf{let } v = b \textbf{ in } s}^{c_2} \Rightarrow\; q : r_2 \end{array}}{m \vdash p : \textbf{let } v = b \textbf{ in } s \;\Rightarrow\; q : \mathcal{T}(r_1, r_2)}$$

MS-IF-T* $$\frac{m \vdash p : s_1 \;\Rightarrow\; q : r}{m \vdash p : \textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2 \;\Rightarrow\; q : r} \qquad a = \text{True}$$

MS-IF-F* $$\frac{m \vdash p : s_2 \;\Rightarrow\; q : r}{m \vdash p : \textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2 \;\Rightarrow\; q : r} \qquad a = \text{False}$$

MS-LET* $$\frac{\begin{array}{r} m \vdash p : s_1 \;\Rightarrow\; q : y \\ m \vdash p : s_2^{\{v \mapsto y\}} \;\Rightarrow\; q : r \end{array}}{m \vdash p : \textbf{let } v = s_1 \textbf{ in } s_2 \;\Rightarrow\; q : r}$$

MS-LET-SPLIT $$\frac{\begin{array}{r} m \vdash p : s_1 \;\Rightarrow\; q : r_1 \\ m \vdash p : s_2^{\{v \mapsto u\}} \;\Rightarrow\; q : r_2 \end{array}}{m \vdash p : \textbf{let } v = s_1 \textbf{ in } s_2 \;\Rightarrow\; q : \textbf{let } u = r_1 \textbf{ in } r_2} \qquad \begin{array}{l} r_1 \notin \mathit{CData} \\ u \text{ new} \end{array}$$

MS-CALL* $$\frac{m \vdash p : f^{pa} \;\Rightarrow\; q : r}{m \vdash p : \textbf{call } f\ a \;\Rightarrow\; q : r} \qquad \text{Vars}(f^p) \subseteq \text{Dom}(a)$$

MS-GEN $$m \vdash p : f^{ma} \;\Rightarrow\; q : \textbf{call } f\ a$$

Fig. 6. Monogenetic specialization

The reason for the introduction of an auxiliary mapping m is that the inference of a specialization judgment is actually a proof by induction of the equivalence of source and residual programs with respect to initial terms, a mapping m being an induction hypothesis and the definition of consistency describing the induction step.

Semantics. The rules marked by an asterisk define usual natural semantics for the object language when residual program q is empty, $q = \{\}$ and $m = \{\}$, and transient driving in general case.

The unmarked rules extend the interpretation semantics to specialization. Notice the correctness of the specification is to be proved with respect to the semantics, that is, the full set of rules is to be correct with respect to the part.

Fork. Rules MS-IF-T* and MS-IF-F* define the interpretation of the **if** term. The conditional a is compared to True or False without evaluation due to the syntactic restriction to arguments consisting of constructors and variables only (see Fig. 1).

Rule MS-FORK uses the result of driving of a Boolean primitive to build a branching in residual code explained in Section 3.2 above.

Notice the absence of a rule for the case of the **if** term where the conditional a is a configuration variable. It is useless due to the syntactic restriction on the conditional introduced in Section 2.1.

Let. Rule MS-LET* defines the call-by-value interpretation of the **let** term.

Rule MS-LET-SPLIT expresses a simple case of splitting a configuration: a **let** configuration splits into two ones, which are specialized separately, and are then composed into the residual **let** term.

Call. Rule MS-CALL* defines unfolding of a function call. Notation f^{pa} means extraction of the body f^p of function f from the program p and substitution of arguments supplied in the argument binding a.

Generalization. Rule MS-GEN defines the notion of generalization of a configuration together with folding into a residual **call** term. Reading the judgment $m \vdash p : f^{ma} \Rightarrow q : \textbf{call}\ f\ a$ from left to right along the algorithm of supercompilation, we say that some configuration $s_1 = f^{ma}$ is decomposed into a configuration s_2 and a substitution a such that $s_1 = s_2^a$. The act of such decomposition is referred to as generalization. Then the configuration s_2 is bound to residual function f in the mapping m or checked that it is there already, and the term **call** $f\ a$ is residualized.

This rule uses the mapping m as an induction hypothesis.

3.4 Correctness

Since the semantics of the object language is represented by a part of the inference rules, the correctness of the specification with respect to the semantics is its internal property that can be expressed as follows.

Definition 2. *A value x is* ground *if it does not contain configuration variables, $x \in Data$. A contraction c is* ground *if it maps configuration variables to ground values,* $\mathrm{Rng}(c) \subseteq Data$.

Proposition 1. *For all m, p, s, q, r such that*

$$m \vdash p : s \Rightarrow q : r$$

it holds that for all ground contractions c and ground values x the following judgments are both deducible or both non-deducible:

$$\{\} \vdash p : s^c \Rightarrow \{\} : x,$$
$$\{\} \vdash q : r^c \Rightarrow \{\} : x.$$

The last two judgments mean interpretation of a source and residual programs p and q with values supplied to the initial terms s and r respectively by the contraction c, produces the same result x in both cases.

4 Conclusion and Related Work

This paper presents a method of specifying specializers by inference rules in style of natural semantics [5] that define a relation between source and residual programs, which partial evaluation and supercompilation (restricted to monogenetic case) obey. The specification captures the essential notions of supercompilation: *configuration*, *driving*, *generalization of a configuration*, and a simple case of *splitting a configuration*, while abstracting from algorithmic problems of when, what and how to generalize and when to terminate. It provides the basis for the correctness proofs of supercompilers and an alternative to that of partial evaluators [1,3]. By nature of natural semantics, the specification allows for automated derivation of specializers that satisfy it.

The first version of this specification was presented at the Dagstuhl Seminar on Partial Evaluation in February 1996, when only abstract [6] was published. It continues the work started in [2] and aimed at clarifying and formalizing the ideas of supercompilation.

In Turchin's original papers ([11,12] and others), the essential ideas of supercompilation and technical details of algorithms were not separated enough to give their short formal definition. Later on, several works have been done to fill this gap, *e.g.*, [2,8,9,10]. All of them formalize the function of the supercompiler, while our work is, to our knowledge, the first attempt to define an input-output relation, which specializers satisfy. The closest related work is [8,9] where the notion of the graph of configurations is formalized by inference rules that deduce the arcs of the graph.

Future work includes proving various properties of the specification including the correctness, and constructing supercompilers that obey the specification and hence are provably correct.

References

1. Charles Consel and Siau Cheng Khoo. On-line and off-line partial evaluation: semantic specifications and correctness proofs. *Journal of Functional Programming*, 5(4):461–500, October 1995.
2. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Static Analysis Symposium. Proceedings*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.

3. Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM TOPLAS*, 14(2):147–172, 1992.
4. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
5. Gilles Kahn. Natural Semantics. In F. G. Brandenburg, G. Vidal-Naquet, and W. Wirsing, editors, *STACS87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
6. Andrei V. Klimov. A specification of a class of supercompilers. In O. Danvy, R. Glück, and P. Thiemann, editors, *Draft Proceedings of the Dagstuhl Seminar on Partial Evaluation*, page 232. Technical Report WSI-96-6, Universität Tübingen, Germany, 1996.
7. Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *ESOP'90, Copenhagen, Denmark*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer-Verlag, May 1990.
8. Jens Peter Secher and Morten Heine B. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2000.
9. Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.
10. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
11. Valentin F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
12. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

Satisfiability of Viability Constraints for Pfaffian Dynamics[⋆]

Margarita Korovina[1] and Nicolai Vorobjov[2]

[1] Fachbereich Mathematik, Theoretische Informatik, Universität Siegen, Germany, and IIS SB RAS, Novosibirsk, Russia
korovina@brics.dk
http://www.brics.dk/~korovina
[2] Department of Computer Science, University of Bath, Bath BA2 7AY, England
nnv@cs.bath.ac.uk
http://www.bath.ac.uk/~masnnv

Abstract. We investigate the behavior of a Pfaffian dynamical system with respect to viability constraints and invariants. For Pfaffian dynamical systems we construct an algorithm with an elementary (doubly-exponential) upper complexity bound for checking satisfiability of viability constraints. This algorithm also provides a useful tool for checking invariance properties of given sets.

1 Introduction

In this paper we study continuous dynamical systems which are called Pfaffian, and first introduced in [9,10]. These systems are defined by Pfaffian functions, either implicitly (via triangular systems of ordinary differential equations) or explicitly (by means of equations and inequalities involving *Pfaffian functions*). Such functions naturally arise in applications as real analytic solutions of triangular first order partial differential equations with polynomial coefficients, and include polynomials, algebraic functions, exponentials, and trigonometric functions in appropriate domains [8]. Pfaffian functions form the largest natural class of real analytic functions which have a uniform description and an explicit characterisation of complexity of their representations in terms of *formats*.

One of the important problems in the theory of dynamical systems is understanding of the behavior of a dynamical system with respect to viable and invariant sets. In this paper we consider this problem for Pfaffian dynamical systems. Viability constraints and invariants naturally arise when some evolutions of a dynamical system do not satisfy the imposed requirements. These constraints include state constraints in control theory and verification of safety-critical systems, power constraints in game theory, ecological constraint in genetics, etc [1]. Therefore, the goal is to select evolutions which are viable in the sense that they satisfy these constraints at each point in time.

⋆ This research was partially supported by Grant Scientific School-4413.2006.1, Deutsche Forschungsgemeinschaft Project WE 843/17-1, and Project GZ: 436 RUS 113/850/01.

In mathematical settings this problem is formalised in the following way. We consider a continuous dynamical system $\gamma : G_1 \times T \to G_2$, where $G_1 \subseteq \mathbb{R}^{k_1}$ is a set of control parameters, T is an interval of time and $G_2 \subseteq \mathbb{R}^{k_2}$ is a state space. Let U be a set of control parameters. Viability constraints are described by subsets of the state space. A subset V is *viable* under the dynamical system γ and the control U if there exits at least one evolution of the system which is viable in the sense that $\forall t \in T \gamma_{\mathbf{x}}(t) \in V,$ where $\mathbf{x} \in U$. We say *a subset $U \subseteq G_1$ satisfies the constraint V* if V is viable under U and the dynamical system γ. In this paper we assume that dynamical systems and sets we are interested in are semi-Pfaffian. Our goal is to characterise the subsets of control parameter space which satisfy a given viability constraint. In order to achieve our goal we use encoding trajectories of a Pfaffian dynamical system by finite words [4,9] and cylindrical cell decomposition for semi-Pfaffian sets [13,5]. Based on this technique we construct an algorithm for checking satisfiability of viability constrains with an elementary exponential upper bound.

The outline of the paper is as follows. Section 1 presents a brief overview of Pfaffian functions, upper bounds on topological complexities of semi- and sub-Pfaffian sets, and algorithms for computing their closures and cylindrical cell decompositions. In Section 2 we recall the notion of Pfaffian dynamical system, viable and invariant sets. We also explain how to associate a word to a trajectory. Finally, in Section 3 we propose an algorithm (with the usual for Pfaffian functions theory oracle) for checking satisfiability of viability constrains. The complexity of the algorithm is doubly exponential in the format of an input system.

2 Basic Definitions and Notions

2.1 Pfaffian Functions and Related Sets

In this section we overview the theory of Pfaffian functions and sets definable with Pfaffian functions. The detailed exposition can be found in the survey [5].

Definition 1. *A* Pfaffian chain *of the order $r \geq 0$ and degree $\alpha \geq 1$ in an open domain $G \subset \mathbb{R}^n$ is a sequence of real analytic functions $f_1, \dots, f_r$ in G satisfying differential equations*

$$\frac{\partial f_j}{\partial x_i} = g_{ij}(\mathbf{x}, f_1(\mathbf{x}), \dots, f_j(\mathbf{x})) \tag{1}$$

for $1 \leq j \leq r$, $1 \leq i \leq n$. Here $g_{ij}(\mathbf{x}, y_1, \dots, y_j)$ are polynomials in $\mathbf{x} = (x_1, \dots, x_n, y_1, \dots, y_j)$ of degrees not exceeding α.

A function

$$f(\mathbf{x}) = P(\mathbf{x}, f_1(\mathbf{x}), \dots, f_r(\mathbf{x})),$$

where $P(\mathbf{x}, y_1, \dots, y_r)$ is a polynomial of a degree not exceeding $\beta \geq 1$, the sequence $f_1, \dots, f_r$ is a Pfaffian chain of order r and degree α, is called a Pfaffian function *of order r and degree (α, β).*

In order to illustrate the definition let us consider several examples of Pfaffian functions.

(a) Pfaffian functions of order 0 and degree $(1, \beta)$ are polynomials of degrees not exceeding β.
(b) The exponential function $f(x) = e^{ax}$ is a Pfaffian function of order 1 and degree $(1, 1)$ in $\mathbb{R}$, due to the equation $df(x) = af(x)dx$. More generally, for $i = 1, 2, \ldots, r$, let $E_i(x) := e^{E_{i-1}(x)}$, $E_0(x) = ax$. Then $E_r(x)$ is a Pfaffian function of order r and degree $(r, 1)$, since $dE_r(x) = aE_1(x) \cdots E_r(x)dx$.
(c) The function $f(x) = 1/x$ is a Pfaffian function of order 1 and degree $(2, 1)$ in the domain $\{x \in \mathbb{R} |\ x \neq 0\}$, due to the equation $df(x) = -f^2(x)dx$.
(d) The logarithmic function $f(x) = \ln(|x|)$ is a Pfaffian function of order 2 and degree $(2, 1)$ in the domain $\{x \in \mathbb{R} | x \neq 0\}$, due to equations $df(x) = g(x)dx$ and $dg(x) = -g^2(x)dx$, where $g(x) = 1/x$.
(e) The polynomial $f(x) = x^m$ can be viewed as a Pfaffian function of order 2 and degree $(2, 1)$ in the domain $\{x \in \mathbb{R} |\ x \neq 0\}$ (but not in $\mathbb{R}$), due to the equations $df(x) = mf(x)g(x)dx$ and $dg(x) = -g^2(x)dx$, where $g(x) = 1/x$. In some cases a better way to deal with x^m is to change the variable $x = e^u$ reducing this case to (b).
(f) The function $f(x) = \tan(x)$ is a Pfaffian function of order 1 and degree $(2, 1)$ in the domain $\bigcap_{k \in Z} \{x \in \mathbb{R} |\ x \neq \pi/2 + k\pi\}$, due to the equation $df(x) = (1 + f^2(x))dx$.
(g) The function $\cos(x)$ is a Pfaffian function of order 2 and degree $(2, 1)$ in the domain $\bigcap_{k \in Z} \{x \in \mathbb{R} |\ x \neq \pi + 2k\pi\}$, due to equations $\cos(x) = 2f(x) - 1$, $df(x) = -f(x)g(x)dx$, and $dg(x) = \frac{1}{2}(1 + g^2(x))dx$, where $f(x) = \cos^2(x/2)$ and $g(x) = \tan(x/2)$. Also, since $\cos(x)$ is a polynomial of degree m of $\cos(x/m)$, the function $\cos(x)$ is Pfaffian of order 2 and degree $(2, m)$ in the domain $\bigcap_{k \in Z} \{x \in \mathbb{R} | x \neq m\pi + 2km\pi\}$. The same is true, of course, for any shift of this domain by a multiple of π. However, $\cos(x)$ is not a Pfaffian function in the whole real line.

As we can see, apart from polynomials, the class of Pfaffian functions includes real algebraic functions, exponentials, logarithms, trigonometric functions, their compositions, and other major transcendental functions in appropriate domains (see [5,6]). Now we introduce classes of sets definable with Pfaffian functions. In the case of polynomials they reduce to *semialgebraic* sets whose quantitative and algorithmic theory is treated in [2].

Definition 2. *A set $X \subset \mathbb{R}^n$ is called* semi-Pfaffian *in an open domain $G \subset \mathbb{R}^n$ if it consists of the points in G satisfying a Boolean combination of some atomic equations and inequalities $f = 0, g > 0$, where f, g are Pfaffian functions having a common Pfaffian chain defined in G. A semi-Pfaffian set X is* restricted *in G if its topological closure lies in G.*

Definition 3. *A set $X \subset \mathbb{R}^n$ is called* sub-Pfaffian *in an open domain $G \subset \mathbb{R}^n$ if it is the image of a semi-Pfaffian set under a projection into a subspace.*

It is worth noting that according to the Tarski-Seidenberg Theorem, the projection of a semialgebraic set is again semialgebraic.

In the sequel we will be dealing with the following subclass of sub-Pfaffian sets.

Definition 4. *Suppose $\bar{I} \subset \mathbb{R}$ is a closed interval. Consider the closed cube $\bar{I}^{m+n}$ in an open domain $G \subset \mathbb{R}^{m+n}$ and the projection map $\pi : \mathbb{R}^{m+n} \to \mathbb{R}^n$. A subset $Y \subset \bar{I}^n$ is called restricted sub-Pfaffian if $Y = \pi(X)$ for a restricted semi-Pfaffian set $X \subset \bar{I}^{m+n}$.*

Note that a restricted sub-Pfaffian set need not be semi-Pfaffian.

Definition 5. *Consider a semi-Pfaffian set*

$$X := \bigcup_{1 \le i \le M} \{\mathbf{x} \in \mathbb{R}^n |\, f_{i1} = 0, \dots, f_{il_i} = 0, g_{i1} > 0, \dots, g_{ij_i} > 0\} \subset G, \tag{2}$$

where f_{is}, g_{is} are Pfaffian functions with a common Pfaffian chain of order r and degree (α, β), defined in an open domain G. Its format *is the tuple (r, N, α, β, n), where $N \ge \sum_{1 \le i \le M} (l_i + j_i)$. For $n = m + k$ and a sub-Pfaffian set $Y \subset \mathbb{R}^k$ such that $Y = \pi(X)$, its* format *is the format of X.*

We will refer to the representation of a semi-Pfaffian set in the form (2) as to the *disjunctive normal form (DNF)*.

Remark 1. In this paper we are concerned with complexities of computations, as functions of the format. In the case of Pfaffian dynamical systems these sizes and complexities also depend on the domain G. So far our definitions imposed no restrictions on an open set G, thus allowing it to be arbitrarily complex and to induce this complexity on the corresponding semi- and sub-Pfaffian sets. To avoid this we will always assume in the context of Pfaffian dynamical systems that G is "simple", like $\mathbb{R}^n$, or I^n for open $I \subseteq \mathbb{R}$.

Remark 2. In this paper we construct and examine complexities of algorithms for checking satisfiability of viability constraints. In order to estimate the "efficiency" of a computation we need to specify more precisely a *model of computation.* As such we use a *real number machine* which is an analogy of a classical Turing machine but allows the exact arithmetic and comparisons on the real numbers. Since we are interested only in upper complexity bounds for algorithms, there is no need for a formal definition of this model of computation (it can be found in [3]). In some of our computational problems we will need to modify the standard real number machine by equipping it with an *oracle* for deciding feasibility of any system of Pfaffian equations and inequalities. An oracle is a subroutine which can be used by a given algorithm any time the latter needs to check feasibility. We assume that this procedure always gives a correct answer ("true" or "false") though we do not specify how it actually works. An *elementary step* of a real number machine is either an arithmetic operation, or a comparison (branching) operation, or an oracle call. The *complexity* of a real number machine is the number of elementary steps it makes in the worst case until termination, as a function of the format of the input.

In the special case of semialgebraic sets, the oracle can be replaced by a proper real number machine, so the algorithm for checking of satisfiability of viability constraints can be realized as a standard real number machine.

2.2 Cylindrical Cell Decompositions

Now we define cylindrical decompositions of semi- and sub-Pfaffian sets in a cube $\bar{I}^n$, where $\bar{I}$ is a closed interval.

Definition 6. *A* cylindrical cell *in $\bar{I}^n$ is defined by induction as follows.*

1. *A cylindrical 0-cell in $\bar{I}^n$ is an isolated point.*
2. *A cylindrical 1-cell in $\bar{I}$ is an open interval $(a, b) \subset \bar{I}$.*
3. *For $n \geq 2$ and $0 \leq k < n$ a cylindrical $(k+1)$-cell in $\bar{I}^n$ is either a graph of a continuous bounded function $f : C \to \mathbb{R}$, where C is a cylindrical $(k+1)$-cell in $\bar{I}^{n-1}$ and $k < n-1$, or else a set of the form*
$$\{(x_1, \ldots, x_n) \in \bar{I}^n |\, (x_1, \ldots, x_{n-1}) \in C \text{ and }$$
$$f(x_1, \ldots, x_{n-1}) < x_n < g(x_1, \ldots, x_{n-1})\},$$
where C is a cylindrical k-cell in $\bar{I}^{n-1}$, and $f, g : C \to \bar{I}$ are continuous bounded functions such that $f(x_1, \ldots, x_{n-1}) < g(x_1, \ldots, x_{n-1})$ for all points $(x_1, \ldots, x_{n-1}) \in C$.

Definition 7. *A* cylindrical cell decomposition *$\mathcal{D}$ of a subset $A \subset \bar{I}^n$ with respect to the variables $x_1, \ldots, x_n$ is defined by induction as follows.*

1. *If $n = 1$, then $\mathcal{D}$ is a finite family of pair-wise disjoint cylindrical cells (i.e., isolated points and intervals) whose union is A.*
2. *If $n \geq 2$, then $\mathcal{D}$ is a finite family of pair-wise disjoint cylindrical cells in $\bar{I}^n$ whose union is A and there is a cylindrical cell decomposition of $\pi(A)$ such that $\pi(C)$ is its cell for each $C \in \mathcal{D}$, where $\pi : \mathbb{R}^n \to \mathbb{R}^{n-1}$ is the projection map onto the coordinate subspace of $x_1, \ldots, x_{n-1}$.*

Definition 8. *Let $B \subset A \subset \bar{I}^n$ and $\mathcal{D}$ be a cylindrical cell decomposition of A. Then $\mathcal{D}$ is* compatible *with B if for any $C \in \mathcal{D}$ we have either $C \subset B$ or $C \cap B = \emptyset$ (i.e., some subset $\mathcal{D}' \subset \mathcal{D}$ is a cylindrical cell decomposition of B).*

Definition 9. *For a given finite family $f_1, \ldots, f_N$ of Pfaffian functions in an open domain G we define its* consistent sign assignment *as a non-empty semi-Pfaffian set in G of the kind*
$$\{\mathbf{x} \in G \,|\, f_{i_1} = 0, \ldots, f_{i_{N_1}} = 0, f_{i_{N_1}+1} > 0 \ldots, f_{i_{N_2}} > 0, f_{i_{N_2}+1} < 0, \ldots, f_{i_N} < 0\},$$
where $i_1, \ldots, i_{N_1}, \ldots, i_{N_2}, \ldots, i_N$ is a permutation of $1, \ldots, N$.

Theorem 1. [6,12] *Let $f_1, \dots, f_N$ be a family of Pfaffian functions in an open domain $G \subset \mathbb{R}^n$, $G \supset \bar{I}^n$ having a common Pfaffian chain of order r, and degrees (α, β). Then there is an algorithm (with the oracle) producing a cylindrical cell decomposition of $\bar{I}^n$ which is compatible with each consistent sign assignment of $f_1, \dots, f_N$. Each cell is a sub-Pfaffian set represented as a projection of a semi-Pfaffian set in DNF. The number of cells, the components of their formats and the complexity of the algorithm are less than*

$$N^{(r+n)^{O(n)}}(\alpha+\beta)^{(r+n)^{O(n^3)}}.$$

We summarize main properties of Pfaffian functions in the following propositions.

- Pfaffian functions can be considered as generalisation of algebraic functions.
- Pfaffian functions have the uniform description and the explicit characterization of complexity of their representations.
- The class of Pfaffian functions includes *exp*, trigonometrical functions defined in appropriate domains, and more generally solutions of a large class of differential equations.
- The structure $\mathbb{R} = \langle \mathbb{R}, +, *, 0, 1, <, \{f_1, \dots, f_N\} \rangle$ is o-minimal, i.e. definable sets have only a finite number of connected components, in the other words, it has finiteness property.

3 Pfaffian Dynamical Systems

3.1 Pfaffian Dynamics and Related Sets

We now recall definitions concerning Pfaffian dynamical systems.

Definition 10. *Let $G_1 \subset \mathbb{R}^{k_1}$ and $G_2 \subset \mathbb{R}^{k_2}$ be open domains. A* Pfaffian dynamical system *is a map*

$$\gamma : G_1 \times (-T, T) \to G_2$$

with a semi-Pfaffian graph, where G_1 is a set of control parameters, $(-T, T)$ is an interval of time, and G_2 is a state space.

For a given $\mathbf{x} \in G_1$ the set

$$\Gamma_{\mathbf{x}} = \{\mathbf{y} | \exists t \in (-T, T)\, (\gamma(\mathbf{x}, t) = \mathbf{y})\} \subset G_2$$

is called the trajectory *(or evolution) determined by $\mathbf{x}$, and the graph*

$$\widehat{\Gamma}_{\mathbf{x}} = \{(t, \mathbf{y}) |\, \gamma(\mathbf{x}, t) = \mathbf{y}\} \subset (-T, T) \times G_2$$

is called the integral curve *determined by $\mathbf{x}$.*

Definition 11. *Let $U \subseteq G_1$. A set $V \subseteq G_2$ is called* viable *under the dynamical system γ and the control U if there exists $\mathbf{x} \in U$ such that for all $t \in T$, $\gamma_{\mathbf{x}}(t) \in V$. We say* a subset $U \subseteq G_1$ satisfies the constraint V *if V is viable under U and the dynamical system γ.*

Definition 12. *Let $U \subseteq G_1$. A set $Inv \subseteq G_2$ is* invariant *under the dynamical system γ and the control U if for all $x \in U$ and for all $t \in T$, $\gamma_x(t) \in Inv$.*

In the next sections we investigate the behavior of a Pfaffian dynamical system with respect to a given semi-Pfaffian viability constraint.

3.2 Encoding Trajectories by Words

We now introduce, following [4,9], a technique of encoding trajectories of dynamical systems by words. Consider a Pfaffian dynamical system $\gamma : G_1 \times (-T, T) \to G_2$, where $G_1 \subset \mathbb{R}^{k_1}$ and $G_2 \subset \mathbb{R}^{k_2}$ are open domains, and a partition $\mathcal{P} := \{P_1, \dots, P_s\}$ of G_2 into s semi-Pfaffian sets P_j. Let the graph of γ and each set P_j have a format (r, N, α, β, n), where $n \geq k_1 + k_2 + 1$, and all Pfaffian functions involved have a common Pfaffian chain. Fix $\mathbf{x} \in G_1$. Define the set of points and open intervals in $\mathbb{R}$:

$$\mathcal{F}_{\mathbf{x}} := \{J |\ J \text{ is a point or an interval in } (-\mathrm{T}, \mathrm{T}) \text{ maximal w.r.t. inclusion for the}$$
$$\text{property } \exists i \in \{1, \dots, s\} \forall t \in J\ (\gamma(\mathbf{x}, t) \in P_i)\}.$$

Let the cardinality $|\mathcal{F}_{\mathbf{x}}| = r$ and $y_1 < \cdots < y_r$ be the set of representatives of $\mathcal{F}_{\mathbf{x}}$ such that $\gamma(\mathbf{x}, y_j) \in P_{i_j}$. Then define the word $\omega := P_{i_1} \cdots P_{i_r}$ in the alphabet $\mathcal{P}$. Informally, ω is the list of names of elements of the partition in the order they are visited by the trajectory $\Gamma_{\mathbf{x}}$. In our setting ω is called the *type of trajectory* $\Gamma_{\mathbf{x}}$. Introduce the set of words $\Omega := \{\omega |\ \mathbf{x} \in G_1\}$.

Theorem 2. [4,9] *The set Ω is finite and the number of different trajectory types of γ with respect to the partition $\mathcal{P}$ is less than*

$$(sN)^{(r+n)^{O(n)}} (\alpha + \beta)^{(r+n)^{O(n^3)}} \tag{3}$$

Theorem 3. *There is a cell decomposition of the control parameter space G_1 such that if $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to the same cell then $\Gamma_{\mathbf{x}_1}$ and $\Gamma_{\mathbf{x}_2}$ are labelled by the same word.*

Proof. Consider the family $\mathcal{F} = \{f_1, \dots, f_k\}$ of Pfaffian functions in the domain $G_1 \times (-T, T) \times G_2$ consisting of all functions in variables $\mathbf{x}, t, \mathbf{y}$ involved in the defining formulas for the graph of the map $\gamma :\ (\mathbf{x}, t) \mapsto \mathbf{y}$, and for all sets P_j. According to Theorem 1, there is a cylindrical decomposition $\mathcal{D}$ of $G_1 \times (-T, T) \times G_2$ with respect to the variables $\mathbf{x}, t, \mathbf{y}$ having the following properties.

1) $\mathcal{D}$ is compatible with each consistent sigh assignment of $f_1, \dots, f_k$.
2) There are at most (3) cylindrical cells.
3) Each of these cells is sub-Pfaffian.
4) $\mathcal{D}$ induces a cylindrical decomposition on G_1 which we denote by $\mathcal{E}$.

We claim that for any cell $C \in \mathcal{E}$ and any two points $\mathbf{x}_1, \mathbf{x}_2 \in C$ the trajectories $\Gamma_{\mathbf{x}_1}, \Gamma_{\mathbf{x}_2} \in G_2$ are intersecting sets $P_1, \dots, P_s$ in the same order (i.e., are encoded by the same word from Ω). Indeed, let $\pi : G_1 \times (-T, T) \times G_2 \to G_1$ be the projection on G_1. The decomposition $\mathcal{D}$ induces cylindrical decompositions $\mathcal{D}_1$ and $\mathcal{D}_2$ on $\pi^{-1}(\mathbf{x}_1)$ and $\pi^{-1}(\mathbf{x}_2)$ respectively. In particular, each of the integral curves $\widehat{\Gamma}_{\mathbf{x}_1}$ and $\widehat{\Gamma}_{\mathbf{x}_2}$ is decomposed into a sequence of alternating points and open intervals. Due to basic properties of cylindrical decomposition, there is a natural bijection $\psi : \mathcal{D}_1 \to \mathcal{D}_2$ such that

(i) the restriction of ψ to the set of all cells in $\widehat{\Gamma}_{\mathbf{x}_1}$ is a bijection onto the set of all cells in $\widehat{\Gamma}_{\mathbf{x}_2}$;
(ii) for each $1 \leq j \leq s$ the restriction of ψ to the set of all cells in $(-T, T) \times P_j \cap \pi^{-1}(\mathbf{x}_1)$ is a bijection onto the set of all cells in $(-T, T) \times P_j \cap \pi^{-1}(\mathbf{x}_2)$.
(iii) the bijection ψ preserves the order in which cells appear in the trajectories.

It follows that if a cell $B \in \mathcal{D}_1$ is a subset of $\widehat{\Gamma}_{\mathbf{x}_1} \cap ((-T, T) \times P_j)$ for some $1 \leq j \leq s$, then $\psi(B) \subset \widehat{\Gamma}_{\mathbf{x}_2} \cap ((-T, T) \times P_j)$. Moreover, if for cells B_1, $B_2 \in \mathcal{D}_1$ there exist t_1, $t_2 \in (-T, T)$ such that $t_1 < t_2$ and $\gamma(\mathbf{x}_1, t_1) \in B_1 \wedge \gamma(\mathbf{x}_1, t_2) \in B_2$ then there exist t_1', $t_2' \in (-T, T)$ such that $t_1' < t_2'$ and $\gamma(\mathbf{x}_2, t_1') \in \psi(B_1) \wedge \gamma(\mathbf{x}_2, t_2') \in \psi(B_2)$. The claim is proved.

It follows that the cardinality of Ω does not exceed the cardinality of $\mathcal{E}$ which does not exceed the cardinality of $\mathcal{D}$ which in turn is at most (3).

4 An Algorithm for Checking Satisfiability of Viability Constraints

Consider a Pfaffian dynamical system $\gamma : G_1 \times (-T, T) \to G_2$ and semi-Pfaffian sets: a subset of control parameters $U \subseteq G_1$, and a subset of the state space $V \subseteq G_2$. Let the graph of γ and the sets U, V have a format (r, N, α, β, n), and all Pfaffian functions involved have a common Pfaffian chain. Let us note that the set $\bar{V} = G_1 \setminus V$ is semi-Pfaffian and has the same format.

Theorem 4. *There is an algorithm which checks whether the control U satisfies the viability constraint V. The complexity of this algorithm does not exceed*

$$(2N)^{(r+n)^{O(n)}} (\alpha + \beta)^{(r+n)^{O(n^3)}} \tag{4}$$

Proof. We are going to show the main steps of our algorithm. First the algorithm produces the set of words Ω corresponding to the Pfaffian dynamical system $\gamma : G_1 \times (-T, T) \to G_2$ and the partition $\mathcal{P} := \{P_1, P_2\}$, where $P_1 := V$ and $P_2 := \bar{V}$. Consider the family of Pfaffian functions in the domain $G_1 \times (-T, T) \times G_2$ consisting of all functions in variables $\mathbf{x}, t, \mathbf{y}$ involved in the defining formulas for the graph of the map $\gamma : (\mathbf{x}, t) \mapsto \mathbf{y}$, and for the set V. According to Theorem 1, there is a cylindrical decomposition $\mathcal{D}$ with respect to $(\mathbf{x}, t, \mathbf{y})$ which is compatible with this family and consists of at most (4) cylindrical cells.

This cell decomposition $\mathcal{D}$ induces the cell decomposition $\mathcal{E}$ (see the proof of Theorem 3). Using the oracle, which decides feasibility of any system of Pfaffian equations and inequalities, the algorithm selects the cells from $\mathcal{D}$ which are subsets of $\{(\mathbf{x}, t, \mathbf{y}) | \mathbf{y} = \gamma(\mathbf{x}, t)\}$. Denote the set of the selected cells by $\mathcal{B}$. Observe that for any fixed $\mathbf{x}' \in G_1$ the set $\bigcup_{B \in \mathcal{B}} B \cap \{(\mathbf{x}, t, \mathbf{y}) | \, \mathbf{x} = \mathbf{x}'\}$ coincides with the integral curve $\widehat{\Gamma}_{\mathbf{x}'}$. Then the algorithm determines the order in which the cells $B \in \mathcal{B}$ intersected with $\{(\mathbf{x}, t, \mathbf{y}) | \, \mathbf{x} = \mathbf{x}'\}$ appear in the trajectory $\Gamma_{\mathbf{x}'}$.

More precisely, for each pair of distinct cells $B_1, B_2 \in \mathcal{B}$ the algorithm decides, using the oracle, whether

$$\exists \mathbf{x} \exists t_1 \exists t_2 \exists \mathbf{y}_1 \exists \mathbf{y}_2 \, ((\mathbf{x}, t_1, \mathbf{y}_1) \in B_1 \wedge (\mathbf{x}, t_2, \mathbf{y}_2) \in B_2 \wedge (t_1 < t_2)).$$

For a given $C \in \mathcal{E}$, after all pairs of cells are processed we get the ordered set of cells $B_1, \dots, B_k$ in $\mathcal{D}$ such that for any $1 \leq i \leq k$ and any $\mathbf{x}' \in C$ the sequence of points and intervals

$$B_1 \cap \{(\mathbf{x}, t, \mathbf{y}) | \, \mathbf{x} = \mathbf{x}'\}, \dots, B_k \cap \{(\mathbf{x}, t, \mathbf{y}) | \, \mathbf{x} = \mathbf{x}'\}$$

forms the integral curve $\widehat{\Gamma}_{\mathbf{x}'}$. By the definition of cylindrical decomposition, for any pair B_i, P_j either $B_i \subset (C \times (-T, T) \times P_j)$ or $B_i \cap (C \times (-T, T) \times P_j) = \emptyset$. The algorithm uses the oracle to decide for every pair which of these two cases takes place. As the result, the sequence $B_1, \dots, B_k$ becomes partitioned into subsequences of the kind

$$(B_1, \dots, B_{k_1}), (B_{k_1+1}, \dots, B_{k_2}), \dots, (B_{k_{\ell-1}+1}, \dots, B_k),$$

where for any i, $0 \leq i \leq \ell - 1$, the cells $B_{k_i+1}, \dots, B_{k_{i+1}}$ lie in $C \times (-T, T) \times P_{j_i}$ for some j_i, while $B_{k_i} \cap C \times (-T, T) \times P_{j_i} = \emptyset$ and $B_{k_{i+1}+1} \cap C \times (-T, T) \times P_{j_i} = \emptyset$. Then the word $\omega := P_{j_0} \cdots P_{j_{\ell-1}}$ corresponds to the cell C. Considering all cells in $\mathcal{E}$ the algorithm finds Ω.

Then the algorithm collects all cells from $\mathcal{E}$ which correspond to the word $\omega = P_1$. Final step is to check intersections of these cells with the given set U. If at least one of them is nonempty then the set U satisfies the viability constraint V. This completes the description of the algorithm.

A straightforward analysis shows that the complexity of the algorithm does not exceed (4), taking into account the bounds from Theorem 1.

Corollary 1. *There is an algorithm checking viability and invariant properties of a set of state space V under the dynamics γ and the control U. The complexity of this algorithm does not exceed $(2N)^{(r+n)^{O(n)}} (\alpha + \beta)^{(r+n)^{O(n^3)}}$.*

5 Conclusion and Future Research

We have proposed an algorithm for checking satisfiability of viability constraints on the control of a Pfaffian dynamical system. This research has been motivated by verification problems of safety-critical large scale continuous and hybrid systems. First step in the suggested procedure is to construct a cylindrical cell decomposition which is compatible with each sign assignment of the Pfaffian functions involved in the definitions of a continuous dynamic and a viability constraint. In the second step we encode trajectories of the Pfaffian dynamical system by finite words. By the construction of cylindrical cell decomposition, the space of parameters is decomposed to cells in such way that each cell corresponds to one word. In other words, if $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to the same cell the trajectories $\Gamma_{\mathbf{x}_1}$ and $\Gamma_{\mathbf{x}_2} \in G_2$ are encoded by the same word. This induces a natural marking the cells of parameters by the words. In the final step we check intersections of a given set of control parameters and the cells of parameters which marked by the special word. If at least one of them is nonempty then the given set of control parameters satisfies the viability constraint. This algorithm is based on the cylindrical cell decomposition technique and, accordingly, has a double exponential upper complexity bound. It seems feasible to construct an algorithm with *single exponential* complexity using the approach employed in the paper [10].

References

1. Jean-Pierre Aubin, *Viability Analysis*, Birkhauser, Boston, 1991.
2. S. Basu, R. Pollack and M.-F. Roy, *Algorithms in Real Algebraic Geometry*, Springer, Berlin-Heidelberg, 2003.
3. L. Blum, , F. Cucker, M. Shub, and S. Smale, *Complexity and Real Computation*, Springer, New York, 1997.
4. T. Brihaye, C. Michaux, C. Riviere, C. Troestler, On o-minimal hybrid systems, in: *Hybrid Systems: Computation and Control*, R. Alur, G. J. Pappas, (Eds.), LNCS, **2993**, Springer, Heidelberg, 2004, 219–233.
5. A. Gabrielov, N. Vorobjov, Complexity of computations with Pfaffian and Noetherian functions, in: *Normal Forms, Bifurcations and Finiteness Problems in Differential Equations*, Yu. Ilyashenko et al., (Eds.), NATO Science Series II, **137**, Kluwer, 2004, 211–250.
6. A. Gabrielov, N. Vorobjov, Complexity of cylindrical decompositions of sub-Pfaffian sets, *J. Pure and Appl. Algebra*, **164**, 1–2, 2001, 179–197.
7. A. Gabrielov, N. Vorobjov, Betti numbers of semialgebraic sets defined by quantifier-free formulae, to appear in: *Discrete and Computational Geometry*, 2004.
8. A. Khovanskii. *Fewnomials.* Number 88 in Translations of Mathematical Monographs. American Mathematical Society, Providence, RI, 1991.
9. M. Korovina and N. Vorobjov, Pfaffian hybrid systems. In *Springer Lecture Notes in Comp. Sci.*, volume 3210 of *Computer Science Logic'04*, 2004, 430–441.
10. M. Korovina and N. Vorobjov, Upper and lower Bounds on Sizes of Finite Bisimulations of Pfaffian Hybrid Systems. In *Proceedings of CiE'06*, invited talk, LNCS 3988, 2006, 235–241.
11. G. Lafferriere, G.J. Pappas, S. Sastry, O-minimal hybrid systems, *Math. Control Signals Systems*, **13**, 2000, 1–21.
12. S. Pericleous, N. Vorobjov, New complexity bounds for cylindrical decompositions of sub-Pfaffian sets, in: *Discrete and Computational Geometry. Goodman-Pollack Festschrift*, B. Aronov et al. (Eds.), Springer, 2003, 673–694.
13. L. van den Dries. *Tame Topology and O-minimal Structures.* Number 248 in London Mathematical Society Lecture Notes Series. Cambridge University Press, Cambridge, 1998.

On the Importance of Parameter Tuning in Text Categorization

Cornelis H.A. Koster[1] and Jean G. Beney[2]

[1] Dept. Comp. Sci., University of Nijmegen, The Netherlands
kees@cs.kun.nl
[2] Dept. Informatique, INSA de Lyon, France
jean.beney@insa-lyon.fr

Abstract. Text Categorization algorithms have a large number of parameters that determine their behaviour, whose effect is not easily predicted objectively or intuitively and may very well depend on the corpus or on the document representation. Their values are usually taken over from previously published results, which may lead to less than optimal accuracy in experimenting on particular corpora.

In this paper we investigate the effect of parameter tuning on the accuracy of two Text Categorization algorithms: the well-known Rocchio algorithm and the lesser-known Winnow. We show that the optimal parameter values for a specific corpus are sometimes very different from those found in literature. We show that the effect of individual parameters is corpus-dependent, and that parameter tuning can greatly improve the accuracy of both Winnow and Rocchio.

We argue that the dependence of the categorization algorithms on experimentally established parameter values makes it hard to compare the outcomes of different experiments and propose the automatic determination of optimal parameters on the train set as a solution.

Keywords: Text Categorization, automatic classification, Winnow, Rocchio, parameter tuning.

1 Introduction

Information Retrieval is an interdisciplinary subject with a long history. Recently, it has been much influenced by theory and methods from Machine Learning (automatic Text Categorization) and Language and Speech technology (Language Modelling, Linguistic Techniques). It has a proud history of strict experimental methodology, exemplified by a sequence of TREC and SIGIR conferences, and a steady progress in its theory.

Text Categorization (for a recent overview see [17]) is a perfect area for experimentation in Information Retrieval, because the abundance of documents labeled with categories provides a solution for the otherwise vexing problem of computing Recall.

An automatic Text Categorization algorithm learns from examples, train documents tagged with the categories to which they belong. It determines from the examples which are characteristic of the categories learned. But these algorithms are to some extent based on formulae that include empirical constants whose value depends on the document set but can not easily be predicted objectively or intuitively, like

– the Term Selection criterion, and the number of terms selected
– the Term Weighting technique (e.g. Boolean, linear, square root, logarithmic, ltc)
– the document length normalization technique
– some parameters typical for the algorithm used (the number of neighbours in Knn, the choice of Kernel Function in SVM, . . .), and
– parameters that determine the kind of classification desired (mono- or multi-, the Utility Function to use, the thresholding technique) and others.

These parameters influence to a large extent either the accuracy of the classification algorithm or its speed, or both. When comparing various classification algorithms on the same task, the differences in performance found may well be attributable to a large extent to differences in tuning, rather than to inherent qualities of the algorithms, falsifying the experiment (see also [6]).

In principle, given enough train documents the optimal choice of a technique or parameter value can be computed from the train set by brute force, but this is time-consuming and difficult to repeat for every new train set. Therefore researchers tend to reuse choices and parameter values which have been reported in literature.

The ideal Text Categorization system should have only parameters whose effect is intuitively clear and predictable, and as few of them as possible. Otherwise the user of the system might well feel lost in a high-dimensional parameter space (like the Nuclear Physicist of the late sixties, trying to manually fit the Optical Model to his data by twiddling 16 parameters).

In this note we'll investigate the typical parameters of the Rocchio and Winnow Text Categorization algorithms and their interaction with Term Selection and propose a technique to make these algorithms self-tuning.

2 Experimental Setup

In our investigations we made use of the LCS classification engine, which implements the Winnow and Rocchio algorithms (see the next sections), automatically learns class thresholds from the train data and has a choice of Term Selection algorithms [15].

Our experimental approach is as follows: we shall first tune each algorithm on a variety of corpora, i.e. determine the optimal values of their parameters, without performing any Term Selection. Then we shall determine the effect of optimal Term Selection on the same corpora with and without tuning, and interpret the results.

Our experimental approach is as follows: we shall tune each algorithm on a variety of corpora, and determine optimal values for their parameters on each corpus, maximizing the accuracy (measured by the micro-averaged F1 value).

2.1 The Corpora

We have experimented with many different corpora, exemplifying different document representations and classification tasks, in order to make the results more general. In this publication we shall use three:

- ModApte – The well-known Apté subset of the Reuters 21578 corpus, consisting of 12902 newspaper stories in 135 TOPICS categories, with an average length of 116 words [1].
- The EPO1A corpus consists of 16000 abstracts of patent applications in English from the European Patent Office, with an average length of 143 words (see [11,13]). For this corpus, we shall also show results using a bag-of-phrases representation (EPO1Afr, see [12]) besides the customary bag-of-words representation (EPO1A kw).
- EPO1F – These are the full-text patent applications corresponding to the EPO1A abstracts, of about 2000 words each; The total collection has a size of 4611 M-bytes.

All these corpora are in English, but they differ in other properties, as shown in table 1, representing different classification tasks.

Table 1. Some quantitative differences between the corpora

Corpus	doc size	classes	nmb docs/class
ModApte corpus	short	135 multi	widely varying
EPO1A abstracts	short	16 mono	1000 documents
EPO1F full-text	long	16 mono	1000 documents

The ModApte corpus has been used (in slightly different subsets) in many experiments reported in literature, allowing comparison with other work.

2.2 The Experiments

In each experiment, the corpus used was split into 4 subsets of equal size, chosen at random, in a four-fold cross-validation (training on one subset while using the union of the other three as test set). In each run, 25% of the documents were used as train documents and 75% as test documents. The relatively large number of test documents was chosen in order to reduce variance (and because testing is much faster than training). As a Measure of Accuracy we used the micro-averaged F1 value. The train sets were kept small, since the goal was to make many comparisons, rather than to achieve the highest possible accuracy. In testing we made no use of the information that the EPO1A/F corpora are mono-classified, allowing 0-3 classifications per document, and 0-16 for the ModApte corpus, so that the results are also applicable to multi-classification. We used the same term weighting (ltc) and the same document length normalisation (cosine) in all experiments.

For the keyword representation, no pre-processing was applied, apart from de-capitalization and the removal of some special characters. For the phrase representation, the EPO1A corpus was parsed and translated to Head/Modifier pairs and unnested as in [12].

In the graphs we shall usually not indicate the variance, because this would make the graphs illegible, but the amount of variance in the measurement results is mostly obvious from the irregularity in the graphs.

3 Tuning Rocchio

In origin [16], the Rocchio algorithm was conceived for retrieval with relevance feedback [18]

$$Q_{new} = \alpha \times Q_{orig} + \beta \times \frac{1}{R} \sum_{D \in Rel} D - \gamma \times \frac{1}{N-R} \sum_{D \notin Rel} D$$

in which each document is represented by a vector of term frequencies. In the Rocchio classification algorithm a class profile is computed as the centroid of the documents relevant to the class, subtracting the irrelevant documents (there is no original document).

A more sophisticated form of the Rocchio algorithm [4] assigns to each individual term t a weight for the class c, according to the formula

$$w(t,c) = max(0, \frac{\beta}{|D_c|} \sum_{d \in D_c} s(t,d) - \frac{\gamma}{|\overline{D}_c|} \sum_{d \in \overline{D}_c} s(t,d))$$

where

- $s(t,d)$ is the normalized strength of the term t in the document d, using some sub-linear function of the frequency of the term and compensating in some way for variations in document length
- D_c is the set of documents that are labeled with class c and $\overline{D}_c$ the set of non-c documents.

The score of a document for a class is the inproduct of the weights of its terms times their strength, and a document is assigned to class c if its score for c exceeds a class threshold which is computed from the train set.

Notice also that terms that would yield a negative contribution are eliminated in the above formula. This contradicts the original intuition of the centroid computation. We shall therefore investigate both variants.

3.1 Choice of Beta and Gamma

The Rocchio formula contains two parameters β and γ, whose values are well-known to be 16 and 4, respectively (according to many publications, including very recent ones like [5,3]). Why 16 and 4? The reason for these curious values is that there was originally also a factor α, which is zero in the present formula.

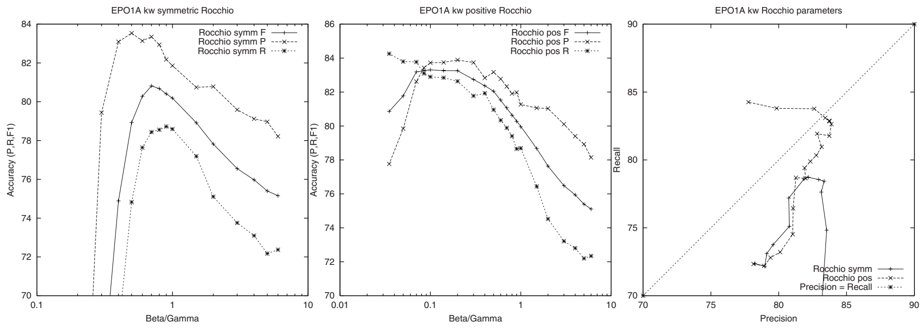


Fig. 1. Varying Rocchio's beta parameter

Obviously, dividing both parameters by the same factor makes no difference, apart from a shift of the threshold, so we can fix γ at 1, leaving only one parameter, which can be interpreted as the relative weight attributed to positive examples. The standard value of β is 4; but is this optimal?

3.2 Symmetric Rocchio and Positive Rocchio

We shall now try to find optimal values of β for various corpora, without Term Selection, i.e. without discarding any of the terms, keeping $\gamma = 1$ and distinguishing between a variant in which negative term weights are allowed ("symmetric Rocchio") and one in which negative term weights are omitted ("positive Rocchio"). Figure 1 shows the the result of varying β in classifying the EPO1A corpus (without Term Selection).

The left graph shows that the optimum β-value for symmetric Rocchio is not 4 but 0.7; the middle one shows that positive Rocchio reaches an even higher Accuracy at $\beta = 0.1$. The elimination of negative terms pays off. The rightmost graph (which is hard to interpret) shows the trajectory followed by Precision and Recall when reducing the β-value, for both variants. When β becomes too small, positive Rocchio fails catastrophically through loss of Recall and symmetric Rocchio through loss of Precision.

Figure 2 compares the accuracy achieved by the two Rocchio variants at various β-values for the EPO1A corpus and the ModApte and EPO1F corpora.

They show roughly the same behaviour, and practically the same optimal parameter values. In all cases, positive Rocchio performs better than symmetric Rocchio, with an optimum near $\beta = 0.1$.

3.3 Interaction with Term Selection

Next, we investigate the interaction between parameter tuning and term selection, by showing the Accuracy on EPO1A as a function of the number of terms per class, selected by the Simplified χ^2 criterion. The results are shown in Fig. 3 for both versions of Rocchio at optimal tuning and only one version at standard tuning (where they are indistinguishable).

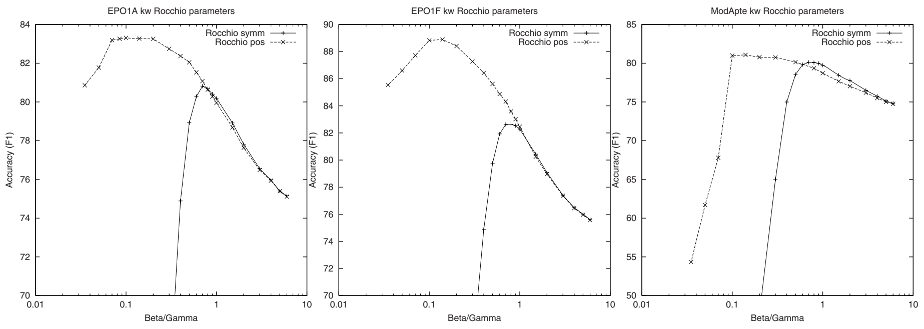


Fig. 2. Comparing two Rocchio variants on 3 corpora

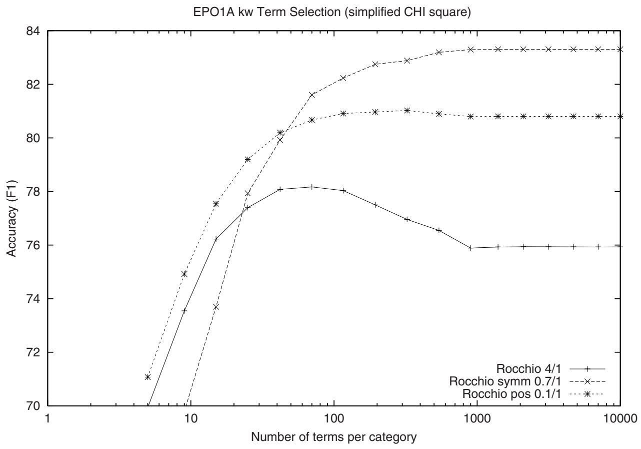


Fig. 3. Tuning versus Term Selection for Rocchio

As was found in our earlier experiments, (cf. [15]), at standard parameter values Rocchio reaches its optimal Accuracy when 100 terms per class are selected. After tuning, the Accuracy (which is much higher) is no longer improved by Term Selection, although selecting at most 1000 terms/category may still be useful for performance reasons. An optimal choice of β/γ has much more effect than optimal Term Selection. At optimal parameter values, Rocchio itself eliminates the noisy terms.

3.4 Discussion

The positive Rocchio variant in each case reaches a higher accuracy than the symmetric one, at a much lower β value. It is surprising to see that the β parameter must be smaller (or even much smaller) than the γ parameter, so that a greater weight is given to negative examples, even though there are many more of them than positive examples. A plausible explanation of the behaviour of positive Rocchio is that, as β gets smaller, more noisy terms are eliminated (because their weight becomes negative) and the Accuracy is improved, until eventually too many terms are eliminated to achieve Recall.

Similarly, for symmetric Rocchio the effect of giving more weight to negative examples is initially beneficial, but when β gets too small, documents tend to get classified on the non-occurrence of negative terms, rather than on the occurrence of positive ones (keep in mind that the threshold will shift automatically with the sinking document scores).

At the traditional value $\beta = 4$ the contribution of negative terms is very small and the two Rocchio variants behave indistinguishably. This is probably the reason that we found only one article in literature [14] which describes and explains the superiority of the positive Rocchio variant.

4 Tuning Winnow

The Balanced Winnow algorithm is a child of the Perceptron, as is clear from the formulation given in [9]:

```
**BalancedWinnow**$(\overrightarrow{w}, \overrightarrow{z}, (\overrightarrow{x}, y))$ :
    if sign $(\overrightarrow{w} \cdot \overrightarrow{x}) \neq y$ then
    begin $\overrightarrow{z} := \overrightarrow{z} + \alpha y \overrightarrow{x}$
          $\overrightarrow{w} := 2\overrightarrow{sinh}(z)$
    end
```

where $y = 1$ for a relevant document and $y = -1$ for an irrelevant one, and the weights are exponentiated by the function *sinh*.

In the description of Balanced Winnow given in [7], for every class c and for every term t two weights W_t^+ and W_t^- are kept. The single parameter α of Winnow has been split into a promotion parameter α and a demotion parameter β.

The score of a document d for a class c is computed as

$$SCORE(d, c) = \sum_{t \in d} (W_{t,c}^+ - W_{t,c}^-) \times s(t, d)$$

where $s(t, d)$ is the normalized strength of the term t in d. A document d belongs to a class c if $SCORE(d, c) > \theta$, where the threshold θ is usually taken to be 1.

Winnow learns multiplicatively, driven by mistakes, one document at a time: When a train document belonging to some class c scores below θ, the weights of its terms t in W_t^+ are multiplied by a constant $\alpha > 1$ and those in W_t^- multiplied by $\beta < 1$; and conversely for documents *not* belonging to c which score above θ.

Winnow is by origin an *on-line* algorithm, meaning that the weights are adjusted for each incoming train document, but it is also possible to iterate over a set of train documents. In the following experiments, we shall at first keep the number of iterations at 5, and later study the effect of different numbers of iterations.

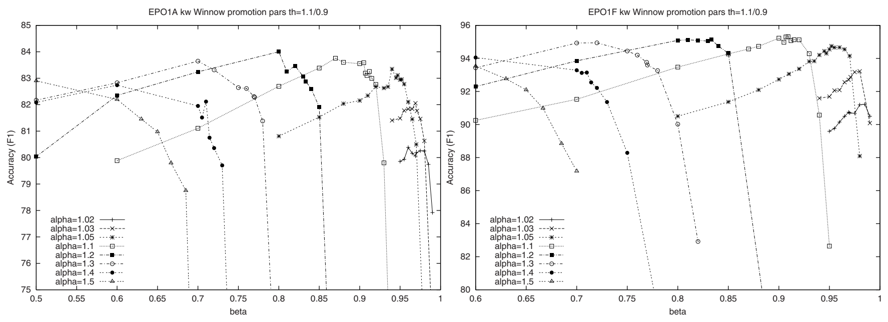


Fig. 4. Varying Winnow's promotion parameters

4.1 The Winnow Promotion Parameters

According to [9] β should be equal to $1/\alpha$. The values suggested in [7] are 1.1 and 0.9, whose product is in fact not quite equal to one.

There are good reasons to choose β smaller than $1/\alpha$: If β is smaller than $1/\alpha$, Winnow learns faster from positive examples than from negative examples, which may be justified since there are fewer positive examples. Furthermore, consider a term that is promoted and demoted a large but equal number of times. When $\alpha \times \beta < 1$, both W_t^+ and W_t^- will tend to zero: this noisy term is eliminated. When $\alpha \times \beta = 1$, they keep their initial values. One would expect a smaller value of α to lead to slower but more precise convergence.

Measuring the Accuracy as a function of α and β for EPO1A kw and EPO1F gives the results shown in Fig. 4.

In spite of the four-fold cross-evaluation, there is a lot of variance (especially for EPO1A kw), which makes it hard to choose an optimal value. From these graphs (and many others not shown here), it appears that $\beta = 2 - \alpha$ is a better choice than $1/\alpha$. The optimal choice of α depends on the number of training documents. For larger numbers of docs it is better to choose a smaller alpha: On the EPO2F corpus (which is like the EPO1F corpus but with 68418 instead of 16000 train documents) we found the highest accuracy at $\alpha = 1.04$ when training on all train documents and at $\alpha = 1.1$ when training on a tenth of the documents [2].

In the following experiments we will stick to Dagan's choice [7] (1.1/0.9) for the promotion parameters, which is quite good for all three corpora, and tune the other parameters accordingly.

4.2 The Thick Threshold Heuristic

Again following [7], the Accuracy of Winnow can be improved by means of the *thick threshold* heuristic: In training, we try to force the score of relevant documents up above $\theta^+ > 1.0$ (rather than just 1) and irrelevant documents below $\theta^- < 1.0$. This resembles the "query zoning" [18] heuristic for Rocchio, in

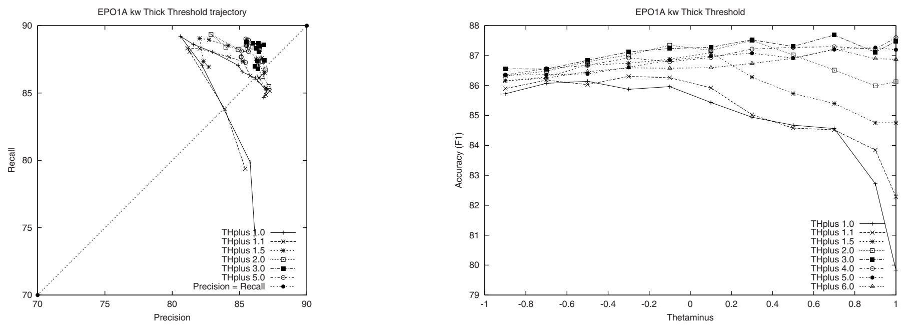


Fig. 5. Effect of Thick Threshold Heuristic (1)

the sense that documents on the borderline of a class (scoring between θ^- and θ^+) receive extra attention.

According to [7] the optimal values for these Thick Threshold parameters are 1.1 and 0.9, respectively (just like the α and β). A heuristical argument suggests that the best value for θ^- might be $1/\theta^+$, since they each represent a number of steps (multiplications by α or β) away from the threshold 1. Figure 5 shows the effect of varying the thickness of the threshold for the EPO1A corpus.

The left graph in Fig. 5 plots Precision against Recall. Each line is a trajectory (going approximately first upwards and then to the right) which represents one value of θ^+ together with values of θ^- going down from 0.9 to 0.1. The dotted line represents Precision = Recall.

The right graph is easier to read. The F1-value fluctuates wildly, but by and by an increase of θ^+ improves the Accuracy, and 3.0/0.7 raises the F1-value over 1.1/0.9 by more than 2 points.

The Thick Threshold graphs in Fig. 6 show roughly similar behaviour on the different corpora, but it is clear that the optimal values for the Thick Threshold parameters depend strongly on the corpus. When increasing θ^+, the Accuracy first increases, then falls off again. The curves are rather erratic due to high variance. The intuition that the best value for θ^- is $1/\theta^+$ is not supported by the measurements.

4.3 Interaction with Term Selection

Figure 7 shows the effect of Term Selection with and without tuning of Winnow. Again, it appears that optimal parameter tuning removes the noisy terms much more effectively than Term Selection.

The relation between Winnow training and Term Selection is quite obvious: given enough promotions and/or demotions of documents belonging to some class, all terms occurring in the profile for that class will reach one of the following states:

– **positive terms** – terms speaking for the category will obtain a relatively stable positive value for W^+ and W^- will be (practically) zero

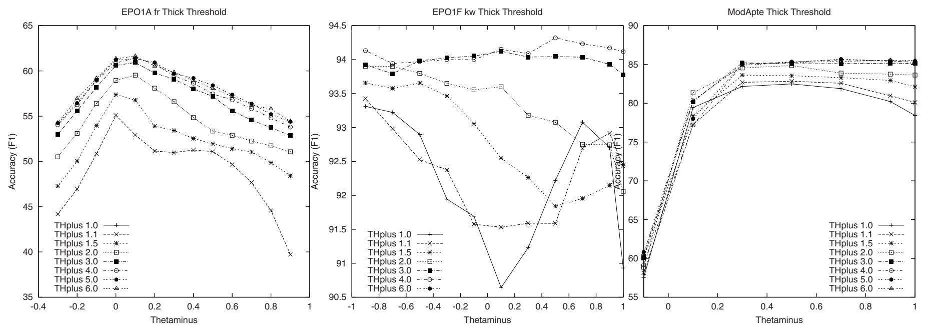


Fig. 6. Effect of Thick Threshold Heuristic (2)

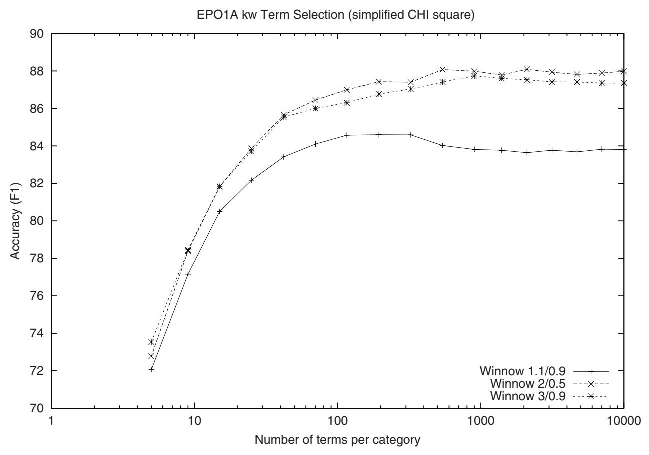


Fig. 7. Tuning versus Term Selection for Winnow

– **negative terms** – terms speaking against the category will have a positive W^- while W^+ is zero
– **noisy terms** – terms that are not reliable will have both W^+ and W^- (practically) zero.

Experiments not reported here show that omitting the negative terms (positive Winnow) causes a significant reduction in the Accuracy. Symmetric Winnow is in all cases better than positive Winnow, in contrast with Rocchio, which appears to cope well enough with noisy terms, but can not use the information provided by negative terms. This may explain why, in our experiments, Winnow always outperforms Rocchio.

4.4 The Number of Iterations

In our previous experiments with Winnow we fixed the number of iterations at five: every document (in random order) was seen five times by the classification

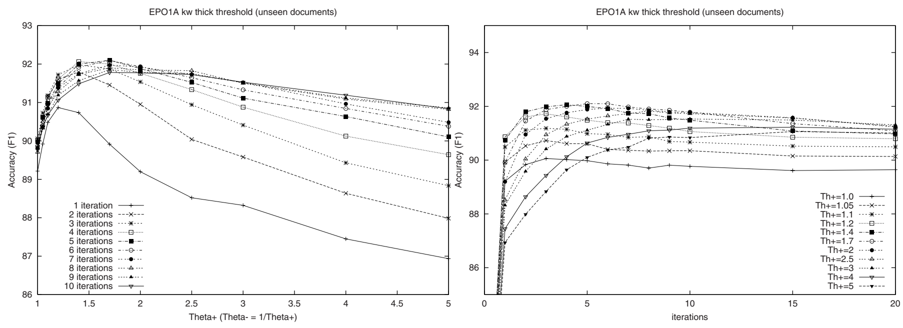


Fig. 8. Iterations and Thresholds

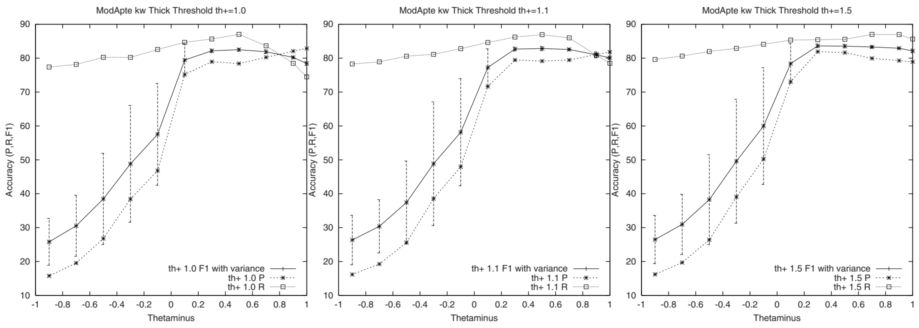


Fig. 9. Thick Threshold – details

algorithm, and only in case the classification computed for it was not correct, the document was trained.

Training more than once may increase Accuracy, especially when there are few documents, but training too often may lead to overtraining. The optimal number of iterations is dependent on the corpus, and on other parameters. We shall now investigate experimentally the interaction between the number of iterations and the optimal Thick Threshold values.

Since it is hard to make an understandable graph depicting the relationship between the three parameters, we fix $\theta^- = 1/\theta^+$ ("geometrical symmetry"), even though according to Fig. 4 and Fig. 5 this is not optimal.

The two graphs in Fig. 8 show the Accuracy achieved on EPO1A kw as a function of θ^+ and the number of iterations, respectively.

The left graph shows that the Accuracy increases with θ^+ at a suitable number of iterations, and then slowly goes down. The right graph shows that, for a given value of θ^+, the Accuracy increases with the number of iterations, until it is reduced again by overtraining. The overtraining point increases with the thickness of the threshold. It appears that choosing 3 to 5 iterations is best, at appropriate Theta values. The other corpora show similar behaviour.

Precision/Recall/F-value graphs for θ^+ against θ^- (like those shown in Fig. 9 for ModApte) show rich details for which no explanatory theory is available. Notice that the Precision = Recall point is not the optimum, and for $\theta^- < 0$ the severe drop in Precision and the large increase in variance.

5 Discussion

We have experimentally shown the importance of properly tuning the parameters of the Rocchio and Winnow classification algorithms. The optimal parameter values found in our experiments were mostly quite different from those found in literature, and gave a much higher accuracy. The good news is that in many existing applications the accuracy can be improved with just a little effort. The bad news is that it is hard to make meaningful comparisons between different classification algorithms. There is an unvoluntary bias against algorithms not familiar to the experimenter, caused by the difficulty of choosing parameters for an unfamiliar algorithm.

In comparing algorithms, the relevant parameter settings should also be published. And the comparison can not be fair unless all algorithms have been carefully tuned on the train set in order to capture the corpus-dependent aspects.

Since the manual tuning process is difficult and time consuming, every practical classification system should be made self-tuning. The best way to deal consistently with all those parameters is to tune them automatically, rather than using default values, for every new corpus or application.

Judging by the graphs in the paper (and the many thousand others we have seen in other experiments) the Accuracy (F1-value) of Winnow on a given corpus is a concave function of the parameters with a large noise term added (2 to 3 percent of the Accuracy). We conjecture that use of the Swarm Optimization technique (based on [8]) might lead to an efficient optimization scheme.

6 Conclusion

We have shed some light on the effect of the most important parameters of these algorithms, and on some of the interactions between them: Rocchio's weighting parameters, Winnow's promotion parameters and Thick Threshold, and their interaction with (one form of) Term Selection. Unfortunately these effects are quite dependent on the corpus. Only when using optimal parameter values, tuned for the particular corpus, can we claim that the results are really representative for the algorithm, and can we stop worrying about parameters.

For Rocchio it was found that discarding all terms with a negative score contribution (positive Rocchio) leads to a better Accuracy than taking them into account (the symmetric Rocchio algorithm). We have argued that this is an effect of automatic Term Selection.

We found the optimal values given for the Rocchio and Winnow parameters in many publications to be wildly wrong, and conjecture that many published results comparing these with other algorithms are misleading.

At optimal parameter values, both Rocchio and Winnow show no positive effect of Term Selection on the Accuracy; at best, performance may be somewhat improved. When optimally tuned, these algorithms are well capable of selecting their own terms. Tuning is much more important than term selection.

But many questions remain open: What happens for other forms of Term Selection? What happens when we use the macro- instead of micro-average (emphasizing the accuracy on small categories)? What is the influence of the Term Weighting and Document Size Normalization technique used? Are the optimal parameter values the same for larger and smaller subsets of the corpora? In fact, we are still groping around in high-dimensional space. More theoretical analysis (in particular of the fascinating Winnow algorithm) is needed.

References

1. Apté, C. and Damerau, F. (1994) Automated learning of decision rules for text categorization. *ACM Transactions on Information Systems* 12,3,233-251.
2. J.G.Beney and C.H.A. Koster (2003), Classification supervisée de brevets: d'un jeu d'essai au cas réel. Proceedings of the XXIeme congre's Inforsid, pp.50-59. http://www.loria.fr/conferences/inforsid2003/ActesWorkshopRI.pdf (last visited October 2003).
3. M.F. Caropreso, S. Matwin and F. Sebastiani (2000), A learner-independent evaluation of the usefulness of statistical phrases for automated text categorization, In: A. G. Chin (Ed.), *Text Databases and Document Management: Theory and Practice*, Idea Group Publishing, Hershey, US, pp. 78–102.
4. W.W. Cohen and Y. Singer (1999), Context-sensitive learning methods for text categorization. *ACM Transactions on Information Systems* 13,1,100-111.
5. K. Crammer and Y. Singer, A New Family of Online Algorithms for Category Ranking, Proceedings SIGIR '02, pg. 154
6. Walter Daelemans, Vronique Hoste, Fien De Meulder and Bart Naudts, Combined Optimization of Feature Selection and Algorithm Parameter Interaction in Machine Learning of Language. In: Proceedings of the 14th European Conference on Machine Learning (ECML-2003), Cavtat-Dubrovnik, Croatia, pp. 84-95, 2003.
7. I. Dagan, Y. Karov, D. Roth (1997), Mistake-Driven Learning in Text Categorization. In: *Proceedings of the Second Conference on Empirical Methods in NLP*, pp. 55-63.
8. Eberhart, R. C. and Kennedy, J. A new optimizer using particle swarm theory. Proceedings of the Sixth International Symposium on Micromachine and Human Science, Nagoya, Japan. pp. 39-43, 1995.
9. A. Grove, N. Littlestone, and D. Schuurmans (2001), General convergence results for linear discriminant updates. Machine Learning 43(3), pp. 173-210.
10. D. Hiemstra (1998), A Linguistically Motivated Probabilistic Model of Information Retrieval, European Conference on Digital Libraries 1998, pp. 569-584.
11. C.H.A. Koster, M. Seutter and J. Beney (2003), Multi-classification of Patent Applications with Winnow, Proceedings PSI 2003, Springer LNCS 2890, pp 545-554.
12. C.H.A. Koster and M. Seutter (2003), Taming Wild Phrases, Proceedings 25th European Conference on IR Research, Springer LNCS 2633, pp 161-176.
13. M. Krier and F. Zaccà (2002), Automatic Categorisation Applications at the European Patent Office, World Patent Information 24, pp. 187-196.

14. A. Moschitti (2003), A Study on Optimal Parameter Tuning for Rocchio Text Classifier, Proceedings 25th European Conference on IR Research, Springer LNCS 2633, pp 420-435.
15. C. Peters and C.H.A. Koster, Uncertainty-based Noise Reduction and Term Selection in Text Categorization (2002), Proceedings 24th BCS-IRSG European Colloquium on IR Research, Springer LNCS 2291, pp 248-267.
16. Rocchio, J.J. (1971). Relevance feedback information retrieval. In: Salton, G. (ed.) *The smart retrieval system—experiments in automatic document processing* p. 313-323. Prentice-Hall, Englewood Cliffs, NJ.
17. F. Sebastiani (2002), Machine learning in automated text categorization, ACM Computing Surveys, Vol 34 no 1, 2002, pp. 1-47.
18. A. Singhal, M. Mitra and C. Buckley (1997). Learning Routing Queries in a Query Zone. Proceedings SIGIR '97, pp. 25-32.

Preconditions for Structural Synthesis of Programs

Vahur Kotkas

Software Department
Institute of Cybernetics at TTU
12618 Tallinn, Estonia
vahur@cs.ioc.ee

Abstract. The current paper presents an extension to the logical language used in Structural Synthesis of Programs (SSP) and describes a modified synthesis algorithm to handle branching in program synthesis. The origin of the extension is from practical experience and introduces statements with preconditions to the logical language of SSP. Using these preconditions one can describe variable domain restrictions in its domain oriented models and gain higher flexibility while doing engineering modeling.

1 Introduction

Program Synthesis is a method of software engineering used to generate programs automatically. It has long been considered as a core AI task. Generally, we can distinguish three approaches to program synthesis: transformational, inductive, and deductive. Already in 60's works of G. Genzten [1], C. Green [2] and R. Waldinger [3] were published that put program synthesis on a firm logical basis.

In this paper we consider a deductive approach of program synthesis, where programs are constructed from existing computational units. These units are given by equations or pre-programmed functions that can be used for program composition without considering their internal behavior [10]. Computational units are described by specifications that present their input and output domains using propositional variables and during the synthesis process only their structural properties are taken into account. Hence, the method is called Structural Synthesis of Programs (SSP).

The usage of SSP is very practically oriented. By using SSP in a programming environment we are able to avoid writing code for each case of execution that we need during data processing; instead we rely on run-time synthesis, which provides the necessary computational units when necessary.

Our aim is to introduce some of the properties of functional programming (like lazy evaluation) into modern commercially accepted programming languages (e.g. Java, C# etc.) by extending their programs with Structural Specifications. In addition these specifications help programmers to understand the behavior and purpose of written programs more easily.

Application area of the program synthesis may vary from mobile handheld devices where we need to synthesize programs suitable for execution in the local environment to large information systems where implementing programs for all possible cases appears unfeasible.

In section 2 we explain the rationale of applying SSP to modern programming languages. Section 3 gives a short overview of SSP and in section 4 we introduce preconditions into the logical language of SSP and define structural synthesis rules to handle them during the proof search process. Section 5 presents proof search strategies of automatic program synthesis and in section 6 we demonstrate SSP at work through a simple example. We provide some concluding remarks and ideas for future work in section 7.

2 Program Synthesis for Modern Programming Languages

Most of the widely used modern programming languages (like Java or C#) do not facilitate automated program synthesis as-is. This is because of a missing semantic connection between the components or variables (later referred to as variables) and methods or functions (we call them functions in the following).

The semantics of libraries – the rules of their usage – are usually described in comments or in a manual (if they are described at all) that are too difficult to parse automatically and it takes much effort to learn them even for a skilled programmer. The typing rules applied in the modern programming languages are not sufficient for program synthesis.

We propose an extension to the programming languages in the form of specifications that are machine and human readable and enable automated (re)use of given functions. These specifications describe which variables can be used in the synthesized program and through which functions they are (inter)related, without looking at their real values, i.e., we take into account only the structural properties of the relations (functions) and hence call them structural specifications.

The idea is to invoke program synthesis from within a running program, synthesize a new program and execute it to perform the needed tasks or calculations and then return to the initial program. The whole process is fully automatic and needs no human intervention.

In the proposed structural specifications variables are defined as:

var *Variablenames* **:** *Type*

and relations as:

rel *Label* **:** *InputParameters* **->** *OutputParameters* {*FunctionName*}

Label gives a name to the relation that is meaningful for debugging, *InputParameters* is a comma-separated list of arguments to the function called *FunctionName* and *OutputParameters* are the results (variables) the function returns. One function can be referred to in many relation specifications, where the input or output parameters vary.

How structural specifications can be incorporated into programming languages is considered in [13] and is not described here. We concentrate on the logics of program synthesis to reduce the level of detail and to make the approach applicable to a number of imperative and/or object-oriented programming languages.

3 Structural Synthesis of Programs in a Nutshell

In the framework of Structural Synthesis of Programs (SSP), propositional variables are used as identifiers for conceptual roles and a propositional logic is applied to solve synthesis problems. This allows us to take advantage of the proof-as-programs property of intuitionistic logic, where program synthesis is equated to proof search [6].

SSP relies on an implicative fragment of intuitionistic propositional calculus, which provides an expressive logical language for problem description and feasible proof search efficiency.

In the following we present the logical language and structural synthesis rules used for proof search as they are described in [5, 9, 14].

3.1 Logical Language

The logical language (LL) of SSP consists of the following formulae:

1. Propositional variables: *A; B; C …*
The propositional variable A corresponds to an object variable a of a source program, and expresses the state of a. A is considered proven if a is evaluated (has a value) or there exists a known way for its computation. A propositional variable will be termed an atom in the following.

2. Unconditional computability statements correspond to a relation specification `rel` `a`$_1$`,…,a`$_n$ `->` `b`$_1$`,…,b`$_m$ `{f}` and are defined in the LL as:

$$\overline{A} \xrightarrow[f]{} \overline{B}$$

An unconditional computability statement presents the fact that $\overline{B}$ is derivable from $\overline{A}$ (we use an abbreviation $\overline{X}$ to denote a conjunction of variables $X_1 \wedge \ldots \wedge X_n$). By this we express the computability of the value of each object variable b_i corresponding to B_i from values of $a_1 \ldots a_n$ corresponding to $A_1 \ldots A_n$. with a single computational step f (by executing a predefined module, calling a method or function f with arguments $a_1 \ldots a_n$). We refer to such computational units as functions.

We would like to stress that at the logical level the statement is equivalent to a set of statements $\overline{A} \xrightarrow[f_i]{} B_i$ and f can be considered as a function composed of a set of subfunctions f_i.

3. Conditional computability statements

$$\overline{(\overline{A} \xrightarrow[g]{} \overline{B})} \wedge \overline{C(c)} \xrightarrow[f(\overline{g},\overline{c})]{} \overline{D}$$

A conditional computability statement expresses computability of $\overline{D}$ from $\overline{C}$ depending on the derivability of $\overline{B}$ from $\overline{A}$. We use an abbreviation $\overline{(\overline{A} \to \overline{B})}$ for $(\overline{A_1} \to \overline{B_1}) \wedge \ldots \wedge (\overline{A_k} \to \overline{B_k})$ also here. In this statement g stands for a function to be synthesized (a subtask), which is used by the body of function f to perform the necessary computations. In other words function f is usable if $\overline{C}$ is proven (c's are evaluated) and g's are synthesized (subtasks are solved).

Such language construction is useful for recursion and loop specification.

4. Computability statements with disjunction

$$W \xrightarrow[f]{} \underline{B}$$

In computability statement with disjunction the antecedent (W) can be the same as in case of unconditional or conditional computability statements. $\underline{B}$ is an abbreviation of a disjunctive normal form $\overline{B_1} \vee \ldots \vee \overline{B_n}$. Computability statements with disjunction allow us to handle possible exceptions thrown by function f or to deal with branching that function f may create.

5. Falsity

$$W \xrightarrow[f]{} \perp$$

In this computability statement W can be the same as in case of unconditional or conditional computability statements. The right-hand side is the falsity constant. This statement enables termination of a synthesized program that may be needed in case of exceptions or branching.

We would like to stress that the formulae of LL represent only the computability of object variables, not their particular values. Each formula of LL is used as an axiom during the proof search.

3.2 Structural Synthesis Rules

The Structural Synthesis Rules (SSR) are the inference rules used to carry out the proof search i.e. to build the proof tree. Proof term derivation (the lambda term construction) is carried along with the deductions. Every propositional variable used in the rules corresponds to an object variable of the programming language and the lambda-terms are the programs to compute their values. Hence, provability and computability have the same meaning here.

The corresponding object variable or lambda-term is provided in the parentheses after the propositional variable.

Let us briefly discuss the inference rules of SSR.

Rule (1) is called Implication Elimination. It permits the application of an axiom to the proof and widens the set of computable variables by B.

$$\frac{\overline{A}(x) \xrightarrow[f]{} B \quad \overline{\Gamma \Rightarrow A(a)}}{\overline{\Gamma} \Rightarrow B\left(f(\overline{a})\right)} \tag{1}$$

a is a realization of the proof of A out of Γ as a term. The following two notations are identical:

$$\Gamma \Rightarrow A \qquad \begin{array}{c} \Gamma \\ \vdots \\ A \end{array}$$

We would like to notice that notation *A(a)* does not lead us to a first order logic but enables the program extraction from the proof.

Rule (2) is called Implication Introduction. This rule creates a new axiom based on an existing proof. The lambda-term shows that in the resulting function a_i's are bound variables. In other words the rule introduces a new function that uses a_i's as inputs and computes the value of *b* out of them. This rule is useful to form realizations of subtasks and in case of synthesis of a new function.

$$\frac{\overline{A(a)} \Rightarrow B(b)}{\overline{A} \xrightarrow[\lambda \overline{a}.b]{} B} \tag{2}$$

Rule (3) is called Double Implication Elimination. Here, computability of *D* depends not only on the set of C_i's, but also on the solvability of subtasks calculating *B* from A_i's. The rule is applicable only in the case the terms b_i are derivable.

$$\frac{\overline{(\overline{A} \to B)}, \overline{C} \xrightarrow[f]{} D \quad \overline{\Gamma, \overline{A(a)} \Rightarrow B(b)} \quad \overline{\Sigma \Rightarrow C(c)}}{\overline{\Gamma}, \overline{\Sigma} \Rightarrow D\left(f(\overline{\lambda \overline{a}.b}, \overline{c})\right)} \tag{3}$$

One can think of the rule consisting of two steps: first introducing an implication A→B (application of Rule 2) and then doing the substitution (passing the right function(s) as argument(s) to *f*).

We can distinguish two kinds of subtasks – dependent and independent. A subtask is called independent when $\Sigma \cap \Gamma = \varnothing$ and dependent otherwise. In practice independent subtasks are often executed on a separate object whereas dependent subtasks use the same context as the main function, even if there is no overlap between the premises of the subtask and the premises of the main function.

Rule (4) is called Disjunction Elimination.

$$\frac{\overline{W} \xrightarrow[f]{} \overline{A} \vee \overline{B} \quad \overline{\Gamma \Rightarrow W(w)} \quad \Sigma, \overline{A} \Rightarrow G(g) \quad \Delta, \overline{B} \Rightarrow G(h)}{\overline{\Gamma}, \Sigma, \Delta \Rightarrow G\left(let\ \sigma = f(\overline{w}),\ in\ case\ type\,\sigma\ match\left\langle type\overline{A} \mapsto g(\sigma) \middle| type\overline{B} \mapsto h(\sigma)\right\rangle\right)} \tag{4}$$

This rule tells us that depending on the type of the variable that function *f* returns we decide with which branch (continuation) we proceed. The new output *G* is called an innermost goal as it is better to find it such that it is reachable with as few computational steps as possible in order to synthesize a compact program.

At the logical level

$$(W \xrightarrow[f]{} A \vee B,\ A \xrightarrow[g]{} G,\ B \xrightarrow[h]{} G)$$

is transformed to

$$(W \wedge (A \xrightarrow[g]{} G) \wedge (B \xrightarrow[h]{} G) \xrightarrow[f']{} G)$$

This gives us instead of an axiom with disjunction a new axiom with as many subtasks as many disjuncts we had in the original axiom. Here we present only the case with two disjuncts, but the transformation to the case with *n* (*n*>2) disjuncts is very straightforward.

The call to original function *f* is encapsulated into the body of a new function *f'*, which contains a case structure to select the subtask to proceed with according to the type of the output that *f* returns. This forms a branching to an executable code. The types of the disjuncts must be different; otherwise it is not possible to detect which branch was selected during the execution of *f*.

Rule (5) is called Falsity Elimination. Falsity is introduced to handle exceptions that can be raised during the execution of the synthesized program and yield to the need of its abortion. Intuitively we can think of the rule as a way to prove the goal *G* at once.

$$\frac{\Gamma \Rightarrow \perp}{\Gamma \Rightarrow G\left(abort^{G}\right)} \perp^{-} \qquad (5)$$

Every application of an elimination rule corresponds to a computational step in a synthesized program. A problem to be solved is formulated as a theorem to be proved. The proofs of soundness and completeness of SSP can be found in [6] and [12] in depth study of a complexity of intuitionistic propositional calculus and a similar proving system is done in [7] and [8].

4 Preconditions for SSP

It may occur that functions covering the whole variable domain are not available, but there are functions meant to work under certain conditions. This could happen for example when empirical models are created based on measurements. This is quite common practice in engineering modeling. Such models are meant to represent the behavior or real world with satisfactory precision in a limited range of variable domain and we need a way to restrict the applicability of axioms. For that we extend the SSP with preconditions.

4.1 Preconditions in Specifications and Corresponding Synthesized Program

In a structural specification a precondition is added to a relation specification as a logical expression:

rel *Label* **:** {*PreCondition*} *InputParameters* -> *OutputParameters* {*ModuleName*}

PreCondition describes the conditions when the relation is applicable. In principle it would be enough to use only one Boolean variable in the *PreCondition* expression and specify it in other statements, but in practice for (human) readability it is much better if the whole expression is presented in the specification.

Some examples of conditional expressions are the following:

```
a<3.5
b!=0 AND a>3.5
FindValue(a,b)<MaxValue()
```

Handling of exceptions that may be thrown during the execution of the logical expression cannot be predefined. In such case it is advisable to specify such computations in separate relation definitions. For example if *FindValue*, from the example logical expressions, can throw an exception then it is better to introduce a new variable *curValue* and introduce a new relation changing the original precondition as follows:

```
rel a,b -> curValue | anException {FindValue}
rel {curValue<MaxValue} ...
```

The aim of using preconditions is to enable automatic construction of if-then-else statements in synthesized programs. For example let us consider the following relations:

```
rel {x<4} a,b -> c {method1}
rel {x==4} a,b -> c {method2}
rel {x>4} a,b -> c {method3}
```

We would like to synthesize an if-then-else statement:

```
if (x<4) then c=method1(a,b)
else if (x==4) then c=method2(a,b)
else c=method3(a,b)
```

However, this is not possible while using propositional calculus as we do not look into the logical expressions and cannot analyze the coverage (whether the whole variable domain is covered) of the expressions. We can synthesize the following if-then-else statement:

```
if (x<4) then c=method1(a,b)
else if (x==4) then c=method2(a,b)
else if (x>4) then c=method3(a,b)
```

There is an issue with the case where we do not have complete coverage of the variable domain i.e. for example we do not have the third declaration in our specification:

```
rel {x<4} a,b -> c {method1}
rel {x==4} a,b -> c {method2}
```

To solve this we have to synthesize a proper program termination for the case *a* is greater than 4 and warn the user about such decision:

```
if (x<4) then c=method1(a,b)
else if (x==4) then c=method2(a,b)
else abort
```

Having this in mind we come to the logical language and structural synthesis rules extensions provided in the next two sections.

4.2 Extended Logical Language

In the extended LL we keep everything that was presented in section 3.1. In addition we define the following formula:

6. Computability statements with precondition

$$\left\langle \overline{A} \xrightarrow[g]{} Bool \right\rangle W \xrightarrow[f]{} D$$

The computability of d corresponding to D depends on inputs W and the Boolean value function g evaluates to. Function f is applicable only in the case g evaluates to *true*. $\left\langle \overline{A} \xrightarrow[g]{} Bool \right\rangle$ is not a subtask as it has the function given in the statement and needs not to be synthesized. We call it the precondition of function f. W can be similarly to statements with disjunction an unconditional or conditional expression.

We would like to stress that g would become evaluated only during the execution of the synthesized program and not during the synthesis process and should be given in a form of Boolean expression of the underlying programming language. Hence, we still remain in the framework of propositional calculus.

4.3 Extended Structural Synthesis Rules

We keep the rules described in section 3.2 and add a new one that handles the computability statements with preconditions:

Rule (6) Precondition elimination.

$$\frac{\left\langle \overline{A} \xrightarrow[g]{} Bool \right\rangle \overline{C} \xrightarrow[f]{} D \quad \overline{\Sigma \Rightarrow A(a)} \quad \overline{\Gamma \Rightarrow C(c)} \quad \Delta, D \Rightarrow G(h) \quad \overline{\Sigma}, \overline{\Gamma}, \Delta \Rightarrow G(u)}{\overline{\Sigma}, \overline{\Gamma}, \Delta \Rightarrow G\left(if\ g(\overline{a})\ then\ h(f(\overline{c}))\ else\ u\right)} \tag{6}$$

Here we use a branching idea similar to the case of disjunction elimination. The axiom can be applied if there exists another possible branch to calculate G. We call that branch a supplementary subtask.

It is advisable to keep the branching local and to collect all possible axioms with preconditions together with their continuations that allow us to compute G, into a single if-then-else structure of a resulting synthesized program. In order to achieve that and still synthesize robust code we need to take care of situations where the whole variable domain is not covered by the applicable axioms with preconditions (the branching is not complete).

To ensure the robustness of resulting program it is required that computability statements with preconditions have complementary falsity statements. The complementary falsity statement should have exactly the same left side (input variables and subtask definitions) without the precondition as the statement with precondition has. It is not necessary to explicitly specify those falsity statements in the specifications; rather the program synthesizer automatically adds them. Complementary falsity statement also ensures the solvability of the supplementary subtask.

5 Proof Search Strategy

The proof search strategy of SSP relies mostly on ideas of Partial Deduction [9]. Proof synthesis algorithm with unconditional and conditional computability statements is presented in [4]. The main proof search strategy of SSP is assumption-driven forward search as well as goal-driven backward search. The assumption-driven forward search is used to select unconditional computability statements. The goal-driven backward search is used to select and solve subtasks if there are no applicable unconditional statements left.

Additionally an algorithm to handle statements with disjunctions is presented in [14]. Some ideas for the proof search distribution are presented in [15]. There are several rules of thumb for search tactics and stopping criteria given in [9] and [12].

5.1 Assumption-Driven Forward Search

In principle there are two kinds of synthesis tasks: first, giving some input variables we need to reach certain goal where the goal is usually a variable or a set of variables we need to calculate. Second, we want to calculate everything possible. The latter is common in engineering modeling where we enter the values we know and want to see what can be calculated and what remains unknown.

In the following we denote by saying that "a variable is evaluated" the fact that a propositional variable of the LL has been proved and "not evaluated" otherwise. We do not refer to object variables here. By saying that a computability statement is applicable we mean that all its input variables are evaluated, at least one of its output variables is not evaluated and its subtasks are solvable.

In the assumption-driven forward search we use the following strategy:

1. Sort the computability statements according to their complexity measure from simpler to harder: 1) unconditional computability statements, 2) conditional computability statements with independent subtasks, 3) conditional computability statements with dependent subtasks, 4) computability statements with disjunctions, 5) computability statements with preconditions, 6) falsity eliminations.
2. Spawn proof searches of independent subtasks as parallel threads to the main proof search. If proof search of subtask returns "unsolvable" then remove all computational statements that have such subtask in their specification from the search space. If proof search of a subtask returns "solvable" then mark in all computational statements that subtask being "solvable".
3. Check whether the goal is achieved, i.e. whether the variables included in the goal are evaluated. If so, stop all parallel threads searching for solutions to independent subtasks. Continue with minimization (see section 5.3). Otherwise restore all "temporarily not applicable" statements to state applicable (if there are any) and continue at 4.
4. Apply simplest applicable computability statement. Evaluate all its output variables. Go back to 3. If there is no applicable statement left and if the search for independent subtasks solutions is still in progress then wait for solutions, otherwise the problem is unsolvable for the first kind of synthesis problems described above.

When testing the applicability of a conditional computability statement with dependent subtasks we have to try solving their subtasks. If any of the subtasks appear being unsolvable we mark that statement "temporarily not applicable". This status is changed whenever a new variable becomes evaluated (see step 3 in the strategy). The subtasks that were solvable are marked accordingly and we do not need to solve them again.

Applicability of statements that cause branching is also dependent on whether we can find proper innermost goal for the branches. In case our synthesis task has a goal consisting of a single variable, it may be used initially also as an innermost goal for these statements [14]. However, in case we have our goal composed of a set of variables or we try to synthesize a program that calculates everything possible, we have to try solving the subtasks for each variable. The latter corresponds to the second synthesis task described above.

In the following we propose a strategy to find an innermost goal:

1. Form a dependent subtask for each branch. That is, we create a group of copies of the search space where a copy is made for each branch. In each search space we additionally evaluate variables corresponding to that branch (disjunct or output variables of the statement with precondition) and exclude the statement(s) that formed the group. In case of computability statement with disjunction the branches correspond to the disjuncts, in case of computability statements with preconditions the group is formed of all applicable statements with precondition.
2. Solve each subtask.
3. Find common patterns in the solutions. A common pattern can be for example a common statement that is applied or a common variable that becomes evaluated. The variable or the output of the statement that is present in each member of the group is declared as an innermost goal.
4. In the case of statements with preconditions the largest subset is selected from the group that has common pattern. The rest of statements are excluded. This is because we may search for an if-then-else structure at the moment where several are possible and we would like to detect only one of them at the time.
5. Minimization is used to form the actual solutions of the subtasks.
6. In the case there is no common pattern the problem is declared unsolvable for the first kind of synthesis task and a branching, that cannot be minimized, is synthesized in the second kind.

In order to take advantage of parallel processing in the main proof search process we propose an addition to the proof search strategy.

The main search task is divided into subsets and spawned in separate threads. The subsets can be selected based on the object structure of the program for example, i.e, we form a separate search space out of each component of the main object. In these parallel threads depth-first search is used. In practice there is no need to go deeper than 5 levels in the object hierarchy. This gives us additional termination criteria to avoid stepping into a well of recursion.

The main search process uses breadth-first search at the same time. When a thread finishes its work it returns its results to the main process that incorporates the statements applied to its search space and also evaluates the evaluated variables.

In the worst case the whole search space must be covered iteratively. This leads us to the theoretical PSPACE complexity. However, in practice it is usually much lower [14].

5.2 Goal-Driven Backward Search

The goal-driven backward search is used to solve subtasks. This is done using practically the same algorithm as in case of assumption-driven forward search, but it starts from the goal and moves toward assumptions. This approach is better for solving subtasks as usually then the goal contains only few variables and narrows the initial set of applicable statements.

5.3 Minimization

Minimization is used to exclude computability statements from the synthesized algorithm that are not needed to reach the goal. Moving in the opposite direction to the one used during the search can easily do this. That is, if we used assumption-driven forward search, then the minimization algorithm should work from the goal towards assumptions and in case of goal-driven backward search from assumptions toward goal.

6 An Example of Program Synthesis

Let us consider a triangle problem as an example. The problem specification is the following:

1. We have a triangle (see Fig. 1) with its side *as* and its internal angles *ca* and *aa* given.
2. Our goal is to calculate the area of the triangle.

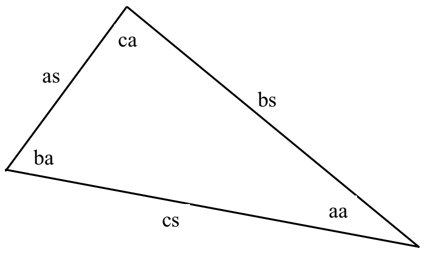


Fig. 1. Triangle problem

We create the following problem specification (see Fig. 2) describing its object variables and computability statements. We put all our knowledge about triangles into the specification. At this point we apologize for our imperfect knowledge on triangle geometry. We have implemented two functions *calcAreaRA* and *calcAreaSA*, first of which is applicable in the case the triangle is right-angled (*ba*=90°) and the other when *aa*<90°.

The var statements define object variables of the problem domain. When moving to the domain of logical language we define corresponding propositional variables for each object variable: *AS*(*as*), *BS*(*bs*), etc.

The rel statements specify the computability statements where braces enclose the preconditions and the function names. The labels (*K:*, *L:*, *M:*, *Q:*) on the left side of the specifications do not have any affect on proof search process. They are only used to simplify the understanding of proof search process below and to improve readability of the algorithm synthesized.

```
var as,bs,cs : side
var aa,ba,ca : angle
var s : area
rel K: aa+ba+ca=180
rel L: {ba==90} as,cs -> s {calcAreaRA}
rel M: {aa<90} aa,bs,cs -> s {calcAreaSA}
rel Q: as/sin(aa)=cs/sin(ca)
```

Fig. 2. Declarative specification of class Triangle

The synthesis problem is given in the form [*as*, *aa*, *ca* → *s*]. That means propositional variables *AS*, *AA*, *CA* are evaluated in the beginning of proof search process and the goal is *S*.

Using the proof search strategies provided above we build a proof tree for the goal *S* and extract a program for its computation:

$$\dfrac{\dfrac{\dfrac{AA,CA \xrightarrow[K']{} BA \quad AA(aa)\ CA(ca)}{AA,CA \Rightarrow BA(K'(aa,ca))}(1) \quad \dfrac{AS,AA,CA \xrightarrow[Q']{} CS \quad AS(as)\ AA(aa)\ CA(ca)}{AS,AA,CA \Rightarrow CS(Q'(as,aa,ca))}(1)}{\langle ba==90\rangle AS,CS \xrightarrow[calcAreaRA]{} S \quad AA,CA \Rightarrow BA(ba) \quad AS(as) \quad AS,AA,CA \Rightarrow CS(cs) \quad *}(6)}{\dfrac{AA,CA,AS \Rightarrow S(if\ (ba==90)\ then\ calcAreaRA(as,cs)\ else\ h)}{AS,AA,CA \xrightarrow[\lambda as,aa,ca.if\,(ba==90,calcAreaRA(as,cs),h)]{} S}(2)}$$

The proof tree above continues from * below.

$$\dfrac{\langle aa<90\rangle AA,BS,CS \xrightarrow[calcAreaSA]{} S \quad AA(aa) \quad BS(bs) \quad CS(cs) \quad \dfrac{AS,BS,CS \Rightarrow \perp}{AA,BS,CS \Rightarrow S(abort)}(5)}{\dfrac{AA,BS,CS \Rightarrow S(if\ (aa<90)\ then\ calcAreaSA(aa,bs,cs)\ else\ abort)}{*\ AA,CA,BS,BA,AS,CS \xrightarrow[\lambda aa,bs,cs.if\,(aa<90,calcAreaSA(aa,bs,cs),abort)]{} S(h)}(2)}(6)$$

From here (the lambda terms) we can extract the algorithm (a sequence of methods and language constructions to be applied) that solves the problem (see Fig. 3). Program derivation from the algorithm is straightforward and is not presented here.

```
K: aa,ca -> ba {K'}
Q: as,aa,ca -> cs {Q'}
if (ba==90)
 then
  L: as,cs -> s {CalcAreaRA}
else if (aa<90)
 then
  M: aa,bs,cs -> s {CalcAreaSA}
else
  abort
Return s
```

Fig. 3. Synthesized algorithm

7 Concluding Remarks and Future Work

In this paper we presented an extension to the Structural Synthesis of Programs (SSP) in order to better meet the needs on program synthesis for engineering modeling tasks. We introduced computability statements with preconditions to the logical language and added a new rule for structural synthesis rules.

A proof search strategy is presented. The main emphasis is set on handling the computability statements that cause branching; the remaining parts remain similar to the algorithms described earlier. Some ideas on synthesized program minimization are also presented.

An interesting continuation to the current work would be synthesis of loops. In principle we have everything necessary for their specifications available: preconditions as guards and other statements for body. However it is unclear whether it is doable by using propositional calculus.

Parallelization of the synthesis process as well as synthesis of parallel programs is another promising field for future work. Recent results on Disjunctive Horn Linear Logic [16] show availability of efficient methods to synthesize a parallel program for fixed number of processors. With some modifications it should be applicable also in the framework of SSP.

References

1. Szabo, M. E. Collected Papers of Gerhard Gentzen. North-Holland (1969).
2. Green, C. C. Application of theorem proving to problem solving. In Proceedings IJCAI '69 (1969). 219--240.
3. Waldinger, R. J. Constructing programs automatically using theorem proving. Ph.D. Thesis, 1969, Carnegie-Mellon U., Pittsburgh, Pa.
4. Harf, M., Tyugu, E. Algorithms of structured synthesis of programs. Programming and Computer Software, 6 (1980), pp 165-175.
5. Tyugu, E. The structural synthesis of programs, LNCS 122 (1981), pp. 290-303.
6. Mints, G., Tyugu, E. Justification of the Structural Synthesis of Programs. Science of Computer Programming, 2(3) (1982): 215-240.
7. Mints, G. Complexity of subclasses of the intuitionistic propositional calculus. Special issue: Selected Papers form Workshop on Programming Logic, Bästad, Sweden, 21-26 May 1989.
8. Kanovich, M. I. 1991. Efficient program synthesis: Semantics, logic, complexity. In T. Ito and A. R. Meyer, editors, Theoretical Aspects of Computer Software, LNCS 526, Springer 1991, pp. 615-632.
9. Matskin, M., Komorowski, J. Partial Structural Synthesis of Programs, Fundamenta Informaticae, IOS Press, 31(1997) pp. 125-144.
10. Tyugu, E. On the border between functional programming and program synthesis. Proc. Estonian Acad. Sci. Engng. (1998), pp. 119-129.
11. Harf, M., Kindel, K., Kotkas, V., Küngas, P., Tyugu, E. Automated Program Synthesis for Java Programming Language. LNCS 2244, Springer 2001, pp. 157-164
12. Matskin, M., Tyugu, E. Strategies of Strucutral Synthesis of Programs and Its Extensions. Computing and Informatics, Vol. 20 (2001), 1-25.
13. Kotkas, V. A distributed program synthesizer. Acta Cybernetica 15 (2002), pp. 567-581.
14. Lämmermann, S. Runtime service composition via logic-Based Program Synthesis. PhD Thesis. Royal Institute of Technology, Stockholm, Sweden. Technical Report TRITA-IT AVH 02:03, ISSN 1403-5286, ISRN KTH/IT/AVH-02/03-SE (2002).
15. Kotkas, V. Synthesis of Distributed Programs. In Proceedings of the Eighth Symposium on Programming Languages and Software Tools. Kuopio 2003.
16. Kanovich , M., Vauzeilles, J. Coping Polynomially with Numerous but Identical Elements within Planning Problems. LNCS 2803 (2003), pp. 285-298.

How to Verify and Exploit a Refinement of Component-Based Systems*

Olga Kouchnarenko[1] and Arnaud Lanoix[1,2]

[1] LIFC, FRE 2661 CNRS, Besançon, France
{kouchna,lanoix}@lifc.univ-fcomte.fr
[2] LORIA, INRIA Lorraine & CNRS, Nancy, France
Arnaud.Lanoix@loria.fr

Abstract. In order to deal with the verification of large systems, compositional approaches postpone in part the problem of combinatorial explosion during model exploration. The purpose of the work we present in this paper is to establish a compositional framework in which the verification may proceed through a refinement-based specification and a component-based verification approaches.

First, a constraint synchronised product operator enables us an automated compositional verification of a component-based system refinement relation. Secondly, safety LTL properties of the whole system are checked from local safety LTL properties of its components. The main advantage of our specification and verification approaches is that LTL properties are preserved through composition and refinement.

Keywords: component-based systems, modules, refinement, LTL properties, composition, verification.

1 Introduction

Nowadays, formal methods are used in various areas, from avionics and automatic systems to telecommunication, transportation and manufacturing systems. However, the increasing size and complexity of these systems make their specification and verification difficult. Compositional reasoning is a way to master this problem.

The purpose of the work we present in this paper is to establish a compositional framework in which an algorithmic verification of a refinement of component-based systems by model exploration of components can be associated with the verification of LTL properties. In our compositional framework, we give ways (see Fig. 1) to preserve LTL properties through:

1. The composition operator for preserving safety LTL properties, meaning that a property satisfied by a separate component is also satisfied by a whole component-based system.

* Work partially funded by the French Research ACI *Geccoo*.

2. The refinement relation for preserving both safety and liveness LTL properties, meaning that a property established for an abstract system model is ensured when the system is refined to a richer level of details.

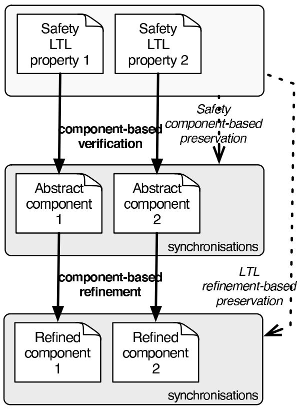


Fig. 1. Verification Principle

To achieve the goal of compositional verification and to model synchronous and asynchronous behaviours of components, we define two operators: a composition of the modules and a constraint synchronised product of transition systems.

We show that the modules [12,13,2] – subsystems sharing variables – whose composition is often used in a concurrent setting, are suitable to compositionally verify a kind of τ-simulation, called the weak refinement. Unfortunately, this model does not allow analysing the strict refinement – a divergence-sensitive completed τ-simulation – from the separate refinements of its modules. That is why we introduce a constraint synchronised product operator. Moreover, the semantics of the component-based systems using this operator makes it possible to verify the strict refinement more efficiently.

The main result of this paper is the theorem claiming that the strict refinement of a component-based system can be established by checking the weak refinement of its expanded components viewed as the modules. The main advantage of the component-based refinement we have been developing is that it allows us to master the complexity of specifications and verifications with a step by step development process without building the whole system. All steps of our compositional approach have been implemented in an analysis tool called SynCo [9].

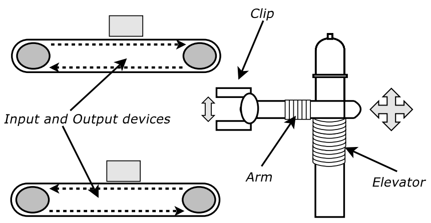


Fig. 2. Production Cell

The main concepts of the paper are illustrated on an example of a simple controller of a production cell moving pieces from an input device to an output device. A pictorial representation of this running example is given in Fig.2. The cell is composed of an arm having horizontal moves, a clip, and an elevator moving vertically. Sensors notify the controller about the production cell changes.

This paper is organised as follows. After giving preliminary notions, we recall in Section 2, the semantics of our refinement relation and its properties. Then Section 3 presents the modules, their composition and the weak refinement of the composition of the modules, called modular refinement. In Section 4, the constraint synchronised product is introduced to specify component-based systems, and the modular refinement is used to establish the strict refinement of component-based systems more efficiently.

2 Preliminaries: LTS Refinement and LTL Properties

We introduce labelled transition systems to specify component behaviours and properties. We then recall a notion of system top-down refinement preserving for the refined system the abstract system LTL properties.

2.1 Preliminaries

Transition systems we consider are interpreted over a finite set of variables V. Let AP_V be a set of atomic propositions over V.

Definition 1 (Labelled Transition System (LTS)). *A LTS S is a tuple (Q, Q_0, E, T, V, l) where*
- *Q is a set of states,*
- *$Q_0 \subseteq Q$ is a set of initial states,*
- *E is a finite set of transition labels,*
- *$T \subseteq Q \times E \times Q$ is a labelled transition relation,*
- *V is a set of variables, and*
- *$l : Q \rightarrow 2^{AP_V}$ is a total function that labels each state with the set of atomic propositions true in that state; we call l the interpretation.*

We consider a (finite or infinite) sequence of states $\sigma = q_0, q_1, \ldots$[1] in Q. σ is a path of S iff $\forall i.(i \geq 0 \Rightarrow \exists e_i.(e_i \in E \wedge (q_i, e_i, q_{i+1}) \in T))$[2]. Given a path σ, we denote by $tr(\sigma)$ its trace $e_0, e_1, \ldots$. $\Sigma(S)$ designates the set of paths of S. A state q_i is reachable from q_0 iff there exists a path of the form $\sigma = q_0, q_1, \ldots, q_i$.

In this paper, dynamic properties of systems are expressed by formulae of propositional Linear Temporal Logic (LTL) [15] given by the following grammar: $\phi,\phi' ::= ap \mid \phi \vee \phi' \mid \neg\phi \mid \bigcirc\phi \mid \phi\mathcal{U}\phi'$, where $ap \in AP_V$.

Definition 2 (LTL semantics). *Given LTL properties ϕ,ϕ' and a path σ, we define ϕ to be satisfied at $i \geq 0$ on a path σ, written $(\sigma, i) \models \phi$, as follows.*
- *$(\sigma, i) \models ap$ iff $ap \in l((\sigma, i))$*
- *$(\sigma, i) \models \neg\phi$ iff it is not true that $(\sigma, i) \models \phi$*
- *$(\sigma, i) \models \phi \vee \phi'$ iff $(\sigma, i) \models \phi$ or $(\sigma, i) \models \phi'$*
- *$(\sigma, i) \models \bigcirc\phi$ iff $(\sigma, i+1) \models \phi$*
- *$(\sigma, i) \models \phi\mathcal{U}\phi'$ iff $\exists j.(j \geq i \wedge (\sigma, j) \models \phi' \wedge \forall k.(i \leq k < j \Rightarrow (\sigma, k) \models \phi))$*

We also use the notations $\Diamond\phi \equiv true\mathcal{U}\phi$, $\Box\phi \equiv \neg\Diamond\neg\phi$, and $\phi\mathcal{W}\phi' \equiv \Box\phi \vee \phi\mathcal{U}\phi'$. A LTL property ϕ is satisfied by S when $\forall\sigma.(\sigma \in \Sigma(S) \wedge (\sigma, 0) \in Q_0 \Rightarrow (\sigma, 0) \models \phi)$.

Moreover, we often consider the set $SP_V \stackrel{\text{def}}{=} \{sp, sp_1, sp_2, \ldots\}$ of state propositions over V defined by $sp, sp' ::= ap \mid sp \vee sp' \mid \neg sp$, where $ap \in AP_V$. In

[1] (σ, i) designates the i^{th} state of σ.

[2] An element of T is also denoted $q \xrightarrow{e} q'$.

this setting, an *invariance property* is a proposition $sp \in SP_V$ satisfied by every state of S, i.e. $\forall q.\ (q \in Q \Rightarrow q \models sp)$, written $S \models sp$.

To handle a product of LTSs, the satisfaction of a state proposition sp by a state is extended to tuples of states. For example, a formula $sp \in SP_{V_1 \cup V_2}$ is satisfied by (q_1, q_2) iff either $sp \in SP_{V_1}$ and $q_1 \models sp$, or $sp \in SP_{V_2}$ and $q_2 \models sp$.

2.2 Refinement

In this paper, we exploit the system top-down refinement relation we have introduced in [5]. Let $SA = (Q_A, Q_{0A}, E_A, T_A, V_A, l_A)$ be an abstract LTS and $SR = (Q_R, Q_{0R}, E_R, T_R, V_R, l_R)$ a more detailed LTS. The syntactic features of the refinement are as follows. First, refinement introduces new actions, so $E_A \subseteq E_R$. Second, some new variables can be introduced and the old ones are renamed: $V_A \cap V_R = \varnothing$. Third, a propositional calculus formula gp over $V_A \cup V_R$, called *gluing predicate*, links variables of both LTSs.

Definition 3 (Gluing relation). *Let $gp \in SP_{V_A \cup V_R}$ be a gluing predicate. The gluing relation $\mu \subseteq Q_R \times Q_A$ is defined w.r.t. to gp: a state $q_R \in Q_R$ is* glued *to $q_A \in Q_A$ by gp, written $q_R\ \mu\ q_A$, iff $(q_A, q_R) \models gp$.*

The refinement relation with the semantic features below is a restriction of μ.

1. The transitions of SR, the labels of which are in E_A (i.e. labelled by the "old" labels) are kept.
2. New transitions introduced during the refinement design (i.e. labelled in $E_R \smallsetminus E_A$) are considered as being non-observable; they are labelled by τ and called τ-transitions in the system SR.
3. Moreover, new transitions should not introduce new deadlocks.
4. Finally, new transitions should not take control forever. So, infinite sequences of τ-transitions, i.e. τ-cycles, are forbidden.

Definition 4 (Refinement Relation). *Let SA and SR be two LTSs, and $e \in E_A$. Let μ be the gluing relation. The refinement relation $\eta \subseteq Q_R \times Q_A$ is defined as the greatest binary relation included in μ and satisfying the following clauses:*
1) strict transition refinement $(q_R\ \eta\ q_A\ \wedge\ q_R \xrightarrow{e} q'_R \in T_R) \Rightarrow \exists q'_A.\ (q_A \xrightarrow{e} q'_A \in T_A\ \wedge\ q'_R\ \eta\ q'_A)$,
2) stuttering transition refinement $(q_R\ \eta\ q_A \wedge q_R \xrightarrow{\tau} q'_R \in T_R) \Rightarrow (q'_R\ \eta\ q_A)$,
3) lack of new deadlocks $(q_R\ \eta\ q_A\ \wedge\ q_R \not\rightarrow) \Rightarrow (q_A \not\rightarrow)$[3],
4) lack of τ-cycles $q_R\ \eta\ q_A \Rightarrow (\forall \sigma.\ \sigma \in \Sigma(q_R)\ \Rightarrow\ tr(\sigma) \neq \tau^\omega)$.

The relation η is a partial order. ¿From now on, we say that SR *refines* SA, written $SR \sqsubseteq_\eta SA$, when $\forall q_R.(q_R \in Q_R \Rightarrow \exists q_A.\ (q_A \in Q_A\ \wedge\ q_R\ \eta\ q_A))$.

It has been shown in [5] that the refinement relation can be classified as a *divergence stability respecting completed simulation* in the van Glabbeek' spectrum [19]. Consequently, it is more expressive than the trace inclusion, and hence the closed systems refinement [18] by Shankar. An algorithmic verification of the

[3] We note $q \not\rightarrow$ when $\forall q', e.\ (q' \in Q \wedge e \in E \Rightarrow q \xrightarrow{e} q' \notin T)$.

relation η can be done by model exploration of the refined system which complexity order is $O(|SR|)$ where $|SR| = |Q_R| + |T_R|$.

Figure 3 gives a refinement of a part of the controller. In the abstract $Clip_A$ system, the sensor has two positions, *open* and *close*, whereas in the refined $Clip_R$ system, there are two more positions, *toOpen* and *toClose*. The relation η between $Clip_R$ and $Clip_A$ is established, so $Clip_R \sqsubseteq_\eta Clip_A$.

The refinement relation η being a kind of τ-simulation, it preserves safety properties [17,6]. Moreover, we have shown in [8] that the refinement relation η preserves LTL – safety and liveness – properties. In other words, any abstract LTL property satisfied by an abstract system SA is, modulo gp, satisfied by a corresponding refined system SR. Here the satisfaction relation $\models_{gp}$ taking gp into account, can be defined by induction on the structure of a formula ϕ like $\models$ in Definition 2. For example, $sp_A \in SP_{V_A}$ is satisfied by a state $q_R \in Q_R$, modulo gp, written $q_r \models_{gp} sp_A$, iff $\bigwedge_{ap \in l_R(q_R)} ap \wedge gp \Rightarrow sp_A$.

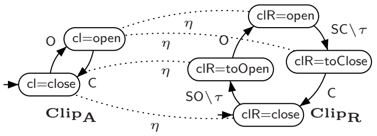


Fig. 3. $Clip_R \sqsubseteq_\eta Clip_A$

Theorem 1 (LTL Component-based Preservation [8]). *Let SA and SR be two LTSs. Let gp be their gluing predicate, and ϕ_A an abstract LTL property. If T_A is total then*

$$\frac{SR \sqsubseteq_\eta SA,\ SA \models \phi_A}{SR \models_{gp} \phi_A}$$

otherwise, only safety LTL properties are preserved.

To handle a product of LTSs which brings about new deadlocks and cycles of τ-transitions, the weak refinement relation is defined by

Definition 5 (Weak Refinement Relation). *Let SA and SR be two LTSs, μ the gluing relation. Let $D \subseteq Q_R$ be the deadlock set. The weak refinement $\rho \subseteq Q_R \times Q_A$ is the greatest binary relation included in μ and satisfying the following clauses:*

***1) strict refinement** and **2) stuttering refinement** from definition 4,*
***3') old or new deadlocks** $((q_R\ \rho\ q_A \wedge\ q_R \not\rightarrow) \Rightarrow ((q_A \not\rightarrow \wedge q_R \notin D) \vee (q_A \rightarrow \wedge q_R \in D)))$*

We say that SR *weakly refines* SA, written $SR \sqsubseteq_\rho^D SA$, when $\forall q_R.(q_R \in Q_R \Rightarrow \exists q_A.\ (q_A \in Q_A\ \wedge\ q_R\ \rho\ q_A))$. Like η, the weak refinement relation ρ is a kind of τ-simulation too. Consequently, ρ preserves safety LTL properties.

The definition above does not mention the τ-cycles. Let $div^\tau(SR, SA)$ be a predicate meaning that SR contains some τ-cycles w.r.t. SA. It is easy to see that the refinement and the weak refinement are linked by

Property 1 (Refinement vs. Weak Refinement).
$SR \sqsubseteq_\eta SA$ iff $SR \sqsubseteq_\rho^D SA$ et $D = \varnothing$ et $\neg\ div^\tau(SR, SA)$.

In the rest of the paper, we use properties proven in [14].

3 Modular Refinement

3.1 Modules

In a concurrent setting, systems are often modelled using a parallel composition of subsystems sharing variables, called *modules* in [12,13,2]. We consider the modules sharing global variables from the set V. Let M^1 and M^2 be two modules. We introduce the parallel composition of M^1 and M^2, denoted $M^1 \| M^2$, which is a module that has exactly the behaviours of M^1 and M^2. *Local* behaviours of M^1 (resp. M^2) are the transitions labelled in $E^1 \setminus E^2$ (resp. $E^2 \setminus E^1$), whereas *global* behaviours are the transitions labelled in $E^1 \cup E^2$.

Definition 6 (Parallel Composition). *Let* $M^1 = (Q^1, Q_0^1, E^1, T^1, V, l^1)$ *and* $M^2 = (Q^2, Q_0^2, E^2, T^2, V, l^2)$ *be two modules. Their composition is defined by* $M^1 \| M^2 = (Q, Q_0, E, T, V, l)$*, where*
- $Q = Q^1 \cup Q^2$,
- $Q_0 = Q_0^1 \cup Q_0^2$,
- $E = E^1 \cup E^2$,
- $T = T^1 \cup T^2$,
- $\forall q \in Q^1 \cup Q^2, l(q) = \begin{cases} l^1(q), \text{ if } q \in Q^1 \\ l^2(q), \text{ if } q \in Q^2 \end{cases}$

Notice that the above definition contains no procedure to obtain modules M^1 and M^2; it is used to prove the component-based refinement in Section 4.

Figure 4 gives two modules M_{Clip} and M_{Arm} of the controller. They interact by modifying the global variables *cl* and *ar* to give the parallel composition $M_{Clip} \| M_{Arm}$.

Property 2 (Commutativity of $\|$*).*
$\forall M^1, M^2.\ M^1 \| M^2 = M^2 \| M^1$

Property 3 (Associativity of $\|$*).*
$\forall M^1, M^2, M^3. (M^1 \| M^2) \| M^3 = M^1 \| (M^2 \| M^3)$

Property 4 (Invariance Preservation of $\|$*).*
Given $sp \in SP_V$, if $M^1 \models sp$ and $M^2 \models sp$ then $M^1 \| M^2 \models sp$.

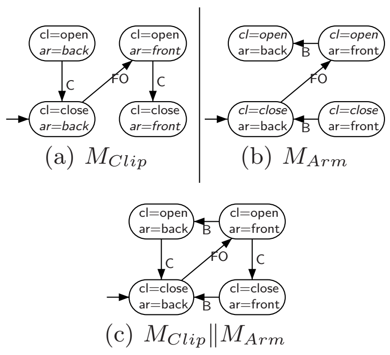


Fig. 4. Modules

The above property can be extended to *dynamic invariants*, that are LTL properties of the form $\Box(sp_1 \Rightarrow sp_2)$.

3.2 Modules vs. Weak Refinement

Instead of verifying the refinement of the composition of the modules, we propose verifying the refinement of each module separately reaching a conclusion about the refinement of their parallel composition automatically. Unfortunately, the strict refinement relation cannot be compositionally established because of interleaving of τ-transitions in the modules composition as illustrated in Fig. 5.

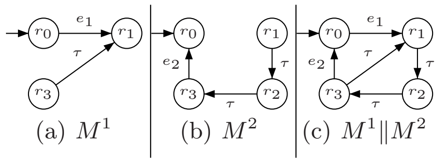


Fig. 5. Interleaving of τ-transitions

Let us examine the weak refinement relation clauses. It is easy to see that the τ-simulation (the strict transition refinement and the stuttering transition refinement) can be compositionally verified since modules only use shared global variables in V. For compositional deadlock checking, the idea is as follows. Suppose that $MR^1 \sqsubseteq_\rho^{D_1} MA^1$ and $MR^2 \sqsubseteq_\rho^{D_2} MA^2$. A state q_r in $D_1 \cup D_2$ is a deadlock in $MR^1\|MR^2$ iff

– q_r is a deadlock in both modules: $q_r \in D_1 \cap D_2$;
– q_r is a deadlock in a module and not a state in the other one: $q_r \in D_1 \setminus Q_R^2$ or $q_r \in D_2 \setminus Q_R^1$.

This deadlock reduction, denoted $D_1 \triangle D_2$, can be computed by

$$D_1 \triangle D_2 = (D_1 \cap D_2) \cup (D_1 \setminus Q_R^2) \cup (D_2 \setminus Q_R^1) \tag{1}$$

Property 5 (Associativity of $\triangle$). $(D_1 \triangle D_2) \triangle D_3 = D_1 \triangle (D_2 \triangle D_3)$.

Now we are ready to establish – in a compositional manner – the weak refinement of the composition of the modules.

Theorem 2 (Modular Refinement). *Let $MA^1\|MA^2$ and $MR^1\|MR^2$ be modules compositions. One has*

$$\frac{MR^1 \sqsubseteq_\rho^{D_1} MA^1, \quad MR^2 \sqsubseteq_\rho^{D_2} MA^2}{MR^1\|MR^2 \sqsubseteq_\rho^{D_1 \triangle D_2} MA^1\|MA^2}$$

Proof is in Appendix A.1. Theorem 2 can be generalised to n modules thanks to Property 3 and Property 5.

4 Component-Based Refinement

The model of the modules is not well-adapted to verify the strict refinement in a compositional way. To this end, a constraint synchronised product is introduced allowing specifying the synchronised behaviours of components.

4.1 Component-Based System

Let consider independent interacting components. The whole component-based system is a rearrangement of its separate part, i.e. the components and their interactions. Let '$-$' denote the fictive action "skip". To specify interactions between components, a synchronisation set *syn* is defined.

Definition 7 (Synchronisation Set). *Let S^1 and S^2 be two components. A synchronisation set syn is a subset of* $\{(e_1, e_2)/sp \mid e_1 \in E^1 \cup \{-\} \wedge e_2 \in E^2 \cup \{-\} \wedge sp \in SP_{V_1 \cup V_2}\}$.

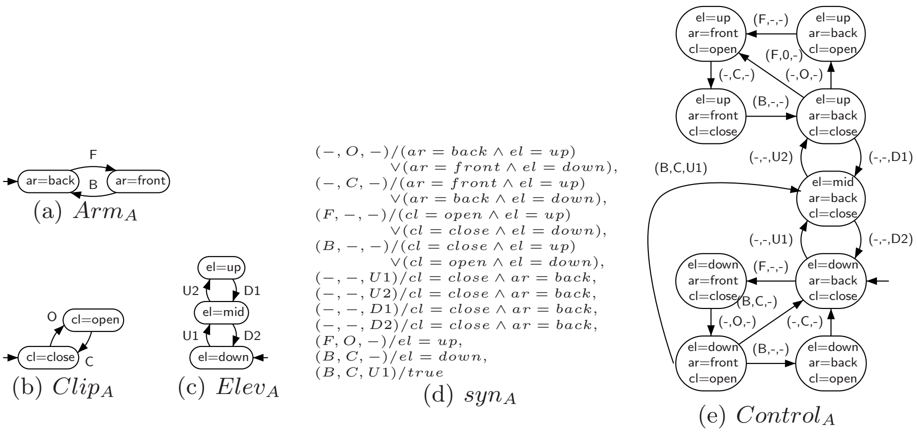
In words, the set *syn* contains tuples of labels (e_1, e_2) with feasibility conditions *sp* scheduling the behaviours of components. Given the whole system (S^1, S^2, syn), the rearrangement of its parts is described by

Definition 8 (Constraint Synchronised Product). *Let* (S^1, S^2, syn) *be a component-based system. The constraint synchronised product* $S^1 \times_{syn} S^2$*, is the tuple* (Q, Q_0, E, T, V, l) *where*

- $Q \subseteq Q^1 \times Q^2$,
- $Q_0 \subseteq Q_0^1 \times Q_0^2$,
- $V = V^1 \cup V^2$,
- $E = \{(e_1, e_2) \mid (e_1, e_2)/sp \in syn\}$,
- $l((q_1, q_2)) = l^1(q_1) \cup l^2(q_2)$,
- $T \subseteq Q \times E \times Q$ *is obtained by :*

[8.1] $(q_1, q_2) \xrightarrow{(e_1,-)} (q_1', q_2) \in T$ *if* $(e_1, -)/sp \in syn$, $q_1 \xrightarrow{e_1} q_1' \in T^1$ *and* $(q_1, q_2) \models sp$,
[8.2] $(q_1, q_2) \xrightarrow{(-,e_2)} (q_1, q_2') \in T$ *if* $(-, e_2)/sp \in syn$, $q_2 \xrightarrow{e_2} q_2' \in T^2$ *and* $(q_1, q_2) \models sp$, *or*
[8.3] $(q_1, q_2) \xrightarrow{(e_1,e_2)} (q_1', q_2') \in T$ *if* $(e_1, e_2)/sp \in syn$, $q_1 \xrightarrow{e_1} q_1' \in T^1$, $q_2 \xrightarrow{e_2} q_2' \in T^2$ *and* $(q_1, q_2) \models sp$

For our running example, Fig. 6 presents the components Arm_A, $Clip_A$ and $Elev_A$, the synchronisation set syn_A, and the computed entire system $Control_A$ $= (Arm_A, Clip_A, Elev_A, syn_A)$.



(a) Arm_A

(b) $Clip_A$

(c) $Elev_A$

$(-, O, -)/(ar = back \wedge el = up)$
$\quad \vee(ar = front \wedge el = down)$,
$(-, C, -)/(ar = front \wedge el = up)$
$\quad \vee(ar = back \wedge el = down)$,
$(F, -, -)/(cl = open \wedge el = up)$
$\quad \vee(cl = close \wedge el = down)$,
$(B, -, -)/(cl = close \wedge el = up)$
$\quad \vee(cl = open \wedge el = down)$,
$(-, -, U1)/cl = close \wedge ar = back$,
$(-, -, U2)/cl = close \wedge ar = back$,
$(-, -, D1)/cl = close \wedge ar = back$,
$(-, -, D2)/cl = close \wedge ar = back$,
$(F, O, -)/el = up$,
$(B, C, -)/el = down$,
$(B, C, U1)/true$

(d) syn_A

(e) $Control_A$

Fig. 6. $(Arm_A, Clip_A, Elev_A, syn_A)$: Components and Synchronisation Set

Definition 8 can be easily extended to n components. Notice that the synchronised product above is more expressive than the well-known synchronised product by Arnold and Nivat [4,3] because of feasibility conditions. Indeed, each transition of our product operator can involve either joint transitions of components or single transition of one component.

Notice that there is a τ-simulation between the whole system and a component. Consequently, the constraint synchronised product preserves safety LTL properties from local components to the component-based system. Furthermore, the conjunction of local safety LTL properties is ensured for the entire system.

Property 6 (Safety LTL Component-based Preservation). Let ϕ_1 and ϕ_2 be safety LTL properties. If $S^1 \models \phi_1$ and $S^2 \models \phi_2$ then $(S^1, S^2, syn) \models \phi_1 \wedge \phi_2$

In addition, every LTL property being the conjunction of a safety property and a liveness property [1], its safety part is preserved by our product operator.

4.2 Component-Based System vs. Modules

Each component is a context-free component. However, for the compositional refinement verification, its environment has to be taken into account. For that purpose, we define an *expanded* component, that is a component in the context of the other components.

Definition 9 (Expanded component). *Let (S^1, S^2, syn) be a component-based system. The expanded component $[S^1]$ corresponding to S^1 is defined by*

$$[S^1] \overset{def}{=} S^1 \times_{[syn]_{S^1}} S^2$$

where $[syn]_{S^1} \overset{def}{=} \{(e_1, e_2)/sp \mid ((e_1, e_2)/sp) \in syn \wedge e_1 \in E^1 \wedge e_2 \in E^2 \cup \{-\}\}$

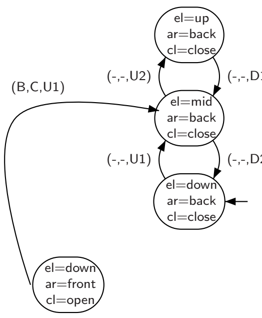


Fig. 7. $[Elev_A]$

In the previous definition, the synchronisation set is restricted to conserve only behaviours involving the considered component. The expanded component $[S^2]$ is similarly defined. Notice that both expanded components are modules (cf. Section 3) defined over the same set of global variables $V^1 \cup V^2$. The parallel composition of these modules gives rise to the whole component-based system.

Property7 (Component-based System vs. Modules). $(S^1, S^2, syn) = S^1 \times_{syn} S^2 = [S_1] \parallel [S_2]$

Figure 7 gives the expanded component $[Eleve_A]$ computed from $(Arm_A,\ Clip_A,\ Elev_A, syn_A)$. To illustrate Property 7 the other expanded components $[Arm_A]$ and $[Clip_A]$ can be built, and we have the whole system $[Arm_A] \parallel [Clip_A] \parallel [Elev_A]$ without building it.

4.3 Component-Based System vs. Refinement

We show now that the constraint synchronised product semantics makes it possible to compositionally verify the strict refinement relation; notably, by resolving the interleaving of τ-transitions and reducing deadlocks more efficiently.

Let (SA^1, SA^2, syn_A) and (SR^1, SR^2, syn_R) be two component-based systems. The lack of τ-cycles in the whole component-based system can be derived from its local checking for each component separately.

Property 8 (Lack of τ-cycles). If $\neg\, div^{\tau}(SR^1, SA^1)$ and $\neg\, div^{\tau}(SR^2, SA^2)$ then $\neg\, div^{\tau}((SR^1, SR^2, syn_R), (SA^1, SA^2, syn_A))$.

Proof is by contradiction. There are 3 cases of relabelling by τ to consider, see Definition 5.7 in [14]. The interested reader can refer to [14] for more detail.

The deadlock reduction can be done more efficiently too. Intuitively, a state inducing a new deadlock in an expanded component does not induce a deadlock in the whole system if there exists an expanded component where this state is not a deadlock state. The checking of whether a state is in an expanded component state space can be done by studying the synchronisation set. Suppose $[SR^1] \sqsubseteq_{\rho}^{D_1} [SA^1]$. The reduced deadlock set RD_1 is defined by

$$RD_1 \overset{\text{def}}{=} D_1 \setminus \{q \mid q \in D_1 \wedge \exists e_2.\ (e_2 \in E_R^2 \wedge (-, e_2)/sp \in syn_R \wedge q \models sp)\} \quad (2)$$

We want to emphasise that for an expanded component, the deadlock reduction is independent from the other expanded components. Property **??** allows us to apply the modular refinement to component-based systems using expanded components as modules.

Theorem 3 (Component-based Refinement). *Let* (SA^1, SA^2, syn_A) *and* (SR^1, SR^2, syn_R) *be two component-based systems. Then*

$$\frac{\neg\, div^{\tau}(SR^1, SA^1),\ [SR^1] \sqsubseteq_{\rho}^{D_1} [SA^1],\ RD_1 = \varnothing, \quad \neg\, div^{\tau}(SR^2, SA^2),\ [SR^2] \sqsubseteq_{\rho}^{D_2} [SA^2],\ RD_2 = \varnothing}{(SR^1, SR^2, syn_R) \sqsubseteq_{\eta} (SA^1, SA^2, syn_A)}$$

Proof is in Appendix A.2. Theorem 3 can be given for n components. It provides a compositional refinement verification algorithm based on the computation, for each refined expanded component $[SR^i]$ separately, of the relation ρ. The complexity order of this refinement verification algorithm is $O(\sum_{i=1}^{n} |[SR^i]|)$. However, the greatest memory space used is $max_{i=1}^{n}|[SR^i]|$, at most: the expanded component building, the weak refinement verification and the deadlock reduction can be done sequentially.

5 Conlusion

In areas like telecommunication or manufacturing, complex systems may be derived from initial models by composition and refinement. Composition combines separate parts and refinements add new details for systematically deriving component-based systems where safety properties are preserved. Furthermore, the state explosion can be alleviated by considering the components one by one.

Our compositional framework is well-adapted for studying components instead of the whole system. Indeed, the weak refinement verification of the composition of the modules can be reduced to the weak refinements of its modules. Furthermore, the constraint synchronised product advocated in this paper allows us to establish the strict refinement of the component-based systems by checking the weak refinements of its expanded components viewed as the modules.

Finally, our compositional framework gives ways to postpone the model-checking blow-up. Actually, a safety LTL property is 1) preserved from an abstract component to a whole abstract component-based system, and 2) preserved from that system to a whole refined component-based system (see Fig. 1). The usefulness of our approach is illustrated by our previous results [10,11] and confirmed by the experimental results [14].

Future work concerns reactive systems. We are going to investigate what model of open systems can be used to obtain a closed system model by adding an environment specification. At that stage, simulation relations should be established again.

Related works. The *assume-guarantee* paradigm (see for instance [6,7,16]) is a well-studied framework for compositional verification. The assume-guarantee paradigm requires the hypotheses on the component environment strong enough to imply any potential constraint. The way out is the *lazy composition* approach by Shankar [18] which works at the level of the specification of component behaviour and discharges proof obligations lazily.

In our approach, the component environment is taken into account by increasing a component model to automatically build an expanded component. Like the lazy composition, the constraint synchronised product allows us to proceed through a refinement-based specification and a component-based verification approaches. The strict refinement relation being a divergence-sensitive completed τ-simulation, we anticipate that it is more expressive than the closed systems refinement by Shankar.

References

1. B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
2. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design (FMSD)*, 15(1):7–48, July 1999.
3. A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Collection Etudes et Recherches en Informatiques. Masson, Paris, 1992.
4. A. Arnold and M. Nivat. Comportements de processus. In *Actes du Colloque AFCET - Les Mathématiques de l'Informatique*, pages 35–68, 1982.
5. F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Fundamental Aspects of Software Engineering (FASE'00)*, volume 1783 of *LNCS*, pages 266–283. Springer Verlag, April 2000.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.
7. J.-M. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, Warsaw, Poland, April 2003. Springer-Verlag.

8. C. Darlot, J. Julliand, and O. Kouchnarenko. Refinement preserves PLTL properties. In D. Bert, J. P. Bowen, S. C. King, and M. Walden, editors, *Formal Specification and Development in Z and B (ZB'2003)*, volume 2651 of *LNCS*, Turku, Finland, June 2003. Springer Verlag.
9. O. Kouchnarenko and A. Lanoix. SynCo: a refinement analysis tool for synchronized component-based systems. In *Tool Exhibition Notes, Formal Methods (FM'03)*.
10. O. Kouchnarenko and A. Lanoix. Refinement and verification of synchronized component-based systems. In K. Araki, S. Gnesi, and Mandrioli D., editors, *Formal Methods (FM'03)*, volume 2805 of *LNCS*, pages 341–358, Pisa, Italy, September 2003. Springer Verlag.
11. O. Kouchnarenko and A. Lanoix. Verifying invariants of component-based systems through refinement. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *LNCS*, pages 289–303, Stirling, Scotland, July 2004. Springer Verlag.
12. O. Kupferman and M. Y. Vardi. Module checking. In Rajeev Alur and T.A. Henzinger, editors, *Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 75–86, New Brunswick, NJ, USA, 1996. Springer Verlag.
13. O. Kupferman and M. Y. Vardi. Module checking revisited. In *9th International Computer Aided Verification Conference*, pages 36–47, 1997.
14. A. Lanoix. *Systèmes à composants synchronisés : contributions à la vérification compositionnelle du raffinement et des propriétés.* PhD thesis, Université de Franche-comté, Septembre 2005.
15. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specifications*. Springer Verlag, 1992.
16. K.L. McMillan. A methodology for hardware verification using compositional model-checking. *Science of Computer Programming*, 37:279–309, 2000.
17. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.
18. N. Shankar. Lazy compositional verification. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 541–564, London, UK, 1998. Springer-Verlag.
19. R.J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR'90*, pages 278–297. Springer-Verlag, 1990.

A Proofs

A.1 Proof of Theorem 2

Let $M \stackrel{\text{def}}{=} \{(q_R, q_A) \mid q_R \in Q^1_R \cup Q^2_R \wedge q_A \in Q^1_A \cup Q^2_A\}$. We show that M verifies all conditions of definition 5 of ρ.

1. *Strict refinement:* suppose $(q_R, e, q'_R) \in T_R$ and $e \in E_A$.
   We must prove there exist $q_A \in Q_A$ and $q'_A \in Q_A$ such that $(q_A, e, q'_A) \in T_A$, $q_R \rho q_A$ and $q'_R \rho q'_A$.
   By definition $E_A = E^1_A \cup E^2_A$. There are 2 cases for e.
   - $e \in E^1_A$. We have $(q_R, e, q'_R) \in T^1_R$. Since $MR^1 \sqsubseteq^{D_1}_\rho MA^1$, there exist $q_A \in Q^1_A$ and $q'_A \in Q^1_A$ such that $q_R \rho q_A$, $(q_A, e, q'_A) \in T^1_A$ and $q'_R \rho q'_A$. q_A and q'_A belong to $Q_A = Q^1_A \cup Q^2_A$.
   - $e \in E^2_A$. We have $(q_R, e, q'_R) \in T^2_R$. Since $MR^2 \sqsubseteq^{D_2}_\rho MA^2$, there exist $q_A \in Q^2_A$ and $q'_A \in Q^2_A$ such that $q_R \rho q_A$, $(q_A, e, q'_A) \in T^2_A$ and $q'_R \rho q'_A$. q_A and q'_A belong to $Q_A = Q^1_A \cup Q^2_A$.

2. *stuttering refinement:* suppose $(q_R, \tau, q'_R) \in T_R$.
   We must prove there exists $q_A \in Q_A$ such that $q_R \rho q_A$ and $q'_R \rho q_A$.
   By definition $E_A = E_A^1 \cup E_A^2$ and $E_R = E_R^1 \cup E_R^2$. There are 2 cases for $e \backslash \tau \in E_R \smallsetminus E_A$.
   - $e \in E_R^1 \smallsetminus E_A^1$. We have $(q_R, e\backslash\tau, q'_R) \in T_R^1$. Since $MR^1 \sqsubseteq_\rho^{D_1} MA^1$, there exists $q_A \in Q_A^1$ such that $q_R \rho q_A$ and $q'_R \rho q_A$. q_A belongs to $Q_A = Q_A^1 \cup Q_A^2$.
   - $e \in E_R^2 \smallsetminus E_A^2$. We have $(q_R, e\backslash\tau, q'_R) \in T_R^2$. Since $MR^2 \sqsubseteq_\rho^{D_2} MA^2$, there exists $q_A \in Q_A^2$ such that $q_R \rho q_A$ and $q'_R \rho q_A$. q_A belongs to $Q_A = Q_A^1 \cup Q_A^2$.
3. *old or new deadlocks:* suppose $q_R \nrightarrow$ and $q_R \in Q_R$.
   We must prove there exists $q_A \in Q_A$ such that $q_R \rho_w q_A$ and $q_A \nrightarrow$ or $q_R \in D_1 \triangle D_2$.
   By definition $Q_R = Q_R^1 \cup Q_R^2 = (Q_R^1 \cap Q_R^2) \cup (Q_R^1 \smallsetminus Q_R^2) \cup (Q_R^1 \smallsetminus Q_R^2)$. There are 3 cases for q_R.
   - $q_R \in Q_R^1 \smallsetminus Q_R^2$. Since $MR^1 \sqsubseteq_\rho^{D_1} MA^1$, either there exists $q_A \in Q_A^1$ such that $q_A \nrightarrow$ and $q_R \rho q_A$. This implies that q_A belongs to Q_A ; or $q_R \in D_1$ and $q_R \rho q_A$. As $q_R \in Q_R^1 \smallsetminus Q_R^2$ and $q_R \in D_1$, $q_R \in D_1 \smallsetminus Q_R^2$. Then q_R belongs to $D_1 \triangle D_2$.
   - $q_R \in Q_R^2 \smallsetminus Q_R^1$. Since $MR^2 \sqsubseteq_\rho^{D_2} MA^2$, either there exists $q_A \in Q_A^2$ such that $q_A \nrightarrow$ and $q_R \rho q_A$. Then q_A belongs to Q_A ; or $q_R \in D_2$ and $q_R \rho q_A$. As $q_R \in Q_R^2 \smallsetminus Q_R^1$ and $q_R \in D_2$, $q_R \in D_2 \smallsetminus Q_R^1$. Then q_R belongs to $D_1 \triangle D_2$.
   - $q_R \in Q_R^1 \cap Q_R^2$. Since$MR^1 \sqsubseteq_\rho^{D_1} MA^1$, there exists $q_A \in Q_A^1$ such that either $q_A \nrightarrow$ and $q_R \rho q_A$ (1) or $q_R \rho q_A$ and $q_R \in D_1$ (2).
     Since $MR^2 \sqsubseteq_\rho^{D_2} MA^2$, there exists $q_A \in Q_A^2$ such that either $q_A \nrightarrow$ and $q_R \rho q_A$ (3) or $q_R \rho q_A$ and $q_R \in D_2$ (4).
     If (1) and (3), there exists $q_A \in Q_A^1 \cup Q_A^2$ such that $q_A \nrightarrow$.
     If (1) and (4), there exists $q_A \in Q_A^1$ such that $q_A \nrightarrow$.
     If (2) and (3), there exists $q_A \in Q_A^2$ such that $q_A \nrightarrow$.
     If (2) and (4), $q_R \in D_1 \cap D_2$: q_R belongs to $D_1 \triangle D_2$.

A.2 Proof of Theorem 3

- $[SR^1] \sqsubseteq_\rho^{D_1} [SA^1]$, $[SR^2] \sqsubseteq_\rho^{D_2} [SA^2]$ and Property **??** imply $(SR^1, SR^2, syn_R) \sqsubseteq_\rho^{D_1 \triangle D_2} (SA^1, SA^2, syn_A)$ by Theorem 2.
- $RD_1 = \varnothing$ and $RD_2 = \varnothing$ imply $D_1 \triangle D_2 = \varnothing$.
- $\neg\ div^\tau(SR^1, SA^1)$ and $\neg\ div^\tau(SR^2, SA^2)$ imply $\neg\ div^\tau((SR^1, SR^2, syn_R), (SA^1, SA^2,\ syn_A))$ by Property 8.

Then we have $(SR^1, SR^2, syn_R) \sqsubseteq_\eta (SA^1, SA^2, syn_A)$ by Property 1.

Refinements in Typed Abstract State Machines

Sebastian Link, Klaus-Dieter Schewe, and Jane Zhao

Massey University, Department of Information Systems &
Information Science Research Centre
Private Bag 11 222, Palmerston North, New Zealand
{s.link|k.d.schewe|j.zhao}@massey.ac.nz

Abstract. While Abstract State Machines (ASMs) provide a general purpose development method, it is advantageous to provide extensions that ease their use in particular application areas. This paper focuses on such extensions for the benefit of a "refinement calculus" in the area of data warehouses and on-line analytical processing (OLAP). We show that providing typed ASMs helps to exploit the existing logical formalisms used in data-intensive areas to define a ground model and refinement rules. We also note that the extensions do not increase the expressiveness of ASMs, as each typed ASM will be equivalent to an "ordinary" one.

1 Introduction

The research reported in this article is part of a project that aims at providing sound and complete refinement rules for the development of distributed data warehouses and on-line analytical processing (OLAP) systems [21]. The rationale is that such a "refinement calculus" would simplify the development task and at the same time increase the quality of the resulting systems. As outlined in [14] we base our work on Abstract State Machines (ASMs, [5]), as they have been already proven their usefulness in many application areas. Furthermore, the ASM method explicitly supports our view of systems development to start with an initial specification called *ground model* [3] that is then subject to *refinements* [4]. This is similar to the approach taken in [12], which contains a refinement-based approach to the development of data-intensive systems using a variant of the B method [1].

The general idea for the data warehouse ground model is to employ three interrelated ASMs, one for underlying operational databases, one for the data warehouse, and one for dialogue types [10] that can be used to integrate views on the data warehouse with OLAP functionality [18]. For the data warehouse level the model of multi-dimensional databases [8], which are particular relational databases, can be adopted. Then a large portion of the refinement work has to deal with view integration as predicted one of the major challenges in this area in [19]. The work in [11] discusses a rule-based approach for this task without reference to any formal method.

However, both the ground model and the refinement rules tend to become difficult, as well-known database features such as declarative query expressions, views, bulk updates, etc. are not well supported. Therefore, we believe it is a good idea to incorporate such features into the ASM method to make it easier to use and hence also more acceptable, when applied in an area such as OLAP. This leads us to typed ASMs (TASMs), for which we focus on the incorporation of relations and bulk update operations. There are other approaches to introducing typed versions of ASMs, e.g. [6] following similar ideas, but with different focus, which lead to using different type systems. The work in [20] presents a very general approach to combine ASMs with type theory.

In Section 2 we give a brief introduction on ASMs method, in Section 3 we introduce TASMs and define their semantics by runs following the usual approach used for ASMs. Then in Section 4 we show how TASMs can be used to define a simple ground model for data warehouses. Finally, in Section 5 we discuss refinement rules for TASMs.

2 Abstract State Machines in a Nutshell

Abstract State Machines (ASMs, [5]) have been developed as means for high-level system design and analysis. The general idea is to provide a through-going uniform formalism with clear mathematical semantics without dropping into the pitfall of the "formal methods straight-jacket".

The systems development method itself just presumes to start with the definition of a *ground model ASM* (or several linked ASMs), while all further system development is done by refining the ASMs using quite a general notion of refinement. So basically the systems development process with ASMs is a refinement-validation-cycle. That is a given ASM is refined and the result is validated against the requirements. Validation may range from critical inspections to the usage of test cases and evaluation of executable ASMs as prototypes. This basic development process may be enriched by rigorous manual or mechanised formal verification techniques. However, the general philosophy is to design first and to postpone rigorous verification to a stage, when requirements have be almost consolidated.

2.1 Simple ASMs

As explained so far, we expect to define for each stage of systems development a collection $M_1, \dots, M_n$ of ASMs. Each ASM M_i consists of a *header* and a *body*. The header of an ASM consists of its name, an import- and export-interface, and a signature. Thus, a basic ASM can be written in the form

```
ASM $M$
IMPORT $M_1(r_{11}, \dots, r_{1n_1}), \dots, M_k(r_{k1}, \dots, r_{kn_k})$
EXPORT $q_1, \dots, q_\ell$
SIGNATURE ...
```

Here r_{ij} are the names of functions and rules imported from the ASM M_i defined elsewhere. These functions and rules will be defined in the body of M_i — not in the body of M — and only used in M. This is only possible for those functions and rules that have explicitly been exported. So only the functions and rules $q_1, \dots, q_\ell$ can be imported and used by ASMs other than M. As in standard modular programming languages this mechanism of import- and export-interface permits ASMs to be developed rather independently from each other leaving the definition of particular functions and rules to "elsewhere".

The *signature* of an ASM is a finite list of function names $f_1, \dots, f_m$, each of which is associated with a non-negative integer ar_i, the *arity* of the function f_i. In ASMs each such function is interpreted as a total function $f_i : \mathcal{U}^{ar_i} \to \mathcal{U} \cup \{\perp\}$ with a not further specified set $\mathcal{U}$ called *super-universe* and a special symbol $\perp \notin \mathcal{U}$. As usual, f_i can be interpreted as a partial function $\mathcal{U}^{ar_i} \nrightarrow \mathcal{U}$ with domain $dom(f_i) = \{\boldsymbol{x} \in \mathcal{U}^{ar_i} \mid f_i(\boldsymbol{x}) \neq \perp\}$.

The functions defined for an ASM including the static and derived functions, define the set of *states* of the ASM.

In addition, functions can be *dynamic* or not. Only dynamic functions can be updated, either by and only by the ASM, in which case we get a *controlled* function, by the environment, in which case we get a *monitored* function, or by none of both, in which case we get a *derived* function. In particular, a dynamic function of arity 0 is a variable, whereas a static function of arity 0 is a constant.

2.2 States and Transitions

If f_i is a function of arity ar_i and we have $f(x_1, \dots, x_{ar_i}) = v$, we call the pair $\ell = (f, \boldsymbol{x})$ with $\boldsymbol{x} = (x_1, \dots, x_{ar_i})$ a *location* and v its *value*. Thus, each *state* of an ASM may be considered as a set of location/value pairs.

If the function is dynamic, the values of its locations may be updated. Thus, states can be updated, which can be done by an *update set*, i.e. a set Δ of pairs (ℓ, v), where ℓ is a location and v is a value. Of course, only *consistent* update sets can be taken into account, i.e. we must have

$$(\ell, v_1) \in \Delta \wedge (\ell, v_2) \in \Delta \Rightarrow v_1 = v_2.$$

Each consistent update set Δ defines *state transitions* in the obvious way. If we have $f(x_1, \dots, x_{ar_i}) = v$ in a given state s and $((f, (x_1, \dots, x_{ar_i})), v') \in \Delta$, then in the successor state s' we will get $f(x_1, \dots, x_{ar_i}) = v'$.

In ASMs consistent update sets can be obtained from *update rules*, which can be defined by the following language:

- the skip rule `skip` indicates no change;
- the update rule $f(t_1, \dots, t_n) := t$ with an n-ary function f and terms $t_1, \dots, t_n, t$ indicates that the value of the location determined by f and the terms $t_1, \dots, t_n$ will be updated to the value of term t;
- the sequence rule r_1 `seq` ... `seq` r_n indicates that the rules $r_1, \dots, r_n$ will be executed sequentially;
- the block rule r_1 `par` ... `par` r_n indicates that the rules $r_1, \dots, r_n$ will be executed in parallel;

– the conditional rule

$$\texttt{if } \varphi_1 \texttt{ then } r_1 \texttt{ elsif } \varphi_2 \dots \texttt{ then } r_n \texttt{ endif}$$

has the usual meaning that r_1 is executed, if φ_1 evaluates to true, otherwise r_2 is executed, if φ_2 evaluates to true, etc.;
– the let rule `let` $x = t$ `in` r means to assign to the variable x the value defined by the term t and to use this x in the rule r;
– the forall rule `forall` x `with` φ `do` r `enddo` indicates the parallel execution of r for all values of x satisfying φ;
– the choice rule `choose` x `with` φ `do` r `enddo` indicates the execution of r for one value of x satisfying φ;
– the call rule $r(t_1, \dots, t_n)$ indicates the execution of rule r with parameters $t_1, \dots, t_n$ (call by name).

Instead of `seq` we simply use ; and instead of `par` we write $\|$. The idea is that the rules of an ASM are evaluated in parallel. If the resulting update set is consistent, we obtain a state transition. Then a *run* of an ASM is a finite or infinite sequence of states $s_0 \to s_1 \to s_2 \to \dots$ such that each s_{i+1} is the successor state of s_i with respect to the update set Δ_i that is defined by evaluating the rules of the ASM in state s_i.

We omit the formal details of the definition of update sets from these rules. These can be found in [5].

The definition of rules by expressions $r(x_1, \dots, x_n) = r'$ makes up the body of an ASM. In addition, we assume to be given an *initial state* and that one of these rules is declared as the *main rule*. This rule must not have parameters.

3 Typed Abstract State Machines

The 3-tier architecture for data warehouses and OLAP systems in [21,22] basically requires relations on the database and the data warehouse tiers, while the OLAP tier requires sets of complex values. Therefore, following the line of research in [13] we start with a *type system*

$$t = b \mid \{t\} \mid a : t \mid t_1 \times \cdots \times t_n \mid t_1 \oplus \cdots \oplus t_n \mid 1\!\!1$$

Here b represents a not further specified collection of base types such as *CARD*, *INT*, *DATE*, etc. $\{\cdot\}$ is a set-type constructor, $a : t$ is a type constructor with label a, which is introduced as attributes used in join operations. $\times$ and $\oplus$ are constructors for tuple and union types. $1\!\!1$ is a trivial type. With each type t we associate a *domain* $dom(t)$ in the usual way, i.e. we have $dom(\{t\}) = \{x \subseteq dom(t) \mid |x| < \infty\}$, $dom(a : t) = \{a : v \mid v \in dom(t)\}$, $dom(t_1 \times \cdots \times t_n) = dom(t_1) \times \cdots \times dom(t_n)$, $dom(t_1 \oplus \cdots \oplus t_n) = \coprod_{i=1}^n dom(t_i)$ (disjoint union), and $dom(1\!\!1) = \{\mathbf{1}\}$.

For this type systems we obtain the usual notation of subtyping, defined by the smallest partial order $\leq$ on types satisfying

– $t \leq 1\!\!1$ for all types t;
– if $t \leq t'$ holds, then also $\{t\} \leq \{t'\}$;
– if $t \leq t'$ holds, then also $a : t \leq a : t'$;
– if $t_{i_j} \leq t'_{i_j}$ hold for $j = 1, \dots, k$, then $t_1 \times \cdots \times t_n \leq t'_{i_1} \times \cdots \times t'_{i_k}$ for $1 \leq i_1 < \cdots < i_k \leq n$;
– if $t_i \leq t'_i$ hold for $i = 1, \dots, n$, then $t_1 \oplus \cdots \oplus t_n \leq t'_1 \oplus \cdots \oplus t'_n$.

We say that t is a *subtype* of t' iff $t \leq t'$ holds. Obviously, subtyping $t \leq t'$ induces a canonical projection mapping $\pi^t_{t'} : dom(t) \to dom(t')$.

The *signature* of a TASM is defined analogously to the signature of an "ordinary" ASM, i.e. by a finite list of function names $f_1, \dots, f_m$. However, in a TASM each function f_i now has a *kind* $t_i \to t'_i$ involving two types t_i and t'_i. We interpret each such function by a total function $f_i : dom(t_i) \to dom(t'_i)$. Note that using $t'_i = t''_i \oplus 1\!\!1$ we can cover also partial functions. In addition, functions can be *dynamic* or not. Only dynamic functions can be updated, either by and only by the ASM, in which case we get a *controlled* function, by the environment, in which case we get a *monitored* function, or by none of both, in which case we get a *derived* function.

The functions defined for a TASM including the static and derived functions, define the set of states of the TASM. More precisely, each pair $\ell = (f_i, x)$ with $x \in dom(t_i)$ defines a *location* with $v = f_i(x)$ as its *value*. Thus, each *state* of a TASM may be considered as a set of location/value pairs.

We call a function R of kind $t \to \{1\!\!1\}$ a *relation*. This generalises the standard notion of relation, in which case we would further require that t is a tuple type of $a_1 : t_1 \times \cdots \times a_n : t_n$. In particular, as $\{1\!\!1\}$ can be considered as a truth value type, we may identify R with a subset of $dom(t)$, i.e. $R \simeq \{x \in dom(t) \mid R(x) \neq \emptyset\}$. In this spirit we also write $x \in R$ instead of $R(x) \neq \emptyset$, and $x \notin R$ instead of $R(x) = \emptyset$.

As with ASMs we define state transitions via *update rules*, which are defined by the following language (we deviate from the syntax used in ASMs):

– the skip rule `skip`, which indicates no change;
– the assignment rule $f(\tau) := \tau'$ with a function $f : t \to t'$ and terms τ, τ' of type t and t', respectively, which indicates that the value of the location determined by f and τ will be updated to the value of term τ';
– the sequence rule $r_1; \dots; r_n$, which indicates that the rules $r_1, \dots, r_n$ will be executed sequentially;
– the block rule $r_1 \| \dots \| r_n$, which indicates that the rules $r_1, \dots, r_n$ will be executed in parallel;
– the conditional rule $\varphi_1\{r_1\} \boxtimes \varphi_2\{r_2\} \boxtimes \cdots \boxtimes \varphi_n\{r_n\}$, which has the usual meaning that r_1 is executed, if φ_1 evaluates to true, otherwise r_2 is executed, if φ_2 evaluates to true, etc.;
– the let rule $\Lambda x = \tau\{r\}$, which means to assign to the variable x the value defined by the term τ and to use this value in the rule r;
– the forall rule $\mathbf{A}x \bullet \varphi\{r\}$, which indicates the parallel execution of r for all values of x satisfying φ;

– the choice rule $@x \bullet \varphi\{r\}$ indicates the execution of r for one value of x satisfying φ;
– the call rule $r(\tau)$ indicates the execution of rule r with parameters τ.

Each update rule r defines an update set $\Delta(r)$ in the same way as for "ordinary" ASMs [5, p.74]. Such an update set is *consistent* iff $(\ell, v_1) \in \Delta \wedge (\ell, v_2) \in \Delta \Rightarrow v_1 = v_2$ holds for all locations. Then state transitions are defined by consistent update sets. Then a *run* of a TASM is a finite or infinite sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ such that each s_{i+1} is the successor state of s_i with respect to the update set Δ_i that is defined by evaluating the rules of the TASM in state s_i.

What is different in TASMs is that the terms used in the rules are typed, i.e. for each type t we obtain a set $\mathbb{T}_t$ of terms of type t. Then also the formulae φ used in the rules change in that equational atoms $\tau_1 = \tau_2$ can only be built from terms τ_1, τ_2 that have the same type. All the rest remains unchanged.

So let us assume that for each type t we are given a set V_t of variables of type t. Then we should have $V_t \subseteq \mathbb{T}_t$ and $dom(t) \subseteq \mathbb{T}_t$, and further terms can be built as follows:

– For $\tau \in \mathbb{T}_t$ and $t \leq t'$ we get $\pi^t_{t'}(\tau) \in \mathbb{T}_{t'}$.
– For $\tau \in \mathbb{T}_{t_1 \times \dots \times t_n}$ we get $\pi_i(\tau) \in \mathbb{T}_{t_i}$.
– For $\tau_i \in \mathbb{T}_{t_i}$ for $i = 1, \dots, n$ we get $(\tau_1, \dots, \tau_n) \in \mathbb{T}_{t_1 \times \dots \times t_n}$, $\iota_i(\tau_i) \in \mathbb{T}_{t_1 \oplus \dots \oplus t_n}$, and $\{\tau_i\} \in \mathbb{T}_{\{t_i\}}$.
– For $\tau_1, \tau_2 \in \mathbb{T}_{\{t\}}$ we get $\tau_1 \cup \tau_2 \in \mathbb{T}_{\{t\}}$, $\tau_1 \cap \tau_2 \in \mathbb{T}_{\{t\}}$, and $\tau_1 - \tau_2 \in \mathbb{T}_{\{t\}}$.
– For $\tau \in \mathbb{T}_{\{t\}}$, a constant $e \in dom(t')$ and static functions $f : t \rightarrow t'$ and $g : t' \times t' \rightarrow t'$ we get $src[e, f, g](\tau) \in \mathbb{T}_{t'}$.
– For $\tau_i \in \mathbb{T}_{\{t_i\}}$ for $i = 1, 2$ we get $\tau_1 \bowtie \tau_2 \in \mathbb{T}_{\{t_1 \bowtie t_2\}}$ using the maximal common subtype $t_1 \bowtie t_2$ of t_1 and t_2.
– For $x \in V_t$ and a formula φ we get $\mathbf{I}x.\varphi \in \mathbb{T}_t$.

The last three constructions for terms need some more explanation. Structural recursion $src[e, f, g](\tau)$ is a powerful construct for database queries [17]. It is defined as follows:

– $src[e, f, g](\tau) = e$, if $\tau = \emptyset$;
– $src[e, f, g](\tau) = f(v)$, if $\tau = \{v\}$;
– $src[e, f, g](\tau) = g(src[e, f, g](v_1), src[e, f, g](v_2))$, if $\tau = v_1 \cup v_2 \wedge v_1 \cap v_2 = \emptyset$. In order to be uniquely defined, the function g must be associative and commutative with e as a neutral element.

We can use structural recursion to specify comprehension, which is extremely important for views. We get $\{x \in \tau \mid \varphi\} = src[\emptyset, f_\tau, \cup](\tau)$ using the static function f_τ with $f_\tau(x) = \{x\}$, if $\varphi(x)$ holds, else $f_\tau(x) = \emptyset$, which can be composed out of very simple functions [13].

For the *join* $\tau_1 \bowtie \tau_2$, using $[\![\cdot]\!]_s$ to denote the interpretation in a state s, we get $[\![\tau_1 \bowtie \tau_2]\!]_s = \{v \in dom(t_1 \bowtie t_2)$
$\mid \exists v_1 \in [\![\tau_1]\!]_s, v_2 \in [\![\tau_2]\!]_s.(\pi^{t_1 \bowtie t_2}_{t_1}(v) = v_1 \wedge \pi^{t_1 \bowtie t_2}_{t_2}(v) = v_2)\}$, which generalises the natural join from relational algebra [13].

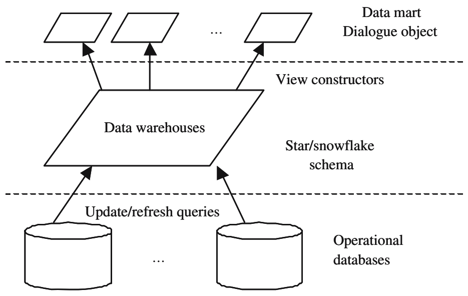


Fig. 1. The general architecture of a data warehouse and OLAP

$\mathbf{I}x.\varphi$ stands for "the unique x satisfying φ". If such an x does not exist, the term will be undefined, i.e. we have $[\![\mathbf{I}x.\varphi]\!]_s = v$, if $[\![\{x \mid \varphi\}]\!]_s = \{v\}$, otherwise it is undefined. If an undefined term appears in a rule, the rule defines an inconsistent update set.

For relations R of kind $t \to \{\mathbb{1}\}$ we further permit *bulk assignments*, which take one of the following forms $R := \tau$ (for replacing), $R :+_k \tau$ (for inserting), $R :-_k \tau$ (for deleting), and $R :\&_k \tau$ (for updating), using each time a term τ of type $\{t\}$ and a supertype k of t. This is an old idea adopted from the database programming language Pascal/R [16]. These constructs are shortcuts for the following TASM rules:

- $R := \tau$ represents $\mathbf{A}x \bullet x \in \tau\{R(x) := \{\mathbf{1}\}\} \| \mathbf{A}x \bullet x \in R \wedge x \notin \tau\{R(x) := \emptyset\}$
- $R :+_k \tau$ represents $\mathbf{A}x \bullet (x \in \tau \wedge \forall y(y \in R \to \pi_k^t(x) \neq \pi_k^t(y))\{R(x) := \{\mathbf{1}\}\}$
- $R :-_k \tau$ represents $\mathbf{A}x \bullet (x \in R \wedge \exists y(y \in \tau \wedge \pi_k^t(x) = \pi_k^t(y))\{R(x) := \emptyset\}$
- $R :\&_k \tau$ represents

$$\mathbf{A}x \bullet (x \in R \wedge \exists y(y \in \tau \wedge \pi_k^t(x) = \pi_k^t(y))\{R(x) := \emptyset\};$$
$$\mathbf{A}x \bullet x \in \tau \wedge \forall y(y \in R \wedge y \neq x \to \pi_k^t(x) \neq \pi_k^t(y))\{R(x) := \{\mathbf{1}\}\}$$

Up to now we have defined a typed ASM. If we translate a TASM $\mathfrak{M}$ into an ASM $\Phi(\mathfrak{M})$, we have the following theorem:

Theorem 1. *For each TASM $\mathfrak{M}$ there is an equivalent ASM $\Phi(\mathfrak{M})$.*

Of course, this theorem also follows immediately from the main results on the expressiveness of ASMs in [2,7].

4 A Data Warehouse Ground Model in TASMs

Following the basic ideas in [22], to separate the output from the input, we get the three-tier architecture of the data warehouse shown in Figure 1. At the bottom tier, we have the operational database model, which has the control of the updates to data warehouse. The updates are abstracted as data extraction

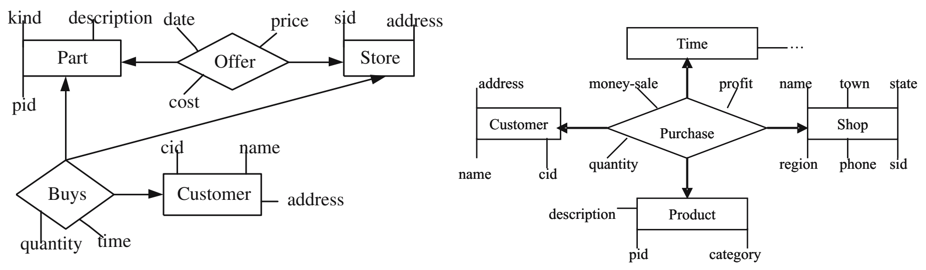


Fig. 2. The operational database and data warehouse schemata

from the operational database to maintain the freshness of the data warehouse. In the middle tier, we have the data warehouse which is modelled in star schema [9]. At the top tier, we have the data marts, which are constructed out of dialogue objects with OLAP operations. Based on this three-tier architecture, we end up with three linked ASMs as in [22], or TASMs when we apply typed ASM. In the following we will use one of them as an example to show the difference between the ground model in ASM and the one in TASM.

We again use the grocery store as the example. In this case we have a single operational database with five relation schemata as illustrated in the left hand HERM diagram in Figure 2, the start schema for the data warehouse in the right hand of the figure.

We present DB-ASM (in ASM) and DB-TASM (in TASM) in the following, with our focus on the affected part, i.e. the data extraction for *Purchase*, the fact table in the data warehouse star schema:

```
ASM DB-ASM
IMPORT DW-ASM(Shop, Product, Customer, Time, Purchase)
EXPORT $extract\_purchase$
SIGNATURE ...
BODY
   $main = \ldots$
   $extract\_purchase =$
     forall $i, p, s, t, p', c$ with $\exists q.\text{Buys}(i,s,p,q,t) \neq \perp \wedge$
       $\exists n, a.\text{Customer}_{db}(i,n,a) \neq \perp \wedge \exists k, d.\text{Part}(p,k,d) \neq \perp \wedge$
       $\exists a'.\text{Store}(s,a') \neq \perp \wedge \exists d.(\text{Offer}(p,s,p',c,d) \neq \perp \wedge \text{Date}(t) = d$
     do let $Q = sum(q \mid \text{Buys}(i,s,p,q,t) \neq \perp), S = Q * p',$
            $P = Q * (p' - c)$
       in $\text{Purchase}(i,p,s,t,Q,S,P) := 1$
     enddo
```

```
ASM DB-TASM
IMPORT DW-TASM(Shop, Product, Customer, Time, Purchase)
EXPORT $extract\_purchase$
```

```
SIGNATURE ...
BODY
    $main = \dots$
    $extract\_purchase$ = Purchase $:\&_{\{cid \times pid \times sid \times time\}}$
      $\{(i, s, p, t, Q, S, P) \mid \exists pr, c.((i, s, p, t, pr, c) \in$
       $\pi_{i,s,p,t,pr,c}(\text{Buys} \bowtie \text{Customer}_{db} \bowtie \text{Part} \bowtie \text{Store} \bowtie \text{Offer} \bowtie \text{Date(t)})$
        $\wedge Q = src[0, \pi_q, +](\{(i', s', p', q, t') \mid (i', s', p', q, t') \in Buys \wedge$
              $i' = i \wedge s' = s \wedge p' = p \wedge t' = t\})$
        $\wedge S = Q * pr \wedge P = Q * (pr - c))\}$
```

As shown above, we have redefined *sum* function using structural recursion constructor, and used the bulk update operation in *extract_purchase* in the typed model DB-TASM.

5 Refinement of Typed ASMs

The general notion of *refinement* in ASMs relates two ASMs $\mathfrak{M}$ and $\mathfrak{M}^*$ in the following way:

– a correspondence between some states s of M and some states s^* of M^*, and
– a correspondence between the runs of $\mathfrak{M}$ and $\mathfrak{M}^*$ involving states s and s^*, respectively.

Keeping in mind that we are looking at the application of TASMs for data warehouses and OLAP systems, we first clarify what are the states of interest in this definition. For this assume that names of functions, rules, etc. are completely different for $\mathfrak{M}$ and $\mathfrak{M}^*$. Then consider formulae $\mathcal{A}$ that can be interpreted by pairs of states (s, s^*) for $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively. Such formulae will be called *abstraction predicates*. Furthermore, let the rules of $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively, be partitioned into "main" and "auxiliary" rules such that there is a correspondence $\rhd$ between main rules r of $\mathfrak{M}$ and main rules r^* of $\mathfrak{M}^*$. Finally, take initial states s_0, s_0^* for $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively.

Definition 1. *A TASM $\mathfrak{M}^*$ is called a* (weak) refinement *of a TASM $\mathfrak{M}$ iff there is an* abstraction predicate *$\mathcal{A}$ with $(s_0, s_0^*) \models \mathcal{A}$ and there exists a correspondence between main rule r of $\mathfrak{M}$ and main rule r^* of $\mathfrak{M}^*$ such that for all states $s, \overline{s}$ of $\mathfrak{M}$, where $\overline{s}$ is the successor state of s with respect to the update set Δ_r defined by the main rule r, there are states $s^*, \overline{s}^*$ of $\mathfrak{M}^*$ with $(s, s^*) \models \mathcal{A}$, $(\overline{s}, \overline{s}^*) \models \mathcal{A}$, and $\overline{s}^*$ is the successor state of s_m^* with respect to the update set Δ_{r^*} defined by the main rule r^*.*

While Definition 1 gives a proof obligation for refinements in general, it still permits too much latitude for data-intensive applications. In this context we must assume that some of the controlled functions in the signature are meant to be persistent. For these we adopt the notion of *schema*, which is a subset of the signature consisting only of relations. Then the first additional condition should be that in initial states these relations are empty.

The second additional requirement is that the schema $\mathcal{S}^*$ of the refining TASM $\mathfrak{M}^*$ should dominate the schema $\mathcal{S}$ of TASM $\mathfrak{M}$. For this we need a notion of computable query. For a state s of a TASM $\mathfrak{M}$ let $s(\mathcal{S})$ define its restriction to the schema. We first define isomorphisms starting from bijections $\iota_b : dom(b) \to dom(b)$ for all base types b. This can be extended to bijections ι_t for any type t as follows:

$$\iota_{t_1 \times \cdots \times t_n}(x_1, \dots, x_n) = (\iota_{t_1}(x_1), \dots, \iota_{t_n}(x_n))$$
$$\iota_{t_1 \oplus \cdots \oplus t_n}(i, x_i) = (i, \iota_{t_i}(x_i))$$
$$\iota_{\{t\}}(\{x_1, \dots, x_k\}) = \{\iota_t(x_1), \dots, \iota_t(x_k)\}$$

Then ι is an *isomorphism* of $\mathcal{S}$ iff for all states s, the permuted state $\iota(s)$, and all $R : t \to \{\mathbb{1}\}$ in $\mathcal{S}$ we have $R(x) \neq \emptyset$ in s iff $R(\iota_t(x)) \neq \emptyset$ in $\iota(s)$. A query $f : \mathcal{S} \to \mathcal{S}^*$ is *computable* iff f is a computable function that is closed under isomorphisms.

Definition 2. *A refinement $\mathfrak{M}^*$ of a TASM $\mathfrak{M}$ with abstraction predicate $\mathcal{A}$ is called a* strong refinement *iff the following holds:*

1. *$\mathfrak{M}$ has a schema $\mathcal{S} = \{R_1, \dots, R_n\}$ such that in the initial state s_0 of $\mathfrak{M}$ we have $R_i(x) = \emptyset$ for all $x \in dom(t_i)$ and all $i = 1, \dots, n$.*
2. *$\mathfrak{M}^*$ has a schema $\mathcal{S}^* = \{R_1^*, \dots, R_m^*\}$ such that in the initial state s_0^* of $\mathfrak{M}^*$ we have $R_i^*(x) = \emptyset$ for all $x \in dom(t_i^*)$ and all $i = 1, \dots, m$.*
3. *There exist computable queries $f : \mathcal{S} \to \mathcal{S}^*$ and $g : \mathcal{S}^* \to \mathcal{S}$ such that for each pair (s, s^*) of states with $(s, s^*) \models \mathcal{A}$ we have $g(f(s(\mathcal{S}))) = s(\mathcal{S})$.*

Definition 2 provides a stronger proof obligation for refinements in the application area we are interested in. Furthermore, this notion of strong refinement heavily depends on the presence of types.

Let us finally discuss refinement rules for data warehouses and OLAP systems. According to [21] the integration of schemata and views on all three tiers is an important part of such refinement rules. Furthermore, the intention is to set up rules of the form

$$\frac{\mathfrak{M} \rhd aFunc, \dots, aRule, \dots}{\mathfrak{M}^* \rhd newFunc, \dots, newRule = \dots} \varphi$$

That is, we indicate under some side conditions φ, which parts of the TASM $\mathfrak{M}$ will be replaced by new functions and rules in a refining TASM $\mathfrak{M}^*$. Furthermore, the specification has to indicate, which relations belong to the schema, and the correspondence between main rules.

Example 1. As a sample refinement rule adopted from [11] take

$$\frac{\mathfrak{M} \rhd R : (t_{11} \oplus \cdots \oplus t_{1k}) \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\}}{\begin{array}{rl} \mathfrak{M}^* \rhd & R_1^* : t_{11} \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\} \\ & R_2^* : t_{12} \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\} \\ & \vdots \\ & R_k^* : t_{1k} \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\} \end{array}}$$

Then the corresponding abstraction predicate $\mathcal{A}$ is as

$$\forall x_1, \dots, x_n (R_i^*(x_1, \dots, x_n) = 1 \leftrightarrow R((i, x_1), \dots, x_n) = 1)$$

The corresponding computable queries f and g can be obtained as

$$R_1 := \{(x_1, \dots, x_n) \mid ((1, x_1), x_2, \dots, x_n) \in R\} \| \dots \| \\ R_k := \{(x_1, \dots, x_n) \mid ((k, x_1), x_2, \dots, x_n) \in R\}$$

and

$$R := \bigcup_{i=1}^{k} \{((i, x_1), x_2, \dots, x_n) \mid (x_1, \dots, x_n) \in R_i\},$$

respectively.

6 Conclusion

In this paper we introduced a typed version of Abstract State Machines. The motivation for this is that our research aims at applying the ground model / refinement method supported by ASMs in the area of data warehouses and OLAP systems. For this it is advantageous to have bulk data types (sets, relations) and declarative query expressions available, but these are not available in ASMs. So, the major reason for adding them is to increase the applicability of ASMs. We did, however, show that these extensions do not increase the expressiveness of ASMs, as each typed ASM is equivalent to an “ordinary” one.

In a second step we approached refinement in typed ASMs. We clarified what we want to achieve by refinements in data-intensive application areas. In particular, we distinguish between weak and strong refinement, the latter one being more restrictive with respect to changes to the signature, in order to preserve the semantics of data in accordance with schema dominance as discussed in [11].

This brings us closer to the goal of our research project to set up sound and complete refinement rules for the development of distributed data warehouses and OLAP systems. The general approach to defining such rules has already been discussed in [21], while the current work clarifies, what kind of rules we have to expect. In [15] we already discussed rules for distribution, but the more challenging task consists in rules for data integration. So our next steps will address such rules following among others the work on view integration in [11].

References

1. Abrial, J.-R.: *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. Blass, A., Gurevich, J.: Abstract State Machines Capture Parallel Algorithms, *ACM Transactions on Computational Logic*, **4**(4), 2003, 578–651.

3. Börger, E.: The ASM Ground Model Method as a Foundation for Requirements Engineering, *Verification: Theory and Practice*, 2003.
4. Börger, E.: The ASM Refinement Method, *Formal Aspects of Computing*, **15**, 2003, 237–257.
5. Börger, E., Stärk, R.: *Abstract State Machines*, Springer-Verlag, Berlin Heidelberg New York, 2003.
6. Del Castillo, G., Gurevich, Y., Stroetmann, K.: Typed Abstract State Machines, 1998, Unpublished, available from http://research.microsoft.com/ gurevich/Opera/137.pdf.
7. Gurevich, J.: Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, **1**(1), 2000, 77–111.
8. Gyssens, M., Lakshmanan, L.: A foundation for multidimensional databases, *Proc. 22nd VLDB Conference, Mumbai (Bombay), India*, 1996.
9. Kimball, R.: *The Data Warehouse Toolkit*, John Wiley & Sons, 1996.
10. Lewerenz, J., Schewe, K.-D., Thalheim, B.: Modelling Data Warehouses and OLAP Applications Using Dialogue Objects, in: *Conceptual Modeling – ER'99* (J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, E. Métais, Eds.), vol. 1728 of *LNCS*, Springer-Verlag, 1999, 354–368.
11. Ma, H., Schewe, K.-D., Thalheim, B., Zhao, J.: View Integration and Cooperation in Databases, Data Warehouses and Web Information Systems, *Journal on Data Semantics*, **IV**, 2005, 213–249.
12. Schewe, K.-D.: *Specification and Development of Correct Relational Database Programs*, Technical report, Clausthal Technical University, Germany, 1997.
13. Schewe, K.-D.: On the Unification of Query Algebras and their Extension to Rational Tree Structures, in: *Proc. Australasian Database Conference 2001* (M. Orlowska, J. Roddick, Eds.), IEEE Computer Society, 2001, 52–59.
14. Schewe, K.-D., Zhao, J.: ASM Ground Model and Refinement for Data Warehouses, *Proc. 12th International Workshop on Abstract State Machines – ASM 2005* (D. Beauquier, E. Börger, A. Slissenko, Eds.), Paris, France, 2005.
15. Schewe, K.-D., Zhao, J.: Balancing Redundancy and Query Costs in Distributed Data Warehouses – An Approach based on Abstract State Machines, in: *Conceptual Modelling 2005 – Second Asia-Pacific Conference on Conceptual Modelling* (S. Hartmann, M. Stumptner, Eds.), vol. 43 of *CRPIT*, Australian Computer Society, 2005, 97–105.
16. Schmidt, J. W.: Some High Level Language Constructs for Data of Type Relation, *ACM Transactions on Database Systems*, **2**(3), 1977, 247–261.
17. Tannen, V., Buneman, P., Wong, L.: Naturally Embedded Query Languages, in: *Database Theory (ICDT 1992)* (J. Biskup, R. Hull, Eds.), vol. 646 of *LNCS*, Springer-Verlag, 1992, 140–154.
18. Thomson, E.: *OLAP Solutions: Building Multidimensional Information Systems*, John Wiley & Sons, New York, 2002.
19. Widom, J.: Research Problems in Data Warehousing, *Proceedings of the 4th International Conference on Information and Knowledge Management*, ACM, 1995.
20. Zamulin, A. V.: Typed Gurevich Machines Revisited, *Joint Bulletin of NCC and IIS on Computer Science*, **5**, 1997, 1–26.
21. Zhao, J., Ma, H.: ASM-Based Design of Data Warehouses and On-Line Analytical Processing Systems, *Journal of Systems and Software*, **79**, 2006, 613–629.
22. Zhao, J., Schewe, K.-D.: Using Abstract State Machines for Distributed Data Warehouse Design, in: *Conceptual Modelling 2004 – First Asia-Pacific Conference on Conceptual Modelling* (S. Hartmann, J. Roddick, Eds.), vol. 31 of *CRPIT*, Australian Computer Society, Dunedin, New Zealand, 2004, 49–58.

Report on an Implementation of a Semi-inverter

Torben Ægidius Mogensen

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
torbenm@diku.dk

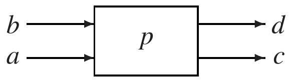
Abstract. Semi-inversion is a generalisation of inversion: A semi-inverse of a program takes some of the inputs and outputs of the original program and returns the remaining inputs and outputs.

We report on an implementation of a semi-inversion method. We will show some examples of semi-inversions made by the implementation and discuss limitations and possible extensions.

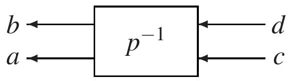
1 Introduction

Inversion of programs [6,3,8,13,11] is a process that transforms a program that implements a function f into a program that implements the inverse function of f, f^{-1}. A related technique was used in [4] to add backtracking to a program by adding code to rewind earlier computation.

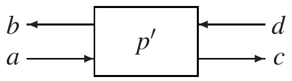
Semi-inversion [12,15] is a generalisation of inversion that allows more freedom in the relation between the function implemented by the original program and the function implemented by the transformed program. The difference can be illustrated with the following diagrams. Assuming we have a program p with two inputs and two outputs:



we can invert this into a program p^{-1} by "reversing all the arrows":



With semi-inversion, we can choose to retain the orientation of some of the arrows while reversing others, to obtain a program p':

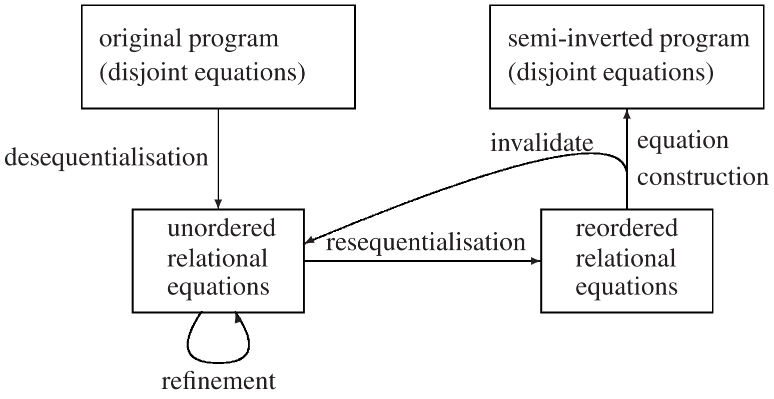


p' takes one of the inputs of p and one of its outputs and returns the remaining input and output. More formally, if running p with inputs (a,b) produces (c,d), then running p' with inputs (b,c) produces (a,d).

In [12], some theory about semi-inversion and a method for it is described, but no implementation existed at the time of writing. This paper describes an implementation of the semi-inversion method described in [12], reports experiences using it on some examples and conclude by discussing limitations and possible extensions.

2 The Semi-inversion Transformation

The steps of the semi-inversion transformation is shown in the following diagram:



The following sections will elaborate on each of these steps, further detail can be found in [12].

We will use a simple example to illustrate the process. The program below takes two lists of equal length and returns a pair of two lists, one containing the pairwise sum of the arguments and the other the pairwise difference.

```
pm ([],[]) = ([],[]);
pm (a:as,b:bs) =
  let (ps,ms) = pm (as,bs) in
    ((a+b):ps,(a-b):ms);
```

The syntax of the language we use is similar to Haskell, but there are a few semantic differences that are not important to this example.

2.1 Desequentialisation

Desequentialisation makes patterns and expressions into unordered sets of relations, where dependencies are carried through shared variables, much like in Prolog. The guarded equation $f\ P \mid G = E$ is desequentialised as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{f\ (P_1,P_2) = R_1 \cup R_2 \cup R_3} \\
\multicolumn{3}{l}{\text{where}} \\
\quad (P_1,R_1) & = I_p(P) & (I_p : \mathit{pattern} \rightarrow \mathit{var}^+ \times \mathit{relationset}) \\
\quad (P_2,R_2) & = I_e(E) & (I_e : \mathit{expression} \rightarrow \mathit{var}^+ \times \mathit{relationset}) \\
\quad R_3 & = I_g(G) & (I_g : \mathit{guard} \rightarrow \mathit{relationset})
\end{array}$$

The functions I_p, I_e and I_g are detailed in [12]. Our example is desequentialised as

```
pm (w,x,y,z) where {w=[], x=[], y=[], z=[]}
pm (w,x,y,z) where {w=:(a,as), x=:(b,bs), pm(as,bs,ps,ms),
                    u=+(a,b), v=-(a,b), y=:(u,ps), z=:(v,ms)}
```

Note that arguments and results of functions are not distinguished. We will collectively call inputs and outputs "in-outs".

2.2 Refinement

Refinement rewrites the relation set to make some relations more explicit:

- Related operators on the same arguments are combined to a single operator:
  $z = \texttt{+}(x,y),\ v = \texttt{-}(x,y)$ is replaced by $(z,v) = \texttt{+-}(x,y)$
- Equalities of tuples are split into equalities of the elements:
  $(x,y) = (z,v)$ is replaced by $x = z,\ y = v$

We can apply the first of the above rules to our program to get:

```
pm (w,x,y,z) where {w=[], x=[], y=[], z=[]}
pm (w,x,y,z) where {w=:(a,as), x=:(b,bs), pm(as,bs,ps,ms),
                    (u,v)=+-(a,b), y=:(u,ps), z=:(v,ms)}
```

2.3 Resequentialisation

Up to now, the steps do not depend on which in-outs will be arguments and results of the semi-inverted program. The next steps do, so we need to specify this. We do this by a *division* that by 1s and 0s indicate which in-outs are, respectively, inputs and outputs of the semi-inverted function. The division can also specify a name for the semi-inverted function. We will use the division `pm(0,1,0,1)=mp` to indicate that the second and last in-outs will be inputs while the first and third are outputs. The semi-inverse of `pm` will be called `mp`.

Each primitive operator has a list of possible semi-inverses, for example

$$\begin{array}{ll} z = \texttt{+}(x,y) & \Rightarrow x = \texttt{-}(z,y),\ y = \texttt{-}(z,x) \\ (p,q) = \texttt{+-}(x,y) & \Rightarrow (p,x) = \texttt{+-24}(q,y),\ldots,(x,y) = \texttt{+-12}(p,q) \end{array}$$

where $\texttt{+-24}(q,y)$ evaluates to $(q+y, q+2y)$ and $\texttt{+-12}(p,q)$ evaluates to $((p+q)/2,(p-q)/2)$. For `+-`, any two in-outs are sufficient to calculate the remaining, so there are 11 cases for `+-` (all cases where at least two in-outs are known).

Resequentialisation uses dependency analysis to list relations in possible evaluation order. Initially, the input variables are known, but as new operations are added to the sequence, more variables become known.

Given the division `pm(0,1,0,1)=mp`, the example program resequentialises to

```
pm (w,x,y,z) where {x=[], z=[], w=[], y=[]}
pm (w,x,y,z) where {x=:(b,bs), z=:(v,ms), (u,a)=+-24(b,v),
                    pm(as,bs,ps,ms), y=:(u,ps), w=:(a,as)}
```

For user-defined functions, it is initially assumed that any subset of input/output to the function can define the remaining input/output, but these assumptions may later be invalidated.

Resequentialisation may fail to succeed. If this happens for the top-level function call (that we wish to semi-invert), the whole semi-inversion process fails, but if it happens for an auxiliary function call, it may be caused by incorrectly assuming that the the input/output subset used for resequentialisation is sufficient. Hence, we mark this subset as invalid. This may require resequentialisation for other functions to be redone.

2.4 Constructing Equations

We now transform back from the relational form to equational form, while obeying the evaluation order found during resequentialisation. We do this in three steps:

1. Construct patterns from structural relations for new inputs.
2. Make guards from non-structural relations for new inputs.
3. Make expression from remaining relations.

The semi-inverse of the example program gives the following reconstructed equations:

```
mp([],[]) = ([],[]);
mp(b:bs,v:ms) =
  let (u,a) = +-24(b,v) in
    let (as,ps) = mp(bs,vs) in
      let y = u:ps in
        let w = a:as in (w,y);
```

After transforming each equation as described above, equations for the same function must by their patterns and guards divide the input into disjoint classes, i.e., there can be no overlap. If this is not the case, we mark the input/output subset for the function as invalid and backtrack.

In our example, the equations are clearly disjoint, so the semi-inverse is valid. The semi-inverter additionally rewrites special operators like `+-24` into "normal" operations and unfolds trivial or linear let-definitions. The actual output from the semi-inverter (after addition of a few newlines) is:

```
mp ([],[]) = ([],[]);
mp (b : bs,e_10_ : ms) =
  let (as,ps) = (mp (bs,ms)) in
    let a = (e_10_+b) in (a : as,(a+b) : ps);
```

3 Design Details

[12] leaves some implementation details unspecified. We have chosen the following:

Refinement. Addition and subtraction are combined, as shown above. Additionally, the constraints `p = /(x,y), q = %(x,y)`[1] are combined to `(p,q) = /%(x,y)`,

[1] Where `%` is the remainder operator.

which when `p, q` and `y` are known can be semi-inverted to `x = p*y+q`. Equalities between tuples are split, and if a variable is equated with a tuple, other occurrences of the variable are replaced by the tuple.

Resequentialisation. When there are several possible relations that can be selected during resequentialisation, the following priorities are used:

1. Tests with all parameters known.
2. Primitive operators with sufficient instantiation for semi-inversion.
3. Calls to already desired semi-inverses of user-defined functions, including the one that is currently being resequentialised.
4. Other calls to user-defined functions.

Backtracking. If the semi-inverter fails to semi-invert a desired semi-inverse, it prints a message, marks the semi-inverse invalid and starts over. Sometimes, it may find other semi-inverses that can be used instead, otherwise the message may help the user rewrite the program to get better results.

Starting over is clearly not the most efficient way of doing back-tracking, but in our experience the semi-inverter rarely back-tracks very much, if at all.

Determining disjointedness of equations. First, the variables are renamed to make the two equations use disjoint variables. Next, the patterns are unified. If unification fails, the equations are disjoint, otherwise the unifying substitution is applied to the guards. If the conjunction of the guards can never be true, the equations are disjoint. To test the guard, intervals of the possible values of integer variables are maintained, and if an interval becomes empty, the guard can't become true. A few additional cases of unsatisfiable constraints such as $e/=e$ are also considered.

There is room for improvement, as conjunctions like `x<y && y<x` are not recognised as unsatisfiable. The general problem of determining non-overlap of guards is undecidable, so there will always be cases that aren't handled.

Resequentialisation of a single equation is quadratic in the worst case, but back-tracking can make semi-inversion take exponential time, as a large fraction of the exponentially many possible divisions of a function can be tried. In the (admittedly small) examples we have tried, little backtracking occurs, so we don't believe this to be a problem in practice.

4 A More Ambitious Example: Multiplication of Binary Numbers

The semi-inverter can semi-invert the primitive multiplication operator into a division operator when one argument and the result are known.

But can we repeat this if we instead represent numbers as lists of bits?

The binary multiplication algorithm can be described by the recursive equations:

$$\begin{array}{ll} 1 & \times\, y = y \\ 2x & \times\, y = 2(x \times y) \\ (2x+1) & \times\, y = 2(x \times y) + y \end{array}$$

Note that we use 1 as a base case, as multiplication by zero doesn't have a unique left-inverse.

The last equation uses addition, so our first step is to semi-invert addition of binary numbers to obtain subtraction. To make the result of the subtraction unambiguous, we assume that numbers do not have leading zeroes (zero is represented by an empty list of bits).

In the addition above, the first argument is always at least as large as the second. This allows us to simplify the algorithm a bit, so addition (for little-endian lists of bits) looks like:

```
add(m,n) = adc(m,n,0);

adc(as,[],0) = as;
adc(as,[],1) = inc(as);
adc(a:as,b:bs,c) =
  let (s,c1) = add3(a,b,c) in s:adc(as,bs,c1);

add3(0,0,0) = (0,0);
add3(0,0,1) = (1,0);
add3(0,1,0) = (1,0);
add3(0,1,1) = (0,1);
add3(1,0,0) = (1,0);
add3(1,0,1) = (0,1);
add3(1,1,0) = (0,1);
add3(1,1,1) = (1,1);

inc [] = [1];
inc (0:as) | as/=[] = 1:as;
inc (1:as) = 0 : inc as;
```

Note that absence of leading zeroes is explicitly tested in `inc`. This test implicitly informs the semi-inverter of the invariant, which allows the semi-inverter to use the invariant to determine disjointedness of equations. It is often required to assert invariants to make the semi-inverter work, we shall see more of this when we tackle the multiplication rules.

With the division `add(0,1,1) = sub` (which says that we know the second argument and the result of `add` and that the semi-inverse should be named `div`) we get the following output from the semi-inverter:

```
sub (n,e_75_) = (adc_0111 (n,0,e_75_));

adc_0111 ([],0,as) = as;
adc_0111 ([],1,e_78_) = (inc_01 e_78_);
adc_0111 (b : bs,c,s : e_82_) =
  let (a,c1) = (add3_01110 (b,c,s)) in a : (adc_0111 (bs,c1,e_82_));

add3_01110 (0,0,0) = (0,0);
add3_01110 (0,1,1) = (0,0);
add3_01110 (1,0,1) = (0,0);
```

```
add3_01110 (1,1,0) = (0,1);
add3_01110 (0,0,1) = (1,0);
add3_01110 (0,1,0) = (1,1);
add3_01110 (1,0,0) = (1,1);
add3_01110 (1,1,1) = (1,1);

inc_01 [1] = [];
inc_01 1 : as | as/=[] = 0 : as;
inc_01 0 : e_91_ = 1 : (inc_01 e_91_);
```

The test for absence of leading zeroes in `inc` is now a guard that makes the two first rules for `inc_01` disjoint. Note that the semi-inverter automatically finds the required semi-inverses of `adc`, `add3` and `inc`, using names that indicate the division used.

Getting multiplication to semi-invert is a bit more tricky. We start with a straight rewrite of the equations for multiplication into the syntax of the language:

```
mul([1],bs) = bs;
mul(0:as,bs) = 0:mul(as,bs);
mul(1:as,bs) | as/=[] = add(0:mul(as,bs),bs);
```

Note that the guard `as/=[]` is added to ensure non-overlapping equations. We give the semi-inverter this program and a division `mul(0,1,1)=div`. The result is an error-message:

```
Overlap:
div (bs,bs)
div (bs,0 : e_4_)
```

The semi-inverter has found that (at least) two equations of the semi-inverse of `mul` overlaps. This isn't too surprising, as the known argument `bs` isn't used to select equations, so we have only the result to do this, and the right-hand sides of the equations for `mul` are not clearly disjoint. Letting the first argument of `mul` be the known argument doesn't help either, as this makes both arguments to `add` unknown, so clearly no semi-inverse of this can be found. Instead, we must make pattern-matching on `bs` while still keeping it as one of the arguments to the addition.

We have three cases: `[1]`, `0:bs` and `1:bs`, where `bs/=[]` in the last case. The two first cases are so simple that we don't need to special-case on `as`, but instead use mirror-images of the rules for `as = [1]` and `as = 0:as'`. The remaining case gives us the three original rules specialised for `1:bs` with `bs/=[]`:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] = add(0:mul(as,1:bs),1:bs);
```

In the last rule, we can unroll a step of the addition, so we obtain:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] = 1:add(mul(as,1:bs),bs);
```

We are nearly there, but the semi-inverter reports overlap between the third and last equation:

```
Overlap:
div (1 : bs,1 : bs) | bs/=[]
div (1 : bs,1 : e_23_) | bs/=[]
```

But we can see that `e_32_`, which is the result of the addition in the last line, must be different from `bs`, so we can add this as an assertion:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] =
  let xs = add(mul(as,1:bs),bs)
    in let True() = xs/=bs in 1:xs;
```

With this test in place, we can call the semi-inverter with the division `mul(0,1,1) = div` and successfully obtain a semi-inverse (which divides its second argument by its first argument):

```
div ([1],as) = as;
div (0 : bs,0 : e_66_) = (div (bs,e_66_));
div (1 : bs,1 : bs) | bs/=[] = [1];
div (1 : bs,0 : e_77_) | bs/=[] = 0 : (div (1 : bs,e_77_));
div (1 : bs,1 : xs) | bs/=[] && xs/=bs =
  let as = (div (1 : bs,(add_011 (bs,xs)))) in
    let (True ()) = (as/=[]) in 1 : as;
```

The above is taken straight from the output of the semi-inverter, only adding a few newlines and omitting the definition of `add_011`, which is identical to `sub`.

Note that the assertion we inserted in the `mul` function is now part of the guard of the last equation to `div`. This test (that ensures disjoint equations) corresponds to the test that in the traditional algorithm for binary division stops the repeated doubling of the divisor. Note, also, that the pattern-matching on `as` has become an assertion.

Since the division is a semi-inverse of multiplication, it is only defined when the divisor divides evenly into the other argument.

This development shows that semi-inversion in its present state is far from being something you can just use without thought. This is similar to how you will often need to rewrite programs (using *binding-time improvements* [9,5,1]) to get good results (or even termination) from partial evaluators.

5 Tail-Recursive Programs

Programs with tail-recursive functions can give problems for the semi-inversion transformation.

Glück and Kawabe [6] observe that tail-calls after inversion are similar to left-recursive productions, and that overlapping equations are similar to overlapping productions in a grammar. They suggest using methods inspired by LR-parsing to solve these problems, just like LR-parsing handles the equivalent problems in grammars. Their method might work also for semi-inversion, but a simpler method is sufficient for most cases. Consider the reverse function:

```
reverse(xs) = rev(xs, []);

rev([], acc) = acc;
rev(x : xs, acc) = rev(xs, x : acc);
```

The essense of the problem is that, with the current method, the call-trees of the original and semi-inverted programs are isomorphic, the only difference being the order of children of each node. But to invert the above, we really want to run the iteration in `rev` backwards, i.e., change the call-tree. Glück and Kawabe's solution is to apply a program transformation to the inverted program to change the call-tree, but we propose to apply a simpler transformation prior to (semi-)inversion instead. The idea is that tail-recursion can be seen as iteration, so we introduce an explicit iteration construct in the language: `loop` f e calls f with the value v_0 of e as argument. If no rule of f matches, it returns v_0 unchanged, but if a rule matches, it applies f to v_0 to get v_1 and repeats the procedure, i.e., checks if there is a matching rule for v_1 and so on. In short, the loop applies f repeatedly until no rule matches, at which point it returns the current value of the argument.

We can rewrite the `reverse` function to use this loop construct:

```
reverse(xs) = let ([], acc) = loop revStep (xs, []) in acc;

revStep(x : xs,acc) = (xs, x : acc);
```

Note that `revStep` corresponds to the recursive case of `rev`, but instead of making the tail-recursive call, it just returns what would have been the arguments to the call. The non-recursive call has now been incorporated into the function that does the looping. Notably, both the initialization and termination patterns are part of this function.

We can now invert `reverse into`

```
reverse2 acc = let (xs, []) = loop revStep_0011 ([], acc) in xs;

revStep_0011 (xs, x : acc) = (x : xs, acc);
```

The loop has been inverted by swapping argument and result and inverting the iteration-step function in the usual way. We can, finally, transform the result back to using a tail-recursive function by making the let-expression of the calling function into the base case for the recursion, mirroring the transformation from tail-recursive function to loop:

```
reverse2 acc = revStep_0011 ([], acc);

rev_0011 (xs, []) = xs;
rev_0011 (xs, x : acc) = rev_001(x : xs, acc);
```

It is interesting to note that the non-recursive rule of `rev_0011` comes from the parameters to the call of the original `rev` and vice-versa. Hence, in retrospect it does not seem reasonable to expect inversion of the program by tansforming each function in isolation. Transforming into the loop form is a way of bringing together the information needed for the successful transformation.

Note, also, that the resulting tail-recursive function has disjoint equations, as required. If we can not rewrite the loops in a (semi-)inverted program back into loop-free disjoint equations, the semi-inversion is not valid, in which case we must backtrack and possibly eventually fail to do (semi-) inversion at all.

As an example of this, consider the function

```
revapp(xs, ys) = rev(xs, ys);

rev([], acc) = acc;
rev(x : xs, acc) = rev(xs, x : acc);
```

Note that `rev` is the same as before. In loop form, this becomes

```
revapp(xs, ys) = let ([], acc) = loop revStep (xs, ys) in acc;

revStep(x : xs,acc) = (xs, x : acc);
```

which we can invert initially to

```
revapp_001(acc) = let (xs, ys) = loop revStep_0011 ([], acc) in (xs, ys);

revStep_0011 (xs, x : acc) = (x : xs, acc);
```

When we rewrite the loop back into functional form, we get

```
revapp_001(acc) = rev_0011 ([], acc);

rev_0011 (xs, ys) = (xs, ys);
rev_0011 (xs, x : acc) = rev_0011 (x : xs, acc);
```

whis has overlapping patterns for `rev_0011`. It should not really be a surprise that we can't invert `revapp`, as it is not injective. But note that the difference is not in the form of the tail-recursive function, but in its calling context. Hence, it is clear that we can not treat a tail-recursive function separate from its context.

We can predict whether a loop can be converted back to functional form after inversion by looking at the original call to the tail-recursive function: If the argument to the call is disjoint from the right-hand sides of the equations for the tail-recursive function, this will make the equations of the inverted function disjoint too. So we can choose to only rewrite to loop form only calls that have this property. This requirement is similar to the requirements for loops in [8], where the precondition of a loop is required to not overlap the postcondition of the loop body.

It does not seem possible to semi-invert a loop other than by full inversion or no inversion at all.

Not all tail-recursion is easily rewritten to a loop. Indirect tail-recursion will require more extensive rewriting, as will tail-recursive function with several non-recursive rules. Multiple recursive rules are no problem, as long as their inverses are disjoint equations.

At the moment, we have not implemented the transformations between tail-recursive functions and loops, but the example above (and a few more complicated examples) written with explicit loops in the source text have been transformed by the current implementation.

This includes inverting a compiler from queue-machine code [14] to syntax trees to get a compiler from syntax tres to queue machine code[2]. In a queue machine, instructions take their operands from the front of a queue and put their results at the end of the queue. Building a syntax tree is easily done by letting the queue contain subtrees rather than values and let instructions build larger subtrees from the operator in the instruction and the subtrees in the queue:

```
fromQueue(prog) = let ([],[t]) = loop fq (prog,[]) in t;

fq(Const n : p, q) = (p, Num n : q);
fq(Bop op : p, q) = let (a,q1) = dequeue q in
                      let (b,q2) = dequeue q1 in
                        (p, Binop (op,a,b) : q2);
fq(Uop op: p, q) = let (a,q1) = dequeue q in
                     (p, Unop (op, a) : q1);

dequeue([x]) = (x,[]);
dequeue(x:xs) | xs/=[] = let (y,ys) = dequeue(xs) in
                          (y,x:ys);
```

The queue (q) is represented as a list where the front of the queue is the end of the list. dequeue returns a pair of the head of the queue and the rest of it. Inverting with the division fromQueue(0,1)=toQueue; produces the following compiler from syntax trees to queue code:

```
toQueue t = let (prog, []) = loop fq_0011 ([], [t]) in prog;

fq_0011 (p, (Num n) : q) = ((Const n) : p, q);
fq_0011 (p, (Binop (op, a, b)) : q2) =
   ((Bop op) : p, dequeue_011 (a, dequeue_011 (b, q2)));
fq_0011 (p, (Unop (op, a)) : q1) =
   ((Uop op) : p, dequeue_011 (a, q1));
```

[2] Thanks to Rustan Leino of Microsoft Research for suggesting this problem.

```
dequeue_011 (x, []) = [x];
dequeue_011 (y, x : ys) = let xs = dequeue_011 (y, ys) in
                          let True () = xs/=[] in x : xs;
```

`dequeue_011` is now an enqueue operation, but on a reversed queue.

6 Conclusion

As the multiplication example shows, it is sometimes necessary to rewrite programs and add assertions of invariants to get successful semi-inversion. This example, though small, is rather complex, so we don't expect nearly as much rewriting to be necessary for the majority of programs. Even so, semi-inversion will probably remain a tool for experts until the technology matures.

Ideally, a semi-inversion method should discover such invariants, but it is unrealistic to expect it to always do so, as discovery of nontrivial invariants is uncomputable. As a consequence, it may sometimes be necessary to provide such invariants as extra information to the semi-inversion process. Adding such redundant assertions is conceptually similar to using *binding time improvements* [9,5,1] to improve the result of partial evaluation.

In addition to making invariants explicit, it may be necessary to combine several user-defined functions, just like some predefined operators are combined at the refinement stage. For example, the function that returns the last element of a list can not be inverted on its own, nor can the function that returns all but the last element of a list. In combination, they can, but only if the functions are merged into a single function (such as `dequeue` above). Such merging of functions is called *tupling* and can be automated [2, 16].

Another non-trivial program that has been semi-inverted by the semi-inverter is an interpreter of the invertible stack language used in [6]. The interpreter was semi-inverted by specifying the program and output as known while the input is unknown. The result is an *inverse interpreter* [7]. This example did not require any assertion of invariants or similar tricks but, admittedly, the language was designed by Glück and Kawabe to be easily invertible.

Tail-recursive functions are in the semi-inverter handled by transformation to explicit loops, and they can only be left unchanged or fully inverted.

Another limitation of the current method is that it only works on first-order equations. We are currently working on extending it to handle simple cases of higher-order functions.

References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10. ACM Press, 1992.
2. Wei-Ngan Chin and Hu Zhenjiang. Towards a modular program derivation via fusion and tupling. In *Proceedings of the first ACM SIGPLAN Conference on Generators and Components*, pages 140–155. ACM Press, 2002.

3. Edsger W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, LNCS 69, pages 54–57. Springer-Verlag, 1978.
4. Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.
5. Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.
6. Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming. Proceedings*, LNCS 2998, pages 291–306. Springer-Verlag, 2004.
7. Robert Glück, Youhei Kawada, and Takuya Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 10–19. ACM Press, 2003.
8. David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
9. Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In [10], pages 83–100, 1991.
10. Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*. Springer-Verlag, August 1991.
11. Ed Knapen. *Relational programming, program inversion and the derivation of parsing algorithms*. Master's thesis, Eindhoven University of Technology, 1993.
12. Torben Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE'05*, Lecture Notes in Computer Science 3676, pages 189–204. Springer-Verlag, 2005.
13. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In Dexter Kozen, editor, *Mathematics of Program Construction. Proceedings*, LNCS 3125, pages 289–313. Springer-Verlag, 2004.
14. Bruno Richard Preiss. *Data Flow on a Queue Machine*. PhD thesis, Department of Electrical Engineering, University of Toronto, 1987. 185 pp.
15. A. Y. Romanenko. The generation of inverse functions in Refal. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 427–444. North-Holland, 1988.
16. Jens Peter Secher. Driving in the jungle. In Olivier Danvy and Andrzej Filinsky, editors, *Programs as Data Objects. Proceedings*, LNCS 2053, pages 198–217. Springer-Verlag, 2001.

Loop Parallelization in Multi-dimensional Cartesian Space

Saeed Parsa and Shahriar Lotfi

Faculty of Computer Engineering
Iran University of Science and Technology
parsa@iust.ac.ir, shahriar_lotfi@iust.ac.ir

Abstract. Loop parallelization is of great importance to automatic translation of sequential into parallel code. We have applied Diophantine equations to compute the basic dependency vector sets covering all possible non-uniform dependencies between loop iterations. To partition the resultant dependencies space into multi-dimensional tiles of suitable shape and size, a new genetic algorithm is proposed in this article. Also, a new scheme based on multi-dimensional wave-fronts is developed to convert the multi-dimensional parallelepiped tiles into parallel loops. The problem of determining optimal tiles is NP-hard. Presenting a new constraint genetic algorithm in this paper the tiling problem is for the first time solved, in Cartesian spaces of any dimensionality.

1 Introduction

The aim is to present a complete and comprehensible approach to convert nested sequential loops into parallel loops. The conversion is performed in three stages. In the first stage, data dependency analysis is performed to find dependencies between the loop iterations [4, 18]. We have applied Diophantine equations [4, 18] to find basic dependence vector sets, BDVS, covering all possible non-uniform dependencies in between the loop iterations.

A main point to consider is to work out the optimal number of iterations to be assigned to each processing element such that the inter-processors communication and synchronization cost is minimized. To achieve this, in the second stage, the loop iterations are partitioned into chunks, called *tiles*, with minimal inter-dependencies [10, 12, 15]. The main problem in tiling for loop parallelization, considered in this paper, is to find optimal sizes and shapes for the tiles such that each tile covers a predefined number of interrelated points considering inter-tile communication restrictions such as: (1) the chain of vectors connecting any two points in a tile, reside in that tile, (2) the number of vectors connecting the tile to its neighboring tiles are minimal and (3) any other constraints imposed by the user.

One of the first papers on tiling dates back to the mid-1980s by Wolfe [16, 17]. In Wolfe's approach, the loop iteration space is transformed into parallelepiped tiles such that the chain of vectors connecting any two dependent iteration points resides in a same tile. The major difficulty with this approach is to find the optimal transformation such that the number of memory accesses to execute the loop iterations is minimized. To resolve the difficulty a heuristic algorithm has been devised by Wolfe. The problem is also addressed by Griebl *et al.* [11]. In the approach proposed below in this paper, the problem is solved by finding an optimal shape for the tiles instead of using rectangular tiles and transforming the iteration space.

In an approach presented by Kandemir *et al.* [12], to create optimal tiles a complete set of constraints, described below in Section 2.3, are defined. In Kandemir's approach, it is shown how to represent multi-dimensional tiles. However, only 2-dimensional tiles can be created with the Kandemir's approach. In addition in this approach, the direction of one side of the tile has to be vertical. As described in Section 2.3.3 below, the shape and direction of a tile directly affects the number of dependencies between the tile and its neighboring tiles.

Multi-dimensional tiles are created by an approach presented by Cociorva in reference [6]. However, in this approach the objective is not loop tiles and each tile represents a part of an array to be reused within a nested loop. None of the existing loop tiling approaches can create multi-dimensional tiles with minimum tiles inter-dependencies, considering the complete set of constraints defined by Kandemir. It has been argued that since shallow loop nests are common and important in scientific applications, none of the existing approaches emphasize on generation of tiles with more than 2-dimensions, for loop parallelization. However, we can not restrict the world of scientific programming to loops with a maximum depth of three. For instance, using a multi-grid approach, adds at least an extra dimension to the nested loops for the numerical solution of integral and differential equations on a finite difference or finite element grid.

In general, loop tiling is an NP-hard problem [1, 5, 10, 12, 15]. Determining an optimal size and shape for tiles is an optimization problem. These types of optimization problems can be best solved using evolutionary algorithms [3, 8, 9]. We have developed a new genetic algorithm (GA) to create suitable parallelepiped tiles of any dimensionality and size. In the third stage, considering the shape and size of the tiles, parallel loops are generated. We have developed a new scheme based on the wave-fronts [10, 15] to create parallel loops.

The remaining sections are organized as follows: In Section 2, a constrained genetic algorithm is presented that uses a new encoding scheme to represent multi-dimensional tiles. Section 3 offers a new scheme based on wave-fronts to convert the tiled space into parallel loops. In Section 4, a complete example is given and finally, in Section 5, an evaluation that reflects the reliability, stability and performance of our proposed genetic algorithm is discussed.

2 A New Genetic Tiling Algorithm

A tile in this paper is a set of loop iterations to be executed on a single processor. The aim is to partition loop iterations into tiles with minimal inter-dependencies, where each of the iterations is represented as a point in Cartesian space. The problem space is considered as a multi-dimensional Cartesian space where each dimension of the Cartesian space represents a loop index. In this section a new genetic algorithm for solving the tiling problem in Cartesian spaces of any dimensionality is presented.

Genetic algorithms are adaptive heuristic search algorithms based upon Darwin's evolutionary ideas of natural selection and survival rule [3, 8, 9]. Below, in Fig. 1, the main body of our genetic tiling algorithm is presented.

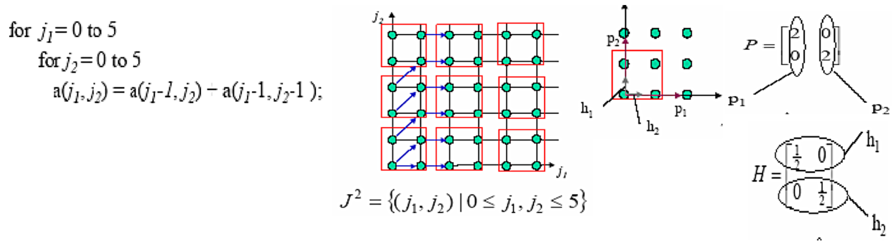
```
Genetic Algorithm:
  Generation number := 0
  Generate the initial population at random using a normal distribution
  Evaluate the fitness of each chromosome in the initial population
  Keep the fittest chromosome in the population
  for Generation number := 1 to Number of generations do
  begin
    Select the fittest chromosome using fitness and penalty functions
    Generate an Intermediate Population of the selected chromosomes
    New population := Intermediate population
    Randomly select chromosomes with a predefined probability
    Recombine the selected chromosomes into the new population
    Mutate the recombined chromosomes with a predefined probability
    Evaluate the fitness of each chromosome in the new population
    Keep the fittest chromosome in the current generation (Elitism)
    Keep the fittest chromosome in the last generations
  end
```

Fig. 1. Genetic loop tiling algorithm

Genetic algorithms normally start with an initial population of randomly generated solutions. In the above algorithm, an initial population of the problem solutions is created randomly. Each solution represents a tile and is called a *chromosome*. A chromosome is an individual in a genetic population. To represent a tile as a chromosome, an encoding operator is presented in Section 2.1. The objective is to generate suitable tiles, which fit into the local memory of the parallel processors and when loaded into the memory execute at once without any delay. The objectives are fully described in Section 2.3. Evaluation of quality of the tiles is discussed in Section 2.3. The tiles with highest quality or in the other words the fittest chromosomes in the population are selected and transferred into an intermediate population. Then, the individuals in the intermediate population are recombined to produce a new population. To combine chromosomes, an operator called crossover, described in Section 2.2, is used. To achieve genetic diversity a mutation operator, described in Section 2.2, is used. The fittest chromosome in a generation is moved to the next generation. The fittest chromosome within the generations is selected as the final solution. This process is repeated for a number of generations until the evolution is completed and the highest quality chromosome is found.

2.1 Encoding Scheme and Initial Population

Tiles should be represented in a form that can be processed by the genetic algorithm. For this purpose, each parallelepiped shaped tile is coded as a matrix. Each column of the matrix corresponds to one edge of the parallelepiped tile, beginning at the leftmost corner of the tile. For instance, the matrix P in Fig. 2 represents a square shaped tile. Since the data dependency vectors between the iterations are uniformed [2, 7, 14], the corresponding tiles are similar and so only the tile beginning at the origin is considered. Hence, only the first tile beginning at the origin of the Cartesian coordinates is considered as a chromosome. The chromosome is represented by an n×n matrix P = [p_i], where p_i represents the ith edge of the corresponding tile. Also, to simplify certain computations on tiles, described in sections 2.3.2 and 2.3.3, a tile is represented by a matrix H = P^{-1}. Each row of the matrix H represents a vector perpendicular to an edge of the tile. We have applied the Gaussian algorithm to compute the inverse of the matrix.



```
for $j_1$ = 0 to 5
    for $j_2$ = 0 to 5
        a($j_1$,$j_2$) = a($j_1$-1,$j_2$) - a($j_1$-1,$j_2$-1 );
```

$J^2 = \{(j_1, j_2) \mid 0 \le j_1, j_2 \le 5\}$

j_2
j_1
p_2
p_1
h_1
h_2
$P = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
$H = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$

Fig. 2. Encoding a tile to a matrix

In the first stage of our genetic tiling algorithm, an initial population of chromosomes is created by applying a random number generator that uses a normal distribution. To generate a chromosome, the coordinates of the end point of each edge, beginning at the origin, are defined randomly and saved in a separate column of the chromosome matrix. The range of these random numbers is limited to the boundaries of the iteration space.

2.2 Generating Genetic Populations

As described above, at the beginning of a genetic algorithm, an initial population of solutions is randomly created. A new generation is created by selecting couples of parent chromosomes from within the current population. The selected couples are either directly transferred to the new population or combined by applying a crossing over operator. To select two parent chromosomes for the crossing over operation, we have applied the improved roulette wheel selection method [8]. In general, the probability of combining the selected parents is set to a value in the range 0.6 to 0.9. Schema theory [9] discusses the effects of the probability of crossing over and mutation on the convergence rate of genetic algorithms. In our genetic tiling environment the crossing over probability is an input parameter, defined by the user. The crossing over operator is applied, by combining the corresponding columns of the chromosomes matrices, using a uniform recombination operator as follows [3, 9]:

Column number i of the child chromosome =
Column number i of the first parent * λ + Column number i of the second parent * (1 – λ) (1)

In the above relation $0 \le \lambda < 1$ is selected randomly. The columns are selected for the crossing over operation using a binary mask, which is also generated randomly.

In order to maintain the diversity in the new population, some of the newly generated chromosomes are selected for the mutation. The probability of applying mutation is normally set to a value within the range 0.001 to 0.1. The mutation operator alters the shape of a selected tile randomly. The alteration is achieved by randomly selecting two rows in the matrix representation of the tile and then swapping the selected rows. Using this approach the absolute value of the determinant of the mutated tile, representing the tile size, remains intact.

2.3 Tiling Objectives

In order to evaluate the fitness of a tile, *x*, the quality of the tile and the penalty values for violating the tiling constraints have to be evaluated [13]. The quality of a tile represents its fitness. The fitness of a chromosome is estimated by the following relation:

$$\textbf{Fitness (x) = Objective (x) + Penalty (x)} \tag{2}$$

There are 5 sub-objectives, described in this section, concerning a tile shape and size that have to be covered by the objective function as follows [12]:

$$\textbf{Objective (x)} = \mathbf{T_{cost}} = \mathbf{p_{nn}C_{I/O}} + \mathbf{V_{comp}t_{I/O}} + \lceil \mathbf{V_{comm}} / \mathbf{K} \rceil C_{comm} + \mathbf{V_{comm}t_{comm}} + (\mathbf{V_{comp}} - \mathbf{M})^2 \tag{3}$$

In the above relation, p_{nn} is the last element of the matrix P, $C_{I/O}$ is startup cost for an I/O read (write), $t_{I/O}$ is the cost of reading (writing) an element from (into) a file, C_{comm} is startup cost for communication, t_{comm} is the cost of communicating an element, V_{comp} is the computation volume, V_{comm} is the communication volume, M is the size of the available local memory of multi-processors and K is maximum message length, passed in between the processors. The overall objective is to minimize the objective function T_{cost}. To minimize the objective function, its sub-objectives have to be minimized. The sub-objectives are further described below.

2.3.1 I/O Cost

The I/O cost of a tile is determined by the number of file accesses I/O calls, $p_{nn}C_{I/O}$, required to read it from disk. Under row-major layout assumption, in order to minimize the number of file accesses, the number of (sub)-row reads should be minimized. This may not be always possible in practice as minimization of I/O calls can increase the cost of communications. In Fig. 3, tile (1) and tile (2) have the same computation volume (8 data points); however, I/O cost (number of sub-rows) of tile (2) is 4 while that of tile (1) is 2. Everything else being equal, tile (1) is a better choice than tile (2). Note that this constraint is unique to iteration space tiling [12].

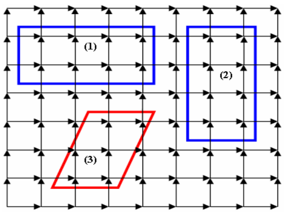


Fig. 3. Three different shapes of a tile

2.3.2 Load Time

The load time of a tile is determined by $V_{comp}t_{I/O,}$ where $t_{I/O}$ is the cost of reading or writing an element from (into) a file and V_{comp} for a tile represented by a matrix $P = H^{-1}$ is equivalent to determinant of P which is the number of iterations or index points contained in a tile [10, 12, 15]:

$$\mathbf{V_{comp}\,(H) = 1\,/\,|det\,(H)|} \tag{4}$$

As an example consider the tile in Fig. 4.(1). The absolute value of determinant of H is 1 / 20, and there are 20 points in the tile.

2.3.3 Communication Startup Cost

Dependency vectors of a loop are conventionally defined as columns of a matrix D. The number of dependency vectors starting in a tile and ending in its neighboring tiles is equal to sum of the elements of the matrix (1 / |det (H)|) H.D [10, 15].

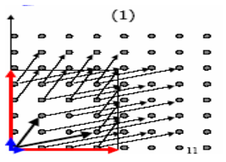
$$\mathbf{V_{comm}\,(H) = (1\,/\,|det\,(H)|)\sum_{i=1}^{n}\;\sum_{k=1}^{n}\;\sum_{j=1}^{m} h_{i,\,k}\,d_{k,\,j}} \tag{5}$$

The reason is that each edge of the tile is perpendicular to a vector represented as a row of H and H.D computes the inner product of the vectors perpendicular to tile edges and the dependency vectors. Here, H.D represents the proportion of the iteration points within a tile which start vectors intersecting the tile edges.

Communication startup cost is determined by multiplying the number of messages to be passed between the tiles, $\lceil V_{comm} / K \rceil$, by the startup cost for communication, C_{comm}, which is the cost of transferring a single message. Here, K is the maximum length of a message and V_{comm} is the communication cost of a tile.

For instance, in Fig. 4.(1), $(H.D)_{1,\,1}$ is equal to 3 / 4 and the number of points in the tile which is the absolute value of the determinant of the matrix P is 20. Hence, the number of vectors d_1= (3, 1) cutting the first edge of the tile is computed as 20 × 3 / 4 = 15 which is exactly the same as the number of vectors passing through the horizontal edge of the tile. There may be dependence vectors passing through more

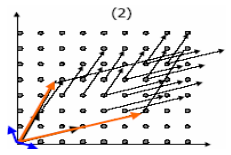
```
for j₁ = 0 to 7 do
   for j₂ = 0 to 7 do
      a (j₁, j₂) = a (j₁ - 3, j₂ -1) + a (j₁ - 1, j₂ - 2);
```



$$D=\begin{bmatrix}3 & 1\\ 1 & 2\end{bmatrix}, P_1=\begin{bmatrix}4 & 0\\ 0 & 5\end{bmatrix}, P_2=\begin{bmatrix}6 & 2\\ 2 & 4\end{bmatrix}$$

$$V_{comp1}=V_{comp2}=20$$

$$V_{comm1}=27,\; V_{comm2}=19$$



$$\mathbf{H_1=\begin{pmatrix}1/4 & 0\\ 0 & 1/5\end{pmatrix}\quad H_1D=\begin{pmatrix}3/4 & 1/4\\ 1/5 & 2/5\end{pmatrix}\qquad H_2=\begin{pmatrix}1/5 & -1/10\\ -1/10 & 3/10\end{pmatrix}\quad H_2D=\begin{pmatrix}1/2 & 0\\ 0 & 1/2\end{pmatrix}}$$

$\mathbf{V_{comp\ a} = |det\,(P_1)| = 20}$ $\mathbf{V^{ideal}_{comm.\ a} = 20 \times (3/4 + 1/4 + 1/5 + 2/5) = 32}$

$\mathbf{V_{comp\ b} = |det\,(P_2)| = 20}$ $\mathbf{V^{ideal}_{comm.\ b} = 20 \times (1/2 + 0 + 0 + 1/2) = 20}$

Fig. 4. Two-dimensional tiles

than one edge of the tile. These vectors are considered twice. For instance, the computed value for V_{comm} of the tile, depicted in Fig. 4.(1), is 32 but it should be 27. Hence the above formula approximates the value of V_{comm}.

2.3.4 Communication Time

The total communication time of a tile with its neighbors is determined by multiplying the communication cost, V_{comm}, by the cost of communicating an element, t_{comm}. The reason is that tile iterations are expected to execute serially on a same processor. On the other hand, the smaller the size of a tile the higher the communication cost between the tiles will be [12].

2.3.5 Fitting Tiles in Local Memory

The iteration number and the shared data accessed by the iterations are kept into the local memory of the processor on which the iterations are to be executed. There should be enough space to keep these iterations and shared data in the local memory of a processor. Hence, the number of iteration points residing on a tile, V_{comp}, should not exceed the size of the part, M, of the local memory of the parallel processor assigned to the tile. In other words, the memory constraint can be defined as: $M \geq V_{comp}$ [12]. The tile size, V_{comp}, should be very close to the size of the available memory. This is achieved by minimizing the value of $(V_{comp} - M)^2$ in relation (3) above.

2.4 Penalty Function

Considering a given constraint, the population of solutions in each generation can be divided into two groups of feasible and infeasible solutions. A feasible solution satisfies the desired constraint. The degree of infeasibility of a solution, c, in a genetic generation is considered as the distance, $d(c) = g(c) - Q_{best}$, where g (c) is the constraint value of c and Q_{best} is the quality of the most feasible solution. We have considered a normal distribution for these distances. To penalize an infeasible solution, c, its fitness can be reduced by a penalty factor $P \in [0, 1]$ as follows:

$$\textbf{Penalized\_Fitness (c) = Estimated\_Fitness (c) * P (c)} \tag{6}$$

P (c) = 1 - (u_p (c) - u_p (c$_0$)) / (u_p (c$_f$) - u_p (c$_0$))
u_p (c) = N (d (c), μ, σ$_t$), where N is a normal distribution function and μ is the mean value

$$\mu = 0, \quad \sigma_t^2 = \quad d(c_i), \forall c_i \in \text{ population in generation t} \tag{7}$$

In the above relation, c_0 and c_f are two solutions such that the distance d (c_0) from feasible solutions is maximum and d (c_f) is minimum; u_p (c) is unadjusted penalty value and p (c) is the penalty value in range [0, 1]. Using the above relations, a penalty value of 0 is assigned to the most infeasible solution and a penalty value of 1 is assigned to each feasible solution. In our genetic tiling algorithm two constraints described below are applied to the tiles. To evaluate the penalty value for violation of these two constraints the following relation is used:

$$\begin{aligned}&\textbf{Fitness (x) = Objective (x) + P (x) * Objective (x),}\\&\mathbf{P = (W_1 * \text{Legality constraint} + W_2 * \text{Boundary Constraint}) / (W_1 + W_2)}\\&\textbf{If x is feasible Then P (x) = 0 Else P (x) > 0, 0 \quad P (x) \quad 1}\end{aligned} \tag{8}$$

2.4.1 Tile Legality

The shape and size of a tile should be determined in such a way that the sequence of vectors connecting any two dependent loop iterations or points in the tile, reside in that tile. This issue is referred to as the atomic or tile legality constraint. To satisfy the legality constraint, there should be no mutual dependencies between the tiles. To achieve this, the length and direction of a tile sides should be defined in such a way that the set of vectors connecting any two dependent loop iterations or points in the tile, reside inside the tile.

Arbitrary clustering of data space points into tiles might result in dependency vector cycles between the tiles. Tile shapes that produce effective dependency vector cycles are called illegal tiles. The reason for illegality is that processing of tiles must be atomic in the sense that a tile must take all the data it requires from outside before execution of the iterations addressed by the tile begins, and all the data required by other tiles should be available after the execution is completed. Allowing effective dependency vector cycles among tiles violates this requirement. As an example, consider the data space graph shown in Fig. 3, tiles (1) and (2) are legal, while tile (3) is illegal [10, 12, 15].

As described in Section 2.3.3 the inner product of vectors the matrices $H = P^{-1}$ and D for a tile P, indicates the number of dependency vectors beginning at P and passing through the edges of P. If the element $(H.D)_{i,\,j}$ is zero, it means that the dependency vector d_j is in parallel with the i^{th} edge of P. If $(H.D)_{i,\,j}$ is positive, it means that d_j cuts the edge and is in the same direction as the flow of execution of the tiles otherwise it is in opposite direction and therefore, the tile can not be atomic. In summary for a tile P to be legal, it must hold $H.D \geq 0$ where, $H = P^{-1}$ [10, 12, 15].

2.4.2 Boundary Constraint Tile Diameter

The diameter of a tile should not exceed the boundaries of the iteration space. Obviously if a tile exceeds the boundaries of the iteration space then there will be some iteration points within the tile which remain outside the scope of the loop indices. The reason behind this constraint is to have at least one complete tile in a loop iteration space.

3 Parallel Loop Generation

After computing the shape and size of the tiles, in the third stage, the iteration space, J^n, is tiled. The tiled space is called J^{sn}. Within the tiled space each tile is considered as a point. The coordinates of a point $J^s = (j^s_1, j^s_2, \ldots, j^s_k, \ldots, j^s_n)$ in the tiled space which includes an iteration point $J = (j_1, j_2, \ldots, j_k, \ldots, j_n)$ in the original iteration space is computed as follows [10, 15]:

$$\mathbf{J^s = (\lfloor HJ^T \rfloor)^T} \tag{9}$$

Given a point J^s in the tiled space, the coordinates of the lower left corner, $J = (j_1, j_2, \ldots, j_k, \ldots, j_n)$, of the tile in the original loop iteration space is computed as follows [10, 15]:

$$\mathbf{J = (PJ^{sT})^T} \tag{10}$$

Applying relation (9) to the last point, ub_k, of the kth dimension of the loop iteration space, the lower bound, lb^s_k, and the upper bound, ub^s_k, for the kth dimension of the tiled space is computed as follows:

$$\mathbf{lb^s_k = min\ (\lfloor h_{ki}\, ub_i \rfloor),\ 1 \quad i \quad n,\ i \neq k} \tag{11}$$

$$\mathbf{ub^s_k = \lfloor h_{kk} ub_k \rfloor} \tag{12}$$

In order to generate the final parallel loop, a wave-front approach can be used. A wave-front contains a collection of tiles which have no inter-dependencies and can execute in parallel. Here, the problem is to locate tiles residing on a same wave-front. Considering the fact that our basic dependency vectors are all in positive directions all those points with equal sum of coordinates in the tiled space, J^{sn}, could be assumed to reside on a same wave-front. This sum of the coordinates is addressed as the wave-front number. Since the edges of the first hyper-rectangular tile reside on the coordinates axis, the lowest wave-front number, lwn, is apparently zero. If the tiles are parallelepiped then there will be tiles which are not completely within the boundaries of the iteration space. In this case lwn is considered as -1. To compute the last tile coordinates relation (9) is applied to the last iteration point $UB = (ub_1, ub_2, \ldots, ub_k, \ldots, ub_n)$ in the iteration space. To compute the highest wave-front number, hwn, the sum of the coordinates of the last tile is computed as follows:

$$\mathbf{hwn = (\Sigma \lfloor \Sigma\ h_{ki} ub_i \rfloor) + c} \tag{13}$$

In the above relation the value of c is defined as zero if the tile shape is hyper rectangular or the tile is not completely in the iteration space otherwise c is defined as 1. To find out whether a tile completely resides in the iteration space the following condition is checked:

$$\mathbf{(P\ (\lfloor H\ UB^T \rfloor_{n\times 1} + [1]_{n\times 1}) - [1]_{n\times 1})^T = UB} \tag{14}$$

In the above relation $UB = (ub_1, ub_2, \ldots, ub_k, \ldots, ub_n)$ indicates the last iteration point and transpose of a vector such as UB is represented as UB^T.

To find the coordinates of the lower left corner, llc, and upper right corner, urc, of a tile J^s, within the iteration space, relation (10) is applied to the tile coordinates $(j^s_1, j^s_2, \ldots, j^s_k, \ldots, j^s_n)$. Below are the relations applied to the tile J^s to compute the coordinates of its corners, $llc = (llc_1, llc_2, \ldots, llc_k, \ldots, llc_n)$ and $urc = (urc_1, urc_2, \ldots, urc_k, \ldots, urc_n)$:

$$\mathbf{llc = (PJ^{sT})^T,\ urc = (P(J^{sT} + [1]_{n\times 1}) - [1]_{n\times 1})^T} \tag{15}$$

$$\mathbf{llc_k = max\ (0, \Sigma\ p_{ki}\, j^s_i),\ \ urc_k = min\ (ub_k, \Sigma\ p_{ki}\, (j^s_i + 1)\ -1)} \tag{16}$$

In order to generate the final parallel loop, the range [lwn, hwn] of the wave-fornt numbers, the lower bound $LB^s = (lb^s_1, lb^s_2, \ldots, lb^s_k, \ldots, lb^s_n)$, the upper bound $UB^s = (ub^s_1, ub^s_2, \ldots, ub^s_k, \ldots, ub^s_n)$ of the tiled space and the coordinates llc and urc, are used. The final parallel loop is shown in Fig. 5, below:

```
Sequential Code:
for $j_1 = 0$ to $ub_1$ do
    for $j_2 = 0$ to $ub_2$ do
        …
            for $j_k = 0$ to $ub_k$ do
                …
                    for $j_n = 0$ to $ub_n$ do
                        a $(j_1, j_2, …, j_k, …, j_n)$;
```

```
Parallel Code:
for WaveFrontsNumber = lwn to hwn do
    for $j^s_1 = lb^s_1$ to $ub^s_1$ do in parallel
        for $j^s_2 = lb^s_2$ to $ub^s_2$ do in parallel
            …
            for $j^s_k = lb^s_k$ to $ub^s_k$ do in parallel
                …
                for $j^s_n = lb^s_n$ to $ub^s_n$ do in parallel
                    if $\sum_{k=1}^{n} j^s_k$ = WaveFrontNumber then
                        for $j_1 = llc_1$ to $urc_1$ do
                            for $j_2 = llc_2$ to $urc_2$ do
                                …
                                for $j_k = llc_k$ to $urc_k$ do
                                    …
                                    for $j_n = llc_n$ to $urc_n$ do
                                        if $(\lfloor HJ^T \rfloor)^T = J^s$ Then
                                            a $(j_1, j_2, …, j_k, …, j_n)$;
```

Fig. 5. A sequential loop and its corresponding parallel loop

4 An Example

Consider the following perfectly nested loop and its corresponding dependence equations set:

```
for $j_1 = 0$ to 9 do
    for $j_2 = 0$ to 8 do
        a $(-4j_1 - j_2, -j_1 - 3j_2 + 3)$ = …
        … = … a $(-j_1 + 1, -j_2 + 4)$ …
```

$$-4j_1 - j_2 = -j'_1 + 1$$
$$-j_1 - 3j_2 + 3 = -j'_2 + 4$$
$$\text{or } -4j_1 - j_2 + j'_1 = 1$$
$$-j_1 - 3j_2 + j'_2 = 1$$

Applying the Diophantine approach to solve the above set of loop dependence equations, the loop dependence vectors and basic dependence vectors set, BDVS, can be computed as shown in Fig. 6. Applying the proposed tiling algorithm, the tile space for the above loop will be as shown below in Fig. 6.

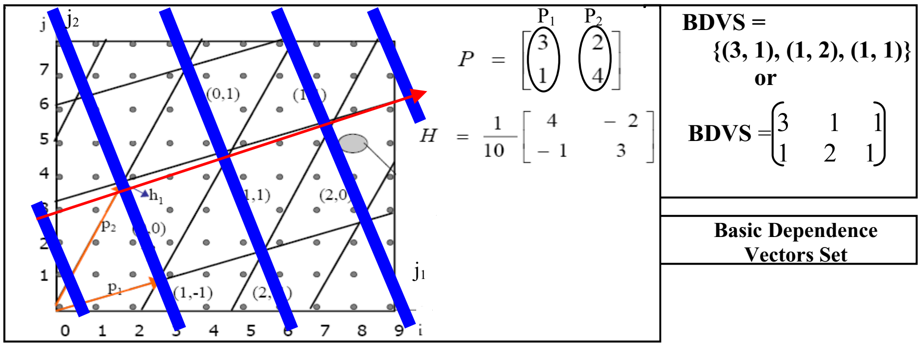

j
j2
7
6
5
4
3
2
1
0 1 2 3 4 5 6 7 8 9
i
j1
(0,1)
(1,1)
(1,1)
(2,0)
(1,0)
(1,-1)
(2,0)
h1
P2
P1
P1 P2
$P = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$
$H = \frac{1}{10}\begin{bmatrix} 4 & -2 \\ -1 & 3 \end{bmatrix}$
BDVS = {(3, 1), (1, 2), (1, 1)}
or
$\text{BDVS} = \begin{pmatrix} 3 & 1 & 1 \\ 1 & 2 & 1 \end{pmatrix}$
Basic Dependence Vectors Set

Fig. 6. An example of a tiled iteration space

Relations (11) and (12) are applied to the last iteration point UB, to compute the coordinates of the lower bounds, LB^s, and upper bounds UB^s of the tiled space respectively. To generate the final parallel loop the tiled space, J^{s2}, is scanned.

$$\mathbf{UB = (9, 8) \Rightarrow LB^s = (-2, -1), UB^s = (3, 2)}$$

$$\mathbf{J^2 = \{J = (j_1, j_2) \mid 0 \le j_1 \le 9, 0 \le j_2 \le 8\} \Rightarrow J^{s2} = \{J^s = (j^s_1, j^s_2) \mid -2 \le j^s_1 \le 3 , -1 \le j^s_2 \le 2\}}$$

```
for WaveFrontNumber = -1 to 3 do
    for $j^s_1$ = -2 to 3 do in parallel
        for $j^s_2$ = -1 to 2 do in parelle
            if $j^s_1 + j^s_2$ = WaveFrontNumber then
                for $j_1 = \max(0, 3j^s_1 + 2j^s_2)$ to $\min(9, 3j^s_1 + 2j^s_2 - 4)$ do
                    for $j_2 = \max(0, j^s_1 + 4j^s_2)$ to $\min(8, j^s_1 + 4j^s_2 - 4)$ do
                        if $(\lfloor HJ^T \rfloor)^T = J^s$ then
                            $a(-4j_1 - j_2, -j_1 - 3j_2 + 3) = \ldots$
                            $\ldots = \ldots a(-j_1 + 1, -j_2 + 4) \ldots$
```

5 Evaluation

Tiling is an NP-hard problem. Increasing the number of a tile dimensions, the time required to find an optimal tile increases exponentially. However, the time complexity of the proposed genetic tiling algorithm, is O (n^3) and the space complexity is O (n^2), because: (1) the main data structures used in the proposed algorithm are n×n matrices, each representing a multi-dimensional tile, and (2) the costly processing of the algorithm are the computation of the determinant and inverse of the tile matrix to find the volume of computation and the volume of communication, respectively. A unique capability of our proposed genetic algorithm is to create tiles with more than two dimensions. In this, tiles of 2 to 6 dimensions are created and it is shown that the resulting tiles are both reliable and stable. As shown in this Fig. 7, increasing a tile dimensionality, the amount of time to find the first feasible solution and the optimal tile increases in polynomial magnitude. The execution times shown in Fig. 7 are all the average for 10 times of running the algorithm for tiles of dimensions 2 to 6.

The volume of computation, volume of communication and fitness of the 10 tiles resulted from 10 times of running the algorithm for the 3, 4 and 5 dimensional tiles, is

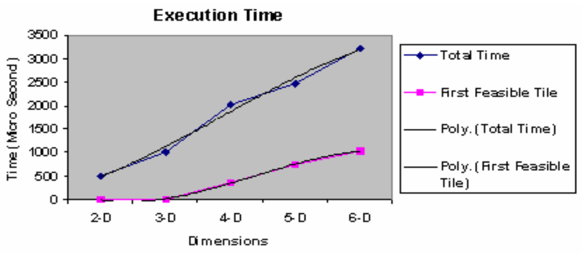
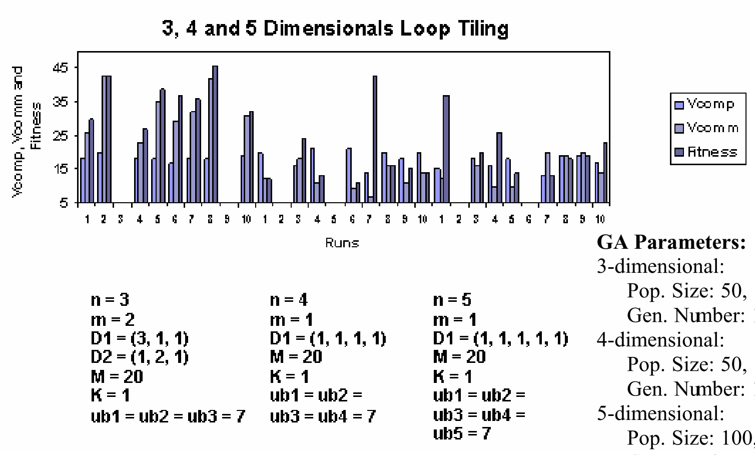


Fig. 7. Execution Time of the proposed algorithm for 2 to 6 dimensional tiles

3-dimensional nested loop:
```
for $j_1 = 0$ to 7
  for $j_2 = 0$ to 7
    for $j_3 = 0$ to 7
      $a(j_1, j_2, j_3) =$
        $a(j_1 - 3, j_2 - 1, j_3 - 1) +$
        $a(j_1 - 1, j_2 - 2, j_3 - 1);$
```

4-dimensional nested loop:
```
for $j_1 = 0$ to 7
  for $j_2 = 0$ to 7
    for $j_3 = 0$ to 7
      for $j_4 = 0$ to 7
        $a(j_1, j_2, j_3, j_4) =$
          $a(j_1 - 1, j_2 - 1, j_3 - 1, j_4 - 1);$
```

5-dimensional nested loop:
```
for $j_1 = 0$ to 7
  for $j_2 = 0$ to 7
    for $j_3 = 0$ to 7
      for $j_4 = 0$ to 7
        for $j_5 = 0$ to 7
          $a(j_1, j_2, j_3, j_4, j_5) =$
            $a(j_1 - 1, j_2 - 1, j_3 - 1, j_4 - 1, j_5 - 1);$
```



n = 3
m = 2
D1 = (3, 1, 1)
D2 = (1, 2, 1)
M = 20
K = 1
ub1 = ub2 = ub3 = 7

n = 4
m = 1
D1 = (1, 1, 1, 1)
M = 20
K = 1
ub1 = ub2 = ub3 = ub4 = 7

n = 5
m = 1
D1 = (1, 1, 1, 1, 1)
M = 20
K = 1
ub1 = ub2 = ub3 = ub4 = ub5 = 7

GA Parameters:
3-dimensional:
Pop. Size: 50,
Gen. Number: 100
4-dimensional:
Pop. Size: 50,
Gen. Number: 100
5-dimensional:
Pop. Size: 100,
Gen. Number: 200

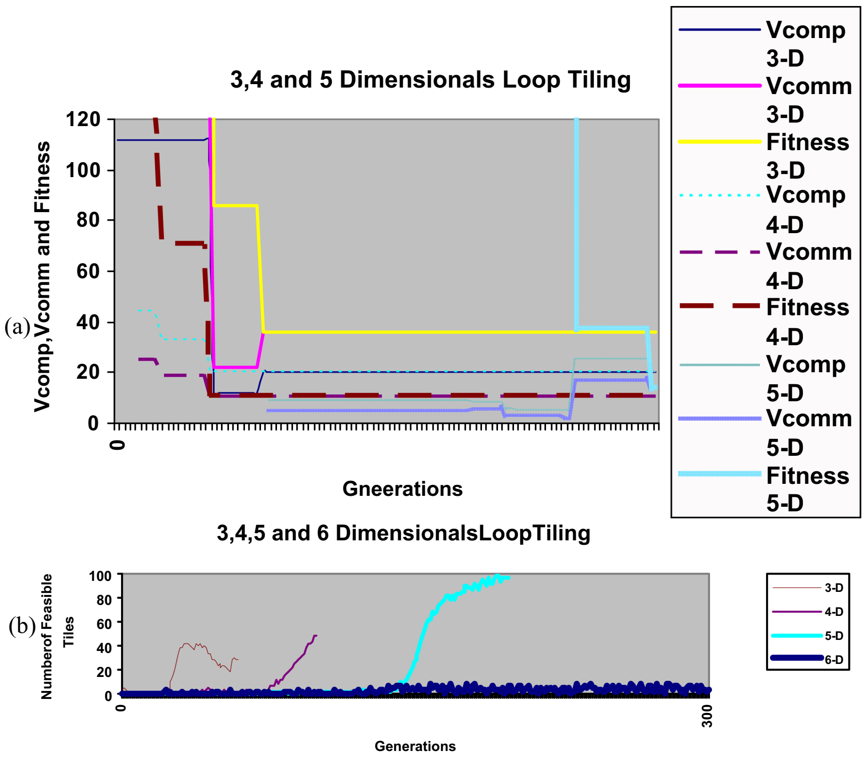
Fig. 8. Stability of the proposed algorithm

shown in Fig. 8. This experiment, demonstrates the stability of the proposed genetic tiling algorithm. M and K in Fig. 8 are already defined in section 2.3.

Fig. 9 demonstrates the reliability of the algorithm for the generation of 3 to 6 dimensional tiles. In Fig. 9.a, the convergence of the proposed genetic tiling algorithm is demonstrated by depicting the quality of the best feasible tile in each generation of the evolutionary process of finding the optimal tile. In Fig. 9.b, to demonstrate the reliability of the proposed algorithm, it is shown that the number of feasible tiles gets closer to the population size as the number of generations increases. As shown in Fig. 9.b, the number of 3-dimensional feasible solutions is very low for the generation numbers 0 to 25. As the number of dimensions of the tiles increases, it takes longer to find the first optimal tile, For instance, as shown in Fig. 9.b, it takes longer for the algorithm to find the first feasible 6-dimensional tile, than the time required to find the first 2 to 5 dimensional tiles.

6 Conclusions and Future Works

In order to construct a uniform iteration space, Diophantine equations can be applied to solve the set of equations describing dependencies loop iterations. The iteration space can be best tiled by applying an evolutionary non-deterministic approach, because tiling is a NP-hard problem. However, to solve the problem the previous



GA Parameters:
3-dimensional: Population Size: 50, Generation Number: 60
4-dimensional: Population Size: 50, Generation Number: 100
5-dimensional: Population Size: 100, Generation Number: 200
6-dimensional: Population Size: 100, Generation Number: 300

Fig. 9. Reliability of the proposed algorithm

works use deterministic approaches. Even the award wining work described in reference [12] solves the tiling problem in only 2 dimensions, using a deterministic approach called *data space tiling*. In this deterministic approach to reduce the size of the search space for finding suitable tiles, the direction of one side of the tiles has to be vertical. Limiting the search space obviously, reduces the chance of finding an optimal solution.

Using a genetic approach as described in our paper, the entire search space is scanned for finding an optimal tile. Due, to the non-deterministic nature of evolutionary approaches such as genetics, the search will be faster than any exhaustive searches in the same space. The genetic approach, proposed in this paper, generates parallelepiped multi-dimensional tiles of any size which best fit into the local memory of parallel processors. The constraints applied to the genetic algorithm ensure the minimum inter-processors communication, maximum usage of the local memory of parallel processors and continuous execution of the iterations residing in each tile. The proposed multi-dimensional wave-front approach can be applied to convert the ultimate iteration space represented as a multi-dimensional parallelepiped tiles into a parallel nested loop.

Iteration spaces are not necessarily rectangular. We are currently working to extend our parallel loop generation algorithm to irregular iteration spaces. Also, as a part of future work we intend to work on parallel loop scheduling algorithms applying multi-dimensional tiled spaces.

References

1. Ahmed N., Mateev N. and Pingali K., "Tiling Imperfectly-nested Loop Nests", *IEEE*, 2000, pp. 1-14.
2. Allen R. and Kennedy K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2001.
3. Bak T., *Evolutionary Algorithms in Theory and Practice*, Oxford University, 1996.
4. Banerjee U., *Loop Transformations for Restructuring Compilers, The Foundations,* Kluwer Academic, 1993.
5. Boulet P., Dongarra D., Robert Y. and Vivien F., "Tiling for Heterogeneous Computing Platforms", 1998, pp. 1-19.
6. Cociorva D., Wilkins J. W., Lam C., Baumgartner G. and Ramanujam J., "Loop Optimization for a class of Memory-Constrained Computations", *15th International Conference on Supercomputing, Sorrento, Italy, ACM*, 2001, pp. 103-113.
7. Darte A., Robert Y. and Vivien F., *Scheduling and Automatic Parallelization*, Birkhäuser, 2000.
8. Gen M. and Cheng R., *Genetic Algorithms & Engineering Design*, John Wiley & Sons, 1997.
9. Goldberg D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
10. Goumas G., Sotiropoulos A. and Koziris N. "Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping", *IEEE*, 2001, pp. 1-10.
11. Griebl M., Faber P. and Lengauer Ch., "Space-time mapping and tiling: a helpful combination", *Concurrency and Computation: Practice and Experience*, John Wiley & Sons, 2004, pp. 221-246.
12. Kandemir M., Bordawekar R., Choudhary A. and Ramanujam J., "A Unified Tiling Approach for Out-of-Core Computations", 1997, pp. 1-12.
13. Leopold C., "Exploiting Non-Uniform Reuse for Cache Optimization: A Case Study", *Symposium on Applied Computing, Las Vegas, ACM*, 2001, pp. 560-564.
14. Miyandashti F. J., *Loop Uniformization in Shared-Memory MIMD Machine*, Master Thesis, Iran University of Science and Technology, 1997 (Persian).
15. Rastello F. and Robert Y., "Automatic Partitioning of Parallel Loops with Parallelepiped-Shaped Tiles", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 5, May 2002, pp. 460-470.
16. Wolfe M. E., "Iteration space tiling for memory hierarchies", In Gary Rodrigue, *3rd Conference on Parallel Processing for Scientific Computing*, December 1989, pp. 357-361.
17. Wolfe M. E., "More iteration space tiling", *Supercomputing'88*, November 1989, pp. 655-664.
18. Zima H. and Chapman B., *Super Compilers for Parallel and Vector Computers*, Addison-Wesley, 1991.

An ASM Semantics of Token Flow in UML 2 Activity Diagrams

Stefan Sarstedt and Walter Guttmann

University of Ulm, 89069 Ulm, Germany
sarstedt@acm.org, walter.guttmann@uni-ulm.de

Abstract. The token flow semantics of UML 2 activity diagrams is formally defined using Abstract State Machines. Interruptible activity regions and multiplicity bounds for pins are considered for the first time in a comprehensive and rigorous way. The formalisation provides insight into problems with the UML specification, and their solutions. It also serves as a basis for an integrated environment supporting the simulation and debugging of activity diagrams.

1 Introduction

The Unified Modeling Language (UML) is widely used for specification and documentation purposes in the software development process. UML activity diagrams model behaviour aspects of software systems, particularly control and data flow. To provide tool support beyond drawing assistance, and to use activity diagrams effectively, it is necessary to exactly understand their meaning.

While the UML specification [1] is a step forward to define activity diagrams more precisely, it is insufficient for several reasons. First, it is vague, leaving much space for interpretation – as will become evident throughout this paper. Second, it is informal, thus a large gap has to be bridged until it can be usefully applied for model execution and automated reasoning. Third, it contains implausible requirements, e.g., for nested interruptible activity regions as discussed in Sect. 6, which reduces its usability.

We propose a solution to these shortcomings by defining the semantics of activity diagrams using Abstract State Machines [2]. The Abstract State Machine (ASM) specification is precise and therefore it enables to understand the meaning of a model to the utmost detail. It is formal and can therefore serve as a foundation for the implementation of tools. Finally, it helps to ensure that the specified behaviour meets the intuition of the modeller.

The state of the art in semantics for UML 2 activity diagrams covers three distinct approaches: mapping to Petri-nets, using graph transformation rules, or providing pseudo-code. A detailed discussion of related work is given in Sect. 8.

We improve on existing work by imposing less restrictions on activity diagrams, e.g., treating multiplicity bounds for pins and interruptible activity regions. The construction using ASMs leads to enhanced clarity and reveals problematic issues in the UML specification. Our solution also shows how to deal with several of these problems without inflicting any biased decision.

The scope of this paper is to describe the ASM semantics of token flow. This specifies the meaning of a transition from one state to another within an activity diagram. It is, however, only one part of a complete ASM semantics of activity diagrams. The remaining parts deal with events and multiple activity and action executions, and are elaborated in [3].

In Sect. 2 we introduce the basic concepts of activity diagrams and Abstract State Machines. The structure of the token flow semantics is presented in Sect. 3, and the following sections describe its aspects. Token offers are computed in Sect. 4 and selected in Sect. 5. Restrictions we have to impose on activity diagrams are also discussed there. Interruptible activity regions and configurable semantics are dealt with in Sect. 6. A brief summary of our techniques to solve problems we have encountered with the UML specification is given in Sect. 7.

2 Basics

In this section we detail the basic concepts of UML activity diagrams, also called "activities" in [1], and Abstract State Machines [2] to a level needed for the following development. In this paper, by writing UML we mean UML 2.0 unless stated otherwise.

2.1 Activity Diagrams

UML facilitates the modelling of control and object (or data) flow by means of activity diagrams, comprising a multitude of concepts. Several levels are defined that support different parts of these concepts. This paper mainly addresses the *intermediate level* that includes object nodes, concurrent flows with guards, and decisions. We additionally discuss *interruptible activity regions* as an example of a useful feature having a vague semantics.

The fundamental elements of activity diagrams are *actions* that are connected by edges to indicate control and data flow. Actions specify transformations on the state of the system that are not further decomposed within the given diagram. They are either implementation-dependent or more specific, e.g., used to send and receive signals or to invoke behaviour specified in other diagrams. Since this distinction is of no concern for the purposes of the paper at hand, the most general term "action" is used.

Edges connecting actions may pass through *control nodes* that coordinate the flows in an activity diagram. A *decision node* chooses between different outgoing edges and the corresponding *merge node* unites several independent flows. On the other hand, a *fork node* splits a flow into concurrent flows along all outgoing edges and the corresponding *join node* synchronises all incoming flows. Moreover, flows may originate in *initial nodes* and terminate in *final nodes*.

Object nodes allow for object flows in addition to control flows. They arise as *input pins* and *output pins* attached to actions, indicating the delivery of data. On the level of activities, objects can be passed through *activity parameter nodes*. Objects may also be buffered in *central buffer nodes*.

An interruptible activity region is a subset of nodes and edges supporting the termination of parts of an activity diagram. It is further examined in Sect. 6.

Example. Several of these building-blocks are illustrated in the activity diagram shown in Fig. 1 that acts as the running example throughout this paper. It contains actions A and B, activity parameter nodes C and D, central buffer node E, input pin F, control flows e_1–e_3, object flows e_4–e_8, the diamond-shaped decision and merge nodes, the bar-shaped join node, and the bullet-shaped initial node. The decision node's outgoing edges are decorated with guards that indicate the conditions for passing the edges.

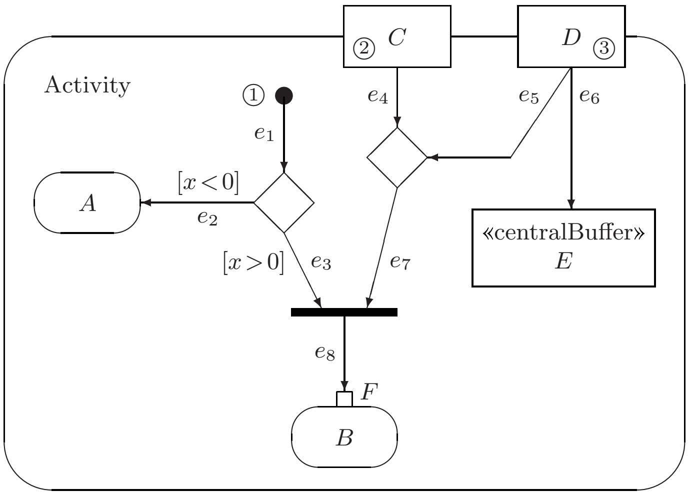


Fig. 1. Activity diagram used as a running example

While Petri-nets are not adequate to describe the semantics of activity diagrams (see Sect. 8), our example can be explained at least in terms of tokens. Upon start of the activity diagram, tokens are available on the nodes C and D, and on the edge e_1. There are three different situations, depending on the value of the attribute x:

– $x < 0$: Token ① may enable action A and token ③ may move to buffer E. The join node must not be traversed.
– $x = 0$: Only token ③ may move to E.
– $x > 0$: Token ② may move to input pin F, enabling action B. Independently, token ③ may move either to F or to E, but not to both nodes.

Note that in any of the three situations, any of the indicated flows may take place, but is not required to.

2.2 Abstract State Machines

Basic ASMs may be viewed as "pseudo-code over abstract data" [2], and indeed we use them as a convenient way to describe computations in this paper. We present a brief overview of the most important concepts and refer to [2] for details, including an operational semantics.

An ASM comprises transition rules that operate on a state composed of functions defined over a base set. The *update rule* $f(s_1, \ldots, s_n) := t$ modifies the value of f at $(s_1, \ldots, s_n)$ to t. In general, several transition rules execute in parallel, requiring that individual updates do not conflict each other. Such conflicts may be avoided by enforcing sequential execution with **seq**. Further constructs include the no-operation **skip**, abstractions using **let** . . . **in**, the conditional, and rule calls with call-by-name semantics. The rule **forall** x **with** φ **do** R executes R in parallel for each x satisfying φ. The rule **choose** x **with** φ **do** R chooses some x satisfying φ and then executes R. The rule **iterate** R executes R until it provides no further or inconsistent updates. Borrowed from AsmL, the **add** . . . **to** and **remove** . . . **from** rules denote non-conflicting, partial updates to sets [4].

3 Flow Computation

In this section we present our main ASM rule for the computation of transitions, describing the structure of the token flow semantics. Relevant terms used by the UML specification are introduced as needed.

The semantics of activity diagrams is specified in terms of tokens [1]. Our transition rule is called whenever tokens are available at actions, initial nodes, or object nodes. According to the UML specification, these nodes *offer* the tokens on their outgoing edges. Tokens can be rejected by edges because their guards evaluate to false, or by nodes not accepting them.

Offered tokens may move, if they are accepted by all intermediate edges and control nodes, as well as their destination nodes. The latter comprise actions, final nodes, and object nodes. The *traverse-to-completion* principle [5] requires that the whole path from the original node to the destination is traversed at once. The definitive goal of our rule is to determine which tokens move, triggering which destinations, and to perform the entailed transition.

The exact working of the propagation of token offers and their selection at destination actions and object nodes, however, is neither formally defined, nor adequately discussed in the specification. Our proposal for transition computation and execution consists of the following main ASM rule that is executed repeatedly as long as control or data tokens are available:

```
    InitialiseFlowsForControlFlowSources
    InitialiseFlowsForObjectFlowSources (see Sect. 4.1)
seq PropagateFlowInformation (see Sect. 4.2)
seq SelectTokenOffers (see Sect. 5)
seq RemoveFlowsInInterruptedRegions (see Sect. 6)
seq ActivateAcceptEventActions
seq ExecuteTransition
```

The InitialiseFlows and PropagateFlowInformation macros spread token offers from the source nodes, where the actual control and data tokens rest, through the activity graph. These macros are described in the following section. After all possible offers have been computed, subsets are selected at destination actions, object and final nodes, preparing the traversal of the associated tokens. The selection mechanism is described in Sect. 5. Note that selecting token offers can invalidate other, conflicting token offers.

Since aborting interruptible activity regions can prevent tokens from traversal, the rule RemoveFlowsInInterruptedRegions removes those selections. Problematic cases unconsidered by the UML specification, including the handling of nested interruptible activity regions, are discussed in Sect. 6.

Accept event action nodes without incoming edges, contained in interruptible activity regions, are initialised by ActivateAcceptEventActions. The actual execution of the token traversal and the termination of actions in interrupted regions is performed by ExecuteTransition. For want of space, both rules are not detailed in this paper but presented in [3] that also deals with event handling and multiple activity and action executions.

4 Computation of Token Offers

The distribution of token offers is performed in two steps. First, new token offers are created for tokens resting at outgoing edges of actions or initial nodes (being sources of control flows), or at object nodes (being sources of object flows). Second, the token offers are propagated through the activity graph towards the consuming destination nodes, namely actions, object nodes, and final nodes.

4.1 Creation of Token Offers

Offers are created by the InitialiseFlows macros. We show the rule for object flow sources, and omit the similar one for control flow sources. The latter joins control tokens offered by the same edge by creating only one token offer.

We use static ASM functions to model the activity diagram being worked on, according to the UML meta model [1]. The domain *ObjectFlowSource* subsumes output pins, central buffer nodes, and incoming activity parameter nodes.

```
InitialiseFlowsForObjectFlowSources ≡
  forall $n$ with $n \in ObjectFlowSource \wedge |dataTokens(n)| > 0$ do
    let $m = |outgoing(n)|$ in
      forall $i$ with $1 \leq i \leq m$ do $t(i) := new(TokenOffer)$
      seq
      forall $i$ with $1 \leq i \leq m$ do
        let $e = outgoing(n, i)$ in
          if $IsGuardTrue(e, Self.context, head(dataTokens(n)))$ then
            $t(i).offeredToken := head(dataTokens(n))$
            $t(i).paths := \{[e]\}$
            $t(i).exclude := \{t(j) \mid 1 \leq j \leq m \wedge i \neq j\}$
            $t(i).include := \emptyset$
            $offers(e) := \{t(i)\}$
```

The function *dataTokens* yields the data tokens currently available at a node. If there is more than one, only the first is considered, assuming a FIFO-ordering [1, p. 380]. For each outgoing edge, if its guard evaluates to true for that token, a new instance of the *TokenOffer* structure is initialised. Since the syntax and the implementation of guards are left open by the UML specification, the evaluation is performed by the monitored ASM function *IsGuardTrue*.

The *TokenOffer* structure consists of the following components:

$$\textbf{domain}\ \mathit{TokenOffer} =_{def} \{\mathit{offeredToken} : \mathit{Token};\ \mathit{paths} : \mathcal{P}(\mathit{ActivityEdge}^*);\ \mathit{exclude} : \mathcal{P}(\mathit{TokenOffer});\ \mathit{include} : \mathcal{P}(\mathit{TokenOffer})\}$$

The component *offeredToken* contains the actual data token, whose possible traversal is represented by this offer. The component *paths* represents the path beginning from the source node of the token to the current position of the offer. Actually, a *set* of paths is needed, since control flows must be included when combined with object flows by a join node, as described in Sect. 4.2.

Furthermore, according to the specification [1, p. 381], "a token in an object node can traverse only one of the outgoing edges". Our algorithm must therefore ensure that the offers on these edges exclude each other, as is the case in Sect. 4.2 for decision nodes with non-exclusive guards. Tool implementation requires efficiency, and we therefore avoid to try out all possible combinations for several nodes with competing edges. An appropriate backtracking mechanism is also ruled out, although for other reasons, by [6]. Note that lifting this problem to the interpreting level, e.g., to the ASM choice construct, does not solve it.

Therefore, the component *exclude* of *TokenOffer* collects all conflicting offers. It is initialised to contain the offers on all edges except the current, since all outgoing edges of object nodes compete with each other. By the way, this is not the case for control flow sources, where they are initialised as empty.

The component *include*, finally, contains those offers that have contributed to the current offer. Being initial offers, their *include* set is empty. Both the *exclude* and *include* sets are used for selecting token offers at destination nodes in Sect. 5.

The ASM function $\mathit{offers} : \mathit{ActivityEdge} \to \mathcal{P}(\mathit{TokenOffer})$ stores the new token offers. While it initially maps to singleton sets, multiple offers may exist on a single edge at later stages, as stated explicitly in [1, p. 369]. All offers stored on all incoming edges of a node n are returned by $\mathit{offersForNode}(n)$.

4.2 Propagation of Token Offers

After all initial offers have been created, PropagateFlowInformation distributes them by iteratively calling rules for the join, decision, merge, and fork nodes.

Join. We first deal with join nodes, being the most complex kind. The following ASM rule processes all join nodes of the current activity once, ensured by the predicate *visited*. All previously computed offers on the outgoing edge are cleared and, if there are offers on all incoming edges, we differentiate the two cases specified by [1, p. 369].

```
PropagateFlowForJoinNode ≡
  forall $n$ with $n \in JoinNode \wedge AreAllPredecessorsVisited(n) \wedge \neg visited(n)$ do
    let $o = outgoing(n)$ in
      $offers(o) := \emptyset$
      seq
      $visited(n) := true$
      if $\forall e \in incoming(n) : offers(e) \neq \emptyset$ then
        if $IsControlFlow(o)$ then PropagateControlFlowForJoinNode$(n, o)$
                                else PropagateObjectFlowForJoinNode$(n, o)$
```

If some (or all) incoming edges are object flows, all offers from these flows have to be forwarded. By joining all incoming control flows to the *paths* set of each transmitted token offer, they can be removed if the actual transition of the data token takes place. The *include* and *exclude* sets of each offer contain all control and object flows to prevent conflicting offers to flow that might invalidate the join condition.

```
PropagateObjectFlowForJoinNode$(n, o)$ ≡
  forall $e$ with $e \in incoming(n) \wedge IsObjectFlow(e)$ do
    forall $t$ with $t \in offers(e)$ do
      if $IsGuardTrue(o, Self.context, t.offeredToken)$ then
        let $t' = new(TokenOffer)$ in
          $t'.offeredToken := t.offeredToken$
          $t'.paths := \{p +\!\!\!+ o \mid p \in \bigcup\{s.paths \mid s \in controlFlowOffers(n)\} \cup t.paths\}$
          $t'.exclude := \bigcup\{s.exclude \mid s \in offersForNode(n)\}$
          $t'.include := \bigcup\{s.include \cup \{s\} \mid s \in offersForNode(n)\}$
          add $t'$ to $offers(o)$
```

For this procedure to work we have to impose a restriction: We assume that the incoming token offers are *consistent*, as discussed in Sect. 5. Otherwise, sets of token offers would have to be considered to handle the additional dependences.

A similar rule emits only one token offer, if only control flows are joined.

Decision. The specification of decision nodes requires that each incoming token is offered to those outgoing edges whose guards are satisfied. The modeller must ensure that only one outgoing edge is actually traversed. Additionally, [1, p. 349] states that "if multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics is not defined".

It is, however, left unspecified what the "approval" proviso means. We propose that any selection of token offers may be chosen as long as no two outgoing edges are traversed simultaneously by the same token. The following fragment of our algorithm shows the use of the *exclude* sets to implement this:

```
forall $i$ with $1 \leq i \leq |acceptingEdges|$ do $t(i) := new(TokenOffer)$
seq
forall $i$ with $1 \leq i \leq |acceptingEdges|$ do
  $t(i).offeredToken := t.offeredToken$
  $t(i).paths := \{p +\!\!\!+ elementAt(acceptingEdges, i) \mid p \in t.paths\}$
  $t(i).exclude := t.exclude \cup \{t(j) \mid 1 \leq j \leq |acceptingEdges| \wedge i \neq j\}$
  $t(i).include := t.include \cup \{t\}$
  add $t(i)$ to $offers(elementAt(acceptingEdges, i))$
```

The set *acceptingEdges* contains all edges with true guards. Extending the guard mechanism of edges, "a predefined guard 'else' may be defined for at most one outgoing edge" of a decision node [1, p. 349]. As expected, this guard succeeds only if all other guards fail. The else-guard is easily incorporated into the transition rule as a special case.

Merge. The propagation for the *merge nodes* is considerably simpler, since "all tokens offered on incoming edges are offered to the outgoing edge" [1, p. 374]. We immediately check if the token satisfies the guard of the outgoing edge, and calculate the new token offer as follows:

```
**let** $t' = new(TokenOffer)$ **in**
    $t'.offeredToken := t.offeredToken$
    $t'.paths := \{p +\!\!\!+ outgoing(n, 1) \mid p \in t.paths\}$
    $t'.exclude := t.exclude$
    $t'.include := t.include \cup \{t\}$
    **add** $t'$ **to** $offers(outgoing(n, 1))$
```

Fork. The calculation for the *fork nodes* is almost identical, except that token offers are made at each outgoing edge with a true guard. Our algorithm can be extended to deal with the buffering of tokens at fork nodes [1, p. 363]. Note that, when used in combination with guards, fork buffering leads to unexpected behaviour. The extension is presented in [3], along with a discussion of the problems caused by the UML specification.

4.3 Example Computation

Figure 2 contains the computed token offers for our example shown in Fig. 1, assuming $x > 0$. The offers 1, 3, 4 and 5 are computed by the InitialiseFlows rules, whereas the remaining offers are added by the Propagate rules. The next section explains how the propagated offers are selected at destination nodes.

Edge	Offers, $id : (offeredToken, paths, exclude, include)$
e_1	$1 : (-, \{[e_1]\}, \emptyset, \emptyset)$
e_2	no offers
e_3	$2 : (-, \{[e_1, e_3]\}, \emptyset, \{1\})$
e_4	$3 : (②, \{[e_4]\}, \emptyset, \emptyset)$
e_5	$4 : (③, \{[e_5]\}, \{5\}, \emptyset)$
e_6	$5 : (③, \{[e_6]\}, \{4\}, \emptyset)$
e_7	$6 : (②, \{[e_4, e_7]\}, \emptyset, \{3\})$ and
	$7 : (③, \{[e_5, e_7]\}, \{5\}, \{4\})$
e_8	$8 : (②, \{[e_4, e_7, e_8], [e_1, e_3, e_8]\}, \{5\}, \{1, 2, 3, 4, 6, 7\})$ and
	$9 : (③, \{[e_5, e_7, e_8], [e_1, e_3, e_8]\}, \{5\}, \{1, 2, 3, 4, 6, 7\})$

Fig. 2. Computed offers for the example in Fig. 1

5 Selection of Token Offers

Once the token offers are computed, we select subsets of them to participate in the planned transition. The transition may lead, e.g., to the start of a new action as described by the specification [1, p. 302]. Other possibilities are moving tokens to central buffer nodes, outgoing activity parameter nodes, and final nodes. The UML specification, however, does not indicate what to perform if there are enough token offers to conduct several of these operations.

We use the ASM iteration and non-deterministic choice constructs to adhere to the specification. The iteration may be stopped by choosing $n = \mathsf{skipSelection}$ at any stage. Otherwise, we select token offers depending on the kind of node.

```
SelectTokenOffers ≡
  iterate
    choose $n$ with $n \in \{\mathsf{skipSelection}\} \cup \mathit{Action} \cup \mathit{FinalNode}$
        $\cup\ \mathit{CentralBufferNode} \cup \mathit{OutgoingActivityParameterNode}$ do
      if $n \in \mathit{Action}$ then SelectTokenOffersForAction($n$)
      if $n \in \mathit{FinalNode}$ then SelectTokenOffersForFinalNode($n$)
      if $n \in \mathit{CentralBufferNode} \cup \mathit{OutgoingActivityParameterNode}$ then
        SelectTokenOffersForCentralBufferAndParameterNode($n$)
```

In the following, we focus on action nodes. The selection for the other kinds of nodes is dealt with by similar, even simpler rules.

For action nodes, we select a subset of token offers S_i for each input pin i of the action. Conditions for the acceptance of tokens by input pins are that the number of selected tokens is between *lower* and *upper* [1, p. 249], and that the total number of tokens resting on each pin does not exceed its upper bound [1, p. 380]. If an appropriate selection has been found, we commit to it and update the remaining offers according to the specification.

The selected subsets are accumulated in *tokenSelections*. The function *taken* keeps track of the selections for each object node to make sure they do not overflow. The input pins of action n are provided by $\mathit{input}(n)$ according to the UML meta model.

```
SelectTokenOffersForAction($n$) ≡
  if $\forall e \in \mathit{incoming}(n) : \mathit{offers}(e) \neq \emptyset$ then
    let $p = |\mathit{input}(n)|$ in
      choose $S_1, \ldots, S_p$ with $\forall 1 \leq i \leq p : \exists j = \mathit{input}(n,i) : S_i \subseteq \mathit{offersForNode}(j)$
          $\wedge\ \mathit{lower}(j) \leq |S_i| \leq \mathit{upper}(j)$
          $\wedge\ |S_i| + \mathit{taken}(j) + |\mathit{dataTokens}(j)| \leq \mathit{upperBound}(j)$ do
        let $\mathit{selection} = \bigcup\{S_i \mid 1 \leq i \leq p\} \cup \mathit{offersForNode}(n)$ in
          UpdateOffers($\mathit{selection}$)
          $\mathit{tokenSelections} := \mathit{tokenSelections} \cup \{(n, \mathit{selection})\}$
          forall $i$ with $1 \leq i \leq p$ do $\mathit{taken}(\mathit{input}(n,i)) := \mathit{taken}(\mathit{input}(n,i)) + |S_i|$
```

The UpdateOffers rule removes the selection of token offers and all offers inconsistent to it. If any inconsistent offers are removed from a *join* node, we re-propagate this information using the rules introduced in Sect. 4, since the removal may affect other token offers originating at that node.

```
$\textsc{UpdateOffers}(\mathit{selection}) \equiv$
  forall $n$ with $n \in \mathit{JoinNode}$
      $\wedge \exists e \in \mathit{incoming}(n) : \exists t \in \mathit{offers}(e) : \exists t' \in \mathit{selection} : \neg \mathit{AreConsistent}(t, t')$ do
    forall $n'$ with $n' \in \mathit{AllControlNodeSuccessors}(n) \cup \{n\}$ do $\mathit{visited}(n') := \mathit{false}$
  forall $e$ with $e \in \mathit{ActivityEdge}$ do
    $\mathit{offers}(e) := \{t \in \mathit{offers}(e) \mid t \notin \mathit{selection} \wedge \forall t' \in \mathit{selection} : \mathit{AreConsistent}(t, t')\}$
  seq
  $\textsc{PropagateFlowInformation}$
```

The symmetric predicate *AreConsistent* is used to calculate which token offers must be removed according to the specification.

$$\begin{aligned} &\mathit{AreConsistent} : \mathit{TokenOffer} \times \mathit{TokenOffer} \rightarrow \mathit{Boolean} \\ &\mathit{AreConsistent}(t_1, t_2) =_{\mathit{def}} (t_1.\mathit{exclude} \cap (t_2.\mathit{include} \cup \{t_2\}) = \emptyset) \\ &\qquad\qquad\qquad\qquad\quad \wedge (t_2.\mathit{exclude} \cap (t_1.\mathit{include} \cup \{t_1\}) = \emptyset) \end{aligned}$$

To avoid an inefficient search at each destination node, we assume that the incoming token offers are consistent. The modeller can ensure this, and also the consistency required for join nodes in Sect. 4, e.g., by placing appropriate guards on competing edges leading to the same destination nodes. A stronger, syntactic condition is the absence of two paths from the same decision or object node to the same action, final, join, or object node. Here, an action together with its input pins is considered as one node.

Example Selection. Let us discuss the case $x > 0$ of our running example shown in Fig. 1, assuming further that the action node B is chosen by SELECTTOKENOFFERS. Then, either of the offers 8 and 9 shown in Fig. 2 may be selected, or both of them, by SELECTTOKENOFFERSFORACTION. In any case, the *exclude* set of the selection contains the offer 5 that is therefore removed by UPDATEOFFERS. Since all offers into the join node are consistent with the selection, no further propagation is performed.

If, on the other hand, central buffer E was chosen by SELECTTOKENOFFERS, and offer 5 selected, the offers 4, 7, 8, and 9 would be removed. By re-propagating from the join node, offer $8 : (\text{②}, \{[e_4, e_7, e_8], [e_1, e_3, e_8]\}, \emptyset, \{1, 2, 3, 6\})$ is reconstructed and may be chosen in the next iteration.

6 Interruptible Activity Regions

After token offers have been selected for destination nodes, we determine which interruptible activity regions are aborted and eliminate flows that are in conflict with those regions.

If one of the selected offers passes an interrupting edge [1, p. 366], all tokens in the interrupted region must be removed, and therefore all offers of these tokens have to be removed as well. Note that it is permitted to have concurrent, non-interrupting flows out of aborted regions.

The specification, however, gives no information regarding concurrent flows leading *into* aborted regions. Figure 3 shows the offer ① that originates from a node outside of the region, and the offer ② that re-enters the region after having left it. Since either keeping or destroying such tokens can be useful, our algorithm can be adapted to both alternatives. To this end, we introduce the configuration of semantics by using UML tags, a standard extension mechanism of the UML. This mechanism has already been applied successfully to the configuration of signal handling [7]. In Fig. 3, the ignoreFlowIntoInterruptedRegion tag is used that is queried in the following ASM rule.

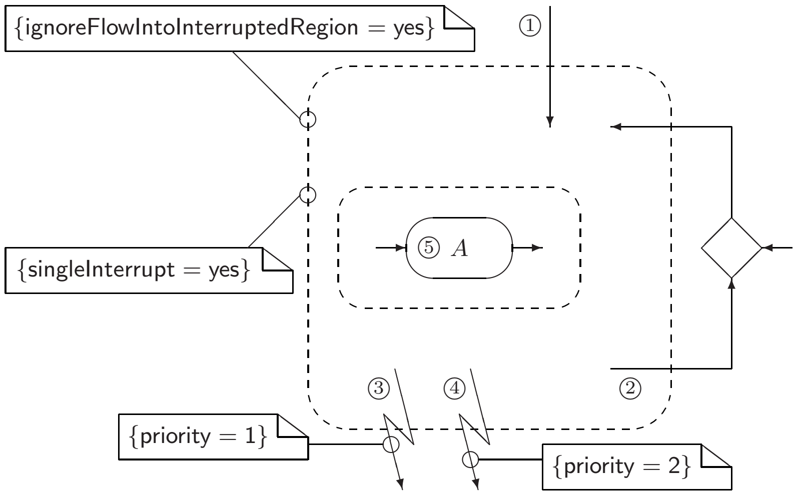


Fig. 3. Issues with interruptible activity regions

In general, multiple interrupting edges can be passed at the same time (see offers ③ and ④ in Fig. 3), leading to another scenario where configuration is useful. To enable the user to specify that only a single edge may be passed, the singleInterrupt and priority tags are defined. If singleInterrupt is enabled for a region, our algorithm selects the edge with the highest annotated priority, implemented by ChooseIntEdge. Offer selections for the other interrupting edges are then discarded.

Interruptible activity regions, being activity groups, are also allowed to be nested. A major deficiency of the UML specification is missing information about how to deal with them. According to the specification [1, p. 323], "no node or edge in a group may be contained by its subgroups or its containing groups, transitively". This means that, when a region is aborted, its nested regions are not. Instead of this unexpected behaviour we propose to interrupt all nested regions. Token ⑤ would thus be removed in Fig. 3 if offer ③ was selected for traversal.

The following ASM rule implements the discussion just carried out. For all regions that are to be aborted, we remove all selected offers that do not leave the region, as checked by *HasInnerFlow*. We furthermore eliminate flows according to the specified configuration tags. Finally, all nested regions are marked for termination.

```
$\textsc{RemoveFlowsInInterruptedRegions} \equiv$
  forall $r$ with $r \in \mathit{InterruptibleActivityRegion} \wedge \mathit{IsInterrupted}(r)$
      $\wedge \nexists r' \in \mathit{parents}(r) : \mathit{IsInterrupted}(r')$ do
    remove $\{s \in \mathit{tokenSelections} \mid \mathit{HasInnerFlow}(s,r)\}$ from $\mathit{tokenSelections}$
    if $\mathit{tagValue}(r, \mathsf{singleInterrupt}) = \mathsf{yes}$ then
      let $e = \textsc{ChooseIntEdge}(\bigcup\{\mathit{interruptingEdge}(s,r) \mid s \in \mathit{tokenSelections}\})$ in
        remove $\{s \in \mathit{tokenSelections} \mid \mathit{Interrupts}(s,r) \wedge e \notin \mathit{interruptingEdge}(s,r)\}$
          from $\mathit{tokenSelections}$
    if $\mathit{tagValue}(r, \mathsf{ignoreFlowIntoInterruptedRegion}) = \mathsf{yes}$ then
      remove $\{s \in \mathit{tokenSelections} \mid \mathit{HasFlowInto}(s,r)\}$ from $\mathit{tokenSelections}$
    add $\{r\} \cup \mathit{children}(r)$ to $\mathit{regionsToInterrupt}$
```

7 Solving Problems of UML

Let us finally compare the problems with the UML specification we have encountered and the ways we solve them. Deliberate under-specifications are modelled by the non-deterministic choice of ASMs (e.g., see Sect. 5). To deal with open issues that can be decided by the modeller we introduce UML tags that are queried from the ASM rules. If the specification should have decided, e.g., concerning non-exclusive guards on competing edges, we propose a decision. Unintuitive consequences of requirements in the specification (e.g., of fork buffering) are discussed in detail, also by providing alternative implementations [3]. Obvious errors, e.g., for nested interruptible activity regions, are corrected.

8 Related Work

Existing work covers the semantics for UML 1.$*$, including an ASM semantics for activity diagrams, excerpts of which are presented in [8]. For historical reasons, however, UML 1.$*$ activity diagrams are special kinds of state charts. In UML 2.0 they have been completely redefined. We therefore discuss only UML 2.0 related work in the following.

Since the UML specification envisions a "Petri-like semantics" for activity diagrams [1, p. 314], it is manifest to propose a mapping between the two. Störrle [9,10] uses different variants of Petri-nets, e.g., coloured Petri-nets for data flow, and procedural Petri-nets for activities. The treatment of join nodes having mixed object and control flows is, however, neither discussed nor obvious. The development culminates in [11] concluding that Petri-nets might, after all, not be appropriate for formalising activity diagrams. Especially mapping advanced concepts, such as interruptible activity regions, is found not to be intuitive. Moreover, the lack of a unified Petri-net formalism, integrating the different variants used to map different concepts, is observed. Assuring the traverse-to-completion semantics is identified as another problem. The related paper [12] also translates to Petri-nets, but focuses on the parameters of actions.

Vitolins and Kalnins [13] present an algorithm for computing the token flow, proposing a forward and backward search by using so-called "push" and "pull" engines. Several far reaching restrictions are, however, imposed on activity diagrams. Decision nodes must have mutually exclusive guards, and object nodes must have no outgoing concurrent edges. This simplifies their algorithm, since they do not have to pull all input tokens in one atomic step – traverse-to-completion is thus not observed. Fork and join nodes must not be on the same path between two actions. Tokens resulting from join nodes are grouped, which is neither prohibited nor stated in the specification.

Hausmann [6] formalises activity diagrams using "Dynamic Meta Modeling", where graph transformation rules operate on an instance of the UML meta model. The transformation engine is responsible to resolve the non-determinism occurring at competing edges of object and decision nodes. This renders the approach too inefficient to serve as a basis for tool support. The semantics of a large part of activity diagrams is described very detailed and problems of the UML specification are discussed. Apart from this, several restrictions apply also to this work. Only one offer is allowed per edge, and – as a consequence – when different data tokens are offered to a join node, only one of them is forwarded. Guards and interruptible activity regions are not supported.

The ongoing UML Semantics Project [14] aims at formalising a subset of UML by providing "a strong foundation for the definition of a UML virtual machine that is capable of executing UML 2.0 models". The Modelware Project [15] implements a tool capable of simulating basic activity diagrams, but only with control flows. Currently, no formalisation of the algorithms behind their execution engine is available.

Our paper shows how to deal with the restrictions mentioned before. Moreover, none of the works discussed so far, and none that we know of, handles the problems presented in Sect. 6 related to interruptible activity regions, including incoming flows, multiple interrupting edges, and nested regions. The useful feature of *lower* and *upper* multiplicity bounds on pins is also not treated elsewhere.

9 Conclusion

We formalise the semantics of token flow in UML 2 activity diagrams in terms of ASM rules. The resulting rules can be traced back to requirements present in or absent from the UML specification. Our contribution deals with several features neglected elsewhere, such as interruptible activity regions and multiplicity bounds for pins. The part presented in this paper is embedded into rules for asynchronous multi-agent ASMs specifying signal handling and activity and action executions [3].

The formalisation is high-level enough to reveal problematic issues with the UML specification. On the other hand, it can be directly executed using the AsmL compiler. Furthermore, it is suitable to serve as a basis for tool support, e.g., for model checking [16] and verification [17]. An integrated environment has been implemented [18], supporting the simulation and debugging of activity diagrams.

References

1. Object Management Group: UML 2.0 Superstructure Specification. (2005)
2. Börger, E., Stärk, R.: Abstract State Machines. Springer-Verlag (2003)
3. Sarstedt, S.: Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams. PhD thesis, Universität Ulm (2006)
4. Gurevich, Y., Tillmann, N.: Partial updates: Exploration. Journal of Universal Computer Science **7**(11) (2001) 917–951
5. Bock, C.: UML 2 activity and action models part 4: Object nodes. Journal of Object Technology **3**(1) (2004) 27–41
6. Hausmann, J.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, Universität Paderborn (2005)
7. Sarstedt, S.: Overcoming the limitations of signal handling when simulating UML 2 activity charts. In Feliz-Teixeira, J., Carvalho Brito, A., eds.: Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05). (2005) 61–65
8. Börger, E., Cavarra, A., Riccobene, E.: An ASM semantics for UML activity diagrams. In Rus, T., ed.: Algebraic Methodology and Software Technology. Volume 1816 of Lecture Notes in Computer Science, Springer-Verlag (2000) 293–308
9. Störrle, H.: Semantics of control-flow in UML 2.0 activities. In: Symposium On Visual Language And Human Centric Computing. IEEE (2004) 235–242
10. Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. In Minas, M., ed.: Workshop on Visual Languages and Formal Methods. Volume 127, Issue 4 of Electronic Notes in Theoretical Computer Science, Elsevier (2005) 35–52
11. Störrle, H., Hausmann, J.: Towards a formal semantics of UML 2.0 activities. In Liggesmeyer, P., Pohl, K., Goedicke, M., eds.: Software Engineering 2005. Volume P-64 of Lecture Notes in Informatics, Gesellschaft für Informatik (2005) 117–128
12. Barros, J., Gomes, L.: Actions as activities and activities as Petri nets. In Jürjens, J., Rumpe, B., France, R., Fernandez, E., eds.: Critical Systems Development with UML: Proceedings of the UML'03 workshop. TUM-I0317 (2003) 129–135
13. Vitolins, V., Kalnins, A.: Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In: Ninth International EDOC Enterprise Computing Conference. IEEE (2005) 181–192
14. UML 2.0 Semantics Project: Web page (2006) http://www.cs.queensu.ca/~stl/internal/uml2/
15. Modelware Project, WP1 Modelling Techniques: D1.3 Model Simulation Scheme: Definition (2005) available from http://www.modelware-ist.org/
16. Winter, K.: Model Checking Abstract State Machines. PhD thesis, Technische Universität Berlin (2001)
17. Gargantini, A., Riccobene, E.: Encoding abstract state machines in PVS. In Gurevich, Y., Kutter, P., Odersky, M., Thiele, L., eds.: Abstract State Machines: Theory and Applications. Volume 1912 of Lecture Notes in Computer Science, Springer-Verlag (2000) 303–322
18. Sarstedt, S., Gessenharter, D., Kohlmeyer, J., Raschke, A., Schneiderhan, M.: ActiveChartsIDE: An integrated software development environment comprising a component for simulating UML 2 activity charts. In Feliz-Teixeira, J., Carvalho Brito, A., eds.: Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05). (2005) 66–73

Well-Structured Model Checking of Multiagent Systems[⋆]

N.V. Shilov and N.O. Garanina

Institute of Informatics Systems, Russian Academy of Science
6, Lavrentiev ave., 630090, Novosibirsk, Russia
{shilov, garanina}@iis.nsk.su

Abstract. We address model checking problem for combination of Computation Tree Logic (CTL) and Propositional Logic of Knowledge (PLK) in finite systems with the perfect recall synchronous semantics. We have published already an (update+abstraction)-algorithm for model checking with detailed time upper bound. This algorithm reduces model checking of combined logic to model checking of CTL in a finite abstract space (that consists of some finite trees). Unfortunately, the known upper bound for size of the abstract space (i.e. number of trees) is a non-elementary function of the size of the background system. Thus a straightforward use of a model checker for CTL for model checking the combined logic seems to be infeasible. Hence it makes sense to try to apply techniques, which have been developed for infinite-state model checking. In the present paper we demonstrate that the abstract space provided with some partial order on trees is a well-structured labeled transition system where every property expressible in the propositional μ-Calculus, can be characterized by a finite computable set of maximal elements. We tried feasibility of this approach to model checking of the combined logic in perfect recall synchronous environment by automatic model checking a parameterized example.

1 Introduction

Combinations of traditional program logics [20,8,25] with logics of knowledge [9,24] become a current research topic due to the importance of study of interactions between knowledge and actions for reasoning about multiagent systems. A number of techniques for (semi)automatic processing of a number of combined logics have been under study [15,6,7,23,14,16,17,13].

Paper [13] has studied the model checking problem in trace-based synchronous perfect recall systems for pairwise fusion of the program logics Computation Tree Logic extended by actions (*Act*-CTL) and the propositional μ-Calculus (μC) with the epistemic logics Propositional Logic of Knowledge for n agents (PLK$_n$) and Propositional Logic of Common Knowledge for n agents (PLC$_n$).

⋆ Supported by joint grant RFBR 05-01-04003-a (DFG project COMO, GZ: 436 RUS 113/829/0-1) and by grant RFBR 06-01-00464-a.

'Trace-based' means that semantics of formulas is defined on traces, i.e. finite sequences of states and actions[1]. Each element of a trace represents a state of the system at some moment of time. So, 'synchronous' means that agents distinguish traces of different lengths. 'Perfect recall' means that every agent can distinguish traces with different sequences of information available for him/her. If $\mathcal{L}$ stands for any of acronym of program logics *Act*-CTL or μC, and PL$\mathcal{X}$ stands for any of acronym of epistemic logics PLK$_n$ or PLC$_n$, then let acronym $\mathcal{L}$-$\mathcal{X}$ stand for fusion of logics $\mathcal{L}$ and PL$\mathcal{X}_n$. For example, *Act*-CTL-K$_n$ denotes fusion of *Act*-CTL and PLK$_n$.

It has been demonstrated in [13] that the model checking problem in the class of finitely-generated trace-based synchronous systems with perfect recall is undecidable for *Act*-CTL-C$_n$, μPLK$_n$, and μPLC$_n$ (where $n > 1$), but is decidable for *Act*-CTL-K$_n$ (with a non-elementary lower bound). It was a 'decidability in principle', and is not oriented towards any implementation.

Paper [26] presents a 'direct' (update+abstraction)-algorithm for model checking *Act*-CTL-K$_n$ in perfect recall synchronous environments. This (update+ abstraction)-algorithm has been inspirited by [21]. It is based on a simple transformation of *Act*-CTL-K$_n$ formulas into formulas of Act^{+n}-CTL (i.e. *Act*-CTL with n fresh action symbols) and on a reduction of infinite synchronous perfect recall system to finite model $TR_k(E)$ which consists of k-trees (special finite trees of height k). Thus the resulting model checking algorithm simply solves formulas of Act^{+n}-CTL on k-trees.

Unfortunately, the upper bound for size of this finite model is a non-elementary function of the size of the background finite system [26]. Hence a straightforward use of a model checker for CTL for model checking Act^{+n}-CTL on k-trees is likely to be a non-feasible task. Roughly speaking, this space is too big to be treated as a finite. It implies that for model checking Act^{+n}-CTL on k-trees it makes sense to try techniques which have been developed for infinite-state model checking.

A very popular approach to infinite-state model checking is formalism of well-structured labeled transition systems. Fundamental papers [1,10] have proved the decidability of liveness (reachability) and progress (eventuality) properties in well-structured single action labeled transition systems. Roughly speaking, a well-structured single action labeled transition system is provided with (pre-)order where transition 'preserves' this (pre-)order, and its labeling forms cones with respect to this (pre-)order. Paper [19] has generalized cited decidability results for disjunctive formulas of the propositional μ-Calculus [18] in well-structured labeled multi-action transition systems.

In the present paper we demonstrate that model $TR_k(E)$ provided with a special sub-tree partial order forms a well-structured labeled transition system where every property expressible in the μ-Calculus can be characterized by a finite computable set of maximal trees that enjoy the property. We tried

[1] Let us remark that we work with traces that are finite sequences. Every finite sequence represents current state of the system (that is the last element of the sequence) and system's past (that is maximal prefix of the sequence).

feasibility of this approach to model checking of *Act*-CTL-K$_n$ in trace-based synchronous perfect recall synchronous environment by automatic model checking simple, but big example[2].

2 Background Logics and Their Fusion

Logics we are going to discuss are propositional polymodal logics. Semantics of these logics is defined in models which are called Kripke structures or labeled transition systems (LTS); it is defined in terms of satisfiability relation $\models$.

Definition 1. Let $\{true, false\}$ be Boolean constants, *Prp* and *Rlt* be disjoint finite alphabets of propositional variables and relational symbols. *Syntax* of our logics consists of formulas which are constructed from Boolean constants, propositional variables, and connectives[3] $\neg$, $\wedge$, $\vee$ and some modalities.

Definition 2. A *transition system* (synonym: Kripke frame) is a tuple (D, I), where the domain D is a non-empty set of elements that are called states (or worlds), and the interpretation I is a total mapping $I : Rlt \to 2^{D\times D}$. For every $r \in Rlt$ an *r-run* is a maximal sequence of states $ws = s_1 \dots s_i s_{i+1} \dots$ such that for all adjacent states within the sequence $(s_i, s_{i+1}) \in I(r)$. For every finite i within ws let ws_i stands for the element s_i. *Kripke model* or *labeled transition system (LTS)* M is a triple (D, I, V), where (D, I) is a Kripke frame, and the valuation V_M maps propositional variables into subsets of D.

Definition 3. A *satisfiability relation* $\models$ between models, worlds, and formulas can be defined inductively with respect to a structure of formulas as follows[4] . For Boolean constants $w \models_M true$ and $w \not\models_M false$ for any world w and model $M = (D, I, V)$. For propositional variables we have: $w \models_M p$ iff $w \in V(p)$. For connectives $\models$ is defined in the standard manner: $w \models_M \neg\phi$ iff $w \not\models_M \phi$, $w \models_M \phi\wedge\psi$ iff $w \models_M \phi$ and $w \models_M \psi$, $w \models_M \phi\vee\psi$ iff $w \models_M \phi$ or $w \models_M \psi$. Definition of $\models$ for modalities is specific for every particular propositional polymodal logic.

A particular example of propositional polymodal logics is Propositional Logic of Knowledge (PLK) [9]. It is the simplest epistemic logic. Informally speaking, PLK is a polymodal variant of the basic propositional modal logic **S5** [3]. A special terminology, notation and models are used in this framework.

Definition 4. *(of Propositional Logic of Knowledge for n agents PLK$_n$)*
Let $n > 0$ be an integer. The alphabet of relational symbols consists of a set of natural numbers $[1..n]$ representing names of agents. Notation for modalities is: if $i \in [1..n]$ and ϕ is a formula, then $(K_i\phi)$ and $(S_i\phi)$ are formulas[5]. For

[2] The size of the initial background environment E is 120000 and the size of the corresponding generated finite model $TR_k(E)$ is about 10^{36000}.

[3] Standard abbreviations $\to$ and $\leftrightarrow$ are admissible too.

[4] Throughout the paper $\not\models$ stays for negation of $\models$.

[5] They are read as '(an agent) i knows ϕ' and '(an agent) i supposes ϕ'.

every agent $i \in [1..n]$ in every model $M = (D, I, V)$ interpretation $I(i)$ is an equivalence, i.e. a symmetric, reflexive, and transitive binary relation on D. Every model M, where all agents in $[1..n]$ are interpreted in this way, is denoted as $(D, \overset{1}{\sim}, \ldots, \overset{n}{\sim}, V)$ instead of (D, I, V) with $I(i) = \overset{i}{\sim}$ for every $i \in [1..n]$. In particular, for every $i \in [1..n]$ and every ϕ,

- $w \models_M (S_i\phi)$ iff for some w': $w \overset{i}{\sim} w'$ and $w' \models_M \phi$,
- $w \models_M (K_i\phi)$ iff for every w': $w \overset{i}{\sim} w'$ implies $w' \models_M \phi$.

Another propositional polymodal logic *Act*-CTL is a variant of the basic propositional branching time temporal logic Computational Tree Logic (CTL) [8,4,5] extended by action symbols.

Definition 5. *(of Act-CTL)*
In the case of *Act*-CTL the alphabet of relational symbols consists of action symbols *Act*. Notation for basic modalities is: if $a \in Act$ and ϕ is a formula, then $(\mathbf{AX}^a\phi)$ and $(\mathbf{EX}^a\phi)$ are formulas. Syntax of *Act*-CTL has also some other special constructs associated with action symbols: if $a \in Act$, ϕ and ψ are formulas, then $(\mathbf{AG}^a\phi)$, $(\mathbf{AF}^a\phi)$, $(\mathbf{EG}^a\phi)$, $(\mathbf{EF}^a\phi)$, $(\mathbf{A}\phi\mathbf{U}^a\psi)$, and $(\mathbf{E}\phi\mathbf{U}^a\psi)$[6] are formulas too. For every model $M = (D, I, V)$ semantics of 'universal' special constructors follows:

- $w \models_M \mathbf{AX}^a\phi$ iff $ws_2 \models_M \phi$ for every a-run ws with $ws_1 = w$,
- $w \models_M \mathbf{AG}^a\phi$ iff $ws_j \models_M \phi$ for every a-run ws with $ws_1 = w$ and every $1 \leq j \leq |ws|$,
- $w \models_M \mathbf{AF}^a\phi$ iff $ws_j \models_M \phi$ for every a-run ws with $ws_1 = w$ and some $1 \leq j \leq |ws|$,
- $w \models_M \mathbf{A}(\phi\mathbf{U}^a\psi)$ iff $ws_j \models_M \phi$ and $ws_k \models_M \psi$ for every a-run ws with $ws_1 = w$, for some $1 \leq k \leq |ws|$ and every $1 \leq j < k$,

Semantics of 'existential' constructors $\mathbf{EX}^a$, $\mathbf{EG}^a$, $\mathbf{EF}^a$, $\mathbf{EU}^a$ is similar but refers to some a-run.

The standard CTL is *Act*-CTL with a singleton alphabet *Act*.

We are going to define a combined Propositional Logic of Knowledge and Branching Time *Act*-CTL-K$_n$.

Definition 6. *(of Act-CTL-K$_n$)*
Let $[1..n]$ be a set of agents $(n > 0)$, and *Act* be a finite alphabet of action symbols. Syntax of *Act*-CTL-K$_n$ admits all knowledge modalities K_i, and S_i for $i \in [1..n]$, and all branching-time constructs $\mathbf{AX}^a$, $\mathbf{AG}^a$, $\mathbf{AF}^a$, $\mathbf{AU}^a$, $\mathbf{EX}^a$, $\mathbf{EG}^a$, $\mathbf{EF}^a$, $\mathbf{EU}^a$. Semantics is defined in terms of satisfiability $\models$. An environment is a tuple $E = (D, \overset{1}{\sim}, \ldots, \overset{n}{\sim}, I, V)$ such that $(D, \overset{1}{\sim}, \ldots, \overset{n}{\sim}, V)$ is a model for PLK$_n$ and (D, I, V) is a model for *Act*-CTL. Satisfiability is defined by induction

[6] $\mathbf{A}$ is read as 'for all futures', $\mathbf{E}$ – 'for some futures', $\mathbf{X}$ – 'next time', $\mathbf{G}$ – 'always', $\mathbf{F}$ – 'sometime', $\mathbf{U}$ – 'until', and a sup-index ^{a} is read as 'in a-run(s)'.

according to semantics of propositional (def. 3), knowledge (def. 4), and branching time constructs (def. 5). For every environment E and every formula ϕ let $E(\phi)$ be the set $\{w \mid w \models_E \phi\}$ of all worlds that satisfies formula ϕ in E.

We are mostly interested in trace-based perfect recall synchronous environments generated from background finite environments. In these environments states are sequences of worlds of initial environments with history of actions that generate them. Agent does not distinguish these sequences if the background system performs the same sequence of actions, and if these sequences have the same number of worlds, and agent can not distinguish these sequences world by world (in the background environment). We can transit from a sequence to another one with respect to an action a by extending the sequence by a state that can be reached by a from the last state of the sequence. Propositionals are evaluated at the last state of sequences with respect to their evaluations in the background environment.

Definition 7. *(of Perfect Recall Synchronous environment)*
Let E be an environment $(D, \overset{1}{\sim}, \ldots, \overset{n}{\sim}, I, V)$. A trace-based Perfect Recall Synchronous environment generated by E is another environment $(D_{PRS(E)}, \overset{1}{\underset{\text{prs}}{\sim}}, \ldots, \overset{n}{\underset{\text{prs}}{\sim}}, I_{PRS(E)}, V_{PRS(E)})$, where

- $D_{PRS(E)}$ is the set of all pairs (ws, as), where[7]
  $ws \in D^+$, $as \in Act^*$, $|ws| = |as| + 1$, and
  $(ws_j, ws_{j+1}) \in I(as_j)$ for every $j \in [1..|as|]$;
- for every $i \in [1..n]$ and for all (ws', as'), $(ws'', as'') \in D_{PRS(E)}$,
  $(ws', as') \overset{i}{\underset{\text{prs}}{\sim}} (ws'', as'')$ iff
  $as' = as''$ and $ws'_j \overset{i}{\sim} ws''_j$ for every $j \in [1..|ws|]$;
- for every $a \in Act$ and for all (ws', as'), $(ws'', as'') \in D_{PRS(E)}$,
  $((ws', as'), (ws'', as'')) \in I_{PRS(E)}(a)$ iff[8]
  $as'' = as'^\wedge a$, and $ws'' = ws'^\wedge w''$, $(w', w'') \in I(a)$, where w' is the last element in ws';
- for every $p \in Prp$ and for every $(ws, as) \in D_{PRS(E)}$,
  $(ws, as) \in V_{PRS(E)}(p)$ iff $ws_{|ws|} \in V(p)$.

3 Bounded Knowledge Update and Abstraction

Below in this section we recall definitions and results from [26] (slightly reformulated for the lack of space) that lead to (update+abstraction)-algorithm and evaluation of its non-elementary complexity. We examine the model checking problem for Act-CTL-K$_n$ in perfect recall synchronous environments generated from finite environments. The following formalization of the problem has been introduced in [26].

[7] For every set S let S^+ be the set of all non-empty finite sequences over S and S^* be the set of all finite sequences over S.

[8] Operation $^\wedge$ stands for the concatenation of finite words.

Definition 8. *(of the model checking problem)*
The model checking problem for *Act*-CTL-K$_n$ in perfect recall synchronous environments is to validate or refute $(ws, as) \models_{PRS(E)} \phi$, i.e. whether ϕ is satisfiable on (ws, as) in $PRS(E)$, where E is a finite environment, $(ws, as) \in D_{PRS(E)}$, ϕ is a formula of *Act*-CTL-K$_n$.

Definition 9. The *knowledge depth* of a formula is the maximal nesting of knowledge operators in that formula. For every $k \geq 0$ let *Act*-CTL-K$_n^k$ be sublogic of *Act*-CTL-K$_n$ with knowledge depth bounded by k.

It is obvious that *Act*-CTL-K$_n = \bigcup_{k\geq 0}$ *Act*-CTL-K$_n^k$.

For every integer $k \geq 0$ we define by mutual recursion a set $\mathcal{T}_k$ of *k-trees over E*, and a set $\mathcal{F}_k$ of *forests of k-trees over E*.

Definition 10. Let $\mathcal{T}_0$ be a set of all tuples of the form $(w, \emptyset, \ldots, \emptyset)$, where w is a world and the number of copies of the empty set $\emptyset$ is equal to the number of agents n. Once $\mathcal{T}_k$ has been defined, let $\mathcal{F}_k$ be the set of all subsets of $\mathcal{T}_k$. Now, define $\mathcal{T}_{k+1}$ as the set of all tuples of the form $(w, U_1, \ldots, U_n)$, where w is a world and $U_i \neq \emptyset$ is in $\mathcal{F}_k$ for each $i \in [1..n]$. Let us denote $\bigcup_{k\geq 0} \mathcal{T}_k$ by $\mathcal{T}$.

Intuitively, a k-tree is a finite tree of height k whose vertices are labeled by worlds of the environment E and edges are labeled by agents. In a tuple $(w, U_1, \ldots, U_n)$, the world w represents the actual state of the universe, and for each $i \in [1..n]$ the set U_i represents knowledge of the agent i.

The following *update functions* G_k^a generate k-trees obtained from some k-tree after action a taking into account knowledge of every agent.

Definition 11. For every number $k \geq 0$, $a \in Act$ and $i \in [1..n]$, functions $G_k^a : \mathcal{T}_k \times D \to \mathcal{T}_k$ and $H_{k,i}^a : \mathcal{F}_k \times D \to \mathcal{F}_k$, are defined by induction on k and mutual recursion. Let $G_0^a(tr, w) = (w, \emptyset, \ldots, \emptyset)$ iff[9] $(root(tr), w) \in I(a)$. Once G_k^a has been defined, we can define for each $i \in [1..n]$ the function $H_{k,i}^a(U, w) = \{G_k^a(tr, w') \mid tr \in U \text{ and } w' \overset{i}{\sim} w\}$. Now let $G_{k+1}^a((w, U_1, \ldots, U_n),\ w')$ be

$$(\ w'\ , H_{k,1}^a(U_1, w'),\ \ldots\ , H_{k,n}^a(U_n, w')\ ) \text{ iff } (w, w') \in I(a).$$

The following model can be associated with the synchronous environment with perfect recall $PRS(E)$.

Definition 12. *(of model $TR_k(E)$)*
For every $k \geq 0$ let $TR_k(E)$ be the following model $(D_{TR_k(E)}, I_{TR_k(E)}, V_{TR_k(E)})$:

- $D_{TR_k(E)}$ is the set of all 0-, ..., k-trees over E for n agents;
- for $a \in Act$: $I_{TR_k(E)}(a) = \{(tr', tr'') \in D_{TR_k(E)} \times D_{TR_k(E)} \mid$
  $tr'' = G_j^a(tr', w)$ for some $j \in [0..k]$ and some $w \in D_E\ \}$;
  for $i \in [1..n]$: $I_{TR_k(E)}(i) = \{(tr', tr'') \in D_{TR_k(E)} \times D_{TR_k(E)} \mid$
  $tr'' \in U_i$ and $tr' = (w, U_1, \ldots, U_n)$ for some $w \in D_E\ \}$;
- $V_{TR_k(E)}(p) = \{tr \mid root(tr) \in V_E(p)\}$ for $p \in Prp$.

[9] Operation *root* on trees returns the root of the argument. In particular, for a k-tree $root(w, U_1, \ldots, U_n)$ returns w.

Definition 13. *(of Act^{+n}-CTL)*
Let Act^{+n} be $Act \cup [1..n]$. A natural translation of formulas of *Act*-CTL-K$_n$ to formulas of Act^{+n}-CTL is simple: just replace every instance of K_i and S_i by corresponding $\mathbf{AX}^i$ and $\mathbf{EX}^i$, respectively ($i \in [1..n]$). For every formula ϕ of *Act*-CTL-K$_n$, let us denote by ϕ^{+n} the resulting formula of Act^{+n}-CTL.

Definition 14. Complete tree is a k-tree $(w, U_1, ..., U_n)$ such that $\{root(tr) | tr \in U_i\} = \{w' \in D | w \overset{i}{\sim} w'\}$ and all trees in U_i are complete trees for every $i \in [1..n]$. Since a state in the root of a complete tree defines the tree uniquely, let us denote the complete tree with root w by $tr(w)$.

Definition 15. Let E be an environment, and $k \geq 0$. A correspondence $tree_k$ between $D_{PRS(E)}$ and k-trees $tree_k : (ws, as) \mapsto tree_k(ws, as)$ is defined by the following.

1. Let tr_1 be complete k-tree $tr(ws_1)$;
2. for every $l \in [2..|ws|]$ let tr_l be $G_k^{as_{l-1}}(tr_{l-1}, ws_l)$;
3. let $tree_k(ws, as)$ be $tr_{|ws|}$.

The following proposition summarizes propositions 4, 5, and 6 from [26].

Proposition 1
For every integer $k \geq 0$ and $n \geq 1$ and every environment E, for every formula ϕ of Act-CTL-K_n with the knowledge depth k at most there exists bijective correspondence $tree_k : D_{PRS(E)} \to D_{TR_k(E)}$ that

$$(ws, as) \models_{PRS(E)} \phi \textit{ iff } tree_k(ws, as) \models_{TR_k(E)} \phi^{+n}.$$

The following (update+abstraction) model checking algorithm is based on the above proposition 1:

1. Input a formula ϕ of *Act*-CTL-K$_n$ and count its knowledge depth k;
2. convert ϕ into the corresponding formula $\psi \equiv \phi^{+n}$ of Act^{+n}-CTL;
3. input a finite environment E and construct the finite model $TR_k(E)$;
4. input a trace (ws, as) and construct the corresponding k-tree tr;
5. model check ψ on tr in $TR_k(E)$.

Its correctness immediately follows from the proposition. In contrasts, complexity of the algorithm is not so straightforward. Paper [26] has proved that its upper bound nonelementary depends on the size of the formula, the number of states, the knowledge depth k and the number of agents n.

4 TR_k as Ideal-Based Model

In principle size of $TR_k(E)$ is finite, but it is simply too big to be treated as finite. Due to this reason for model checking $TR_k(E)$ we would like to try techniques that are in use for model checking infinite systems. In particular, we try a formalism of well-structured labeled transition systems [1,10] that is a very popular approach to infinite-state model checking.

In well-structured labeled transition systems we can represent a set of states (that is semantics of some formula) by some subset that is usually much smaller than the set itself. Usually this representative subset is a collection of minimal or maximal elements of the set of interest. In this case model checking can compute representative subsets and then restore complete semantics of formulas.

Definition 16. Let D be a set. A *partial order* is a reflexive, transitive, and antisymmetric binary relation R on D. A *preorder* is a reflexive and transitive binary relation R on D. We use prefix, infix and postfix notation for preorders: $R(d', d'')$, $d'(R)d''$ (or $d'Rd''$) , and $(d', d'') \in R$. A *well-preorder* is a preorder R where every infinite sequence $d_1, \ldots d_i, \ldots$ of elements of D contains a pair of elements d_m and d_n so that $m < n$ and $d_m(R)d_n$.

Definition 17. Let (D, R) be a *well-preordered set* (i.e. a set D provided with a well-preorder R). An *ideal (synonym: cone)* is an upward closed subset of D, i.e. a set $C \subseteq D$ such that for all $d', d'' \in D$, if $d'(R)d''$ and $d' \in C$ then $d'' \in C$. Every $d \in D$ generates a cone $(\uparrow d) \equiv \{e \in D \mid d(R)e\}$. For every subset $S \subseteq D$, a *basis* of S is a subset $B \subseteq S$ such that for every $s \in S$ there exists $b \in B$ that $b(R)s$.

Definition 18. A *well-preordered transition system (WPTS)* is a triple (D, R, I) such that (D, R) is a well-preordered set and (D, I) is a Kripke frame.

We are mostly interested in well-preordered transition systems with decidable and compatible well-preorder and interpretation. The standard decidability condition for the well-preorder is straightforward: $R \subseteq D \times D$ is decidable.

Definition 19. Let (D, R, I) be a WPTS.

– *Decidability (tractable past) condition*: there exists a computable total function $BasPre : D \times Act \to 2^D$ such that for every $w \in D$, for every $a \in Act$, $BasPre(w, a)$ is a finite basis of $\{u \in D \mid (u, v) \in I(a) \text{ and } w(R)v\}$.
– *Compatibility condition*: preorder R is compatible with interpretation $I(a)$ of every action symbol $a \in Act$. (Three equivalent definitions for compatibility of R and $I(a)$ are presented in Tab. 1.)

Definition 20. *(of ideal-based model)*
A well preordered transition system is said to be well-structured transition system (WSTS) iff its preorder is decidable, it meets tractable past and compatibility conditions. A well-structured labeled transition system (WSLTS) is a quadruple (D, R, I, V), where (D, I, V) is a labeled transition system, and (D, R, I) is a well-structured transition system. An ideal-based model is a well-structured labeled transition system (D, R, I, V), where V interprets every propositional variable $p \in Prp$ by a cone.

Table 1. Equivalent compatibility conditions

notation	
logic	$\forall s_1', s_1'', s_2' \; \exists s_2'' :$ $s_1' \xrightarrow{I(a)} s_1'' \;\&\; R(s_1', s_2') \;\Rightarrow$ $\Rightarrow s_2' \xrightarrow{I(a)} s_2'' \;\&\; R(s_1'', s_2'')$
diagram	$\begin{tikzcd} s_1'' \arrow[r, phantom, "\overset{(R)}{\cdots}"] & s_2'' \\ s_1' \arrow[u] \arrow[r, phantom, "(R)"] & s_2' \arrow[u, dashed] \end{tikzcd}$
algebraic	$R^- \circ I(a) \;\subseteq\; I(a) \circ R^-$

The μ-Calculus of D.Kozen (μC) [18] is a very powerful propositional program logic with fixpoints. It is widely used for specification and verification of properties of finite state systems. We would like to skip formal definition of μC due to space limitations. Please, refer to [25] for the elementary introduction to μC. A comprehensive definition of μC can be found in a monograph [2]. Paper [19] has demonstrated that model checking problem in ideal-based models is decidable for μC formulas without negation, conjunction, boxes, and greatest fixpoints.

In the following we assume we are given an environment E and $k, n \geq 0$.

Definition 21. Let us define *binary relation* $\succ$ *on* $D_{TR_k(E)}$. For all trees of equal height $tr' = (w', U_1', \ldots, U_n')$ and $tr'' = (w'', U_1'', \ldots, U_n'')$ in $D_{TR_k(E)}$, let us write $tr' \succ tr''$ (and say that tr' has a subtree tr'') iff $w' = w''$ and for every $i \in [1..n]$, for every $st'' \in U_i''$ there exists $st' \in U_i'$ that $st' \succ st''$.

Theorem 1. *Binary relation $\succ$ is a partial order on k-trees such that model $TR_k(E)$ provided with this partial order becomes an ideal-based model, where semantics of every formula of μC is a cone with computable finite basis.*

Proof

First, $\succ$ is a partial order since $tr' \succ tr''$ iff all branches in both trees have equal length and the set of vertexes and edges of tr'' is a subset of the set of vertexes and edges of tr' (i.e. just some branches are skipped). It is decidable relation due to the same argument. It is also a well-preorder since $D_{TR_k(E)}$ is finite.

Second, $\succ$ enjoy tractable past since, in principle, we can find preimage of every tree for every 'action' transition and for every 'knowledge' transition (defined by $I_{TR_k(E)}(a)$ and $I_{TR_k(E)}(i)$, respectively for $a \in Act$ and $i \in [1..n]$, in def. 12) by scanning finite space $TR_k(E)$. But there is more effective technic to find preimages based on the notion of complete trees. More efficient algorithm follows. Let $(w, U_1, ...U_n)$ be a k-tree. If $a \in Act$ then for every state u such that $(u, w) \in I(a)$ construct complete tree $tr(u)$; collect all subtrees tr of any of these complete $tr(u)$ (i.e. $tr(u) \succeq tr$) such that $(w, U_1, ...U_n) \succeq G_k^a(tr, w)$. If $i \in [1..n]$ is an agent then just collect all $tr \in U_i$.

But the most efficient[10] algorithm can be described as follows. We consider the case of one agent only, which is easily generalized to the case of n agents. Let us find preimage by induction on a height of a tree. Let $tr = (w, \emptyset)$ be a tree of height 0. Its preimage with respect to an action a is the set of trees of height 0 with roots which are in the preimage for w: $Pre^a(tr) = \{(w', \emptyset) | (w', w) \in I(a)\}$. Let $tr = (w, U)$ be a tree of height k, where root w is a world, and U is a set of trees, which roots are in the set $roots(U) = \{s | s = root(t), t \in U\}$. Its preimage with respect to action a $Pre^a(tr) = Tr'_1 \cup Tr'_2 \cup Tr'$ includes all trees of form $tr'_1 = (w', U'_1) \in Tr'_1$ and $tr'_2 = (w', U'_2) \in Tr'_2$, such that $(w', w) \in I(a)$ and $U'_1 = \{(s', V') | s' \sim w'$, and $\exists s \in root(U) : G^a_{k-1}((s', V'), s) \in U\}$, and $U'_2 = U'_1 \cup \{(s', tr(s') | s' \sim w'$, and $\forall s'' \in a(s') : s'' \nsim w\}$ (Note that each tree of Tr'_1 is k-subtree of some tree in Tr'_2). The set Tr' is defined as follows: $Tr' = \{tr' | tr'_1 \prec tr' \prec tr'_2$ for some $tr'_1 \in Tr'_1$ and $tr'_2 \in Tr'_2\}$. So, roots of trees in U'_1 are all worlds s', which the agent can not distinguish with w', and there exists a world $s \in roots(U)$, such that $(s', s) \in I(a)$, and subtrees of these trees are computed in according to induction assumption. Roots of trees in U'_2 are $roots(U_1)$ supplemented by worlds s'' whose images $a(s'')$ are distinguished with w by the agent, and subtrees corresponding to these roots s'' are complete trees. In addition, preimage includes all "intermediate" trees. We can compute these trees easily due to finiteness of D. Due to definition of update function, it is obvious that $tr = G^a_1(tr', w)$ for every $tr' \in Pre^a(tr)$. There are no other trees in preimage since other roots are impossible, and U'_2 can be extended only by trees with roots which are transformed by action a to worlds indistinguishable with w, but the images of such trees include tr as a subtree.

Third, $\succ$ is compatible with all 'action' transitions and all 'knowledge' transitions. For every action $a \in Act$, and for every pair of trees $tr' \succ tr''$ there exists some sup-tree tr which is a-image of the greater tree tr' and this sup-tree includes a-image of the smaller tree tr'' because a-images are computed recursively by processing each vertex of trees (def. 11). Again we consider the case of one agent only, which is easily generalized to the case of n agents. Let $tr_1 = (w, U_1)$ and $tr_2 = (w, U_2)$ be trees of height k and $tr_1 \prec tr_2$. Note that for every $t_1 \in U_1$ there exists $t_2 \in U_2$ such that $t_1 \prec t_2$ by definition of $\prec$. Let $tr'_1 = (w', U'_1) \in G^a_k(tr_1, w')$. Let us find $tr'_2 = (w', U'_2) \in G^a_k(tr_2, w')$, such that $tr'_1 \prec tr'_2$. Due to def. 11, $(w, w') \in I(a)$ and $U'_1 = \{G^a_{k-1}(tr, w'') | tr \in U_1$ and $w'' \sim w'\}$. By the same definition $U'_2 = \{G^a_{k-1}(tr, w'') | tr \in U_2$ and $w'' \sim w'\}$. Then it is obvious that for every $t'_1 \in U'_1$ there exists $t'_2 \in U'_2$ such that $t'_1 \prec t'_2$, hence $tr'_1 \prec tr'_2$.

For every 'knowledge' action $i \in [1..n]$, and for every pair of trees $tr' \succ tr''$ there exists some sup-tree tr which is i-image of the greater tree tr' and this sup-tree includes i-image of the smaller tree tr'' because computing of i-images of some tree is based on transition to subtrees of this tree (def. 12). Once again we consider the case of one agent only. Let $tr_1 = (w, U_1)$ and $tr_2 = (w, U_2)$ be trees of height k and $tr_1 \prec tr_2$. Note that for every $t_1 \in U_1$ there exists $t_2 \in U_2$ such that $t_1 \prec t_2$ by definition of $\prec$. $(tr_1, t_1) \in I_{TR_k(E)}(1)$ holds for every $t_1 \in U_1$

[10] To the best of our knowledge.

and $(tr_2, t_2) \in I_{TR_k(E)}(1)$ holds for every $t_2 \in U_2$ due to def. 12. it is obviously implies that $\prec$ is compatible with 'knowledge' transitions.

Fourth, $TR_k(E)$ is an ideal-based model. It is obvious that valuation of every propositional variable forms a cone with basis consisting of complete trees with roots which are states where this propositional variable holds, due to def. 12: $V_{TR_k(E)}(p) = \{tr | root(tr) \in V_E(p)\} = \uparrow \{tr(w) | w \in V_E(p)\}$ for $p \in Prp$ since p is satisfiable in every k-subtree of every $tr \in V_{TR_k(E)}(p)$. There are no other trees with this property since the set includes all complete trees with these roots. Note, that negation of propositional variable is a cone also: $V_{TR_k(E)}(\neg p) = \uparrow \{tr(w) | w \notin V_E(p)\}$ for $p \in Prp$.

Finally we prove that semantics of every formula of μC is a cone with computable finite basis by induction on structure of normal formulas in which negation is used in literals[11] only. (Every μC formula is equivalent to some normal formula [25].) Induction basis deals with literals; for propositional variables it is proved already, for their negations proof is similar. Induction step consists of a number of cases: for disjunction $\vee$, conjunction $\wedge$, box $[\ ]$ and diamond $\langle\ \rangle$.

Basis of disjunction of formulas is union of bases of these formulas.

Basis of conjunction of formulas consists of maximal trees which are subtrees of trees from bases of these formulas simultaneously. Let basis of formula ϕ be B_ϕ and basis of formula ψ – B_ψ. Hence, the set $B_{\phi\wedge\psi} = \{tr | tr \prec tr_\phi \in B_\phi$ and $tr \prec tr_\psi \in B_\psi\}$ is computable and finite due to finiteness of sets B_ϕ and B_ψ. This set is a basis for semantics of $\phi \wedge \psi$.

Bases of a box- or diamond-formula is a computable cone due to properties of tractable past and compatibility. Let us find basis of semantics of formula $\phi = \langle a \rangle \psi$[12]. For simplicity let basis of ψ consists of single tree: $B_\psi = \{tr\}$. Denote preimage of tr with respect to action a as $Pre^a(tr)$, defined above. Hence, preimage of all trees in semantics of ψ with respect to action a is the set of all k-subtrees from $Pre^a(tr) = Tr'_1 \cup Tr'_2 \cup Tr'$. Every tree in $Tr'_1 \cup Tr'$ is k-subtree of some tree in Tr'_2. Hence, basis of preimage of semantics of ψ with respect to action a is the finite set $Pre^a(\psi) = Tr'_2$, due to definition of Tr'_2 and knowledge update function a-image of every k-subtree of every tree in Tr'_2 is some tree in the cone generated by tree tr. By definition of diamond modality this set is a basis of semantics of formula $\phi = \langle a \rangle \psi$ also.

The least $\mu x.\phi$ and the greatest fixpoints $\nu x.\phi$ in finite models[13] are equivalent to 'infinite disjunctions' and 'infinite conjunctions':

– $false \vee \phi_x(false) \vee \phi_x(\phi_x(false)) \vee \ldots = \bigvee_{j\geq 0} \phi_x^j(false)$,
– $true \wedge \phi_x(true) \wedge \phi_x(\phi_x(true)) \wedge \ldots = \bigwedge_{j\geq 0} \phi^j(true)$,

where $\phi_x(\psi)$ is a result of substitution of a formula ψ instead of x, $\phi_x^0(\psi)$ is ψ, and $\phi_x^{j+1}(\psi)$ is $\phi_x(\phi_x^j(\psi))$ for $j \geq 0$. In $TR_k(E)$ one can assume these infinite disjunctions and conjunctions to be finite and bounded by the number of k-trees

[11] A literal is a propositional variable or its negation.
[12] Basis of semantics of $[a]\psi$ is treated analogously.
[13] By the finite-case Tarski-Knaster fixpoint theorem.

in $TR_k(E)$. This observation reduces the case of fixpoints to combination of cases for disjunction, conjunction, box and diamond that are proved already. ■

Note that semantics of *every* formula of μC in model $TR_k(E)$ is a computable cone in contrast to [19] where arbitrary ideal-based models have been studied.

It is well-known that standard CTL is expressible in μC (see for example [25]). This translation of CTL to μC can be generalized easily to *Act*-CTL.

$$\begin{array}{ll}
\mathbf{AX}^a\varphi \leftrightarrow [a]\varphi & \mathbf{EX}^a\varphi \leftrightarrow \langle a\rangle\varphi \\
\mathbf{AG}^a\varphi \leftrightarrow \nu x.\ (\varphi \wedge [a]x) & \mathbf{AF}^a\varphi \leftrightarrow \mu x.\ (\varphi \vee [a]x) \\
\mathbf{EG}^a\varphi \leftrightarrow \nu x.\ (\varphi \wedge \langle a\rangle x) & \mathbf{EF}^a\varphi \leftrightarrow \mu x.\ (\varphi \vee \langle a\rangle x) \\
\mathbf{A}(\varphi\mathbf{U}^a\psi) \leftrightarrow \mu x.\ (\psi \vee (\varphi \wedge [a]x)) & \mathbf{E}(\varphi\mathbf{U}^a\psi) \leftrightarrow \mu x.\ (\psi \vee (\varphi \wedge \langle a\rangle x))
\end{array}$$

It implies the following corollary.

Corollary 1. *Semantics of every formula of Act^{+n}-CTL in $TR_k(E)$ is a cone with respect to $\succ$ with computable finite bases.*

5 Conclusion

In this paper we have shown that space $TR_k(E)$ provided with sub-tree partial order forms a well-structured labeled transition system where every property expressible in the propositional μ-Calculus, can be characterized by a finite computable set of maximal trees that enjoy the property. We tried feasibility of this approach to model checking of *Act*-CTL-K$_n$ in trace-based synchronous perfect recall synchronous environment by automatic model checking simple, but big example (the size of model TR_k is about 10^{36000}). Data structures that are used in the experiment are so-called vector-affine trees [12]. A presentation of a background theory and of our experimental model checker is a topic for a future publication.

To the best of our knowledge, the only reported (experimental) model checker for perfect recall synchronous systems is MCK [11]. It works in a linear as well as branching time settings. For perfect recall synchronous systems in temporal dimension MCK supports 'next' operator only, but neither 'always', 'sometimes', nor 'until' (although the model checking theory for the full combination of knowledge with Propositional Logic of Linear Time has been already developed [22]). The present paper has developed a technique that may lead to practical model checking the full combination of knowledge with branching time logics (a là Computation Tree Logic) with 'next' operator as well as with 'always', 'sometimes', and 'until'.

References

1. Abdulla P.A., Ĉerâns K., Jonsson B., and Tsay Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation, v.160 (1-2), 2000, p.109-127.
2. Arnold A. and Niwinski D. *Rudiments of μ-calculus.* North Holland, 2001.

3. Bull R.A., Segerberg K. Basic Modal Logic. In: Handbook of Philosophical Logic. Vol.II. Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.). p. 1–88.
4. Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J. Symbolic Model Checking: 10^{20} states and beyond. Information and Computation, 1992. v.98(2), p. 142–170.
5. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.
6. Dixon C., Fernandez Gago M-C., Fisher M., and van der Hoek W. Using Temporal Logics of Knowledge in the Formal Verification of Security Protocols. In: Proceedings of TIME 2004, 1st-3rd July 2004, Tatihou, Normandie, France. IEEE.
7. Dixon C., Nalon C. and Fisher M. Tableau for Logics of Time and Knowledge with Interactions Relating to Synchrony, Journal of Applied Non-Classical Logics, v.14, n.4, p.397-445, 2004.
8. Emerson E.A. Temporal and Modal Logic. In: Handbook of Theoretical Computer Science. v.B, Elsevier and MIT Press, 1990, p. 995–1072.
9. Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995.
10. Finkel A., Schnoebelen Ph. *Well-structured transition systems everywhere!* Theor. Comp. Sci., v.256(1-2), 2001, p.63-92.
11. Gammie P. and van der Meyden R. MCK: Model Checking the Logic of Knowledge. Springer-Verlag Lect. Notes Comp. Sci., v.3114, 2004, p.479-483.
12. Garanina N.O. Verification of Distributed Systems on base of Affine Data representation and Logics of Knowledge and Actions. Ph.D Thesis, A.P. Ershov Institute of Informatics Systems, 2004 (in Russian).
13. Garanina N.O., Kalinina N.A. and Shilov N.V. Model checking knowledge, actions and fixpoints. Proc. of Concurrency, Specification and Programming Workshop CS&P'2004, Germany, 2004, Humboldt Universitat, Berlin, Informatik-Bericht Nr.170, v.2, p.351-357.
14. Halpern J. Y., van der Meyden R., and Vardi M. Y. Complete Axiomatizations for Reasoning about Knowledge and Time. SIAM Journal on Computing, v.33(3), 2004, p. 674-703.
15. van der Hoek W. and Wooldridge M.J. Model Checking Knowledge and Time. Lecture Notes in Computer Science, v.2318, p.95-111, 2002.
16. Kacprzak M., Lomuscio A., Penczek W. Unbounded Model Checking for Knowledge and Time. Proceedings of the CS&P'2003 Workshop, Warsaw University, v.1, p.251-264.
17. Kacprzak M., Penczek W. Model Checking for Alternating-Time mu-Calculus via Translation to SAT. Proc. of Concurrency, Specification and Programming Workshop CS&P'2004, Germany, 2004, Humboldt Universitat, Berlin, Informatik-Bericht Nr.170, v.2.
18. Kozen D. *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.
19. Kouzmin E.V., Shilov N.V., Sokolov V.A. Model Checking μ-Calculau in Well-Structured Transition Systems. Proceedings of 11th International Symposium on Temporal Representation and Reasoning (TIME 2004), France 2004, IEEE Press, p. 152-155.
20. Kozen D., Tiuryn J. Logics of Programs. In: Handbook of Theoretical Computer Science, v.B., Elsevier and MIT Press, 1990, p. 789–840.
21. van der Meyden R. Common Knowledge and Update in Finite Environments. Information and Computation, 1998, v.140(2), p. 115–157.

22. van der Meyden R., Shilov N.V. Model Checking Knowledge and Time in Systems with Perfect Recall. Springer-Verlag Lect. Notes Comput. Sci., 1999, v.1738, p. 432–445.
23. van der Meyden R. and Wong K. Complete Axiomatizations for Reasoning about Knowledge and Branching Time. Studia Logica, v.75(1), 2003, p. 93-123.
24. Rescher N. Epistemic Logic. A Survey of the Logic of Knowledge. University of Pitsburgh Press, 2005.
25. Shilov N.V. and Yi K. How to find a coin: propositional program logics made easy. In: Current Trends in Theoretical Computer Science, World Scientific, v. 2, 2004, p.181-213.
26. Shilov N.V., Garanina N.O., and Choe K.-M. 2.7. Update and Abstraction in Model Checking of Knowledge and Branching Time. Fundameta Informaticae, v.72, n.1-3, 2006, p.347-361.

Development of a Robust Data Mining Method Using CBFS and RSM

Sangmun Shin[1], Yi Guo[1], Yongsun Choi[1], Myeonggil Choi[1], and Charles Kim[2]

[1] Department of Systems Management & Engineering, Inje University
Gimhae, Kyung-Nam 621-749, South Korea
[2] School of Computer Engineering, Inje University, Gimhae,
Kyung-Nam 621-749, South Korea
sshin@inje.ac.kr, kwakhyok@gmail.com,
(yschoi,mgchoi,charles)@inje.ac.kr

Abstract. Data mining (DM) has emerged as one of the key features of many applications on information system. While Data Analysis (DA) represents a significant advance in the type of analytical tools currently available, there are limitations to its capability. In order to address one of the limitations on the DA capabilities of identifying a causal relationship, we propose an integrated approach, called robust data mining (RDM), which can reduce dimensionality of the large data set, may provide detailed statistical relationships among the factors and robust factor settings. The primary objective of this paper is two-fold. First, we show how DM techniques can be effectively applied into a wastewater treatment process design by applying a correlation-based feature selection (CBFS) method. This method may be far more effective than any other methods when a large number of input factors are considered on a process design procedure. Second, we then show how DM results can be integrated into a robust design (RD) paradigm based on the selected significant factors. Our numerical example clearly shows that the proposed RDM method can efficiently find significant factors and the optimal settings by reducing dimensionality.

1 Introduction

The continuous improvement and application of the information system technology has become widely recognized by industry as critical in maintaining a competitive advantage in the marketplace. It is also recognized that the improvement and application activities are most efficient and cost-effective when implemented during an early process/product design stage. Data mining (DM) has emerged as one of the key features of many applications on computer science. Often used as a means for predicting the future directions, extracting the hidden limitations, and the specifications of a product/process, DM involves the use of data analysis tools to discover previously unknown, valid pattern and relationships from a large database. In the context of a wastewater treatment, DA is often viewed as a potential means to identify the specification of a wastewater treatment process, such as conductivity, Biochemical Oxygen Demand (BOD), and PH because the wastewater treatment process includes a number of input and output factors to improve the water quality.

DA is a term coined to describe the process of sifting through large databases for interesting patterns and relationships. This field spans several disciplines such as database, machine learning, intelligent information system, statistics and expert system. Two approaches that enable standard machine learning algorithms to be applied to a large database are factor selection and sampling. Factor selection is known as an effective method for reducing dimensionality, removing irrelevant and redundant data, increasing mining accuracy, and improving result comprehensibility [1]. Consequently, factor selection has been a fertile field of research and development since 1970's and proven to be effective in removing irrelevant and redundant features, increasing efficiency in mining tasks, improving mining performance like predictive accuracy, and enhancing comprehensibility of learned results. The factor selection algorithm performs a search through the space of feature subsets [2]. In general, two categories of the algorithm have been proposed to solve the factor selection problem. The first category is based on a filter approach that is independent of learning algorithms and serves as a filter to sieve the irrelevant factors. The second category is based on a wrapper approach, which uses an induction algorithm itself as part of the function evaluating factor subset [3]. Because most filter methods is based on a heuristic algorithm for general characteristics of the data rather than a learning algorithm to evaluate the merit of factor subsets as wrapper methods do, filter methods are generally much faster, and has more practical capabilities to utilize high dimensionality than wrapper methods.

Most DA methods associated with the factor selection reported in literature may obtain a number of factors associated with the interesting response factor without providing the detailed information, such as relationships between the input factor and response, statistical inferences, and analyses [4, 5, 6, 7]. Based on this awareness, Witten and Frank [6] suggested alternative DA approach to semiconductor manufacturing problems in order to find significant factors. Further more, Su *et. al* [8] developed an integrated procedure combining a DM method and Taguchi methods. While DA represents a significant advance in the type of analytical tools currently available, there are limitations to its capability on the DA capabilities of identifying a causal relationship [9]. The limitation is that while DA can identify connections between responses and/or factors, it may not necessarily identify a causal relationship.

In order to address this limitation, we develop an enhanced data analysis method incorporating the robust design (RD) principle into DA. Among the process/product design methods currently studied in the science and engineering community, researchers often identify RD as one of the most effective methodology for process/product improvement. Because of their practicability in reducing the inherent uncertainty associated with input factors and process performance, the widespread applications of RD techniques have resulted in significant improvements in process quality, manufacturability and reliability at low cost. However, most RD methods reported in literature may obtain the most favorable solution for a small number of given input control factors without considering reduction of dimensionality for a large database. Although traditional RD methods consider the selection of potential significant factors when they confront a data set including many factors with an interesting response factor, the process is frequently far from objective as individual egos because the selection process is based on drawing insight from a number of readily available sources relying on practitioners' opinion and their experience.

For this reason, we propose an integrated approach, called robust data mining (RDM) as shown in Fig 1, which can reduce dimensionality of the large data set, may provide detailed statistical relationships among the factors and robust factor settings. This RDM approach has not been adequately addressed in the literature nor properly applied to industries. As a result, the primary objective of this paper is two-fold. First, we show how DM techniques can be effectively applied into a wastewater treatment process design by applying a correlation-based feature selection (CBFS) method. This method can evaluate the worth of a subset including input factors by considering the individual predictive ability of each factor along with the degree of redundancy between pairs of input factors. This method is far more effective than any other method when a large number of input factors are considered on a process design procedure. Second, we then show how DM results can be integrated into a RD paradigm based on the selected significant factors. Our numerical example clearly shows that the proposed RDM method can efficiently find significant factors and the optimal settings by reducing dimensionality.

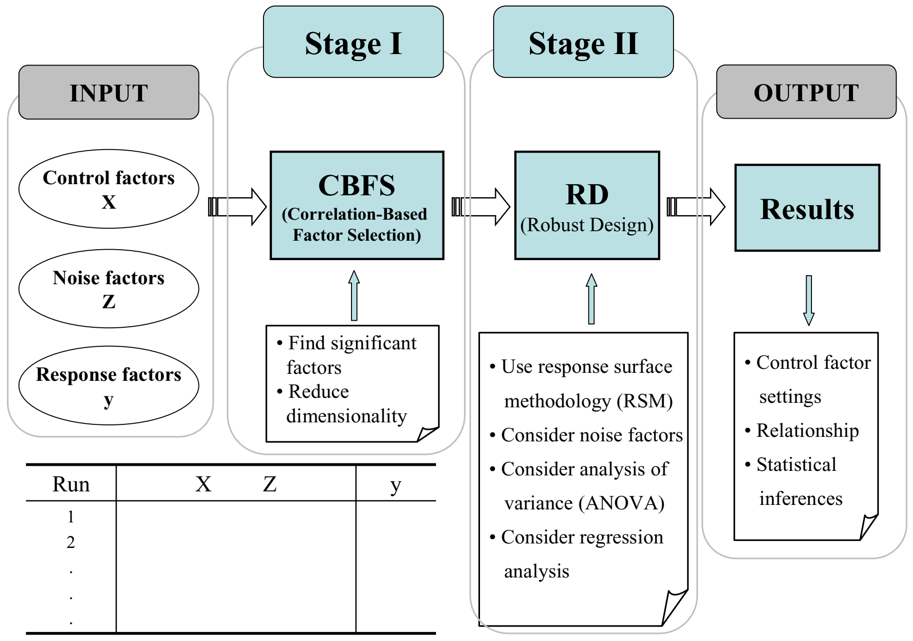

Stage I
Stage II
INPUT
OUTPUT
Control factors X
Noise factors Z
Response factors y
CBFS (Correlation-Based Factor Selection)
RD (Robust Design)
Results
• Find significant factors
• Reduce dimensionality
• Use response surface methodology (RSM)
• Consider noise factors
• Consider analysis of variance (ANOVA)
• Consider regression analysis
• Control factor settings
• Relationship
• Statistical inferences
Run
X
Z
y
1
2
.
.
.

Fig. 1. The overview of the proposed RDA procedure associated with dual stage clearly illustrates roles of data mining and robust design

2 Development of Robust Data Mining Method

2.1 Stage I: Correlation-Based Feature Selection (CBFS) Method

Correlation-Based Feature Selection (CBFS) is a filter algorithm that ranks subsets of input features according to a correlation based heuristic evaluation function. The bias of the evaluation function is toward subsets that contain a number of input factors, which are not only highly correlated with a specified response but also uncorrelated with each other [3, 9, 14]. Among input factors, irrelevant factors should be ignored because they may have low correlation with the given response. Although some selected factors are highly correlated with the specified response, redundant factors must be screened out because they are also highly correlated with one or more of these selected factors. The acceptance of a factor depends on the extent to which it predicts the response in areas of the instance space not already predicted by other factors. The evaluation function of the proposed subset is

$$EV_S = \frac{n\bar{\rho}_{FR}}{\sqrt{n + n(n-1)\bar{\rho}_{FF}}} \tag{1}$$

where EV_S, $\bar{\rho}_{FR}$, and $\bar{\rho}_{FF}$ represents the heuristic evaluation value of a factor subset S containing n factors, the mean of factor-response correlation $(F \in S)$, and the mean of factor-factor inter-correlation, respectively. $\sqrt{n + n(n-1)\bar{\rho}_{FF}}$ and $n\bar{\rho}_{FR}$ indicate the prediction of the response based on a set of factors and the redundancy among the factors. In order to measure the correlation between two factors or a factor and the response, an evaluation of a criterion called symmetrical uncertainty [10].

The symmetrical measure represents that the amount of information gained about Y after observing X is equal to the amount of information gained about X after observing Y. Symmetry is a desirable property for a measure of factor-factor inter-correlation or factor-response correlation. Unfortunately, information gain is not apt to factors with more values. In addition, $\bar{\rho}_{FR}$ and $\bar{\rho}_{FF}$ should be normalized to ensure they are comparable and have the same effect. Symmetrical uncertainty can minimize bias of information gain toward features with more values and normalize its value to the range [0, 1]. The coefficient of symmetrical uncertainty can be calculated by

$$C_{SU} = 2.0 \times \left[\frac{gain}{H(Y) + H(X)} \right] \tag{2}$$

where

$$H(Y) = -\sum_{y \in Y} p(y) \log_2 (p(y))$$

$$H(Y \mid X) = -\sum_{x \in X} p(x) \sum_{y \in Y} p(y \mid x) \log_2 (p(y \mid x))$$

$$gain = H(Y) - H(Y \mid X) = H(X) - H(X \mid Y) = H(Y) + H(X) - H(X,Y)$$

and where $H(Y)$, $p(y)$, $H(Y|X)$, and *gain* represent the entropy of the specified response Y, the probability of y value, the conditional entropy of Y given X, and the information gain that is a symmetrical measure reflects additional information about Y given X, respectively.

2.1.1 Best First Search (BFS) Algorithm

In much literature, finding a best subset is hardly achieved in many industrial situations by using an exhaustive enumeration method. In order to reduce the search spaces for evaluating the number of subsets, one of the most effective methods is the best first search (BFS) method which is a heuristic search method to implement CBFS algorithm [5]. This method is based on an advanced search strategy that allows backtracking along a search space path. If the path being explored begins to look less promising, the best first search can back-track to a more promising previous subset and continue searching from there. The procedure using the proposed BFS algorithm is given by the following steps:

```
Step 1. Begin with the OPEN list containing the start state,
the CLOSE list empty, and BEST• start state (put start state
to BEST).
Step 2. Let a subset, $\theta$ = $arg$ max $EV_s$ (subset), (get the state
from OPEN with the highest evaluation $EV_s$).
Step 3. Remove $s$ from OPEN and add to CLOSED.
Step 4. If $EV_s(\theta) \geq EV_s(BEST)$, then BEST • $\theta$ (put $\theta$ to
BEST).
Step 5. For each next subset $\xi$ of $\theta$ that is not in the OPEN or
CLOSED list, evaluate and add to OPEN.
Step 6. If BEST changed in the last set of expansions, go to
step 2.
Step 7. Return BEST.
```

The evaluation function given in Equation (1) is a fundamental element of CBFS to impose a specific ranking on factor subsets in the search spaces. In most cases, enumerating all possible factor subsets is astronomically time-consuming. In order to reduce the computational complexity, the BFS method is utilized to find a best subset. The BFS method can start with either no factor or all factors. The former search process moves forward through the search space adding a single factor into the result, and the latter search process moves backward through the search space deleting a single factor from the result. To prevent the BFS method from exploring the entire search space, a stopping criterion is imposed. The search process may terminate if five consecutive fully expanded subsets show no improvement over the current best subset.

2.2 Stage II: Connection to RD

Even though a data warehouse contains many factors including both controllable and uncontrollable factors which is known as noise factors. The proposed DA method may provide significant factors associated with the given response. Based on the DA solutions, a further analysis of the given solutions may also be an important part of a process design for applying the detailed and analyzed information to develop a process/product. In this situation, RD principle can be utilized to provide statistical analyses and optimal factor settings for the selected factors associated with the given response by considering the effect of noise factors.

2.2.1 Response Surface Methodology (RSM)

Response surface methodology (RSM) is a statistical tool that is useful for modeling and analysis in situations where the response of interest is affected by several factors. RSM is typically used to optimize the response by estimating an input-response functional form when the exact functional relationship is not known or is very complicated. For a comprehensive presentation of RSM, Box *et al.* [11] and Shin and Cho [12] provide insightful comments on the current status and future direction of RSM.

In many industrial situations, a manufacturing or service process often contains both control and noise factors which may not be handled [13]. Supposing that there are k controllable variables $\mathbf{x} = [x_1, x_2, ..., x_k]$, and r noise variables $\mathbf{z} = [z_1, z_2, ..., z_r]$, the response model incorporating both control and noise factors can be given by

$$y(\boldsymbol{x}, \boldsymbol{z}) = f(\boldsymbol{x}) + h(\boldsymbol{x}, \boldsymbol{z}) + \psi \tag{3}$$

where $f(\mathbf{x})$, $h(\mathbf{x}, \mathbf{z})$, and ψ are the portion of the model that involves only the control factors, the term involving the main effects of the noise factors and the interactions between the control and noise factors, and a random error which is assumed a normal distribution with zero mean and certain variance, respectively. The detailed calculation of $h(\mathbf{x}, \mathbf{z})$ is

$$h(x, z) = \sum_{i=1}^{r} \gamma_i z_i + \sum_{i=1}^{k} \sum_{j=1}^{r} \delta_{ij} x_i z_j \tag{4}$$

where γ_i and δ_{ij} are coefficients of the noise factors and the interactions between the control and noise factors noise factors, respectively. Denoting variance of the noise variables as $\sigma_z{}^2$ and the random errors as ε, and assuming that the noise variables and ε have zero covariance, and then the mean response model by taking the expectation of the response model in Equation (3) can be derived as follows:

$$E_z[y(\boldsymbol{x}, \boldsymbol{z})] = f(\boldsymbol{x}) \, . \tag{5}$$

Using Taylor series expansion, the variance model for the response can be simplified as follows:

$$V_{\mathbf{z}}[y(\mathbf{x}, \mathbf{z})] = \sigma_z{}^2 \sum_{i=1}^{r} \left(\frac{\partial h(\mathbf{x}, \mathbf{z})}{\partial z_i} \right)^2 + \sigma^2 \tag{6}$$

where σ^2 is the mean square error on analysis of variance (ANOVA).

2.2.2 Proposed Robust Data Mining (RDM) Model

Using the two response functions for the process mean and variance given in Equations (5) and (6), a bi-objective problem can be formulated as follows:

$$\begin{aligned} &\text{Minimize} && g_i(\boldsymbol{x}) = [g_1(\boldsymbol{x}), g_2(\boldsymbol{x})]^T = [E_z[y(\boldsymbol{x},z)], V_z[y(\boldsymbol{x},z)]]^T \\ &\text{Subject to} && \mathbf{x} \in \boldsymbol{X} \end{aligned} \tag{7}$$

where g_i: $R^n \rightarrow R^1$ and $g_i(\boldsymbol{x}) \in C^1$, $i = 1, 2$. The set X of feasible solutions given by $X \subseteq \mathrm{R}^n$ is closed and bounded. A point $x^* \in \Omega$ (design space) is said to be a (globally) efficient (Pareto) solution of the bi-objective problem in Equation (3) if there does not exist another $x^* \in \Omega$ for which $g_i(x) \leq g_i(x^*)$, for $i = 1, 2$, with a strict inequality for at least one index i. The image $g_i(x^*)$ of an efficient solution x^* in the objective space is called a (global) Pareto (efficient, nondominated, noninferior) solution. A point $x^* \in \Omega$ is said to be a (globally) weakly efficient solution of the BOP in Equation (3) if there does not exist another $x^* \in \Omega$ for which $g_i(x) < g_i(x^*)$, for $i = 1, 2$. The image $g_i(x^*)$ of a weakly efficient solution x^* in the objective space is called a (global) weak Pareto solution. The BOP in Equation (3) is referred to as the convex BOP if the feasible set X is convex and the objective functions $g_i(x)$, $i = 1,2$, are also convex. It is a well-known fact that the set Z of the convex BOP is convex in R^2 and the Pareto set can be viewed as a convex curve in R^2.

Let $g_1(\boldsymbol{x})$ and $g_2(\boldsymbol{x})$ denote $E_z[y(\boldsymbol{x},z)]$ and $V_z[y(\boldsymbol{x},z)]$, respectively. Since the optimization model subordinates $E_z[y(\boldsymbol{x},z)]$ and $V_z[y(\boldsymbol{x},z)]$ to the constraint associated with ε, the RDM model using the ε−constrained method can be formulated as

$$\begin{aligned} &\text{Minimize} && V_z[y(\boldsymbol{x},z)] \\ &\text{Subject to} && E_z[y(\boldsymbol{x},z)] \leq \varepsilon \\ & && \boldsymbol{x} \in \Omega \end{aligned} \tag{8}$$

where ε represents an upper level of process bias in Equation(8). To solve this RDM problem, the ε–constrained method is used. The ε–constrained method has two important features. The first feature is the interactive nature of a solution process. After each cycle of computation, the results are evaluated before the problem is reconfigured for the next cycle of computation. The second feature is the generation of a series of candidate solutions from which the final solution will eventually be selected. Let x^* be an efficient point and ζ^* be a nondominated criterion vector. Using the ε–constrained method, we can compute the nondominated partial tradeoff rate at $\boldsymbol{x}^*$

$$\left. \frac{V_z(y(\boldsymbol{x},z))}{E_z(y(\boldsymbol{x},z))} \right|_{x^*} \tag{9}$$

between $V_z(y(\boldsymbol{x},z))$ and $E_z(y(\boldsymbol{x},z))$. The nondominated partial tradeoff rate is given by the values of the dual variable associated with $E_z(y(\boldsymbol{x},z))$ in the following model:

$$\begin{aligned} &\text{Minimize} && \eta_1(V_z(y(\boldsymbol{x},z))) = \zeta_1 \\ &\text{Subject to} && \eta_2(E_z(y(\boldsymbol{x},z))) \leq \zeta_2 \\ & && \boldsymbol{x} \in \Omega \end{aligned} \tag{10}$$

where η_1 and η_2 are the gradients of the two objective functions. The partial tradeoff rate given in Equation (9) is “nondominated” in the sense that after a perturbation, the resulting criterion vector is still in the nondominated set. Thus, the ε–constraint method is useful for the generation of locally relevant nondominated trade-off information. The detailed discussion and proof of the ε –constrained method are shown in Shin and Cho [12].

3 Numerical Example

The data set comes from the daily measures of sensors in an urban wastewater treatment plant [15]. We select COND-S that is output conductivity of treated water as the response. Since the conductivity of water is an essential criterion for water purification, the lower the value of conductivity is, the purer the water is. One of the indispensable purposes of water treatment is to reduce the conductivity of water.

If potential significant factors are selected by subjective opinions or experiences, it may often not include important factors on a factor selection process. Our objective is to find the most significant factors to the output response by short time consuming. In particular, during the two-step process of wastewater treatment, we want to make sure whether some input factors can affect the response factor significantly.

3.1 Stage I : Results of Significant Factor Selection Using CBFS Method

The evaluation result of the numeric example calculated by the DM software named “Weka” [15]. Among the BFS, COND-E represents the observation value of initial input conductivity to the plant, therefore, it can hardly be controlled during the RD process. Consequently, we consider COND-E the noise factor, and consider other input factors among BFS the controlled factors.

Table 1. The Water-Treatment Plant Data Set includes 34 factors and 527 instances. Among 34 factors, the output of conductivity (COND-S) may include uncertain effects that are either irrelevant or redundant. Factor2-22 cover all input values measured during the process of two-step treatment, and factor 23-29 cover all output criterion values after two settlers treatment, and factor 30-38 cover the performance criterions.

Q-E	ZN-E	PH-E	DBO-E	DQO-E	...	SED-D	COND-S
35023	3.5	7.9	205	588	...	0.4	2060
29156	2.5	7.7	206	451	...	0.3	1233
39246	2	7.8	172	506	...	0.6	1825
42393	0.7	7.9	189	478	...	0.4	1562
40923	3.5	7.6	146	329	...	0.2	1467
43830	1.5	7.8	177	512	**...**	0.4	1401
...	...	...	...	...	...	...	...

Table 2. As shown in CBFS results, the merit of best subset equals 0.92, the highest value among the calculated 371 subsets. The factor set F = {ZN-E, SED-D, COND-E, COND-P, SS-S, RD-DBO-P, DQO-S} is considered the best factor subset towards the response factor COND-S. Aside from the output factors including SS-S, RD-DBO-P and DQO-S, we get the best input factor subset BFS = {ZN-E, SED-D, COND-E, COND-P}.

Selected Evaluator	The response attribute	COND-S
	Merit of best subset	0.92
	Selected attributes	ZN-E, SED-D, COND-E, COND-P,SS-S, RD-DBO-P, DQO-S
Search method	Search method	Best First
	Search Direction	forward
	Start set	no attributes
	Total number of subsets evaluated	371

3.2 Stage II: Results of RD Using RSM Based on Both Control and Noise Factors

The data mining solution provides the four significant factors as ZH-E, COND-P, SED-D, and COND-E. Among these solutions, a primary input conductivity, COND-E, may often not be controlled in a water treatment process. For this reason, we regard COND-E as a noise factor incorporating the RD principle in order to achieve a robust process and the other factors as control factors. Using MINITAB software package [16], the response function including both control and noise factors can be obtained by

$$f(\boldsymbol{x},\boldsymbol{z}) = 546.13 - 16.00x_1 - 0.43x_2 + 174.80x_3 + 0.79z_1 - 0.63x_1{}^2 - 2.06x_3{}^2 + 0.04x_1x_2$$
$$- 21.71x_1x_3 - 0.01x_1z_1 + 0.39x_2x_3 - 0.44x_3z_1.$$

Using the Equations (8) and (9), the response functions for mean $E_z[y(\boldsymbol{x},\boldsymbol{z})]$ and the response function for variance $V_z[y(\boldsymbol{x},\boldsymbol{z})]$ associated with response COND-S can be found as follows:

$$E_z[y(\boldsymbol{x},\boldsymbol{z})] = 54613 - 16.00x_1 - 0.43x_2 + 17480x_3 - 0.63x_1{}^2 - 2.06x_3{}^2 + 0.04x_1x_2 - 21.71x_1x_3 + 0.39$$

and

$$V_z[y(\boldsymbol{x},\boldsymbol{z})] = \sigma_z{}^2(0.79 - 0.01x_1 - 0.44x_3) + \sigma^2$$

where $\sigma_z{}^2 = 154924.98$, $\sigma^2 = 23053.9$ using the results of ANOVA as shown in Table 3, and σ^2 represents the residual mean-square on the given ANOVA table.

Table 3. As shown in the RSM results, the second-order regression is significant based on the results of the global F-test and its associated p-values. The response model also has 85% R-sq, which implies the model may adequate to utilize as a response function.

Predictors	**Coef**	**SE Coef**	**T**	**P**
Constant	546.13	103.060	5.299	0.000
x_1	-16.00	15.225	-1.051	0.294
x_2	-0.43	0.493	-0.867	0.387
x_3	174.80	113.587	1.539	0.125
z_1	0.79	0.487	1.615	0.107
x_1*x_1	-0.63	0.580	-1.088	0.277
x_2*x_2	-0.00	0.000	-1.309	0.191
x_3*x_3	-2.06	19.137	-0.108	0.914
x_1*x_2	0.04	0.050	0.770	0.442
x_1*x_3	-21.71	10.301	-2.108	0.036
x_1*z_1	-0.01	0.048	-0.306	0.760
x_2*x_3	0.39	0.345	1.117	0.265
x_2*z_1	0.00	0.000	2.327	0.020
x_3*z_1	-0.44	0.344	-1.289	0.198
S = 151.8	R-Sq = 85.0%	R-Sq(adj) = 84.4%		

Analysis of Variance

Source	**DF**	**SS**	**MS**	**F**	**P**
Regression	14	47552154	3396582.4	147.33	0.000
Linear	4	46770281	79532.5	3.45	0.009
Square	4	359208	34363.2	1.49	0.204
Interaction	6	422666	70444.3	3.06	0.006
Residual Error	365	8414673	23053.9		
Total	379	55966827			

Using the proposed RDA model given in Equation (7), we can formulate optimization model as follows:

$$\begin{aligned} &\text{Minimize} && 154924.98(0.79-0.01x_1-0.44x_3)+23053.9 \\ &\text{Subject to} && 546.13-16.00x_1-0.43x_2+174.80x_3-0.63x_1^2-2.06x_3^2 \\ &&& +0.04x_1x_2-21.71x_1x_3+0.39\le\varepsilon \\ &&& \mathbf{x}\in\Omega\,. \end{aligned}$$

Using MATLAB software package [17], we obtained the optimal solution ($x_1^* = 0.5$, $x_2^* = 1153.6$, and $x_3^* = 1.8$).

4 Conclusion

In this paper, we developed a RDM method by integrating a DM method for finding significant factors into an RD method for providing the best factor settings. Based on the results of the DM method, we found important factors for water treatment process among a large set of data. Utilizing BFS method, the CFBS method in its pure form is exhaustive, but the use of a stopping criterion makes the probability of searching the whole data set quickly. We then conducted an RD optimization using RSM and ε-constrained method while incorporating an uncontrollable noise factor. We finally showed that the proposed RDM method could efficiently find significant factors and the optimal settings by reducing dimensionality through the numerical example. In order to achieve better precision of the proposed RDM method, the consideration of outliers of data using expectation maximization (EM) algorithm can be a possible further research issue.

Acknowledgement

This work was supported by Grant from Inje University 2005.

References

1. Yu, L. and Liu, H.: Feature Selection for High-Dimensional Data: A Fast Correlation-Based Filter Solution. The Proceedings of the 20th International Conference on Machine Leaning (ICML-03). Washington D.C. (2003) 856-863
2. Allen, D.: The Relationship between Variable Selection and Data Augmentation and a Method for Prediction. Technometrics. 16 (1974) 125–127
3. Langley, P.: Selection of Relevant Features in Machine Learning. Proceedings of the AAAI Fall Symposium on Relevance. AAAI Press (1994) 140-144
4. Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T.: Numerical Recipes in C. Cambridge University Press, Cambridge UK (1988)
5. Quinlan, R.R.: Induction of Decision Trees. Machine Learning. 1 (1), Hingham, MA (1986) 81-106
6. Gardner, M. and Bieker, J.: Data Mining Solves Tough Semiconductor Manufacturing Problems. Conference on Knowledge Discovery in Data Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining. New York (2000) 376-383
7. Witten, I.W.H. and Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd edn Morgan Kaufmann, San Francisco (2005)
8. Su, C.T., Chen, M.C., and Chan, H.L.: Applying Neural Network and Scatter Search to Optimize Parameter Design with Dynamic Characteristics. Journal of the Operational Research Society 56 (2005) 1132-1140
9. Seifert, J.W.: Data Mining: An Overview. CRS Report RL31798 (2004)
10. Hall, M. A.: Correlation-based Feature Selection for Machine Learning. Ph.D diss. Waikato University, Department of Computer Science. Hamilton, New Zealand (1998)

11. Box, G.E.P., Bisgaard, S., and Fung, C.: An Explanation and Critique of Taguchi's Contributions to Quality Engineering. International Journal of Reliability Management. 4 (1988) 123-131
12. Shin, S. and Cho, B.R.: Bias-specified robust design optimization and its analytical solutions. Computer & Industrial Engineering. 48 (2005) 129-140
13. Montgomery D.C.: Introduction to Statistical Quality Control. 4th edn. John Wiley & Sons, New York (2001)
14. Xu, Q., Kamel, M. and Salama, M.M.A.: Significance Test for Feature Subset Selection on Image Recognition. International Conference of Image Analysis and Recognition (ICIAR-04), LNCS, 3211 Porto, Portugal. (2004) 244-252
15. Merz, C.J. and Murphy, P.M.: UCI Repository of Machine Learning Database. http:// www. ics.uci.edu/~mlearn/MLrepository.html
16. MINITAB: http://www.minitab.com/
17. MATLAB: http://www.mathwork.com/

Pushout: A Mathematical Model of Architectural Merger

Andrew Solomon

Faculty of Information Technology, University of Technology, Sydney
andrews@it.uts.edu.au

Abstract. Although there are many formal representations of architecture, actually determining what an architecture should be when systems are merged is largely based on context and human intuition. The goal of this paper is to find a *mathematical* model which supports this context and determines the architecture when the systems have been merged. A category of architectural models is presented, and the pushout in this category provides the *unique minimal* merger of two architectures by way of an abstraction of the desired intersection. We conclude by identifying deeper aspects of architectural type which should be incorporated into this theory, and how the whole model might be automated.

1 Introduction

An Architecture Description Language (ADL) provides means for a formal model of systems' components and their connections. A model clarifies the purpose of the components and their interactions and, for the most part engineers represent these architectures using graphs [18], such as UML architectural diagrams [5].

It often happens (for example, when companies merge) that their computer systems with overlapping functionality also need to be merged with minimal duplication of functionality in the resulting system. Because the architecture of two systems merged is not a simple cut-and-paste, one hopes for formal and well-defined ways to merge system designs without errors.

1.1 Related Work

Modelling systems has been addressed at many levels from syntax, through logic based theories to abstract graph representations. At each level, maintaining the known properties of systems being merged is a core issue.

At the syntactic level, the process of merging two modifications of the same code has been dealt with extensively using the well established computation of text differences. Although the process of merging them has been semi-automated in projects such as the ArchStudio version control system [3], determining the merger is far from obvious. Niu et al. [14] have modelled the code as a graph labelled by Fuzzy logic values (which are essentially a partially ordered set) and merging the models of two programs has been implemented using the categorical pushout.

At the architectural level, merger has been attempted for systems modelled with a logic based ADL. Moriconi and Qian [13] merge two architectures by a union of their theories and provide a method for determining whether a composition of two systems is *faithful*, which is to say there is no collapse in the separation of components apart from those specified. Using logic to define the smallest architecture containing a set of properties, Caporuscio et al. [2] give a method to test whether the theories are contradictory. While this is not specifically about system merger, it could clearly be a bottom-up approach. With a logic based model of program specifications, Goguen and Burstall [10] provide a very similar categorical approach as presented below, using the colimit (a generalization of pushout) to merge several specifications. In a slight deviation from the logic approach, algebras have also been used to model software specifications [11] with merger being defined to be the pushout when certain conditions are met.

In spite of its rigour, logic does not fit comfortably in an engineer's intuitive graphical approach. The purpose of this paper is to formulate a graph based ADL as a mathematical model which accommodates some of the *context* of the systems being merged [6]. Le Metayer [12] uses a graph grammar to describe the process of adding and removing components of a system, while Baresi et al. [1] use graph homomorphisms to model architectural abstraction. Modelling architecture by graphs labelled by a poset of component types, Denford et al. [4] give an approach to refine an abstract description into a model closer to the implementation level. As with Fahmy and Holt [7,8] one of the main topics is an abstraction with only the parts of the architecture relevant to the activity at hand – in our case, the systems being merged.

1.2 Contents of This Paper

In Section 2 we illustrate an example of systems to be merged, together with the obvious intuitive solutions. Section 3 formalizes the notion of connection and component *types* and gives a mathematical representation for the relationships of their attributes. An architectural type (or as we refer to it, *archetype*) is presented in Section 4 as a graph of component and connection types. Together with the components and connections themselves, an architecture is then defined in Section 5 as a graph projected upon an archetype by graph morphism.

In Section 6 the examples will demonstrate that on this categorical basis, the merging of two architectures using pushout seems close to human intuition. Furthermore, the fact that it is a pushout means that it is the unique and smallest architecture which contains both of its sub-architectures.

In the conclusion we summarize how the aims of this paper have been achieved and identify possible approaches to issues yet to be addressed such as automation.

2 Three Examples of Merging

Suppose two companies have merged and they need their two IT systems (Figure 1) to become a single system.

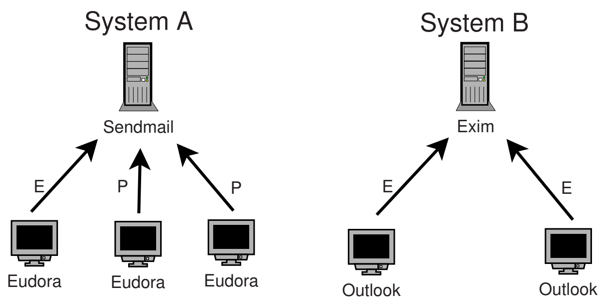


Fig. 1. Email services of two companies, with SMTP connections over Ethernet (E) or PPP (P)

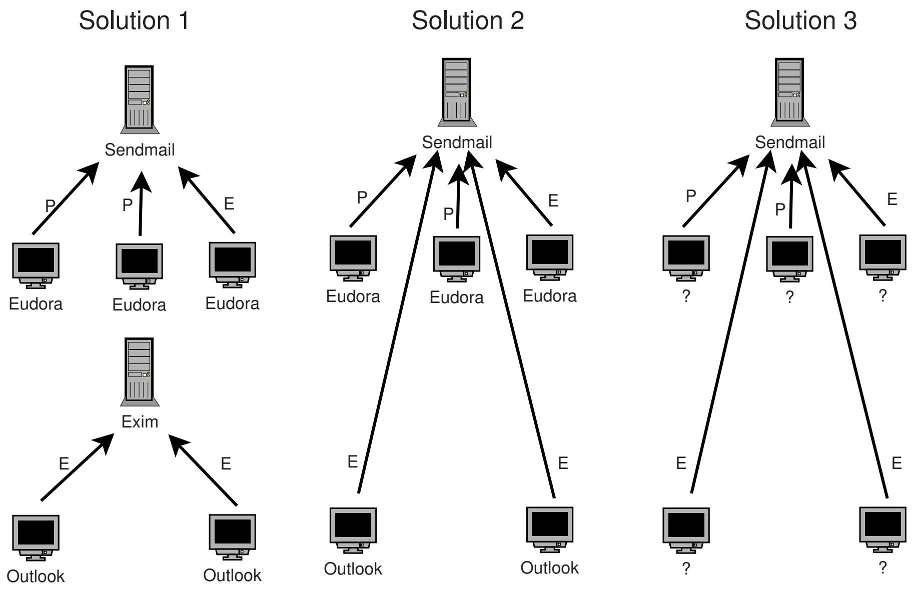


Fig. 2. Three possible email services of the new company where E is an Ethernet based SMTP connection and P is a PPP based SMTP connection

The differences between the two systems account for different activities, for example Sendmail has already been setup for dealing with email over dialup connections (PPP), while Exim has not, and to change (according to management) would be an unnecessary expense. In this sense, Sendmail has more attributes of value to the new organization.

Regarding the client side, many claim [16] that Eudora (relative to Outlook) has serious inefficiency with large folders but is better for control filters.

Generally, there are three ways the new IT team might solve this problem which are depicted in Figure 2.

The first solution is easy but may make any retrenchment of employees infeasible. The second solution is possible because the *Sendmail* server has all the necessary functionality. Having both *Eudora* and *Outlook* may keep the users happy as they both have distinct but useful features, but it would be an expense to the company to have two different client packages to buy and maintain. Therefore, the third solution would be to seek out an entirely different email client which satisfies all the staff requirements.

For the remainder of this paper it is shown that these three approaches can be formally modelled so that known system requirements are satisfied and which ensures that the erroneous models (such as replacing Sendmail with Exim) cannot occur.

3 Posets of Types

In the examples there are two sets of types: *component types* such as Exim and Sendmail, and *connection types* - SMTP over PPP (P) versus SMTP over the Ethernet (E).

There is no exact definition of these types other than the code which implements them. But rather than giving up, we propose formalizing relations of these types based on the attributes which are important to the stake-holders of the given situation.

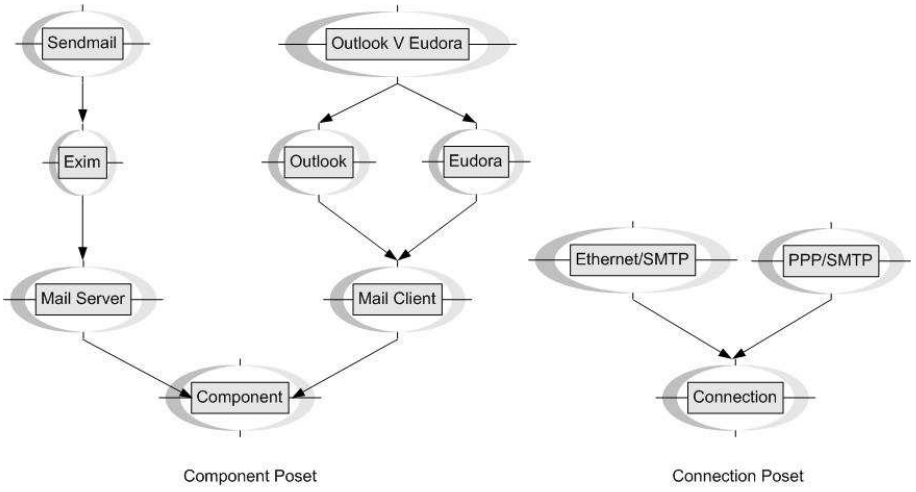


Fig. 3. Posets of component and connection types

A *partially ordered set* or *poset* is a set X together with a binary relation (which we write as $\leq$) which is reflexive (for all $x \in X$, $x \leq x$), antisymmetric (for all $x, y \in X$, $x \leq y$ and $y \leq x$ implies that $x = y$) and transitive (for all

$x, y, z \in X$, $x \leq y$ and $y \leq z$ implies that $x \leq z$). Most importantly, in a *partially* ordered set it may be the case that for some $x, y \in X$, neither $x \leq y$ nor $y \leq x$.

We can form a poset Π of the component types and a poset Λ of the connection types by the definition that $x \geq y$ if and only if x has all the attributes of y in the given situation. We depict the posets related to our example in Figure 3 using a graph with an arrow $x \rightarrow y$ meaning $x \geq y$. The basis of this work is that if $x \geq y$ in the poset of component types, a component of type y can be replaced with a component of type x. Similarly for connection types. Furthermore, writing *Outlook* $\vee$ *Eudora* we mean "some minimal application which covers both of their attributes".

4 Archetypes and Architectures

In this section we extend the notion of type from components and connections to architectures. A architecture's type (in short, an *archetype*) is simply a graph labelled by elements of the posets of component and connection types. An archetype is not in itself an architecture, but merely the description of the component and connection types which exist in an architecture. For example, the archetypes of the architectures depicted in Figure 2 are given in Figure 4.

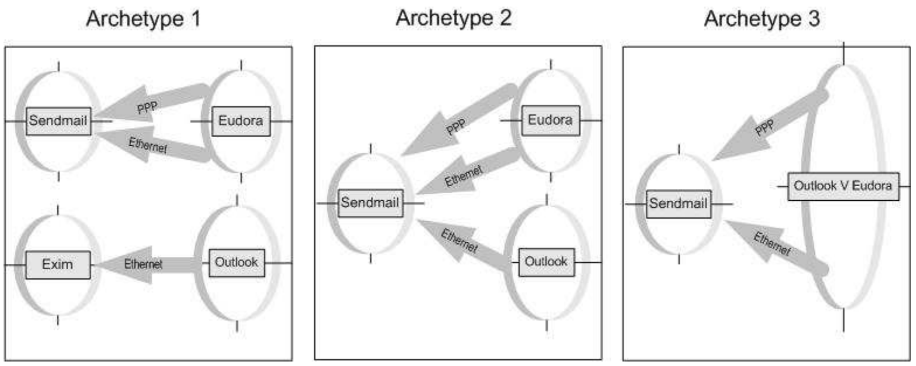


Fig. 4. Archetypes of Solutions 1, 2 and 3 in Figure 2

Now, we show an *architecture* to be a graph morphism from an unlabelled graph (the actual components and their connections) to the graph of its archetype, which defines the types. The following section gives a concise formalization of these notions, and their consequences through category theory. Thenceforth, we are able to show the main result - a formal model of merging architectures.

5 A Category of Architectures

As introduced in Section 3, Π is a poset of component types and Λ is a poset of connection types. An *archetype* (also known as a *poset labelled graph*) G is a tuple $(V_G, E_G, s_G, t_G, \pi_G, \lambda_G)$ where V_G and E_G are sets of *components* and *connections* respectively; $s_G, t_G : E_G \to V_G$ define the *source* and *target* of a connection; and $\pi_G : V_G \to \Pi$ and $\lambda_G : E_G \to \Lambda$ are the types of the components and connections.

A *morphism* $\phi : G \to H$ of archetypes is a pair $(\phi_V : V_G \to V_H, \phi_E : E_G \to E_H)$ such that for all $e \in E_G$: $s_H(\phi_E(e)) = \phi_V(s_G(e))$ and $t_H(\phi_E(e)) = \phi_V(t_G(e))$ meaning ϕ preserves the structure of the graphs; and such that for all $x \in V_G, e \in E_G$, ϕ has *lax* preservation of the types, that is:

$$\pi_H(\phi_V(x)) \geq \pi_G(x) \text{ and } \lambda_H(\phi_E(e)) \geq \lambda_G(e) \tag{1}$$

– a component (connection) of one type can only be mapped to a component (connection) of greater or equal type as per the posets of types. In case the mappings in Equation 1 are all equalities, we say that ϕ is *strict*.

It is easy to see that the composition of morphisms, as pairs of functions, is another morphism. Associativity and identity are inherited from the category of sets and functions. Therefore, with posets Π and Λ fixed as above, archetypes (poset labelled graphs) and their morphisms form a category which we denote by $\mathsf{Graph}_{\Pi,\Lambda}$.

5.1 Formalization of Architecture

Let $U : \mathsf{Graph}_{\Pi,\Lambda} \to \mathsf{Graph}$ be the functor which forgets the labelling of an archetype, and consider the comma category Graph/U. An object of Graph/U is a pair $(T, X : G \to UT)$ (often simply written as X) we call an *architecture*. The architecture X consists of a typed graph T, called the *archetype*, and the graph G, called the *component* graph, equipped with the graph morphism X from G to the underlying ordinary graph of T. An arrow (*architectural morphism*) is a pair (f, t) such that the following diagram commutes

$$\begin{tikzcd}
G \arrow[r, "f"] \arrow[d, "X"'] & H \arrow[d, "Y"] \\
UT \arrow[r, "U(t)"] & UT'
\end{tikzcd}$$

Definition 1. *Given a category* $\mathsf{Graph}_{\Pi,\Lambda}$ *of archetypes and the functor* $U : \mathsf{Graph}_{\Pi,\Lambda} \to \mathsf{Graph}$*, then the comma category* Graph/U *is called a* category of architectures.

Informally identifying T and UT, for any component v of G define the component $X(v)$ of T to be the *archetype* of v while $\pi(X(v)) \in \Pi$ is the *type* of v. Define archetype and type similarly for connections.

Let $c : \mathsf{Graph}/U \to \mathsf{Graph}$ map an architecture to its graph of *components and connections*, and let $a : \mathsf{Graph}/U \to \mathsf{Graph}_{\Pi,\Lambda}$ map an architecture to its *archetype*. It can be shown that there is a natural transformation $l : c \to Ua$.

Figure 5 illustrates the way that objects of Graph/U represent architectures. The top part of the diagram is the object in Graph and the bottom part is its archetype in $\mathsf{Graph}_{\Pi,\Lambda}$.

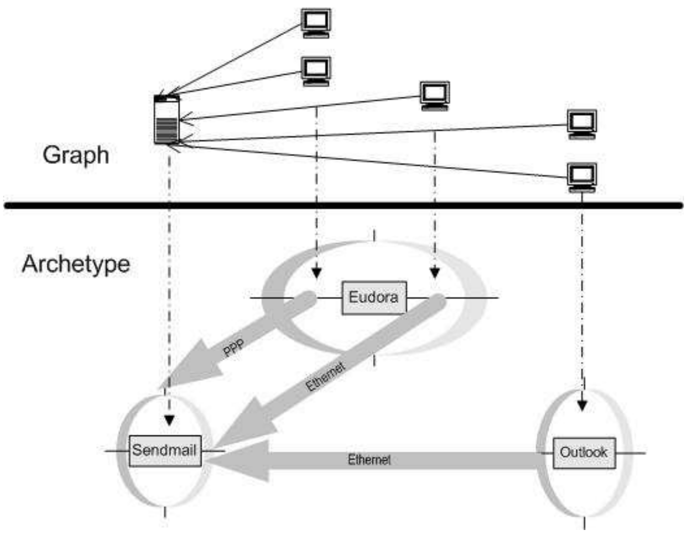


Fig. 5. An architecture – an object of Graph/U

6 Merging Architectures by Pushout

As a quick reminder before launching into the pushout of Graph/U consider the first example of pushout in most textbooks – the union of sets.

For example, let $X = \{1, 2, 3\}$ and $Y = \{2, 4, 6\}$ then the union of these two sets, denoted $X \cup Y$ is equal to $\{1, 2, 3, 4, 6\}$. This is, the unique smallest set containing the elements of both X and Y. One can express this category theoretically by saying, let $f : Z \to X$ and $g : Z \to Y$ be injective functions, which define Z as the *intersection* of X and Y. Then the *pushout* is an object P together with a pair of arrows $i_1 : X \to P$ and $i_2 : Y \to P$ such that the inner square of Figure 6 *commutes* (that is, $i_1 f = i_2 g$), and furthermore, that given any other diagram (j_1, j_2, Q) there is a unique arrow from P to Q making the whole diagram commute. (In the example of sets, this unique arrow is simply indicating that any set which contains both X and Y contains $\{1, 2, 3, 4, 6\}$, but could be larger.)

Therefore, the pushout is a formal construct that ensures the containment is complete, unique (up to isomorphism) and minimal. In the remainder of this section we present the mathematical details of the pushout for architectures.

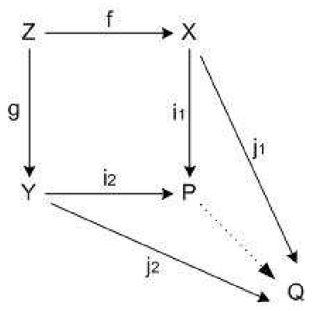


Fig. 6. Pushout diagram

It is a straightforward consequence of [15, Lemma 3.9] that

Proposition 1. $\mathsf{Graph}_{\Pi,\Lambda}$ *has pushouts along strict monomorphisms.*

and more generally

Theorem 1. *In* Graph/U *there are pushouts along arrows* η *if its archetype part* η_a *is a strict monomorphism.*

However in trying to model merger, it is not helpful to demand that η_a be strict, so instead we assert that Π and Λ have least upper bounds. It can be shown that

Theorem 2. *If* Π *and* Λ *have least upper bounds, then* $\mathsf{Graph}_{\Pi,\Lambda}$ *has pushouts (and, in fact, all its colimits) and the architecture category* Graph/U *has all pushouts (and all its colimits).*

Note that although real-world component and connection types will not generally have least upper bounds, the join operation (for example *Outlook* $\vee$ *Eudora* of Figure 3) produces any necessary types which can then be analysed to determine how they will be implemented in practice. Therefore we can now formalize what it means to merge architectures:

Definition 2. *The* merger *of architectures* X *and* Y *over the archetype* Z *with monomorphisms* $X \overset{i}{\leftarrow} Z \overset{j}{\rightarrow} Y$ *is the pushout of* X *and* Y *over* Z*.*

In the next section we use this framework to identify two subsystems of the merging systems which are similar enough that a single subsystem can replace them both in the architecture of the merge.

6.1 Three Examples of Merging (Again)

It should now be clear that the examples in Section 2 are each a pushout in the category of architectures. Solution 1 (see Figure 2) comes about when the intersection architecture is empty. Solution 2 is the pushout when the intersection architecture is a single Email Server. The archetype pushout ensures that as this maps to both Sendmail server and an Exim server, the result is a Sendmail server. The most difficult example (Archetype 3 in Figure 4) is illustrated in Figure 7 where a least upper bound appears as the join of *Outlook* and *Eudora*, alerting one to the decision which needs to be made on the new type of these components.

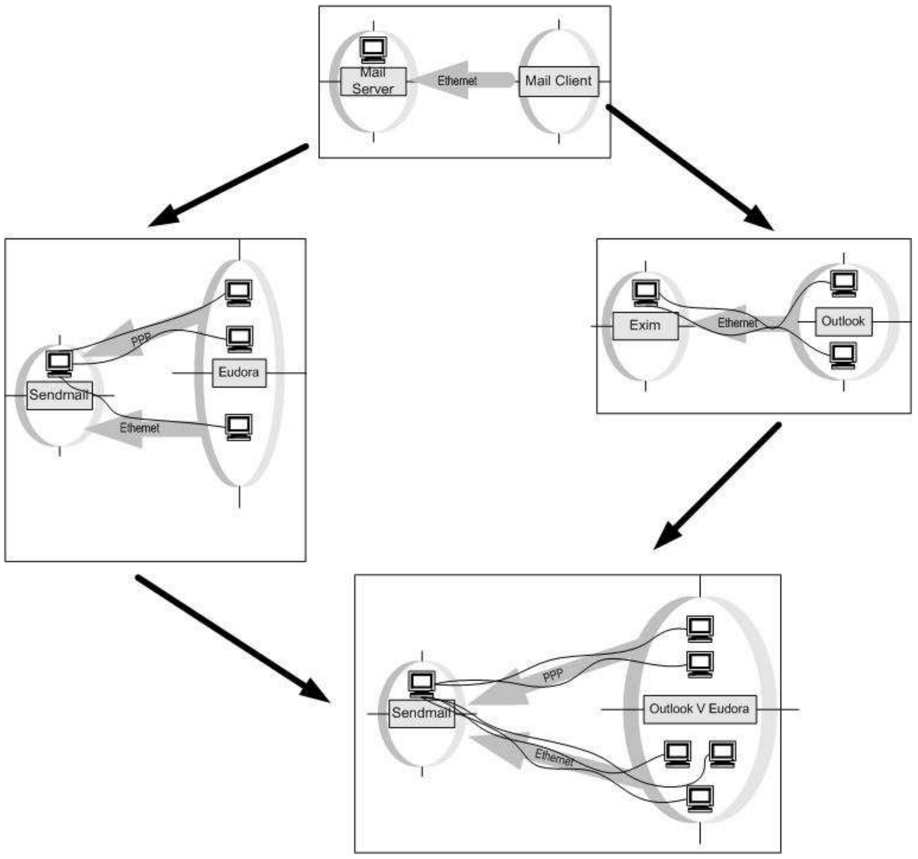


Fig. 7. 'Top-down' view of a pushout diagram in Graph/U, where an architecture is encapsulated in a box. The component graph (the terminal icons) sit above the archetypes (circular objects) which define their types. The pushout results in Solution 3 in Figure 2. Notice that if this diagram were from $\mathsf{Graph}_{\Pi,\Lambda}$ it would not be a pushout and therefore not have the useful properties of being minimal and unique.

7 Conclusion

The goal of this paper was to find a mathematical model of architecture which is intuitive, yet rigorous when determining architectural merger. By "intuitive" it is meant to be close to the way in which engineers think about architectures and this has been done by using graph-like representations.

The rigour required for merging two systems is to maintain the structure of both systems while avoiding any unnecessary duplication. This was achieved by formalizing the contextual and intuitive definition of types and implementing the merger as a categorical pushout. At the core of this process is identifying the intersection of the systems as different parts which, by abstraction, are the same.

Although this is a well controlled model of types, there are many sets of attributes which apply to an archetype - and not merely the sum of the attributes of its parts. For instance, there are several properties determining *architectural style* [9] such as ensuring that a particular type of connection has only one server to many clients. It may therefore be helpful to move from types to a single (infinite) poset of archetypes (together with some constraint on the comma category) creating a more detailed definition of architecture.

An interesting project would be to modify a graph transformation based program such as PROGRES [8] or AGG [17] to incorporate architectures as defined in this paper and automate the pushout once the intersection is chosen by the user.

References

1. L. Baresi, R. Heckel, S. Thöne and D. Varrò, *Style-Based Refinement of Dynamic Software Architectures*, In Proc. WICSA 2004: 4th Working International IEEE/IFIP Conference on Software Architecture Page(s) 155-164, Publisher: IEEE Computer Society.
2. Mauro Caporuscio, Paola Inverardi and Patrizio Pelliccione, *Formal Analysis of Architectural Patterns*, EWSA 2004, LNCS 3047, pp. 10–24, 2004.
3. Eric M. Dashofy, André van der Hoek, Richard N. Taylor, *Towards architecture-based self-healing systems* Proceedings of the first workshop on Self-healing systems Charleston, South Carolina 21 – 26 (2002) ISBN:1-58113-609-9
4. Mark Denford, Andrew Solomon, John Leaney and Tim O'Neill, *Architectural Abstraction as Transformation of Poset Labelled Graphs*, Journal of Universal Computer Science, Special Issue on Formal Specification of Computer Based Systems, Journal of Universal Computer Science, vol. 10, no. 10 (2004), 1408-1428.
5. Martin Fowler and Kendall Scott, "UML Distilled", Second Edition, Addison Wesley Longman, Inc. 2000.
6. R. Kazman, G. Abowd, L. Bass, and P. Clements, *Scenario-Based Analysis of Software Architecture*, IEEE Software, pp.47-55, November 1996.
7. Hoda Fahmy and Richard C. Holt, *Software Architecture Transformations*, ICSM 2000: International Conference on Software Maintenance, San Jose, CA, Oct 2000.
8. Hoda Fahmy and Richard C. Holt, *Using Graph Rewriting to Specify Software Architectural Transformations*, p.187, 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.
9. David Garlan, Robert T. Monroe, and David Wile, *Acme: Architectural Description of Component-Based Systems* Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000, pp. 47–68.
10. Joseph A. Goguen and Rod M. Burstall, *Institutions: abstract model theory for specification and programming*, Journal of the ACM, vol. 39, No. 1, January 1992, pp. 95 – 146.

11. A. E. Haxthausen and F. Nickl, *Pushouts of Order-Sorted Algebraic Specifications*, In Algebraic Methodology and Software Technology (AMAST'96), number 1101 in Lecture Notes in Computer Science, pages 132–148. Springer, 1996.
12. Daniel Le Métayer, *Describing Software Architecture Styles Using Graph Grammars*, IEEE Transactions on Software Engineering, vol. 24, No. 7, July 1998.
13. Mark Moriconi and Xiaolei Qian, *Correctness and Composition of Software Architectures*, Proceedings, ACM SIGSOFT'94, Symposium on Foundations of Software Engineering, New Orleans, Louisiana, December, 1994, pp. 164-174.
14. Nan Niu, Steve Easterbrook, Mehrdad Sabetzadeh, *A Category-theoretic Approach to Syntactic Software Merging*, 21st IEEE International Conference on Software Maintenance, pp. 197-206.
15. F. Parisi-Presicce, H. Ehrig and U. Montanari, *Graph rewriting with unification and composition*, in 'Graph grammars and their application to computer science', Lecture Notes in Computer Science (1986), 496–514.
16. Google Groups Discussion Forum, *comp.mail.eudora.ms-windows.*
17. The Attributed Graph Grammar System, `http://tfs.cs.tu-berlin.de/agg/index.html`, Last Modified: 30 January 2006.
18. Carnegie Mellon University, Software Engineering Institute, `http://www.sei.cmu.edu/architecture/adl.html`, Last Modified: 31 August 2005.

A Formal Model of Data Privacy

Phiniki Stouppa and Thomas Studer

Institut für Informatik und angewandte Mathematik,
Universität Bern, Neubrückstrasse 10, CH-3012 Bern, Switzerland
{stouppa,tstuder}@iam.unibe.ch

Abstract. Information systems support data privacy by constraining user's access to public views and thereby hiding the non-public underlying data. The privacy problem is to prove that none of the private data can be inferred from the information which is made public. We present a formal definition of the privacy problem which is based on the notion of certain answer. Then we investigate the privacy problem in the contexts of relational databases and ontology based information systems.

1 Introduction

The development of automatic information processing has made it necessary to consider privacy protection in relation to personal data. The surveillance potential of powerful computer systems demands for rules governing the collection and sharing of personal information. An overview of the evolution of data protection is presented in [20].

Two of the main international instruments in this context are the Council of Europe's 1981 Convention for Protection of Individuals with regard to Automatic Processing of Personal Data [8] and the Organisation for Economic Cooperation and Development (OECD) Guidelines on the Protection of Privacy and Transborder Flows of Personal Data [19]. These rules describe *personal data* as any information relating to an identified or identifiable individual.

The expression of data protection in various declarations and laws varies. However, all require that personal data must be kept secure. That includes appropriate security measures for the protection of personal data stored in information systems against unauthorized access. Thus, information systems must take responsibility for the data they manage [1]. The main challenge in data privacy is to share some data while protecting personal information.

We provide a theoretical framework to prove that under certain circumstances none of the personal data can be inferred from the information which is made public. The underlying system is given in the form of an *ontology*. Personal data takes the form of a *privacy condition* which is a set of queries. Moreover, the public information is given in terms of a *view instance* and *background knowledge*. A view instance consists of queries and their (actual) answers, while background knowledge includes additional facts about the system that are provided for better understanding of the data in the views. The *privacy problem* is then to decide whether any of the queries in the private condition can be inferred from the view instance and the background knowledge.

In order to state the privacy problem, we employ the notion of *certain answer*: data privacy is preserved for a query with respect to the provided public knowledge if there are no non-negative certain answers of the query with respect to that knowledge. That is, if the certain answer to it is either the empty set or negative ("None" or "No"). The certain answers of a query are those answers that are returned by the query in every 'possible' instance. The problem of answering queries against a set of 'possible' instances was first encountered in the context of incomplete databases [24]. Today, certain answer is a key notion in the theory of data integration [7,14,16] and data exchange [2,13].

Let us demonstrate the above setting: consider an ontology that contains information about the customers of a telecommunication company. The company provides information to the end-users through searching engines on its telephone lists, whereas at the same time some of its customers do not wish to give in publicity their telephone numbers. Thus, the privacy condition would be a set of queries of the form $\mathsf{Owns}(\mathsf{cust}_i, \mathsf{Tel})$, where Owns relates customers to their telephone numbers, cust_i is a constant and Tel is a variable. Since these are retrieval queries, data privacy is preserved when there is no certain answer to each of them. That is, there is no telephone number which is returned by such a query in every 'possible' ontology. If this holds, then the set of certain answers is empty which means that no telephone number of any of cust_is is exhibited. Negative answers might occur only in the case of boolean queries that are not applicable on the ontology, when this is also announced through the public information.

Our work is concerned with the question how much information a given view instance reveals and whether it leaks private data. In [5] the same question is addressed for a variety of confidentiality policies. Following their setting, the privacy problem we deal with corresponds to what is called 'uniform refusal for unknown potential secrets'. Their work is however limited to boolean queries in complete information systems. Moreover, much of the existing work on privacy for information systems deals with privacy preserving query answering. There, the privacy problem is that of inferring a maximal subset of the answer to a query so that no secrets are violated [6,25]. The idea of specifying sensitive information as conjunctive query is pioneered in [18], where the notion of perfect privacy is introduced. However, enforcing perfect privacy for conjunctive queries is highly intractable. A generalization of this model has been studied in [11]. There, checking perfect privacy is even harder. Recently, Machanavajjhala and Gehrke [17] make a connection between perfect privacy and the problem of checking query containment. This allows them to identify many subclasses of conjunctive queries for which enforcing perfect privacy is tractable. Dix et al. [12] established a relationship between privacy problems and non-monotonic logics. Another approach [22] is to generalize the answers to a query in order to provide anonymity.

The rest of the paper is organized as follows: first, we give formal definitions for both the ontology and query answering on it. We define the ontology as a set of first-order sentences, while query answering is done via entailment. This allows for the application of data privacy in both knowledge base and database systems. Thus, the present definition of data privacy is much more general than the one given in [23] which applies to relational databases only. Then, we present a formal model of data privacy using certain answers and show that these can be reduced to logical entailment. Thus, in general, the privacy problem is not decidable. We continue by presenting two applications where the data privacy problem is decidable: in Section 4 we apply data privacy on relational databases with conjunctive queries. In this case, background knowledge consists of a relational schema with constraints imposed on it. Data privacy for this setting is decidable in polynomial time. In Section 5 we apply data privacy on $\mathcal{ALC}$ description logic-based ontologies. In this case, background knowledge might include any TBox or ABox entries. Here, the complexity of data privacy follows the complexity of $\mathcal{ALC}$-reasoning: it is ExpTime-complete for ontologies with a general TBox and PSpace-complete for ontologies with an acyclic TBox. Finally, we summarize the results and give further research directions.

2 The Ontology and Query Answering

We define the *relational first-order language* $\mathcal{L}$ as follows. The collection of $\mathcal{L}$ terms comprises countably many *variables* $x, y, z, \ldots$ and countably many *constant symbols* $a, b, c, \ldots$. We use Const for the set of $\mathcal{L}$ constants. $\mathcal{L}$ includes for every natural number n countably many *relation symbols* $R, S, T, \ldots$ of arity n as well as the binary relation symbol $=$ for equality. If R is an n-ary relation symbol of $\mathcal{L}$ and $t_1, \ldots, t_n$ are $\mathcal{L}$ terms, then $R(t_1, \ldots, t_n)$ is an *atomic $\mathcal{L}$ formula*. $\mathcal{L}$ *formulae* are built up inductively from the atomic formulae of $\mathcal{L}$ by closing under the usual connectives as well as universal and existential quantification. We call an $\mathcal{L}$ formula without free variables $\mathcal{L}$ *sentence*.

We will also make use of the standard notion of *logical entailment*: Let ϕ be a formula and $\mathcal{O}$ a set of formulae. Then $\mathcal{O} \models \phi$ if every model of $\mathcal{O}$ is also a model of ϕ.

Note that the choice of a first-order language for the current presentation is not important. We could as well use any other language that is employed in the context of information systems, such as second order languages or fixed point logics. Now, we can formally introduce the ontology and show how query answering can be defined in terms of entailment:

Definition 1. *An* ontology $\mathcal{O}$ *is a finite set of $\mathcal{L}$ sentences.* $\mathsf{Const}(\mathcal{O})$ *denotes the set of constants that occur in $\mathcal{O}$. A* query q *is an $\mathcal{L}$ formula. If q has no free variables, then q is called* boolean query *otherwise it is a* retrieval query*.*

Definition 2. *The* range of a query q ($\mathsf{range}(q)$) *is given by:*

1. $\{\emptyset, \{\top\}, \{\bot\}\}$ *if q is a sentence,*
2. $\mathsf{Pow}(\mathsf{Const}^n)$ *which is the power set of the n times Cartesian product of* Const *with itself, if q is a formula with $n > 0$ free variables.*

Definition 3. *The* answer to a query q with respect to an ontology $\mathcal{O}$ ($\mathsf{ans}(q, \mathcal{O})$) *is given by:*

$$\begin{aligned}
&\mathsf{ans}(q, \mathcal{O}) := \{\top\} \textit{ if } q \textit{ is a sentence and } \mathcal{O} \models q,\\
&\mathsf{ans}(q, \mathcal{O}) := \{\bot\} \textit{ if } q \textit{ is a sentence, } \mathcal{O} \not\models q \textit{ and } \mathcal{O} \models \neg q,\\
&\mathsf{ans}(q, \mathcal{O}) := \emptyset \textit{ if } q \textit{ is a sentence, } \mathcal{O} \not\models q \textit{ and } \mathcal{O} \not\models \neg q,\\
&\mathsf{ans}(q, \mathcal{O}) := \{\boldsymbol{t} \in \mathsf{Const}(\mathcal{O})^n \mid \mathcal{O} \models q(\boldsymbol{t})\} \textit{ if } q \textit{ has } n > 0 \textit{ free variables.}
\end{aligned}$$

Note that $\mathsf{ans}(q, \mathcal{O}) \in \mathsf{range}(q)$ and is always finite. Finally, a view instance is a set of queries together with their answers:

Definition 4. *A* view instance V_I *is a finite set of tuples* $\langle q_i, r_i \rangle$ *where each* q_i *is a query and* $r_i \in \mathsf{range}(q_i)$*. We say that an* ontology $\mathcal{O}$ entails a view instance V_I *(in symbols* $\mathcal{O} \models V_I$*) if* $r_i = \mathsf{ans}(q_i, \mathcal{O})$ *for every* $\langle q_i, r_i \rangle \in V_I$*.*

3 Data Privacy

As mentioned in the introduction, in addition to the view instance V_I that is provided, public knowledge also includes some other facts, the background knowledge. We will refer to it as the ontology $\mathcal{O}$. We call the tuple $\langle \mathcal{O}, V_I \rangle$ a *data privacy setting.* Also, since querying an ontology makes sense only when the answers it provides do actually hold, we assume that the underlying ontology is consistent.

We give a definition of the problem based on the notion of certain answer: let q be the information we wish to keep private. First, we collect all those ontologies each of which is conceivably the underlying ontology. Afterwards, we collect those answers to q that do *certainly* hold in each of the collected ontologies. A non-negative answer would then mean that q is exhibited and thus, data privacy is not preserved.

Definition 5. *Let* $\langle \mathcal{O}, V_I \rangle$ *be a data privacy setting. We call an ontology* $\mathcal{P}$ possible *with respect to* $\langle \mathcal{O}, V_I \rangle$ *if*

1. *$\mathcal{P}$ is consistent,*
2. *$\mathcal{O} \subseteq \mathcal{P}$, and*
3. *$\mathcal{P} \models V_I$.*

$\mathsf{Poss}_{\langle \mathcal{O}, V_I \rangle}$ *denotes the set of all possible ontologies with respect to* $\langle \mathcal{O}, V_I \rangle$*.*

Definition 6. *The* certain answers *to a query q with respect to a setting* $\langle \mathcal{O}, V_I \rangle$ *are defined by*

$$\mathsf{certain}(q, \langle \mathcal{O}, V_I \rangle) := \bigcap_{\mathcal{P} \in \mathsf{Poss}_{\langle \mathcal{O}, V_I \rangle}} \mathsf{ans}(q, \mathcal{P})$$

Definition 7. *We say* data privacy is preserved for q with respect to $\langle \mathcal{O}, V_I \rangle$ *if* $\mathsf{certain}(q, \langle \mathcal{O}, V_I \rangle) \subseteq \{\bot\}$.

The proposed definition has the advantage that works independently of the underlying language. However, it does not provide a direct solution to the problem as the possible ontologies are infinitely many. For this reason, we first construct a so-called *canonical* ontology that carries minimal, though complete, information about the certain answers to a given query.

Definition 8. *Given a setting $\langle \mathcal{O}, V_I \rangle$, the canonical ontology $\mathcal{C}_{\langle \mathcal{O}, V_I \rangle}$ is defined as*

$$\begin{aligned}\mathcal{C}_{\langle \mathcal{O}, V_I \rangle} := \mathcal{O} \cup & \\ & \{q \mid \langle q, \{\top\}\rangle \in V_I\} \cup \\ & \{\neg q \mid \langle q, \{\bot\}\rangle \in V_I\} \cup \\ & \{q(\boldsymbol{t}) \mid \textit{there is an } A \textit{ with } \langle q, A\rangle \in V_I \textit{ and } \boldsymbol{t} \in A\}\end{aligned}$$

Note that this construction is language-dependent. The following theorem can be easily shown:

Theorem 1. *Given an $\mathcal{L}$ formula ϕ and a data privacy setting $\langle \mathcal{O}, V_I \rangle$, the following holds:*

$$\mathcal{C}_{\langle \mathcal{O}, V_I \rangle} \models \phi \textit{ if and only if } \forall \mathcal{P}.(\mathcal{P} \in \mathsf{Poss}_{\langle \mathcal{O}, V_I \rangle} \rightarrow \mathcal{P} \models \phi).$$

In order to check whether data privacy is preserved for a query q with respect to $\langle \mathcal{O}, V_I \rangle$, we can build the canonical ontology $\mathcal{C}_{\langle \mathcal{O}, V_I \rangle}$ and issue q to it.

Corollary 1. *Data privacy is preserved for q with respect to $\langle \mathcal{O}, V_I \rangle$ if and only if $\mathsf{ans}(q, \mathcal{C}_{\langle \mathcal{O}, V_I \rangle}) \subseteq \{\bot\}$.*

4 Relational Databases

In this section we show that there is a polynomial time solution to the privacy problem for relational databases. Although classical database theory is concerned with model checking, we can make use of Reiter's proof theoretic approach [21] in order to apply our setting to relational databases.

In the context of relational databases, we consider only conjunctive queries.

Definition 9. *An $\mathcal{L}$ formula is called* conjunctive query *if it is built from atomic formulae, conjunctions and existential quantifiers. A* conjunctive view instance *V_I is a view instance such that q_i is a conjunctive query for each $\langle q_i, r_i \rangle \in V_I$.*

Definition 10. *A* data privacy setting for databases *$\langle \mathcal{O}, V_I \rangle$ consists of*

1. *a set of dependencies $\mathcal{O}$. Each element of $\mathcal{O}$ is either a tuple generating dependency [4] of the form*

$$\forall \boldsymbol{x}(\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y} \psi(\boldsymbol{x}, \boldsymbol{y}))$$

or an equality generating dependency [4] of the form

$$\forall \boldsymbol{x}(\phi(\boldsymbol{x}) \to (x_1 = x_2)),$$

where $\phi(\boldsymbol{x})$ and $\psi(\boldsymbol{x}, \boldsymbol{y})$ are conjunctions of atomic formulae and x_1, x_2 are among the variables of $\boldsymbol{x}$,

2. *a conjunctive view instance V_I.*

It is possible to translate the data privacy setting for databases to a data exchange setting [23]. Fagin et al. [13] show that in such a setting, the classical chase can be used to compute certain answers for conjunctive queries. The procedure they present terminates in polynomial time.

Theorem 2. *Given a data privacy setting for databases $\langle \mathcal{O}, V_I \rangle$ and a conjunctive query q. Then we can check in polynomial time whether privacy is preserved for q with respect to $\langle \mathcal{O}, V_I \rangle$.*

5 $\mathcal{ALC}$-Based Ontologies

Description logics build the mathematical core of many modern knowledge base systems [3]. Their language consists of *concepts* (sets of individuals) and *roles* (binary relationships between the individuals).

The basic description logic $\mathcal{ALC}$ consists of the following concepts:

$$C := A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \forall R.C \mid \exists R.C,$$

where A is an atomic concept and R is a role. Each concept C abbreviates an $\mathcal{L}$ formula $C'(x)$ with one free variable x as follows.

$$\begin{aligned}
A'(x) &:= A(x)\\
(\neg C)'(x) &:= \neg C'(x)\\
(C_1 \sqcap C_2)'(x) &:= C_1'(x) \land C_2'(x)\\
(C_1 \sqcup C_2)'(x) &:= C_1'(x) \lor C_2'(x)\\
(\forall R.C)'(x) &:= \forall y.(R(x,y) \to C'(y))\\
(\exists R.C)'(x) &:= \exists y.(R(x,y) \land C'(y))
\end{aligned}$$

In the sequel, we will identify concepts and the corresponding $\mathcal{L}$ formulae. An ontology contains a terminology, that is the vocabulary of an application domain, as well as assertions about named individuals in terms of the vocabulary. The *terminology* consists of concept equality axioms of the form $C_1 \equiv C_2$ abbreviating $\forall x.(C_1(x) \leftrightarrow C_2(x))$. An *assertion* is a formula of the form $C(a)$ or $R(a, b)$ where $a, b \in \mathsf{Const}$ are called *individuals*. An *$\mathcal{ALC}$-based ontology* consists of a terminology (called TBox) and a set of assertions (called ABox).

A TBox is *acyclic* when it satisfies the following: (i) every concept equality is of the form $A \equiv C$, (ii) every atomic formula occurs at most once at the left hand side of an equality and (iii) there are no cycles in the concept equality axioms.

An $\mathcal{ALC}$ *query* is either a concept (retrieval query) or an expression of the form $C(a)$ or $C_1 \equiv C_2$ (boolean query).[1] A setting $\langle \mathcal{O}, V_I \rangle$ is a data privacy setting for $\mathcal{ALC}$-based ontologies if $\mathcal{O}$ is an $\mathcal{ALC}$-based ontology and V_I is given by $\mathcal{ALC}$ queries. For the rest of this section, *query* refers to $\mathcal{ALC}$ query.

The data privacy problem in this setting can be solved following the approach presented in the general setting, that is, by building a canonical ontology that corresponds to the public knowledge $\langle \mathcal{O}, V_I \rangle$. In its current form, the ontology defined in Definition 8 is not an $\mathcal{ALC}$-based ontology, since a negative answer on an equality query $C_1 \equiv C_2$ would include a non-$\mathcal{ALC}$ formula. What actually a negative answer tells about the ontology in this case, is that there is an individual which belongs to C_1 and does not belong to C_2 or vice versa. Thus, we can unfold the view instance by replacing every $\langle C_1 \equiv C_2, \{\bot\} \rangle$ in V_I by $\langle (C_1 \sqcap \neg C_2) \sqcup (\neg C_1 \sqcap C_2)(d), \{\top\} \rangle$, where d is fresh (that is it does not occur in $\langle \mathcal{O}, V_I \rangle$ or in the private query q). We can now construct the canonical ontology based on this unfolded view instance.

Similarly to Theorem 1, it can be shown that the constructed ontology is indeed canonical with respect to the public knowledge. Finally, under this framework, the complexity results for the reasoning problem in $\mathcal{ALC}$-based ontologies [3] apply also to the privacy problem.

Theorem 3. *Given a data privacy setting $\langle \mathcal{O}, V_I \rangle$ for $\mathcal{ALC}$-based ontologies and a query q, the data privacy problem for q with respect to $\langle \mathcal{O}, V_I \rangle$ is* ExpTime-*complete when the* TBox *in $\langle \mathcal{O}, V_I \rangle$ is general and* PSpace-*complete when it is acyclic.*

Note that in the context of description logic ontologies, our approach is not restricted to $\mathcal{ALC}$. We can use the same method also to solve the data privacy problem for ontologies which are given in very expressive description logics. For instance, our technique also applies to logics such as $\mathcal{SHIF}$ and $\mathcal{SHOIN}$ which are the mathematical models for the web ontology languages OWL Lite and OWL DL.

However, if the query language is different from the ontology language, then Definition 8 is not applicable. For instance, if we have a description logic based ontology language and use conjunctive queries to retrieve information, then we need other techniques to solve the privacy problem.

6 Conclusion and Outlook

We have given a formal definition of the general data privacy problem for information systems. This problem is to check whether a given view instance leaks information about the underlying data or knowledge base. We have modeled the privacy problem using the notion of certain answer. Privacy holds for a query q

[1] The problems of querying a concept assertion and querying an equality are known as the instance and equivalence problems, respectively. The well-known subsumption problem is reduced to the equivalence problem.

with respect to a view instance V_I if there are no non-negative certain answers to q with respect to V_I.

Computing certain answers is equivalent to logical entailment. Thus it is in general undecidable. We have investigated two important decidable cases: the privacy problem for relational databases with a set of constraints and the privacy problem for ontology (description logic) based information systems.

We plan to extend our study to other data models. The investigation of the privacy problem for XML databases is an important further task. Like relational databases, XML databases protect data from unauthorized access by allowing users to issue queries solely to views that provide public information only [10]. The computation of certain answers in XML databases has been studied for instance in [2].

Another direction of future work is to investigate the effect of updates to data privacy. Assume we have a query and a view instance for which privacy holds. If we update the underlying database or ontology, can we be sure that privacy still is preserved? Thus, it is important to study privacy preserving updates. That is, the question of which forms of updates do not violate data privacy.

The present definition of the privacy problem consists of deciding whether a given view instance leaks information. There is a second privacy problem: deciding whether already the view definition guarantees that there is no possible leaking. That means, given the view definition, there cannot be a view instance that leaks private information. For example, this is the case in relational databases if values stored in private attributes cannot be inferred via the constraints defined in the database. In ontology based systems, the theory of $\mathcal{E}$-connections [15] and partitioning of ontologies [9] may lead to such secure view definitions. Finally, the study of this second privacy problem will result in a collection of database patterns which are safe with respect to data privacy.

Acknowledgments

We would like to thank Sebastian Link for bringing our attention to the privacy problem for XML databases.

References

1. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. of 28th VLDB Conference*, 2002.
2. M. Arenas and L. Libkin. XML data exchange: Consistency and query answering. In *PODS*, pages 13–24, 2005.
3. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
4. C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.
5. J. Biskup and P. A. Bonatti. Controlled query evaluation for enforcing confidentiality in complete information systems. *International Journal of Information Security*, 3(1):14–27, 2004.

6. P. A. Bonatti, S. Kraus, and V. s. Subrahmanian. Foundations of secure deductive databases. *Transactions on Knowledge and Data Engineering*, 7(3):406–422, 1995.
7. A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *Proc. of CAiSE 2002*, volume 2348 of *LNCS*, pages 262–279. Springer, 2002.
8. Council of Europe. Convention for the protection of individuals with regard to automatic processing of personal data, 1981. Available at `http://conventions.coe.int/Treaty/en/Treaties/Html/108.htm`.
9. B. Cuenca Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Automated partitioning of owl ontologies using e-connections. In *Proceedings of Int. Workshop on Description Logics*, 2005.
10. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Controlling access to XML documents. *IEEE Internet Computing*, 5(6):18–28, 2001.
11. A. Deutsch and Y. Papakonstantinou. Privacy in database publishing. In *ICDT*, 2005.
12. J. Dix, W. Faber, and V. Subrahmanian. The relationship between reasoning about privacy and default logics. In *LPAR*, pages 637–650. Springer, 2005.
13. R. Fagin, P. G. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.
14. A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
15. O. Kutz, C. Lutz, F. Wolter, and M. Zakharyaschev. E-connections of abstract description systems. *Artifical Intelligence*, 156(1):1–73, 2004.
16. M. Lenzerini. Data integration: a theoretical perspective. In *ACM PODS '02*, pages 233–246. ACM Press, 2002.
17. A. Machanavajjhala and J. Gehrke. On the efficiency of checking perfect privacy. To appear in *Proceedings of PODS 2006*.
18. G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. In *SIGMOD*, 2004.
19. OECD. Guidelines on the protection of privacy and transborder flows of personal data, 1980. Available at `http://www.oecd.org`.
20. Privacy International. Overview of privacy, 2004. Available at `http://www.privacyinternational.org/privhroverview2004`.
21. R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233. 1982.
22. P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *PODS*, page 188. ACM Press, 1998.
23. K. Stoffel and T. Studer. Provable data privacy. In K. Viborg, J. Debenham, and R. Wagner, editors, *Database and Expert Systems Applications DEXA 2005*, volume 3588 of *LNCS*, pages 324–332. Springer, 2005.
24. R. van der Meyden. Logical approaches to incomplete information: a survey. In *Logics for databases and information systems*, pages 307–356. Kluwer Academic Publishers, 1998.
25. M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Trans. Database Syst.*, 19(4):626–662, 1994.

Linear Complementarity and P-Matrices for Stochastic Games*

Ola Svensson and Sergei Vorobyov

IDSIA, Lugano, Switzerland and TU Wien, Austria
svorobyov@gmail.com

Abstract. We define the first nontrivial polynomially recognizable subclass of P-matrix Generalized Linear Complementarity Problems (GLCPs) with a subexponential pivot rule. No such classes/rules were previously known. We show that a subclass of Shapley turn-based stochastic games, subsuming Condon's simple stochastic games, is reducible to the new class of GLCPs. Based on this we suggest the new strongly subexponential combinatorial algorithms for these games.

1 Introduction

The *Linear Complementarity Problem* (LCP: find vectors $w, z \geq 0$ satisfying $w = Mz + q$ and $w^T z = 0$ for given real square matrix M and vector q) is a powerful framework for combinatorial and continuous optimization, with a rich theory [21,10] and numerous important applications. The general problem is NP-hard, but there are many rich polynomially solvable subclasses, such as Z-matrix and PSD-matrix (positive semidefinite) LCPs [21,10,17]. For a prominent class of P-matrix LCPs (possessing unique solutions, with positive principal minors of matrix M) there are no currently known polynomial algorithms, but there is strong evidence (NP$\neq$coNP) that the P-matrix LCP is not NP-hard [19]. It is an exciting open problem to invent polynomial or at least subexponential algorithms for nontrivial subclasses of P-matrix LCPs [20].

We consider the *Generalized* LCP (GLCP) introduced by Cottle and Dantzig [9], also referred to as the *Vertical* LCP in the literature. The GLCP subsumes LCP as a particular case, and it is more flexible and convenient in applications, like game-theoretic ones aimed at in this paper.[1] All definitions of matrix classes (P-, Z-, etc.) extend straightforwardly to the GLCP.

In this paper we describe the first nontrivial subclass of P-matrix GLCPs, which we call D-matrix GLCPs or DGLCPs (D- stands for *discounted*), together

* Supported by the Grant IG2003-2 067 from the Swedish Foundation for International Cooperation in Research and Higher Education (STINT) and the Swedish Research Council (VR) grants "*Infinite Games: Algorithms and Complexity*" and "*Interior-Point Methods for Infinite Games*".

[1] The advantage of the GLCP is that it allows for arbitrary, non-binary, complementarity condition (in contrast to the standard LCP), which is more suitable for describing games on graphs with arbitrary outdegree.

with a number of randomized subexponential algorithms based on combinatorial linear programming. The class of D-matrices has a simple syntactic description. We show that D-matrix GLCP is nontrivial by subsuming Shapley's turn-based stochastic games [22] and Condon's simple stochastic games [8] (both currently not known to be polynomial). To our knowledge, prior to this paper there were no nontrivial, polynomially recognizable and subexponentially solvable classes of P-matrix GLCPs.

Our investigation into the GLCP theory was motivated by applications to solving certain full-information infinite adversary games. In [5] we investigated the LCP approach for expressing and solving *Mean Payoff Games* (MPGs) [13,14], and developed the first subexponential LCP-based algorithm for MPGs. It was obvious that reductions to LCPs described in [5] applies to wider classes of games, including stochastic games and a more general framework of *Controlled Linear Programming Problems* (CLPPs) [3,2,4], but prior to this paper the usefulness of such reductions was questionable by lack of existing subexponential algorithms for nontrivial classes of (G)LCPs. Alternative approaches to solving simple stochastic and mean payoff games are described in [7,6].

Paper Outline. After recalling stochastic games in Section 2, in Section 3 we reduce the value problem for these games to the GLCP problem (basic facts about GLCPs are collected in Appendix A). The structure of the resulting matrices, called D-matrices, is explored in Section 4. D-matrices happen to be P-matrices and D-matrix GLCPs possess unique solutions (Section 5). After that we introduce a class of switching/pivoting algorithms for D-matrix GLCPs (Section 6), analyze the structure of the matrices they produce (Sections 7, 8), prove monotonicity of switching (Section 9) and optimality of stable strategies (Section 10), crucial for termination and subexponential analysis. A family of subexponential algorithms is described in Section 11. Algorithms for solving one-player games at the bottom of recursion are described in Section 12. Missing proofs/details can be found in [23].

2 Shapley's Stochastic Games

For $m \in \mathbb{N}$ denote $[m] = \{i \in \mathbb{N} | 1 \leq i \leq m\}$. In a stochastic game [22] there are finitely many N positions, and players Max, Min have finitely many action choices, $[m_k]$, $[n_k]$, respectively, in each position $k \in [N]$. If in position k player Max selects action $i \in [m_k]$ and Min simultaneously selects action $j \in [n_k]$, then Max gets payment a^k_{ij} from Min, with probability $s^k_{ij} > 0$ the play stops, while with probability $p^{kl}_{ij} \geq 0$ the play proceeds to position l. A particular game Γ^k is obtained by specifying the starting position k. Player Max wants to maximize, whereas player Min to minimize the total payoff, which accumulates during the play. Assume, $\sum_{l=1}^{N} p^{kl}_{ij} = 1 - s^k_{ij} < 1 - s < 1,\ |a^k_{ij}| < M$. Then the probability that a play does not stop after t steps is at most $(1-s)^t$, and the maximal payoff does not exceed $\mathbf{M} = M \sum_{i=0}^{\infty} (1-s)^i = M/s$.

Turn-Based Stochastic Games. Simultaneous move stochastic games thus defined are not perfect information. In *turn-based* stochastic games, for every position k at least one of m_k, n_k equals one. For such a game, let $k \in \textsc{Max}$ if $m_k > 1$, $k \in \textsc{Min}$ if $n_k > 1$. We call positions k for which both $m_k = n_k = 1$ *unary* and arbitrarily let $k \in \textsc{Max}$ or $k \in \textsc{Min}$. Turn-based stochastic games are perfect information, solvable in pure positional strategies [22], and the unique value (optimal payoff) for every vertex is determined by the *unique* solution of the system

$$\begin{aligned} v_k &= \max_{i \in [m_k]}(a^k_{i1} + \textstyle\sum_l p^{kl}_{i1}\, v_l), \ \text{ for } k \in \textsc{Max}, \\ v_k &= \min_{j \in [n_k]}(a^k_{1j} + \textstyle\sum_l p^{kl}_{1j}\, v_l), \ \text{ for } k \in \textsc{Min}. \end{aligned} \tag{1}$$

3 Reducing to Generalized LCP

In this section we show that turn-based Shapley stochastic games are reducible to Generalized LCPs (see Appendix A) of a specific structure.

By introducing, if necessary, auxiliary *unary* positions between positions of the same player, and by appropriately modifying stopping and transitional probabilities, we may assume, with no loss of generality, that the game is *bipartite*, i.e., $p^{kl}_{ij} > 0$ implies $k \in \textsc{Max}$ and $l \in \textsc{Min}$ or $k \in \textsc{Min}$ and $l \in \textsc{Max}$.[2] System (1) can be equivalently presented as:

$$\begin{aligned} v_k &= \max\{\, -\mathbf{M},\, a^k_{i1} + \sum_l p^{kl}_{i1}\, u_l \mid i \in [m_k]\}, \ \text{ for } k \in \textsc{Max}, \\ u_k &= \min\{\ \ \mathbf{M},\ \ a^k_{1j} + \sum_l p^{kl}_{1j}\, v_l \mid j \in [n_k]\}, \ \text{ for } k \in \textsc{Min}, \end{aligned} \tag{2}$$

where we reflect bipartiteness by using v_i/u_i for $\textsc{Max}/\textsc{Min}$ variables.

Let us introduce $m_k + 1$ fresh auxiliary *nonnegative* variables $z_k, w^k_1, \ldots, w^k_{m_k} \geq 0$ for each variable $v_k \in \textsc{Max}$, $n_k + 1$ auxiliary *nonnegative* variables $z_k, w^k_1, \ldots, w^k_{n_k} \geq 0$ for each variable $v_k \in \textsc{Min}$, and rewrite the system (2) as

$$\begin{aligned} v_k \quad &= z_k - \mathbf{M}, && \text{for } k \in \textsc{Max}, \\ v_k \quad &= w^k_i + a^k_{i1} + \sum_l p^{kl}_{i1}\, u_l, && \text{for } k \in \textsc{Max},\ i \in [m_k], \\ u_k + z_k &= \mathbf{M}, && \text{for } k \in \textsc{Min}, \\ u_k + w^k_i &= a^k_{i1} + \sum_l p^{kl}_{i1}\, v_l, && \text{for } k \in \textsc{Min},\ i \in [n_k], \end{aligned} \tag{3}$$

additionally stipulating *complementarity*, i.e.,

$$z_k \cdot \prod_i w^k_i = 0 \text{ for each } k \in [N]. \tag{4}$$

[2] Although this may blow up quadratically the number of positions, unary positions introduced do not make worse the resulting complexity.

Excluding variables v_i, u_i from (3), we can rewrite it as

$$\begin{aligned} w_i^k &= z_k + P_i^k(\bar{z}_{|\textsc{Min}}),\ \text{for } k \in \textsc{Max},\ i \in [m_k], \\ w_i^k &= z_k + P_i^k(\bar{z}_{|\textsc{Max}}),\ \text{for } k \in \textsc{Min},\ i \in [n_k], \end{aligned} \tag{5}$$

where: 1) polynomials $P_i^k(\bar{z}_{|\textsc{Min}})$ contain only variables z_j for $j \in \textsc{Min}$, 2) polynomials $P_i^k(\bar{z}_{|\textsc{Max}})$ contain only variables z_j for $j \in \textsc{Max}$, 3) these polynomials have all variable coefficients nonnegative, summing up to < 1 (call such polynomials *discounted*). Note that in obtaining this form of system (5) we essentially use bipartiteness, which guarantees that variables w_i^k and z_k appear with nonnegative coefficients on *different* sides of equations.

4 D-Matrices and Discounted GLCPs

Finding nonnegative values z_k, w_i^k satisfying (5) and (4) is a well known Generalized (or Vertical) LCP [9]; see Appendix A for a reminder of the main definitions. In this paper, motivated by the special structure of the system (5), we introduce a new class of vertical matrices and corresponding GLCPs called *Discounted*, D-matrices and DGLCPs for short. We will demonstrate that D-matrices are P-matrices, and *unique* solutions to DGLCPs can be found in randomized *subexponential* time, which cannot be done (at least not known yet) for general P-matrices [20]. Here is our main

Definition 1 (Discounted Vertical Matrix, Discounted LCP). *A vertical block matrix A is* Discounted*, or D-matrix, if A is of the form depicted in Figure 1 and has the following properties: 1) all elements of A are non-negative; 2) every representative submatrix of A has a unit main diagonal; 3) the remaining nonzero entries are located in the gray area; 4) A is strictly row diagonally dominant. A D-matrix GLCP is called* Discounted *GLCP, or DGLCP for short.* □

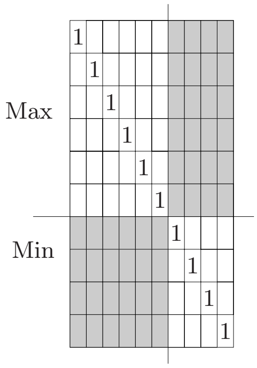


Fig. 1. The structure of a D-matrix. 1's denote column vectors of ones.

Note that the property of being a D-matrix is easily polynomial time recognizable. Assuming the unit main diagonal is just a matter of technical convenience and may be dropped.

The reduction from the previous section shows that the class of DGLCPs is nontrivial:

Theorem 1. *Turn-based Shapley stochastic games and simple stochastic games reduce to the DGLCP.* □

Notational Conventions. From now on we assume that the vertical block matrices considered are of dimension $m \times k$ with $m \geq k$. The upper part of a D-matrix, consisting of $n \leq k$ blocks, is associated with the player Max, and the lower part with the player Min. Thus the range of blocks and columns is split between Max and Min. We will write $i \in \text{Max}$ or $j \in \text{Min}$ meaning that the corresponding index is in the range of one of the players. By $z_{|\text{Min}}$ we will denote vector z with all components of Max replaced with zeros, and similarly for $z_{|\text{Max}}$, $w_{|\text{Max}}$, $w_{|\text{Min}}$.

By U^i_j or L^i_j, respectively, depending on whether $i \in \text{Max}$ or $i \in \text{Min}$, we will denote the j-th row in the i-th block of a D-matrix, with the 1 in the i-th coordinate (main diagonal) replaced with 0. We will call vectors U^i_j and L^i_j *discounted*, because they have nonnegative coordinates summing up to a number < 1. Note that by our conventions $U^i_j\, z_{|\text{Min}} = U^i_j\, z$ and $L^i_j\, z_{|\text{Max}} = L^i_j\, z$.

5 D-Matrices Are P-Matrices, Uniqueness of Solutions

We start by an important property that D-matrices form a subclass of P-matrices.

Theorem 2. *Every D-matrix is a P-matrix.*

Proof. Since every representative submatrix of a D-matrix has positive diagonal and is strictly diagonally dominant, it is a P-matrix [10, p. 152]. □

P-matrix GLCPs have unique solutions [24], therefore,

Theorem 3 (Uniqueness). *Every D-matrix GLCP has a unique solution.* □

6 Strategies, Attractiveness, Switches, Stability

A strategy prescribes which of the complementary variables are to be zeroed.

Definition 2 (Strategy). *A strategy σ_i for the block $i \in$ Max is a selection of either $\sigma_i = \{z_i = 0\}$ or $\sigma_i = \{w^i_j = 0\}$, for some $j \in \{1, \ldots, p_i\}$. A full* Max *strategy σ consists of selections σ_i for all blocks $i \in$ Max and is denoted $\sigma = \{\sigma_1, \ldots, \sigma_n\}$. A partial* Max *strategy consists of strategy selections for some* Max *blocks.* □

After selecting a strategy σ_i for block $i \in$ MAX, we have either $z_i = 0$ or $z_i = -q^i_j - U^i_j \cdot z_{|\text{MIN}}$. Denote by $GLCP_{\sigma_i}(A, q)$ the system (A, q) with the i-th block and i-th column removed and remaining occurrences of z_i replaced with 0 (if $\sigma_i = \{z_i = 0\}$) or with $-q^i_j - U^i_j \cdot z_{|\text{MIN}}$ (if $\sigma_i = \{w^i_j = 0\}$). Note that $GLCP_{\sigma_i}(A, q)$ is not a DGLCP any longer, in general.

Given a MAX strategy σ for all blocks we denote the resulting system, after removing all blocks of MAX and replacing all $z_{|\text{MAX}}$, by $GLCP_\sigma(A, q)$. Note that after finding a solution $w_{|\text{MIN}}$ and $z_{|\text{MIN}}$ for $GLCP_\sigma(A, q)$, the values $w_{|\text{MAX}}$ and $z_{|\text{MAX}}$ are easily and uniquely calculated by substitution but these values may be *negative*. The fact that some values are negative means that we made a mistake in selecting a strategy and some switches have to be made.

Definition 3 (Attractiveness, Switches, and Stability). *For a full* MAX *strategy σ, let w^* and z^* be the complementarity vectors after calculating $w_{|\text{MAX}}$ and $z_{|\text{MAX}}$ from the solution of $GLCP_\sigma(A, q)$, as explained above.*

1. *Say that a pivot to w^i_j or z_i is* attractive*, for $i \in$* MAX*, if $w^{*i}_j < 0$ or $z^*_i < 0$.*
2. *An attractive switch for σ in block $i \in$* MAX *results from making an attractive pivot, replacing σ_i with $\sigma'_i = \{w^i_j = 0\}$ or $\sigma'_i = \{z_i = 0\}$.*
3. *The strategy σ is* stable *if w^* and z^* are nonnegative, i.e., give a solution to the DGLCP(A,q).*

7 GLCPs Resulting from Fixing Partial Strategies

Making a partial (or complete) substitution of a strategy in a DGLCP results in a GLCP with a unique solution, which is a critical invariant for our algorithms:

Theorem 4. *For a $DGLCP(A, q)$ with matrix A of order $p \times k$ and n blocks of* MAX*, the resulting $GLCP_{\sigma_{1,\dots,l}}(A, q)$, for any* MAX *strategy $\sigma_{1,\dots,l}$ ($l \leq n$), has a unique solution.*

Proof. By induction. The inductive hypothesis (IH) is that the matrix in the system $GLCP_{\sigma_{1,\dots,l}}(A, q)$ has the following properties. 1) The MAX partition is as in the Definition 1 of the DGLCP. 2) The MIN partition is strictly row diagonally dominant with a positive diagonal ≤ 1 (in every representative matrix).

For the base case $l = 0$, the original system $DGLCP(A, q)$ satisfies the IH by definition. For the inductive step, assume, IH is true for $l - 1$, with $l \leq n$. We want to prove the IH for the $GLCP_{\sigma_{1,\dots,l}}(A, q)$. Let A be the vertical block matrix for the system $GLCP_{\sigma_{1,\dots,l-1}}(A, q)$.

Case 1 ($\sigma_l = \{z_l = 0\}$). Removing the l-th column and block from A and substituting z_l with 0 in the remainder results is a matrix $GLCP_{\sigma_{1,\dots,l}}(A, q)$ obviously satisfying the IH.

Case 2 ($\sigma_l = \{w^l_j = 0\}$). We remove the l-th block and column and substitute z_l with $-U^l_j \cdot z - q^l_j$. The resulting matrix B will be the matrix for the system

$GLCP_{\sigma_{1,\ldots,l}}(A,q)$. It is immediate that the Max partition of B satisfies the IH. Moreover, every Min row in A can be written as $w_i^d = A_i^d \cdot z + q_i^d = \lambda_1 z_1 + \ldots \lambda_l z_l + \cdots + \lambda_k z_k + q_i^d$.

By positivity of the diagonal and strict row diagonal dominance (IH), we have $\lambda_d > \sum_{x \neq d} |\lambda_x|$. After substituting z_l with $-U_j^l \cdot z - q_j^l\ z_l$, the same row of B will be

$$w_i^d = \lambda_1 z_1 + \cdots - \lambda_l U_j^l \cdot z_{|\text{Min}} + \cdots + \lambda_k z_k + q_i^d - \lambda_l q_j^l.$$

Since $\sum_y |U_{jy}^l| < 1$ implies $|\lambda_l| \sum_y |U_{jy}^l| < |\lambda_l|$, it follows that $\lambda_d > \sum_{x \neq d} |\lambda_x| > \sum_{x \neq d, x \neq l} |\lambda_x| + |\lambda_l| \sum_y |U_{jy}^l|$. Thus, after the substitution of z_l the row remains strictly diagonally dominant with a positive diagonal. Also, the diagonal entry is ≤ 1, because at every step one Max variable z_i with a positive coefficient is replaced by either 0 or a nonpositive linear polynomial depending on variables z_j, $j \in$ Min. Therefore, λ_d may only decrease. Consequently, the Min part of $GLCP_{\sigma_{1,\ldots,l}}(A,q)$ also satisfies the IH. We have proved that the IH holds for all $GLCP_{\sigma_{1,\ldots,l}}(A,q)$ for $l \leq n$. The uniqueness of their solutions follows from the IH by the proof of Theorem 2 and by Theorem 3. □

8 One-Player Case Yields Discounted Z-Matrices

We have a special restricted subclass of block Z-matrices resulting from substituting full Max strategies in DGLCPs, which deserve a special name.

Definition 4. *A vertical block is called a ZD$^+$-matrix if it is:*

1. *a Z-matrix, i.e., all off-diagonal elements (in the representative matrices) are ≤ 0;*
2. *diagonally positive with all diagonal elements in the range* $(0,1]$*;*
3. *strictly row diagonally dominant.* □

Theorem 5. *For every full* Max *strategy σ the matrix in the system $GLCP_\sigma$ (A,q) is a ZD$^+$-matrix possessing a unique solution.*

Proof. Recall that the first n blocks belongs to Max and the remaining m blocks belongs to Min. Every Min row in $DGLCP(A,q)$ can be written as

$$w_j^{n+d} = z_{n+d} + \lambda_1 z_1 + \cdots + \lambda_n z_n + q_j^{n+d}, \tag{6}$$

with $\lambda_l \geq 0$. After selecting a strategy σ, either $z_i = 0$ or $z_i = -U_j^i z_{|\text{Min}} - q_j^i$, for $1 \leq i \leq n$ and some $j \in \{1, \ldots, p_i\}$. Substituting the value for every z_i, $1 \leq i \leq n$, into (6) will result in a nonpositive coefficient in front of z_j, where $n < j \leq k$ and $j \neq n+d$, because U is nonnegative. Hence, all off-diagonal entries of $GLCP_\sigma(A,q)$ will be nonpositive. Consequently, the matrix of $GLCP_\sigma(A,q)$ is a Z-matrix.[3] The remaining conditions 2, 3 of Definition 4 for

[3] This part of the proof does not use any row diagonally dominance or discountedness properties. It only relies on the bipartite structure, shown in Figure 1, and can therefore be generalized.

the matrix of $GLCP_\sigma(A,q)$ and solution uniqueness follow from the inductive proof of Theorem 4.[4] □

Corollary 1. *Every ZD$^+$-matrix is a K-matrix.* □

9 Monotonicity: Attractiveness Is Profitable

Monotonicity of attractive switches/pivots (to be explained shortly) is the crucial property ensuring termination of our pivoting algorithms and allowing for subexponential upper bounds. To simplify the proof (to reduce the number of cases considered), we assume that the algorithms always start from a Max strategy selecting $z_i = 0$ for all $i \in$ Max and proceed by making attractive switches/pivots.[5] A subexponential randomized policy is described in Section 11, but monotonicity proved here guarantees finite termination of any sequence of attractive switches/pivots. For our purposes, making just one attractive switch at a time is enough, but one can consider a generalization when several such pivots are made simultaneously. To simplify notation we make a convention to denote solutions to the GLCPs before and after a switch as non-primed w, z and primed $w^{'}$, $z^{'}$.

Definition 5. *The* value *$val(w,z)$ of a solution (w,z) to a DGLCP equals*

$$\sum_{i \in \text{Max}} z_i - \sum_{k \in \text{Min}} z_k. \tag{7}$$

Monotonicity of attractive switches guarantees that this value *strictly monotonically increases*, which immediately follows from the more general

Theorem 6 (Monotonicity). *For every attractive switch/pivot in any DGLCP instance from solution (w,z) to solution $(w^{'},z^{'})$ one has:*

1. *$z_i^{'} - z_i \geq 0$ for each $i \in$ Max (monotonic* non-decrease*);*
2. *at least one inequality above is* strict, *namely the one in the block where an attractive switch was made;*
3. *$z_k^{'} - z_k \leq 0$ for each $k \in$ Min (monotonic* non-increase*).*

Proof. Suppose, an attractive switch/pivot in block $i \in$ Max results in a new strategy with $\sigma_i^{\prime} = \{w_j^{'i} = 0\}$.

Let us start by proving Claim 3 by contradiction.[6] Since the switch was attractive, the following constraints are satisfied (the first line means attractiveness, the second stipulates that that after a switch we impose $w_j^{'i} = 0$): [7]

$$\begin{aligned} 0 > w_j^i \; &= q_j^i + z_i + U_j^i \; z, \\ 0 = w_j^{'i} &= q_j^i + z_i^{'} + U_j^i \; z^{'}. \end{aligned} \tag{8}$$

[4] This part of the proof depends on diagonal dominance and discountedness.

[5] With this assumption, every switch away from $z_i = 0$, $i \in$ Max, will be definitive, i.e., the algorithm will never switch back to $z_i = 0$. The extension to an arbitrary initial strategy is pretty straightforward.

[6] This is the only part of the proof that relies on discountedness.

[7] It is not important here whether before the switch σ_i was $\{z_i = 0\}$ or $\{w_{j'}^i = 0\}$.

Suppose, toward a contradiction, that some z_k (for $k \in \textsc{Min}$) increases its value, and select k yielding the *largest* increase $c > 0$,

$$c = z_k^{'} - z_k > 0. \tag{9}$$

Subtracting the first line in (8) from the second one, we get $-(z_i^{'} - z_i) < U_j^i(z^{'} - z) \leq \lambda \cdot c$, where the last inequality holds for some $0 < \lambda < 1$, because $U_j^i\, z$ is discounted, depends only on variables of $\textsc{Min}$ ($U_j^i z = U_j^i z_{|\textsc{Min}}$), z, $z^{'}$ are nonnegative, and by the choice of k. Consequently, for the block $i \in \textsc{Max}$ in which the switch was made, $z_i^{'} - z_i > -\lambda \cdot c$. Similarly, in each block $i \in \textsc{Max}$ in which there was no switch, $z_i^{'} - z_i \geq -\lambda \cdot c$. (the only difference consists in replacing $>$ in the first line of (8) with $=$, which results in a non-strict inequality.) Now let us look in the selected block $k \in \textsc{Min}$. For *every* constraint m in this block, before and after the switch, we have:

$$\begin{aligned} w_m^k &= q_j^k + z_k + L_j^k\, z \\ w_m^{'k} &= q_j^k + z_k^{'} + L_j^k\, z^{'} \end{aligned} \tag{10}$$

Subtracting the first line of (10) from the second one we get $w_m^{'k} - w_m^k = (z_k^{'} - z_k) + L_m^k(z^{'} - z) \geq c - \lambda \cdot c > 0$, because $L_m^k\, z$ is a discounted polynomial depending only on variables z_i with $i \in \textsc{Max}$, and for all such we proved $z_i^{'} - z_i \geq -\lambda \cdot c$. The last chain of inequalities leads to a contradiction. Indeed, $w_m^{'k} - w_m^k > 0$ plus nonnegativity of w_m^k imply $w_m^{'k} > 0$ for *every* m in block k. By assumption $z_k^{'} - z_k > 0$ and nonnegativity of z_k, we also have $z_k^{'} > 0$. But this implies that the complementarity $z_k^{'} \prod_{m=1}^{n_m} w_m^{'k} > 0$ in block k is violated. This shows that the increase (9) for z_k, $k \in \textsc{Min}$, cannot happen, which proves Claim 3.

Let us now prove Claims 1 and 2, which depend on non-negativity of coefficients in U_j^i, but not on discountedness. Assume the attractive switch happens in block i and consists in switching from $\sigma_i = \{w_l^i = 0\}$ to $\sigma_i^{'} = \{w_j^{'i} = 0\}$, [8] i.e.:

$$\begin{aligned} 0 = w_l^i\ &= q_l^i + z_i + U_l^i\, z, \\ 0 > w_j^i\ &= q_j^i + z_i + U_j^i\, z, \\ 0 = w_j^{'i} &= q_j^i + z_i^{'} + U_j^i\, z^{'}. \end{aligned} \tag{11}$$

(This is consistent with (8); the second line in (11) coincides with the first line in (8). Line 1 expresses the selection before the switch, line 2 attractiveness, and line 3 the selection after the switch.)

From (11) we derive $z_i = -q_l^i - U_l^i\, z < -q_j^i - U_j^i\, z \leq -q_j^i - U_j^i\, z^{'} = z_i^{'}$, where the inequality $\leq$ holds because U_j^i has nonzero (positive) coefficients only for z_k, $k \in \textsc{Min}$ (recall that by our notational convention $U_j^i\, z = U_j^i\, z_{|\textsc{Min}}$), and by the fact that $z_k^{'} - z_k \leq 0$ proved as Claim 3 above. Therefore, if an attractive switch happened in block i, the corresponding z_i component *strictly increases* $z_i^{'} > z_i$, proving Claim 2.

[8] The case not covered here, but completely analogous, is when the switch is made from $\sigma_i = \{z_i = 0\}$, i.e., the first line is replaced with $z_i = 0$.

A block $i \in$ MAX in which an attractive switch was *not made* corresponds to the system similar to (11):

$$\begin{aligned} 0 &= w_l^i \ = q_l^i + z_i + U_l^i\ z, \\ 0 &= w_l^{'i} = q_l^i + z_i^{'} + U_l^i\ z^{'} \end{aligned} \tag{12}$$

from which we derive, analogously ($\leq$ holds for the same reason) that $z_i = -q_l^i - U_l^i\ z \leq -q_l^i - U_l^i\ z^{'} = z_i^{'}$, which proves Claim 1 $z_i^{'} \geq z_i$ for $i \in$ MAX and finishes the proof. □

10 Stability Implies Optimality for Discounted GLCPs

The following result is essential for correctness and complexity analysis of our algorithm. (We do not claim stable strategies are unique, they are generally not.)

Theorem 7. *In every DGLCP instance every stable* MAX *strategy determines the same solution.*

Proof. Consider any two stable strategies in a DGLCP instance I. Since both are stable, they both determine solutions for I, which are equal by Theorem 3. □

11 Subexponential Algorithms

We now have all the ingredients necessary to describe a class of randomized subexponential algorithms for the D-matrix GLCP, based on combinatorial linear programming schemes due to Kalai [15,16] and Matoušek-Sharir-Welzl [18].

Given a DGLCP instance (A, q), define a *hyperstructure* as a Cartesian product $\mathcal{P} = \prod_{i=1}^{n} S_i$, where $n \leq k$ is the number of MAX blocks, k is the total number of blocks, $S_i = \{0, \ldots, p_i\}$, and p_i the size of the i-th block. Intuitively, $\mathcal{P}$ is the space of all MAX strategies, with 0 corresponding to $\sigma_i = \{z_i = 0\}$ and $j > 0$ corresponding to $\sigma_i = \{w_j^i = 0\}$. Define a *substructure* $\mathcal{P}'$ of $\mathcal{P}$ as a Cartesian product $\mathcal{P}' = \prod_{i=1}^{n} S_i'$, where $0 \in S_i' \subseteq S_i$ for each $i \in \{1, \ldots, n\}$). It corresponds to the set of strategies in a DGLCP instance (A, q), in which some constraints have been deleted (which remains a DGLCP instance).

Define the *valuation* on the hyperstructure $\mathcal{P}$ as follows. For every MAX strategy $\sigma \in \mathcal{P}$ the $GLCP_\sigma(A, q)$ (obtained by substituting σ considered as an assignment of zeros for z_i and w_j^i, as described in Section 6) is a ZD$^+$-matrix GLCP, possessing a unique solution (w, z) (Theorems 4, 5). Find this solution as described in Section 12. Assign to σ the value $\nu(\sigma) = val(w, z)$, as defined by (7). With this valuation,

1. every two neighbors σ and σ' on $\mathcal{P}$ (at Hamming distance 1) corresponding to an attractive switch from σ to σ' have values $\nu(\sigma) < \nu(\sigma')$;
2. on every *substructure* $\mathcal{P}' = \prod_{i=1}^{n} S_i'$ with $0 \in S_i' \subseteq S_i$ for each i, there is a unique stable (optimal) solution/value (cf., Theorem 7).

Now numerous well-known randomized subexponential algorithms [15,16,18] for finding a (globally) maximal valuation (stable strategy solving the DGLCP instance (A, q)) on the structure $\mathcal{P}$ apply. Roughly (we refer the reader to [15,16,18] for details), one of the versions of the algorithm is as follows. Given a structure $\mathcal{P}$, consider the initial strategy $\sigma = \{z_i = 0\}_{i=1,\dots,n}$, corresponding to the point $\hat{\sigma} = (0, \dots, 0) \in \mathcal{P}$ (below, for brevity we identify points of hyperstructures with corresponding strategies)[9], and proceed as follows.

1. if $\hat{\sigma} = (\hat{\sigma}_1, \dots, \hat{\sigma}_n)$ and *the bottom of recursion is hit*, expressed formally as[10]
$$\mathcal{P} = \prod_{i=1}^{n} \{0\} \cup \{\hat{\sigma}_i\}, \tag{13}$$
then solve an instance of ZD$^+$-matrix $GLCP_\sigma(A, q)$; see Section 12;
2. otherwise, consider a substructure $\mathcal{P}' \subset \mathcal{P}$ containing σ, obtained by (temporarily) throwing away a random $c \in S_i$, $c \neq 0$, $c \neq \sigma_i$ for a random i; [11]
3. apply the algorithm recursively to find a stable (globally maximal) σ^* on $\mathcal{P}'$;
4. return back the last c temporarily thrown away and check whether σ^* is stable in $\mathcal{P}$;
5. if yes, return σ^* as stable (globally maximal) on $\mathcal{P}$;
6. if not, make an attractive switch for σ^*, replacing σ_i^* with c; denote the resulting strategy σ and repeat from step 1.

The analyses of [15,16,18] yield the following

Theorem 8. *The above algorithm solves an instance of a DGLCP with n* Max *blocks after expected subexponential $2^{O(\sqrt{n \log n})}$ number of switches and invocations of the subroutine solving ZD$^+$-matrix $GLCP_\sigma(A, q)$ in step 1.* □

In Section 12 we show that ZD$^+$-matrix $GLCP_\sigma(A, q)$ can also be solved in expected subexponential (or in weakly polynomial) time. This results in the first nontrivial subclass of P-matrix GLCPs solvable in expected subexponential time (in the number of variables).

---

[9] For efficiency reasons, it is better to select, in the initial strategy, components $\sigma_l = \{w_1^l = 0\}$ for each unary position l. This neutralizes the effect of introducing many unary positions when reducing to the bipartite case. The correctness of this initial setting is explained by the fact that no switches will ever be made/become attractive in any run of the algorithm.

[10] Technically, we have to keep 0-components in substructures in (13) in order to be able to associate elements of hyperstructures to strategies in GLCPs, since, in general, the strategy switch to $\{z_i = 0\}$ is not excluded. However, starting with the strategy $\hat{\sigma} = (0, \dots, 0)$, and making attractive switches only, by Monotonicity Theorem 6, every switch away from $\{z_i = 0\}$ is *definitive*, since z_i for $i \in$ Max can only increase. Therefore, when (13) holds we immediately know that σ is optimal in $\mathcal{P}$, and it remains to solve the $GLCP_\sigma(A, q)$ to find values to be used in determining further attractive switches.

[11] In other words, we delete a random facet of $\mathcal{P}$ not containing σ.

Monotonicity of attractive switches (Theorem 6) is essential for acyclicity of the algorithm. Uniqueness (Theorems 3, 5) is crucial for subexponential analysis, because after finding an optimum on a substructure $\mathcal{P}'$ and making the next attractive switch, $\mathcal{P}'$ will never be revisited by the algorithm again (by monotonicity, each attractive switch improves the value), and the subexponential analysis based on *hidden dimensions* applies; see [15,16,18] for details.

12 Solving One-Player Z-GLCPs

In the bottom of recursion (when the full Max strategy σ is fixed) the randomized algorithm described in the previous section solves $GLCP_\sigma(A, q)$, an instance of ZD$^+$-matrix GLCP with a unique solution, as explained in Section 8. There are several possible algorithms for this problem.

1) By using the *least element property* [12], solving any feasible Z-matrix GLCP (A, q) amounts to solving a single linear program, minimizing any positive-coordinate linear target function over the feasible domain $\{z : z \geq 0, q + Az \geq 0\}$. There is a multitude of polynomial (but non-strongly) algorithms for that.

2) The above linear programming problem instance can be solved in randomized strongly subexponential time by the algorithms [15,16,18]. Note that this algorithm is subexponential $2^{O(\sqrt{(k-n)\log(k-n)})}$ in the number $k - n$ of Min blocks (equals the number of z-variables remaining in $GLCP_\sigma(A, q)$). The advantage of using options 1 or 2 depends on the size of coefficients in the instance $GLCP_\sigma(A, q)$. Applying option 2, together with Theorem 8 results in

Theorem 9. *A D-matrix GLCP instance with k blocks, n of which belong to* Max*, can be solved in expected subexponential time* $2^{O(\sqrt{n \log n} + \sqrt{(k-n)\log(k-n)})}$.

Further improvement will be achieved if a more efficient, strongly polynomial, algorithm for solving ZD$^+$-matrix GLCP instances is used at the bottom of recursion. This subject is outside the scope of this paper, deserves a separate careful treatment, and the progress will be reported elsewhere; see, e.g., [1].

13 Conclusions

We identified the first nontrivial subclass of P-matrix Generalized LCPs, which is: 1) polynomial time recognizable (in general, the P-matrix property is coNP-complete); 2) has a very simple syntactical structure; 3) subsumes Shapley's turn-based stochastic games and Condon's simple stochastic games (currently not known to be polynomial time solvable). We suggested the first subexponential pivot rule and algorithm for this class of GLCPs; no such rules were previously known, all were either polynomial or exponential. The resulting algorithm for stochastic games has the same asymptotic behavior as other best currently available algorithms for the problem [4,6].

References

1. D. Andersson and S. Vorobyov. Fast algorithms for monotonic discounted linear programs with two variables per inequality. Manuscript submitted to *Theoretical Computer Science*, July 2006. Preliminary version available as Isaac Newton Institute Preprint NI06019-LAA.
2. H. Björklund, O. Nilsson, O. Svensson, and S. Vorobyov. Controlled linear programming: Boundedness and duality. TR DIMACS-2004-56, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, 2004.
3. H. Björklund, O. Nilsson, O. Svensson, and S. Vorobyov. The controlled linear programming problem. TR DIMACS-2004-41, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, 2004.
4. H. Björklund, O. Svensson, and S. Vorobyov. Controlled linear programming for infinite games. TR DIMACS-2005-13, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, 2005.
5. H. Björklund, O. Svensson, and S. Vorobyov. Linear complementarity algorithms for mean payoff games. TR DIMACS-2005-05, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, 2005.
6. H. Björklund and S. Vorobyov. Combinatorial structure and randomized subexponential algorithms for infinite games. *Theoretical Computer Science*, 349(3): 347–360, 2005.
7. H. Björklund and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Discrete Applied Mathematics*, 2006, to appear. Available from `http://www.sciencedirect.com/`, 27 June, 2006.
8. A. Condon. The complexity of stochastic games. *Information and Computation*, 96:203–224, 1992.
9. R. W. Cottle and G. B. Dantzig. A generalization of the linear complementarity problem. *Journal of Combinatorial Theory*, 8:79–90, 1970.
10. R. W. Cottle, J.-S. Pang, and R. E. Stone. *The Linear Complementarity Problem*. Academic Press, 1992.
11. G. Coxson. The P-matrix problem is coNP-complete. *Mathematical Programming*, 64:173–178, 1994.
12. A. A Ebiefung and M. M. Kostreva. The generalized linear complementarity problem: least element theory and Z-matrices. *Journal of Global Optimization*, 11:151–161, 1997.
13. A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journ. of Game Theory*, 8:109–113, 1979.
14. V. A. Gurvich, A. V. Karzanov, and L. G. Khachiyan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 28(5):85–91, 1988.
15. G. Kalai. A subexponential randomized simplex algorithm. In *24th ACM STOC*, pages 475–482, 1992.
16. G. Kalai. Linear programming, the simplex algorithm and simple polytopes. *Math. Prog. (Ser. B)*, 79:217–234, 1997.
17. M. Kojima, N. Megiddo, T. Noma, and A. Yoshise. *A Unified Approach to Interior Point Algorithms for Linear Complementarity Problems*, volume 538 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
18. J. Matoušek, M. Sharir, and M. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

19. N. Megiddo. A note on the complexity of P-matrix LCP and computing the equilibrium. Technical Report RJ 6439 (62557) 9/19/88, IBM Almaden Research Center, 1988.
20. W. D. Morris. Randomized pivot algorithms for P-matrix linear complementarity problems. *Mathematical Programming, Ser. A*, 92:285–296, 2002.
21. K. G. Murty and F.-T. Yu. *Linear Complementarity, Linear and Nonlinear Programming.* Heldermann Verlag, Berlin, 1988. http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/.
22. L. S. Shapley. Stochastic games. *Proc. Natl. Acad. Sci. U.S.A.*, 39:1095–1100, 1953.
23. O. Svensson and S. Vorobyov. A subexponential algorithm for a subclass of P-matrix generalized linear complementarity problems. TR DIMACS-2005-20, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ, 2005.
24. B. P. Szanc. *The Generalized Complementarity Problem.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1989.

A Appendix: Generalized LCP

Definition 6. *A* vertical block matrix of type $(p_1, \ldots, p_k)$ *is a real block matrix A of order $p \times k$, where $p = \sum_{j=1}^{k} p_j$, partitioned by horizontal cuts in blocks A_j of order $p_j \times k$, for $j = 1, \ldots, k$.* □

Note that in A the number of blocks equals the number of columns.

Here comes the main definition.

Definition 7 (GLCP/VLCP). *An instance of the* Generalized or Vertical *LCP is specified as follows.*

Given: *a* vertical block matrix A *of type* $(p_1, \ldots, p_k)$ *and a constant vector q decomposed in conformity with A:*

$$q = \begin{bmatrix} q^1 \\ \vdots \\ q^k \end{bmatrix}, \qquad A = \begin{bmatrix} A^1 \\ \vdots \\ A^k \end{bmatrix}.$$

Find: *a vector $w \in \mathbb{R}^p$ (decomposed as q) and $z \in \mathbb{R}^k$ satisfying*

$$\begin{aligned} & w = q + Az, \\ & w \geq 0, z \geq 0, \\ & z_i \textstyle\prod_{j=1}^{p_i} w_j^i = 0, \textit{ for } i = 1, \ldots, k, \quad \textit{(Generalized Complementarity)} \end{aligned} \tag{14}$$

where $p = \sum_{i=1}^{k} p_i$. □

The standard LCP is a special case of the GLCP (14), with all blocks of size 1 and square matrix A. Many results of the GLCP depend on the matrix structure. The analysis of the matrix structure of a GLCP, often boils down to the investigation of representative submatrices.

Definition 8 (Representative Submatrix). *A square submatrix M of a vertical block matrix A is called a* representative submatrix *if its i-th row is drawn from A^i, the i-th block of A.* □

The following classes of matrices are well investigated in the literature [21,10,17]. Every class is first defined for square matrices and then the definition is extended in a standard way to block matrices by stipulating the property to hold for all representative submatrices.

Definition 9. *A square matrix M is:*

1. *a P-matrix if all principal minors of M are positive;*
2. *a Z-matrix if all off-diagonal elements of M are nonpositive; if M also is a P-matrix it is called a K-matrix;*
3. *strictly row diagonally dominant if $|M_{ii}| > \sum_{j \neq i} |M_{ij}|$ for each row i;*
4. *diagonally positive if all diagonal elements of M are positive.*

A vertical block matrix is a P-matrix, Z-matrix, etc., if all its representative submatrices are square P-matrices, Z-matrices, etc, respectively. □

The property of being a P-matrix is coNP-complete [11].

RapidOWL — An Agile Knowledge Engineering Methodology

Sören Auer and Heinrich Herre

University of Leipzig, 04109 Leipzig, Germany
auer@informatik.uni-leipzig.de
http://www.informatik.uni-leipzig.de/~auer/

1 Problem

The analysis of the application of the existing knowledge engineering methodologies and tools shows that they are up to now virtually not used in practice (see [13, page 16]). This stands in contrast to the often proclaimed necessity for knowledge engineering. What can be the reason for this discrepancy? Most of the existing knowledge engineering methodologies adopt techniques and apply process models from software engineering. However, in many scenarios required knowledge engineering tasks reveal specific characteristics, which an knowledge engineering methodology should be aware of. In the following, we describe briefly some specific characteristics of Knowledge Engineering important for RapidOWL.

Knowledge Engineering is not a Business in itself. There is no market for Knowledge Engineering as there is for Software Development. This is not because Knowledge Engineering is less important in the economic sphere, but due to the fact that the flow of knowledge in most cases accompanies the development of products and services, rather than being an economic asset itself. Hence, Knowledge Engineering services are often required when spatially distributed users have to collaborate on a semantic level. For example, this is the case when a common terminology has to be established, dispersed information must be integrated, or when shared classification systems and taxonomies have to be developed. This type of semantic cooperation is for example often required for Virtual Organizations [1], scientific communities or standardization boards, or intra-organizational use.

Lack of a Unique Knowledge Serialization. Agile methodologies rely heavily on sophisticated versioning and evolution strategies due to their focus on small incremental changes. However, agile methodologies, as well as their respective versioning and evolutions strategies within software development, do not seem to be reasonably applicable to knowledge engineering. For example, contrary to software development paradigms, most knowledge representation paradigms do not provide unique serializations. In other words, the ordering of statements or axioms in a knowledge base is irrelevant, while the ordering of source-code lines in software is fixed. Consequently, the use of existing software versioning strategies (e.g. delta method) and their respective implementations (e.g. CVS, Subversion) would not be efficiently suitable.

Spatial Separation of Parties. Most agile Software Development methodologies assume a small team of programmers working closely (especially spatially) with domain experts. This is a reasonable assumption for commercial software development, where a client requests software developers to implement a certain functionality. But when the involved parties are spatially separated, the use of a formal, tool-supported Knowledge Engineering methodology becomes particularly important. Furthermore, the knowledge engineering tasks of establishing common classification systems, shared vocabularies and conceptualizations are especially important in distributed settings. When teams are co-located implicit knowledge representation in the form of text documents in conjunction with verbal communication turns out to be more efficient and for a long time established.

Involvement of a Large Number of Parties. The growing together of the world by Internet and Web technologies enabled completely new mechanisms of collaboration. Open source software projects as for example the Linux kernel or collaborative content authoring projects as Wikipedia demonstrate this power of scalable collaboration impressively. However, Knowledge Engineering is especially challenging when a large number of domain experts have to be integrated into the knowledge-engineering process. Agile software development methodologies claim to be best suited for small to medium sized development scenarios. This is mainly due to the accent on and need for instant communication. On the other hand, the interlinking of people and tools using internet technologies facilitates scaling of agile cooperation scenarios. Knowledge Engineering scenarios in most cases differ from software development scenarios: it is usually not optional, but crucial to integrate a large number of domain experts, knowledge engineers and finally users of the knowledge bases.

2 Aim of RapidOWL

The aim of this paper is to help make the development and use of knowledge bases more efficient. For that purpose, a new, agile knowledge engineering methodology, called RapidOWL is proposed. RapidOWL is founded on the observation that knowledge must necessarily be modeled evolutionary, in a close collaboration between users, domain experts and knowledge engineers. We argue that existing heavy-weight development methodologies from Software Engineering and Knowledge Engineering are inefficient for certain application scenarios, because they make changes in knowledge models too expensive. Most existing Knowledge Engineering methodologies (e.g. Uschold [19], Grüninger and Fox [10], Methontology [7]) take a task as the starting point, i.e. they suggest performing ontology construction with the ontology's usage scenarios in mind. This requires significant initial effort and makes changes to and reuse of the resulting ontologies inherently hard (cf. [13]). The starting point of RapidOWL is the hypothesis that light-weight (or agile) development processes can be suitable for knowledge engineering, because they stress the importance of supporting frequently changing requirements and models and rely on a minimum of representation artifacts. The application of RapidOWL is supported by concepts for knowledge base versioning and evolution (see [4]) as well as rapid querying and view generation. Both are accompanied by a framework supporting the development of semantic-web applications on the basis of the RDF statement paradigm (see [2,3]). Applications and examples are presented how efficient tool support for RapidOWL can then be implemented on top of the framework in either generic or domain specific way (e.g. [5]).

3 Related Work

Related approaches can be roughly classified into two groups. Accompanied by the formation of knowledge engineering as an independent field of research several Knowledge Engineering methodologies were developed. Most of them are much inspired by Software Engineering methodologies. In the Software Engineering domain, in the 90's several Agile Software Engineering methodologies emerged. Triggered by the fact that flexibility, in particular fast and efficient reactions on changed prerequisites, becomes increasingly important, agile methodologies recently also appeared in other areas than Software Engineering. The most prominent representatives from each of these directions are briefly presented in the following.

Knowledge Engineering. The main goal of Knowledge Engineering is to structure the development and use of knowledge bases. For that purpose, the most widely known Knowledge Engineering approaches (such as CommonKADS [17]) are based on the ontology paradigm (i.e. knowledge should be represented in formal and explicit specifications [12]). Ontologies capture the semantics of the knowledge in a format that is designed to be on the one hand easy to maintain and on the other hand efficient to process by reasoning algorithms. The development of both ontologies and adequate reasoning algorithms is supported by various methodologies, the phases and models of which resemble traditional Software Engineering approaches. These Knowledge Engineering methodologies now also reveal similar problems to traditional Software Engineering approaches. Significant initial efforts are needed to make the purpose of the final ontology explicit and to deduce an appropriate model. It is often hard to estimate the required level of detail for the knowledge structuring a priori. Changes to the knowledge structuring are difficult and costly. Finally, only a few knowledge modeling tools allow easy collaboration between domain experts and knowledge engineers. For these reasons, methods from Knowledge Engineering are often too expensive to apply and rarely used in practice.

Agile Methodologies. Agile methodologies have recently gained growing success in many economic and technical spheres. This is due to the fact that flexibility, in particular fast and efficient reactions to changed prerequisites, is becoming increasingly important in the information society. This development started in Software Engineering after the realization in the mid 1990's that the traditional 'heavy' methodologies do not work well in settings where requirements are uncertain and change frequently. Several adaptive or agile Software Engineering methodologies subsequently evolved (see [6,8,18,15,16]). Agile methodologies are especially suited for small co-located teams and for the development of non life-threatening applications. Since the problem of uncertain, changing requirements is not limited to the Software Engineering domain, the idea of establishing adaptive methodologies, which can react to changing prerequisites, was also adopted by other domains than Software Engineering. These include 'The Wiki Way' [14] for Content Management, Rapid Prototyping [11] for Industrial Engineering. Also, the Lean Management method was used to some extent in the business management domain.

4 Results

The RapidOWL methodology is based on the idea of iterative refinement, annotation and structuring of a knowledge base. Its aim is to bring about a stable state of the knowledge base through small incremental changes from a multiplicity of contributors. A central paradigm for the RapidOWL methodology is the concentration on smallest possible information chunks (i.e. RDF statements). The collaborative aspect comes into play, when those information chunks can be selectively added, removed, annotated with comments or ratings. Design rationales for the RapidOWL methodology are to be light-weight, easy-to-implement, and support of spatially distributed and highly collaborative scenarios. The RapidOWL methodology is presented following other agile methodologies. It is grounded on the paradigms of the generic architecture of knowledge-based systems, knowledge representation for the semantic-web on the basis of the RDF statement paradigm and web technologies. The RapidOWL process then is characterized by values from which (on the basis of the paradigms) principles are derived for the engineering process in general, as well as practices for establishing those principles in daily life (cf. Fig. 1). The values of RapidOWL are Community (to enable collaborative knowledge base development and evolution), Simplicity (to increase knowledge base maintainability), Courage (to be able to escape representation dead-ends) and Transparency (to promote early detection of modeling errors). The practices envisioned to organize the Knowledge Engineering process in daily life include among others: Joint Ontology Design (to ease collaboration between knowledge engineers, domain experts and users), Information Integration (to ground the knowledge elicitation on existing information), View Generation (to provide domain specific views for human users and software systems) and Ontology Evolution (enabling the smooth adoption of modelings and corresponding instance data migration). These values, principles, and practices are the major ingredients of the RapidOWL methodology. In contrast to systematic engineering methodologies, RapidOWL does not prescribe a sequence of modeling activities that should be precisely followed. Furthermore, RapidOWL does not waste resources on comprehensive initial analysis and design activities. RapidOWL is primarily suited for establishing conceptualizations for information integration as well as the establishing of shared classification systems and vocabularies. The idea of applying agile paradigms to Knowledge Engineering with respect to the specific characteristics of Knowledge Representation is the major innovation of the presented approach.

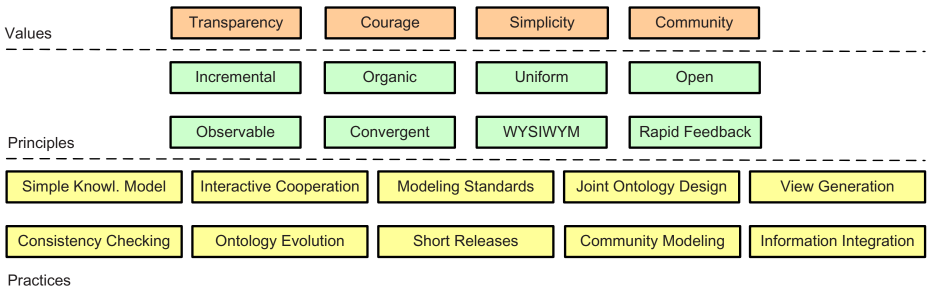


Fig. 1. The building blocks of RapidOWL: Values, Principles, Practices

In [9] a number of criteria for analyzing methodologies was proposed. In the following we discuss RapidOWL in the light of these criteria.

Detail of the methodology. RapidOWL is a rather lightweight methodology. This is primarily due to the recognition that knowledge engineering is usually not a business in itself and thus significant resources for evaluating the methodology and later controlling the compliance of the processes with the methodology are not available. RapidOWL rather banks on tools supporting it than on exhaustive documentation.

Recommendations for knowledge formalization. RapidOWL bases on representation of all knowledge in the form of triples, i.e. RDF statements. A concrete degree of formalization is not prescribed. However, RapidOWL proposes to justify the degree of formalization according to the required reasoning capabilities of the resulting knowledge base.

Strategy for building ontologies. Regarding this criteria it is questioned whether the strategy to develop ontologies is (a) application-dependent, (b) application-semidependent, or (c) application-independent. RapidOWL focuses on the development of rather application-independent ontologies. However, RapidOWL is primarily suited for information integration tasks and tasks related to the establishing of shared classification systems, vocabularies and conceptualizations.

Strategy for identifying concepts. RapidOWL here follows a middle-out strategy, i.e. from the most relevant to the most abstract and most concrete. By stressing the collecting of example or instance data RapidOWL tries to abolish knowledge elicitation by means of face-to-face communication between domain experts and knowledge engineers.

Recommended life cycle. Due to its adaptive nature RapidOWL does not explicitly propose a rigid life cycle. However, many aspects of stages in the life cycle of conventional methodologies can be discovered in RapidOWL's single process.

Differences between the methodology and IEEE 1074-1995. This criteria is related to the conviction that knowledge engineering processes should be similar to conventional software development processes. In this regard RapidOWL is different in two ways: Firstly it stresses the need to react on changed prerequisites, i.e. being agile. Secondly it assumes knowledge engineering to be fundamentally different from software engineering in certain scenarios.

Recommended techniques. RapidOWL stresses the importance of providing concrete techniques for performing the different practices of which the methodology is composed. However, in the description of RapidOWL's practices within this document only starting points on how to put them into effect are mentioned.

Usage and Application. Due to the fact that RapidOWL is rather new and significant resources had not been at our disposal for a broad evaluation the number of successfully realized RapidOWL projects is still small. However, ontologies and applications have been build on the basis of RapidOWL containing approximately 20,000 concepts and serving 3,000 parties (see e.g. [5]).

References

1. Wolfgang Ph. Appel and Rainer Behr. Towards the theory of virtual organizations: A description of their formation and figure. Arbeitspapiere wirtschaftsinformatik, Justus-Liebig-Universität Gießen Fachbereich Wirtschaftswissenschaften, 12 1996.
2. Sören Auer. Powl: A web based platform for collaborative semantic web development. In Sören Auer, Chris Bizer, and Libby Miller, editors, *Proceedings of the Workshop Scripting for the Semantic Web*, number 135 in CEUR Workshop Proceedings, Heraklion, Greece, 05 2005.
3. Sören Auer, Sebastian Dietzold, and Thomas Riechert. Ontowiki - a tool for social, semantic collaboration. In I. Cruz et al., editor, *Proc. of 5th International Semantic Web Conference, Athens, GA, USA, Nov 5th-9th*, number 4273 in LNCS, pages 736–749. Springer-Verlag Berlin Heidelberg, 2006.
4. Sören Auer and Heinrich Herre. A versioning and evolution framework for rdf knowledge bases. In *Proc. of Sixth International Conference Perspectives of System Informatics, Novosibirsk, Russia, Sep 27-30*, 2006.
5. Sören Auer and Bart Pieterse. "vernetzte kirche": Building a semantic web. In *Proceedings of ISWC Workshop Semantic Web Case Studies and Best Practices for eBusiness (SWCASE05)*, 2005.
6. Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, Second Edition.* Addison Wesley Professional, 2004.
7. Mercedes Blázquez, Mariano Fernández, Juan Manuel Garcia-Pinar, and Asunción Gómez-Pérez. Building ontologies at the knowledge level using the ontology design environment. In *Proceedings of the Eleventh Workshop on Knowledge Acquisition, Modeling and Management (KAW-98), Banff, Canada*, 1998.
8. Alistair Cockburn. *Crystal Clear.* Addison-Wesley Professional, 2004.
9. Mariano Fernández-López. Overview of methodologies for building ontologies. In *IJCAI99 Workshop on Ontologies and Problem-Solving Methods: Lessons Learned and Future Trends*, 1999.

10. Mark S. Fox and Michael Gruninger. Methodology for the design and evaluation of ontologies. In *Proceedings of the IJCAI-95 Workshop on Basic Ontological Issues in Knowledge Sharing, Menlo Park, USA*. AAAI Press, 1995.
11. Andreas Gebhardt. *Rapid Prototyping*. Hanser Gardner Pubns, 2003.
12. Tom R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, June 1993.
13. Holger Knublauch. *An Agile Development Methodology for Knowledge-Based Systems*. PhD thesis, University of Ulm, 2002.
14. Bo Leuf and Ward Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley Professional, 2001.
15. Ken Orr and James A. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Co, 2000.
16. Stephen R. Palmer and John M. Felsing. *A Practical Guide to the Feature-Driven Development*. Prentice Hall PTR, 2002.
17. Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, N. Shadbolt, Walter Van de Velde, and Bob J. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology.* MITpress, 2000.
18. Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 1st edition, Oct 2001.
19. Mike Uschold. Building ontologies: Towards a unified methodology. In *16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems*, Cambridge, UK, 1996.

BURS-Based Instruction Set Selection

Dmitri Boulytchev

St.Petersburg State University
198504, Universitetskii pr., 28
St.Petersburg, Russia
db@tepkom.ru

Abstract. Application-specific processors (ASIPs) look very promising as a platform for embedded systems since they comprise both the flexibility of a programmable device and the efficiency of application-specific hardware. A number of approaches to design an application-specific instruction sets were introduced during the last years. We apply the BURS (Bottom-Up Rewrite System) technique which is commonly used for retargetable code generation to this problem. As a result a simple algorithm is presented that generates both instruction set and assembly code from the source program; this algorithm can be used for retargetable code generation as well.

1 Introduction

Two main tasks have to be solved during the development of the embedded system based on an application-specific instruction set processor. On the one hand the suitable architecture has to be developed; on the other hand the compiler has to be retargeted to this architecture. Solving the first task independently from the second can make efficient code generation surprisingly hard. Hardware engineers strive to invent the most appropriate hardware for the given set of tasks; however as a rule their considerations can hardly be effectively utilized by a compiler. The reason is that the compiler has to generate good code for *any* program while instruction set design has been performed with regard to a particular application or a group of applications.

One way to avoid this glitch is to synthesize machine code together with an appropriate instruction set directly from source application i.e. to perform *instruction set selection*. Many different approaches have been developed in this area; we suggest yet another one.

Bottom-Up Rewrite System (BURS) is a simple conventional model for retargetable code generation. Crafting BURS to synthesize both machine code and instruction set we constructed an algorithm which provides for a given instruction-set constraint an instruction set as a set of tree patterns minimizing the cost of the machine program among all instruction sets satisfying that constraint. The algorithm can be used without any detailed knowledge of hardware features. Since these features are not yet clarified due to the lack of the hardware this does not restrict design space.

In comparison to other approaches the suggested one may be considered as a *prototyping step* since it allows to implement executable prototype consisting of machine code and hardware instruction set model from scratch using only high-level software implementation. While generally speaking such a prototype may lack some important features (parallelism, clever resource allocation etc.) then it may be optimized both on software and hardware levels.

2 Related Works

A number of approaches to design an application-specific instruction sets were introduced during the last years.

Instruction set synthesis for pipelined architectures is addressed in [14,13]. For a given parameterized pipelined microarchitecture and the set of benchmarks the design of instruction set is considered as a scheduling problem. To control the tradeoff between instruction set and program quality the number of cycles in the program and the number of instructions in the instruction set are used. Scheduling problem is solved by the simulated annealing algorithm.

Another approach [16] uses combined representation of a datapath and instruction set model. *Bundling* technique is explored to couple sequences of micro-operations and construct datapath. Initial datapath samples are taken from a predefined library; profiling is performed to determine frequently occurring operation sequences.

Template generation as a way for instruction set selection is considered in [15, 3]. Templates are repeated occurrences of possibly interdependent nodes and edges in a dataflow graph. Two types of templates are identified: sequential and parallel. Templates are built iteratively starting from the pairs of adjacent nodes; initially the most frequent pairs are selected. All selected templates are combined into *supernodes*; therefore DAG isomorphism algorithm is used to detect higher-order templates.

Another graph-based approach to instruction-set selection is presented in [6]. All reasonable patterns with regard to the set of constraints are enumerated to collect an instruction candidates. Instruction set selection guided by some cost function is then performed. Finally the target application is mapped into the selected patterns by *binate covering* algorithm. The similar approach is used in [4].

3 Tree Grammars and BURS

Our approach to instruction set selection is based on tree grammars and BURS theory so we have to mention some formal notions.

A tree grammar is a context-free grammar $G = (N, T, S, R)$ where N and T are finite sets of nonterminals and terminals, $S \in N$ is starting nonterminal, R is a set of rules of the form

$$K : p$$

where $K \in N$ is nonterminal, p is *tree pattern* — an (ordered) tree with all interior nodes labeled with terminals and all leaves labeled with either terminals or nonterminals. A *ground tree* is a pattern without nonterminal-labeled leaves. While ordinary "linear" grammars define languages as sets of words tree grammars define languages as sets of ground trees. Similarly to the "linear" case one may define the transition relation $p \xrightarrow{r} q$ for a rule r and two patterns p and q as follows:

$$p \xrightarrow{r} q \text{ iff } q = p[l \leftarrow t]$$

where r is $K : t$, l is a leaf of p with label K, $p[l \leftarrow t]$ is a result of substitution of l with t. The language defined by a tree grammar G is the set of ground trees

$$\mathcal{L}(G) = \{t : S \rightarrow^* t\}$$

where "$\rightarrow^*$" is a reflexive-transitive closure of "$\rightarrow$". More detailed description of tree languages and automata theory can be found in [5].

For any ground tree its derivation corresponds to some partition by patterns of G. This observation explains why tree grammars are used as a convenient formal model for the instruction selection problem: if we interpret patterns of the grammar as machine instructions and nonterminals as storage classes then any derivation of the tree corresponds to some feasible instruction selection. Under these assumptions the problem of optimal instruction selection for trees is the problem of finding the *least-cost derivation* in a tree grammar with weighted rules.

The least-cost derivation can be found by a dynamic programming algorithm that in fact is a variant of Aho-Johnson algorithm for finding optimal code for expression trees [2]; unlike the latter one, however, it does not yield optimal register allocation.

The search is performed in two stages. During the first one, *labeling*, all possible rules are applied in a bottom-up manner and the costs of all derivations are calculated. During the second stage, *reduce*, the least-cost derivation is reconstructed by a top-down walk.

Applications of tree grammars to pattern matching and code generation are discussed in more details in [12,1,9,10].

4 Instruction Set Selection as a BURS Problem

As we have seen in the previous section, optimal instruction selection for a tree can be considered as a search of least-cost derivations in weighted tree grammars. Patterns of the grammar play role of individual machine instructions; any derivation introduces a partition of the source tree. The inverted claim obviously does not always hold: for an arbitrarily chosen partition, there might not be a derivation. Since any partition corresponds to some instruction selection with regard to some instruction set our first task is to build for given tree a grammar which allows all its partitions. We call such grammars *enumerating.*

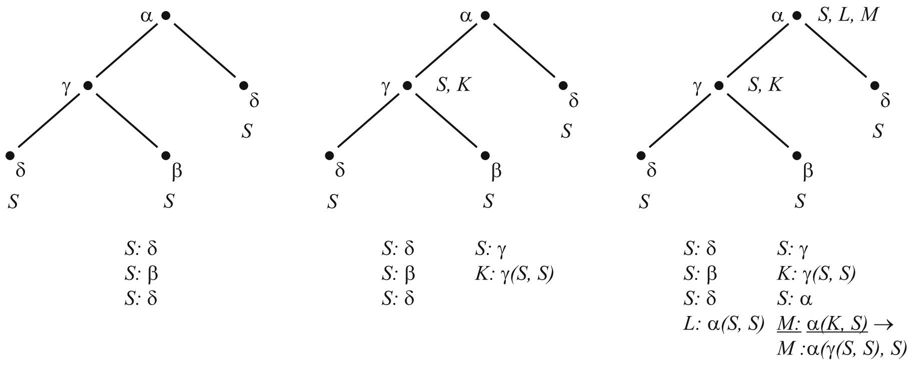


Fig. 1. An example of enumerating grammar construction

This property of enumerating grammars can be interpreted as follows: for given tree t they describe *all possible* instruction selections in *all possible* instruction sets. So least-cost derivation in enumerating grammar is the least-cost instruction selection among all instruction sets.

The enumerating grammar for the given tree t can easily be built via bottom-up breadth-first traversal of the tree t. For each node v with terminal label x we first add a rule $S : x$ to the grammar, where S is starting nonterminal of the grammar; then we apply all rules exactly as during labeling stage of the BURS algorithm; finally we enrich the grammar with rules of the form $K : x(L_1, \ldots, L_k)$, where K is new nonterminal, L_i is nonterminal mark of i-th immediate successor of v and there is no rule with the same pattern yet. An example of enumerating grammar construction is presented in Fig. 1.

After construction of the enumerating grammar one may apply the conventional BURS algorithm to perform instruction selection. The costs of the rules may be assigned in the commonly used manner; for example to minimize code length the cost of 1 has to be assigned to each rule. Least-cost derivation for enumerating grammar yields the best instruction selection among all instruction sets with the regard to constraint used during grammar construction. To select an instruction set we finally remove from the enumerating grammar all rules that do not participate in the least-cost derivation.

The main restriction of the approach being discussed is introduced by limitation of cost function that has to be compliant with dynamic programming algorithm. It is quite simple to estimate the cost of the a program and quite hard to estimate the cost of an instruction set with the function of that kind. On the other hand we can not ignore instruction set cost. For example the standard cost function of a program is its length; obviously the shortest program for given tree among all possible instruction sets contains one instruction — the tree itself. To avoid such degenerative instruction sets from being considered we restrict the set of patterns that can be used in enumerating grammars by means of a constraint — some property that can be effectively checked for a tree.

Table 1. Evaluation Results for the DSPstone Benchmarks

benchmark	**size**	*triad*			*dual mem*			*mem/mul*			*no constraint*		
		I	*A*	*R*	*I*	*A*	*R*	*I*	*A*	*R*	*I*	*A*	*R*
complex_multiply	58	5	38	11	5	36	3	5	38	11	4	34	9
complex_update	78	6	62	14	6	50	4	6	62	14	5	46	13
convolution	70	6	42	8	5	39	2	6	42	8	5	39	7
dot_product	78	6	36	9	6	33	4	6	36	9	5	32	8
fir	83	8	54	12	7	48	2	8	54	12	7	48	11
fir2dim	149	8	147	23	8	136	4	8	147	23	6	132	21
biquad_one_section	86	8	57	14	7	48	2	8	57	14	7	48	14
biquad_N_sections	117	12	91	20	12	76	4	12	91	20	12	73	19
lms	156	9	77	15	9	65	4	9	77	15	8	64	14
matrix	76	9	87	18	9	74	4	9	87	18	8	72	17
matrix1x3	130	6	38	10	6	35	4	6	38	10	5	34	9
n_complex_updates	89	7	101	20	7	89	4	7	101	20	6	85	19
n_real_updates	69	6	56	11	6	53	4	6	56	11	5	52	10
real_update	66	5	28	5	4	25	4	5	28	5	3	24	4
fft	125	38	235	41	42	189	4	35	230	41	42	184	40
g721	866	84	1547	75	149	1049	3	101	1332	75	136	1002	73

During the construction of enumerating grammars we do not consider patterns that violate the chosen constraint.

5 Evaluation

We implemented the described algorithm on top of an ASDL-port of the lcc compiler [8,7,11]. In addition to the instruction set selection we had to implement local register allocator to store intermediate values and build a program from partition. The classic algorithm of Aho and Johnson [2] was used for this purpose.

We ran our algorithm for the DSPstone benchmark [17]. Four types of constraints were used:

1. *triad* — the number of nodes in a pattern is limited by 3, which forces instruction set selector to generate triad-based instruction set;
2. *no constraint* — any pattern is allowed so each tree becomes separate machine instruction;
3. *dual mem* — no more than one occurrence of multiplicative, logic or shift operations and no more than two occurrences of memory access operations are allowed in the pattern;
4. *mem/mul* — same as the above, but no simultaneous memory access operation and multiplication are allowed.

The length of the generated program was used as its cost function.

The results of the evaluation are shown in Table 1. Here I, A, and R stand for instruction-set size, machine program length, and number of registers. The number of registers is relatively large since we did not perform any clever global register allocation but assigned a dedicated register for each local variable, parameter, or temporary. Such a naive allocation does not affect the instruction set selection results. Since our algorithm works only on basic blocks we use some predefined set of control flow instructions to express the control flow in the generated program; these instructions are not included in the presented results. Despite the complexity of the algorithm running time for all these benchmarks did not exceed three seconds on Pentium-III 800 MHz 512MB RAM workstation running Linux.

6 Discussion and Future Work

We see several drawbacks of the presented approach.

First of all, the model of tree covering looks restrictive since even within basic blocks any program is generally represented by a DAG. One has to perform common expression elimination to turn this DAG into forest. We argue that due to the simplicity of the algorithm we may perform various ways of common subexpression elimination prior to the instruction set selection.

The second drawback is that each instruction is implied to be a tree. We think there is no hope to adjust this algorithm to handle instructions as DAGs; however we may suggest to perform instruction set optimization to join simpler instructions into complex ones.

The cost function for instruction sets is quite poor in the current implementation; it does not even allow to select the shortest instruction set among all sets that yield the same program quality. We think though that the algorithm can be extended to handle more sensible instruction-set costs.

These issues are subjects of future research.

On the other hand presented approach has some merits: it allows to synthesize both instruction set and machine code without any special knowledge of processor architecture and so it can be used to discover some intrinsic properties of the target algorithm implementation.

As a member of BURS family the algorithm produces provably-optimal code with regard to all formulated restrictions. Note that the register allocator may be adjusted to avoid introducing anti-dependencies into generated code; thus the scheduling will not require instruction reselection.

Finally, for a fixed instruction set IS this algorithm can easily be turned into code generator: it is enough to specify constraint of the form $p \in IS$, where p is an instruction candidate.

References

1. A. V. Aho, M. Ganapathi, and S. W. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.
2. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1979.

3. P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.
4. N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. *Proc. 36th International Symposium on Microarchitectures*, 2003.
5. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata/, 2002.
6. J. Cong, Y. Fan, G. Han, and Z. Z. Z. Application-specific instruction generation for configurable processor architecture. *Proc. of the 12th International Symposium on Field Programmable Gate Arrays*, pages 183–189, 2004.
7. C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, 1991.
8. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.
9. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, (3):213–226, 1992.
10. C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.
11. D. R. Hanson. Early experience with ASDL in lcc. *Software — Practice & Experience*, 29(5):417–435, 1999.
12. C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1983.
13. I. J. Huang and A. M. Despain. Generating instruction sets and microarchitectures from applications. *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 391–396, 1994.
14. I. J. Huang and A. M. Despain. Synthesis of instruction sets for pipelined microprocessors. *Proc. of the 31th ACM/IEEE Design Automation Conference*, pages 5–11, 1994.
15. R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. on Design Automation of Electronic Systems*, 7(4):605–627, 2002.
16. J. V. Praet, G. Goossens, D. Lanneer, and H. de Man. Instruction set definition and instruction selection for ASIPs. *Proc. of the 7th ACM/IEEE International Symposium on High-Level Synthesis*, pages 11–16, 1994.
17. V. Zivojnovic, J. M. Velarde, C. Schlaeger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. *Proc. International Conference on Signal Processing Applications and Technology*, 1994.

Improved Resolution-Based Method for Satisfiability Checking Formulas of the Language L

Anatoly Chebotarev and Sergey Krivoi

Glushkov Institute of Cybernetics Ukrainian Academy of Sciences
Glushkov's pr., 40, Kiev, 03187, Ukraine
ancheb@gmail.com, krivoi@i.com.ua

Abstract. The language L is used for specifying finite automata, and is a fragment of a first order language with monadic predicates. Checking specification for satisfiability plays an important role in the development of reactive algorithms. Restricted syntax of this language and interpreting it over the integers make it possible to substantially improve resolution-based methods for satisfiability checking. This has been done in previous papers devoted to R- and S-resolution. In this paper, we present yet another improvement based on the restriction of the type of atoms upon which the resolution is allowed.

1 Introduction

The language L is used as a specification language in the system for provably correct design of reactive algorithms from their logical specifications [1]. This language is a subset of a first-order language with monadic predicates interpreted over the set of integers. Checking specifications for satisfiability plays an important role in the design process. The corresponding procedure is used in almost all specification transformations not only to detect the internal inconsistency of the specification but also to verify the designer's decisions (changes in the specification) in the course of interactive development of the algorithm [2,3]. So, a rather high efficiency of satisfiability checking algorithm is required. The algorithm we propose here is based on the resolution inference search procedure. The main reason of inefficiency of resolution-based methods is generating a large number of redundant clauses during the inference search process. A reduction in the number of generated clauses is attained by imposing various restrictions on the application of the resolution rule. An efficient method for checking satisfiability of a language L formula was suggested in [4], where the simplification of the corresponding procedure was achieved by means of restricting the type of atoms upon which the resolving is allowed. The resolution rule in this method was called R-resolution. Additional improvements to this method were made in [5], where a set of clauses was partitioned into several classes and resolving was only allowed between clauses belonging to the same class. This results in significant reduction in the amount of the generated clauses as compared with the R-resolution method. In this paper, we present one more improvement to this method connected with the additional restriction on the type of atoms upon which the resolving is allowed.

2 Basic Notions

First, we recapitulate briefly the basic notions concerning the language L and R-resolution. For more details the reader is referred to [4]. Let T be a class of formulas constructed by means of logical connectives from atoms of the form $p(t+k)$, where p is a monadic predicate symbol, t is a variable ranging over the set of integers $\mathbf{Z}$, and k is an integer constant called the rank of the atom. The language L consists of the formulas of the form $\forall t F(t)$, where $F(t) \in T$ and is interpreted on $\mathbf{Z}$. The example of such a formula is $\forall t(y(t-1)\&x(t) \rightarrow y(t))$, where y and x are predicate symbols and $(t-1)$ is an abbreviation for $(t+(-1))$.

A formula $\forall t F(t)$ is called satisfiable if it has a model, i. e. the interpretation in which it evaluates to true. Since $F(t)$ is interpreted over the set of integers the equivalence $\forall t F(t) \leftrightarrow \forall t F(t+k)$, where $F(t+k)$ denotes the formula obtained from $F(t)$ by adding k to the ranks of all its atoms, holds for any integer k. So, we may assume that the maximum rank of atoms occurring in any formula is equal to 0. Such formulas will be referred to as right-normalized. The formula $F(t)$ in the specification is assumed to be represented in the conjunctive normal form which is viewed as a set of clauses, i. e. disjunctions of literals, where a literal is an atom or its negation. A clause containing no literals is called an empty clause (denoted by $\square$). A set of clauses is called right-normalized if each its clause is right-normalized.

Definition 1. *Let $c_1 = c \vee p(t)$, $c_2 = c' \vee \neg p(t)$ be right-normalized clauses, where $p(t)$ is an atom of rank 0. The clause $c \vee c'$ is called an R-resolvent of c_1 and c_2 upon the atom $p(t)$.*

R-resolution (restricted resolution) is an inference rule which only admits resolving upon atoms of rank 0.

Definition 2. *An R-deduction of a clause c from a set of clauses C is a finite sequence of clauses $c_1, \ldots, c_m$ such that $c_m = c$ and each c_i $(i = 1, \ldots, m)$ either belongs to C, or is an R-resolvent of c_j and c_k for j, $k < i$, or is a result of right-normalization of c_{i-1}.*

Definition 3. *A clause $c_1(t)$ subsumes a clause $c_2(t)$ if there exists $k \in \mathbf{Z}$ such that $c_1(t+k)$ is a subset of $c_2(t)$.*

A clause set C is called unsatisfiable if it specifies an unsatisfiable formula $\forall t F(t)$. The following proposition has been proved in [4].

Proposition 1. *A set C of right-normalized clauses is unsatisfiable if and only if there exists an R-deduction of the empty clause from C.*

The corresponding procedure checking a set of clauses for satisfiability is called an R-completion procedure. In this procedure, the clauses subsumed by other clauses are removed after adding each new clause to the current set of clauses.

3 Separate Resolution Method

Let $p_1 < p_2 < \ldots < p_n$ be an ordering of predicate symbols occurring in the right-normalized set of clauses C. This ordering of predicate symbols is associated with the partition of C into subsets C_i $(i = 1, \ldots, n)$, where C_n consists of all the clauses which contain the atom $p_n(t)$ (the literal $p_n(t)$ or $\neg p_n(t)$) and C_i $(i = 1, \ldots, n-1)$ consists of all the clauses that do not belong to any C_j $(j > i)$ and contain the atom $p_i(t)$.

Definition 4. *An S-deduction of a clause c from the set of clauses C is such an R-deduction of c from C, where the R-resolution rule is only applied to the clauses in the same subset C_i and the only atom the clauses from C_i are resolved upon is $p_i(t)$.*

The method is based on the following theorem.

Theorem 1. *If C is an unsatisfiable set of right-normalized clauses, then for any ordering of the predicate symbols occurring in C, there exists an S-deduction of the empty clause from c.*

First we prove the following proposition.

Proposition 2. *For any right-normalized clause set C with ordered predicate symbols and a clause c containing no atoms of rank grater than -1, the existence of an R-deduction of c from C implies the existence of its S-deduction.*

The validity of Theorem 1 immediately follows from this proposition since the empty clause does not contain any atoms.

To prove Proposition 2, it suffices to consider an R-deduction that does not contain clauses obtained by application of the right-normalization operation. We shall refer to such an R-deduction as a simple R-deduction. Indeed, if we define appropriately a notion of the depth of a deduction (for example, the number of successive applications of the right-normalization operation in the deduction tree) we can easily show by induction on the depth of the deduction that if Proposition 2 holds for a simple R-deduction it also holds for any other R-deduction.

Let c be a clause that does not contain atoms of rank grater than -1. Consider a simple R-deduction of c from the clause set $C = \{c_1, \ldots, c_n\}$. In such a deduction, all clauses, except the last, contain atoms of rank 0. With every clause c_i of this deduction we associate a clause c'_i consisting of all literals of rank 0 contained in the clause c_i. The sequence of clauses c'_i corresponding to the R-deduction of clause c is an R-deduction of $\Box$ from the clause set $C' = \{c'_1, \ldots, c'_n\}$. It is easy to show that if there exists an S-deduction of $\Box$ from C', then there also exists an S-deduction of c from C. Thus the problem is reduced to the propositional case.

We now consider an unsatisfiable set of clauses C' whose atoms are propositional variables ordered in the following way: $p_1 < p_2 < \ldots < p_q$. The existence of an S-deduction of $\Box$ from C' follows from the Davis-Putnam method [6] which can be reformulated as follows.

Proposition 3. *Let C' be a set of propositional clauses and p any propositional variable occurring in C'. If all the resolvents upon variable p which are not tautologies are added to C' and all the clauses containing p or ¬p are removed, then the resulting set of clauses is unsatisfiable if and only if the original set is unsatisfiable.*

Let $W_1, W_2, \ldots, W_q$ be the partition of C' corresponding to the above ordering of the variables. The variable p_q is contained only in the clauses of W_q. Applying the rule of Proposition 3 to W_q and p_q we eliminate the variable p_q from the set of variables occurring in the resulting clause set. Next, we eliminate the variable p_{q-1} applying this rule to the clauses in W_{q-1}. Proceeding in this manner, we obtain an S-deduction of $\Box$ from C'. Thus, if there exists a simple R-deduction of a clause c containing no atoms of rank grater than -1 from the clause set C, then there exists an S-deduction of $\Box$ from C' and hence there exist an S-deduction of c from C. This completes the proof of Proposition 2 as well as Theorem 1.

We now can summarize the main features of the separate resolution method.

1. R-resolving is only allowed between clauses belonging to the same subset of the partition corresponding to the chosen ordering of predicate symbols.

2. In every subset C_i of the clause set partition, resolving is only allowed upon the atom $p_i(t)$.

3. The order in which the subsets of clauses are handled is not essential because the subsets are not removed after generating all resolvents upon the corresponding atom.

4. An S-resolvent c that is not a tautology and not subsumed by any other of the existing clauses is added to the corresponding subset (according to the partitioning rule) in the right-normalized form, and all the clauses in the current set of clauses that are subsumed by c are removed.

4 Example

Consider the partition of the clause set corresponding to the following ordering of its predicate symbols: $x < u < y < z$.

The subset C_4 (corresponds to z):
$(y(t-2) \vee \neg y(t-1) \vee z(t) \vee \neg u(t))$ 1,
$(\neg z(t-2) \vee \neg y(t-1) \vee \neg z(t) \vee y(t))$ 2,
The subset C_3 (corresponds to y):
$(z(t-1) \vee y(t) \vee \neg u(t))$ 3,
$(y(t-2) \vee \neg y(t-1) \vee \neg y(t) \vee u(t))$ 4,
$(z(t-1) \vee \neg y(t-1) \vee y(t) \vee \neg x(t))$ 5,
$(z(t-1) \vee y(t) \vee x(t))$ 6.
The subset C_2 (corresponds to u):
$(z(t-1) \vee \neg u(t-1) \vee \neg u(t) \vee x(t))$ 7,
$(z(t-1) \vee \neg y(t-1) \vee u(t-1) \vee u(t) \vee \neg x(t))$ 8.
The subset C_1 (corresponds to x):
$(\neg z(t-2) \vee \neg y(t-2) \vee u(t-1) \vee x(t))$ 9.

The number of the clause is written to the right of it, and pairs of numbers written to the left of the resolvents indicate the numbers of clauses being resolved.

The process of S-completion proceeds as follows.

(1, 2) $(\neg z(t-2) \vee y(t-2) \vee \neg y(t-1) \vee \neg u(t) \vee y(t))$ 10, is added to C_3.

(4, 5) $(y(t-2) \vee z(t-1) \vee \neg y(t-1) \vee \neg x(t) \vee u(t))$ 11, is added to C_2.

(4, 6) $(y(t-2) \vee z(t-1) \vee \neg y(t-1) \vee x(t) \vee u(t))$ 12, is added to C_2.

(7, 12) $(y(t-2) \vee z(t-1) \vee \neg u(t-1) \vee \neg y(t-1) \vee x(t))$ 13, is added to C_1.

The process terminates after generating four resolvents while in the process of R-completion 35 clauses are generated.

5 Conclusion

We have proposed an efficient resolution based method for satisfiability checking specifications in the language L. This method leads to significant reduction in the number of clauses generating during the satisfiability checking in comparison with the method of R-resolution. One more factor ensuring the efficiency of the method is reduction in the number of clause pairs checking for the possibility to be resolved. The result proved in the paper may be regarded as the proof of completeness of the strategy combining a predicate symbols ordering with R-resolution.

It should be noted that different orderings of predicate symbols lead to different partitions of the clause set that may result in different numbers of clauses generated in the process of satisfiability checking. In turn, different orders of subsets handling may lead to different run times of the procedure. As an appropriate heuristics we recommend to handle subsets of clauses C_i in the decreasing order of their subscripts.

References

1. A. Chebotarev. Provably-correct development of reactive algorithms. *Proc. Int. Workshop "Rewriting Techniques and Efficient Theorem Proving" (RTETP-2000), P. 117 - 133 (2000).*
2. A. Chebotarev. Determinisations of logical specifications of automata. *Cybernetics and Systems Analysis (translated from Russian), v.31, N1, P. 1 - 7 (1995).*
3. A. N. Chebotarev, M.K. Morokhovets. Resolution-based approach to compatibility analysis of interacting automata. *Theoretical Computer Science, 194, P. 183 - 205 (1998).*
4. A. N. Chebotarev, M.K. Morokhovets. Consistency checking of automata functional specifications. *Proc. LPAR'93, Lecture Notes in Artificial Intelligence, v. 698, Springer, Berlin, P. 76 - 85.(1993).*
5. A. Chebotarev. Separate Resolution Method for Checking the Satisfiability of Formulas in the Language L. *Cybernet. Systems Analysis (translated from Russian) v.34, N6, P. 794 - 799. (1998).*
6. C.L. Chang, R.C.T. Lee. Symbolic Logic and mechanical theorem proving. *Academic press. (1973).*

Real-Time Stable Event Structures and Marked Scott Domains: An Adjunction

R.S. Dubtsov

A.P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev avenue, 630090, Novosibirsk, Russia
Phone: +7 3833 30 40 47, Fax: +7 3833 32 34 94
dubtsov@iis.nsk.su

Abstract. Event structures constitute a major branch of models for concurrency. Their advantage is that they explicitly exhibit the interplay between concurrency and nondeterminism. In a seminal work, Winskel has shown that categories of prime and stable event structures can be related to a category of Scott domains by adjunctions. The intention of this note is to show the applicability of the theory-categorical framework to a real-time extension of stable event structures, in order to identify suitable semantical domains for the models. To that end, we first introduce a category **TSES** of real-time stable event structures, and a category **MDom** of a particular class of Scott domains, called marked Scott domains, and then define an adjunction between **TSES** and **MDom**.

1 Introduction

Category theory has been used to unify and classify models for concurrency — see [22] for a survey. The general idea is to formalise that one model is more expressive than another in terms of an 'embedding', most often taking the form of an adjunction.

Domain theory offers a global mathematical setting for sequential computation, and thereby sets programming languages in connection with each other; relates them with the mathematical worlds of algebra, topology and logic; and inspires programming languages type disciplines and methods of reasoning. Although the applicability of classical domain theory to the general theory of concurrent computation is arguable [17], it has been shown in [20] that Scott domains bear close categorical relationships with event structures (and therefore, via a chain of adjunctions, with Petri Nets [14,7]). Furthermore, semantical domains have been identified for prime event structure with priorities [5] and probabilistic event structures [19].

It is generally acknowledged that time plays an important role in many concurrent and distributed systems. This has motivated the extension of the theory of untimed systems to real-time setting. Timed extensions of interleaving models have been studied thoroughly in the last ten years (see, for instance, [1,8]). On the other hand, the incorporation of quantitative information into noninterleaving models has received scant attention: a few extensions are known of pomsets

[4], configurations [15], asynchronous transition systems [3,12], net processes [10], and event structures [9,12,18]. In [6] real-time prime event structures were related to Scott domains, obtaining a coreflection (a special form of adjunction) between categories of the models.

The intention of this note is to show the applicability of the theory-categorical framework to a real-time extension of stable event structures, in order to identify suitable semantical domains for the models. To that end, we first introduce a category **TSES** of real-time stable event structures, and a category **MDom** of a particular class of Scott domains, called marked Scott domains, and then define an adjunction between **TSES** and **MDom**.

The note is organized as follows. Sections 2 and 3 contain notions and notations related to real-time stable event structures and marked Scott domains, respectively. The interrelations between the models are established in Section 4. Conclusions are drawn in Section 5. Appendix A contains basic notions regarding partial orders and Scott domains.

2 Real-Time Event Structures

In this section, we introduce a real-time extension of Winskel's model of event structures [21] by equipping events with time delays.

We first recall the terminology concerning event structures. An event structure is a set of events together with relations of enabling and conflict. The enabling relation models causality, whereas the conflict relation expresses alternative choices between events. Two events which are neither causally dependent nor in conflict may occur concurrently. In this sense, event structures provide explicit and separate representations of causality, choice and concurrency.

An *event structure* is a tuple $S = (E, \#, \vdash)$, where E is a countable set of events; $\# \subseteq E \times E$ is a symmetric and irreflexive relation (the *conflict relation*); $\vdash \subseteq Con \times E$ is an *enabling relation* such that $X \vdash e \ \wedge X \subseteq Y \in Con \Rightarrow Y \vdash e$ (here Con is the set of finite conflict-free subsets of E, i.e. those finite subsets $X \subseteq E$ for which holds: $\forall e, e' \in X \diamond \neg(e\#e')$). An event structure is called *stable* if $X \vdash e \wedge Y \vdash e \wedge X \cup Y \cup \{e\} \in Con \Rightarrow X \cap Y \vdash e$. In the following, we will consider only stable event structures and call them simply event structures.

The set $Conf(S)$ of *configurations* of S consists of those $C \subseteq E$ which are *secured* (i.e. $\forall e \in C \diamond \exists \{e_0, \ldots, e_n = e\} \subseteq C$ s.t. $\forall i \leqslant n \diamond \{e_0, \ldots, e_{i-1}\} \vdash e_i$) and *conflict-free* (i.e. $\forall e,\ e' \in C \diamond \neg(e\#e')$).

We next present a real-time extension of event structures. In our model, a global clock [8,9,12,15] which approximates the time of every snapshot by a natural number is assumed, and we attach a global-clock delay to each event. The interpretation is that an event with a delay t becomes enabled at the time t since the start of the system and may happen at any time from t. Therefore, in our model all events are non-urgent [9,18], allowing idling to be modelled. The occurrences of events themselves take no time, i.e. events happen 'instantaneously' [8,9,15].

Let $\mathbb{N}$ be the set of natural numbers, and $\widetilde{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, where $\infty > n$ for any $n \in \mathbb{N}$.

Definition 1. *A real-time event structure is a tuple $TS = (S, \delta)$, where S is an event structure and $\delta : E \to \mathbb{N}$ is a* delay function*.*

A real-time event structure $TS = (S = (E, \#, \vdash), \delta)$ has *correct timing* iff for any $e \in E$ and $X \vdash_{min} e$[1] holds: $\forall e' \in X \diamond \delta(e') \leqslant \delta(e)$. In what follows, only real-time event structures with correct timing are considered.

A state of an execution of a real-time event structure $TS = (S, \delta)$ is called a *timed configuration* (C, t) which consists of $C \in Conf(S)$ and $t \in \widetilde{\mathbb{N}}$ such that $\delta(e) \leqslant t$ for all $e \in C$. Let $TConf(TS)$ denote the set of timed configurations of TS. Say that there is a *transition* from a timed configuration (C, t) to a timed configuration (C', t') and write $(C, t) \longrightarrow (C', t')$ iff $C \subseteq C'$ and $t \leqslant t'$. It is easy to see that $\longrightarrow$ induces a partial order on $TConf(TS)$.

We shall indicate that θ is a partial function from E_0 to E_1 by writing $\theta : E_0 \to_* E_1$. The image of a set $X \subseteq E_0$ under a partial function θ is denoted by $\theta X = \{\theta(e) \mid e \in X \text{ and } \theta(e) \text{ is defined}\}$.

Given real-time event structures $TS_i = (S_i = (E_i, \#_i, \vdash_i), \delta_i)$ $(i = 0, 1)$, a morphism from TS_0 to TS_1 is a partial function $\theta : E_0 \to_* E_1$ on events such that $(C, t) \in TConf(TS_0)$ implies $(\theta\, C, t) \in TConf(TS_1)$ and $\theta(e) = \theta(e') \Rightarrow e = e'$, for every $e, e' \in C$.

Lemma 1. *Real-time event structures with the morphisms defined above form the category* **<u>TSES</u>**.

3 Marked Scott Domains

In this section we provide notions and notations concerning marked Scott domains. We assume that the reader is familiar with basic domain theory. The definitions of the notions used here, that are related to partial orders and domains, can be found in Appendix A. Recall that a *Scott domain* is a consistently complete algebraic partial order. We are specially interested in a particular kind of Scott domains – finitary, coherent prime algebraic partial orders – which we call simply domains.

We now develop the notion of a *marked domain.* The intuition behind marked domains is rather simple. Prime intervals of a domain can be viewed as activity occurrences. A marking is used to distinguish between two types of activities: instantaneous and delayed, which are marked with 0 and 1, respectively.

Definition 2. *Let D be a domain and I be the set of prime intervals of D. A mapping $m : I \to \{0, 1\}$ is called a* marking*. A pair (D, m) is called a* marked domain*.*

[1] For $X \subseteq E$ and $e \in E$, $X \vdash_{min} e$ iff $X \vdash e \wedge \forall Y \subseteq X \diamond (Y \vdash e \Rightarrow Y = X)$.

Let (D, m) be a marked domain and $\sim$ be the equivalence relation on prime intervals of D. Then, (D, m) is called *correctly marked* iff m respects the $\sim$-relation, i.e $[c, c'] \sim [d, d'] \Rightarrow m([c, c']) = m([d, d'])$. Informally speaking, $\sim$-related prime intervals of a domain represent the same activity of a system, hence they are required to be equally marked. In the following, we shall consider only correctly marked domains and call them simply marked domains.

Let (D, m) be a marked domain. For an element $d \in D$ and a covering chain σ for d, define the *norm of* d *w.r.t.* σ as follows: $\|d\|_\sigma = \sum_{d_j \in \sigma \setminus \{d\}} m([d_j, d_{j+1}]) \in \widetilde{\mathbb{N}}$. The norm of d w.r.t. σ allows us to calculate the number of delayed activities appearing in σ. It is easy to show that the definition of the norm of element $\|d\|$ does not depend on the choice of a covering chain σ for d, since D is prime algebraic. Let $d, d' \in D$ and $i = 0, 1$. Define the following: $d \prec^i d'$ iff $d \prec d'$ and $m([d, d']) = i$; $d \preccurlyeq^i d'$ iff $d \prec^i d'$ or $d = d'$; $\sqsubseteq^i = (\prec^i)^*$ (the transitive and reflexive closure of $\prec^i$); $\uparrow^i d = \{d' \in D \mid d \sqsubseteq^i d'\}$; $P^i = \{p \in P \mid m([d, p]) = i\}$. A marked domain (D, m) is called *linear* iff for any $d \in D$ holds: $\uparrow^1 d$ is a linearly ordered set.

Let (D_0, m_0) and (D_1, m_1) be marked domains and f be a morphism from D_0 to D_1 (see Appendix A). Then f is called $\prec^i$*-preserving* ($\preccurlyeq^i$*-preserving*, respectively) (for $i = 0, 1$) iff for any $d \prec^i d'$ holds $f(d) \prec^i f(d')$ ($f(d) \preccurlyeq^i f(d')$, respectively).

Lemma 2. *Linear marked domains with additive, stable,* $\preccurlyeq^0$*- and* $\prec^1$*-preserving morphisms form the category* **<u>MDom</u>**.

4 Relating the Models

In this section we establish the relationships between real-time event structures and marked Scott domains.

First, consider a mapping $\mathcal{TL}$: **<u>TSES</u>** $\rightarrow$ **<u>MDom</u>**. For a real-time event structure TS, define $\mathcal{TL}(TS) = ((TConf(TS), \longrightarrow), m_{TS})$, where m_{TS} is a mapping $I(TConf(TS)) \rightarrow \{0, 1\}$ defined as follows. Given a prime interval $[TC_0 = (C_0, t_0), TC_1 = (C_1, t_1)]$ of $(TConf(TS), \longrightarrow)$, define

$$m_{TS}\big([TC_0, TC_1]\big) = \begin{cases} 0, \text{ if } C_1 \setminus C_0 = \{e\} \text{ for some } e \in E \text{ and } t_0 = t_1; \\ 1, \text{ if } C_0 = C_1 \text{ and } t_1 = t_0 + 1. \end{cases}$$

It easy to show that m_{TS} is well-defined and $\mathcal{TL}(TS)$ is a linear marked domain.

For real-time event structures TS_i $(i = 0, 1)$ and a morphism $\theta : TS_0 \rightarrow TS_1$, define a mapping $\mathcal{TL}(\theta) : \mathcal{TL}(TS_0) \rightarrow \mathcal{TL}(TS_1)$ by $\mathcal{TL}(\theta)(C, t) = (\theta\, C, t)$ for any element (C, t) of $\mathcal{TL}(TS_0)$.

Lemma 3. $\mathcal{TL}$: **<u>TSES</u>** $\rightarrow$ **<u>MDom</u>** *is a functor.*

Next, consider a mapping $\mathcal{TS}$: **<u>MDom</u>** $\rightarrow$ **<u>TPES</u>**. For a linear marked domain $(D, m) \in$ **<u>MDom</u>**, define $\mathcal{TS}(D, m) = ((P^0, \#, \vdash), \delta)$ as follows:

- P^0 is the set of 0-marked prime elements of (D, m);
- $p \# p'$ iff $p \not\uparrow p'$, for all $p, p' \in P^0$;

– $X \vdash p$ iff $\{p' \in P^0 \mid p' \sqsubset p\} \subseteq X$, for all $p \in P^0$ and $X \in Con$;
– $\delta(p) = \|p\|$, for all $p \in P^0$.

It is easy to see that $\mathcal{TS}(D, m)$ is a real-time event structure.

The action of $\mathcal{TS}$ on morphisms in **TSES** can be described as follows. Let $f : (D_0, m_0) \to (D_1, m_1)$ be a morphism in **MDom** between marked domains (D_0, m_0) and (D_1, m_1) with the sets of 0-marked prime elements P_0^0 and P_1^0, respectively. Each prime element $p \in P_0^0$ has the corresponding class of $\sim$-equivalent prime intervals and because f respects $\sim$ — the content of [20, Lemma 2.5.4] — f determines a partial function from P_0^0 to P_1^0. More precisely, let $p \in P_0^0$ correspond to the class of $\sim$-equivalent prime intervals $[d, d']_\sim$ in D_0. Define $\mathcal{TS}(f)(p) = p' \in P_1^0$ if $f(d) \prec^0 f(d')$ and p' corresponds to $[f(d), f(d')]_\sim$ in D_1, and undefined otherwise.

Lemma 4. $\mathcal{TS}$: **MDom** $\to$ **TSES** *is a functor.*

Proposition 1. *The pair* $(\mathcal{TS}, \mathcal{TL})$: **MDom** $\to$ **TSES** *constitutes an adjunction with left adjoint* $\mathcal{TS}$*.*

Proof (sketch). Let $TS = (S = (E, \#, \vdash), \delta)$ be a real-time event structure. It is routine to show that any 0-marked prime element of $\mathcal{TL}(TS)$ is of the form $(C_e, \delta(e))$, where $e \in E$ and $C_e \in Conf(S)$ such that C_e is a minimal configuration w.r.t. $\subseteq$, containing e. Thus, define a mapping $\varepsilon_{TS} : \mathcal{TS} \circ \mathcal{TL}(TS) \to TS$ as follows: $\varepsilon_{TS}(C_e, \delta(e)) = e$ for all $(C_e, \delta(e)) \in P^0(\mathcal{TL}(TS))$. It is easy to see that ε_{TS} is a **TSES**-morphism for every $TS \in$ **TSES**.

Further we shall show universality of ε_{TS} for every $TS \in$ **TSES**, i.e. for every marked domain $(D, m) \in$ **MDom** and a **TSES**-morphism $\theta : \mathcal{TS}(D, m) \to TS$ there exists a unique **MDom**-morphism $f : (D, m) \to \mathcal{TL}(TS)$ such that $\theta = \varepsilon_{TS} \circ \mathcal{TS}(f)$ (see the diagram below).

$$\begin{tikzcd}
TS & \mathcal{TL}(TS) & & TS & & \mathcal{TS} \circ \mathcal{TL}(TS) \arrow[ll, "\varepsilon_{TS}"'] \\
\mathcal{TS}(D,m) \arrow[u, "\forall\theta"] & (D,m) \arrow[u, dashed, "\exists! f"] & s.t. & \mathcal{TS}(D,m) \arrow[u, "\theta"] \arrow[urr, dashed, "\mathcal{TS}(f)"'] & &
\end{tikzcd}$$

Indeed, define a mapping $f(d) = (\theta(\downarrow d \cap P^0(D, m)), \|d\|)$ for every $d \in D$ (with $\downarrow d = \{d' \in D \mid d' \sqsubseteq d\}$). It is straightforward to check that f is an **MDom**-morphism (using linearity of (D, m)) and that the diagram above commutes. It is routine to show that f is unique. $\square$

5 Conclusions

In this paper, following Winskel's approach [20,21] we have shown the applicability of the theory-categorical framework to a real-time extension of stable event structures, in order to identify suitable semantical domains for the models. For this purpose, we have introduced a category **TSES** of real-time stable event structures, and a category **MDom** of a particular class of Scott domains, called marked Scott domains, and then defined an adjunction between **TSES** and **MDom**.

As a work in progress, we are studying coreflections between timed concurrent models (e.g., timed Petri nets, timed asynchronous transition systems, and timed event structures), hoping to reach for these models results similar to those relating classical untimed models along the lines of [16,22].

Acknowledgements

The author is indebted to the anonymous referees for their helpful comments and suggestions. Thanks also go to Irina Virbitskaite for her time, guidance and assistance on the research and on the paper.

References

1. Alur, R., Dill, D.: The theory of timed automata. *Theoretical Computer Science*, **126** (1994) 183–235.
2. Abramsky, S., Jung, A.: Domain theory. *Handbook of Logic in Computer Science*, vol. 3. Clarendon Press, 1994.
3. Aceto, L., Murphi, D.: Timing and causality in process algebra. *Acta Informatica* **33**(4) (1996) 317–350.
4. Casley, R.T., Crew, R.F., Meseguer J., Pratt V.R.: Temporal structures. *Mathematical Structures in Computer Science* **1**(2) (1991) 179–213.
5. Degano, P., Gorrieri, R. Vigna, S.: On Relating Some Models for Concurrency. *Lecture Notes In Computer Science* **668** (1993) 15–30.
6. Dubtsov, R.: Real-Time Event Structures and Scott Domains. *Lecture Notes In Computer Science* **3606** (2005) 29–32.
7. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: An event structure semantics for general Petri nets. *Theoretical Computer Science* **153** (1996) 129–170.
8. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems. *Lecture Notes in Computer Science*, **600** (1991) 226–251.
9. Katoen, J.-P.: *Quantitative and qualitative extensions of event structures.* Ph.D. thesis, University of Twente, 1996.
10. Lilius, J.: Efficient state space search for time Petri nets. Proc. *MFCS'98 Workshop on Concurrency*, August 1998, Brno (Czech Republic), FIMU Report Series, FIMU RS-98-06 (1998) 123–130.
11. Droste M.: Event Structures and Domains. *Theoretical Computer Science* **68** (1989) 37–47.
12. Murphy, D.: Time and duration in noninterleaving concurrency. *Fundamenta Informaticae* **19** (1993) 403–416.
13. MacLane, S.: Categories for the working mathematician. GTM, Springer-Verlag, 1971.
14. Mesguer, J., Montanari U., Sasonne V.: Process versus unfolding semantics for Place/Transition Petri nets. *Theoretical Computer Science* **153** (1996) 171–210.

15. Maggiolo-Schettini, A., Winkowski, J.: Towards an algebra for timed behaviours. *Theoretical Computer Science* **103** (1992) 335–363.
16. Nielsen, M., G. Winskel.: Petri Nets and Bisimulation. *Theoretical Computer Science* **153** (1996) 211–244.
17. Nygaard M., Winskel, G.: Domain Theory For Concurrency: *Theoretical Computer Science* **103** (2004) 153–190.
18. Virbitskaite I.B., Gribovskaya N.S.: Open maps and observational equivalences for timed partial order models. *Fundamenta Informaticae* **60(1-4)** (2004) 383–399.
19. Varacca, D., Völzer, H., Winskel, G.: Probabilistic event structures and domains. *Lecture Notes in Computer Science*, **3170** (2004) 481–496.
20. Winskel, G.: Event Structures. *Lecture Notes in Computer Science*, **255** (1987) 325–392.
21. Winskel, G.: An Introduction to Event Structures. *Lecture Notes in Computer Science*, **354** (1988) 364–397.
22. Winskel, G., Nielsen, N.: Models for concurrency. *Handbook of Logic in Computer Science*, vol. 4. Clarendon Press, 1995.

A Scott Domains

We recall some notions and notations concerning partial orders from [20,21].

Let $(D, \sqsubseteq)$ be a partial order, $d \in D$ and $X \subseteq D$. Then,

- X is said to be *compatible* (written $X\uparrow$ or $x \uparrow y$ for two elements) iff X has an upper bound. We shall denote the *least upper bound* (*greatest lower bound*), if it exists, as $\bigsqcup X$ ($\bigsqcap X$, respectively). Furthermore, lowest upper bound (greatest lower bound) of a two-element set $\{d, d'\}$ is denoted as $d \sqcup d'$ ($d \sqcap d'$).
- X is said to be *finitely compatible* (written $X\uparrow_{fin}$), iff every finite subset $Y \subseteq_{fin} X$ has an upper bound,
- X is said to be directed iff all its finite subsets $X' \subseteq_{fin} X$ have upper bounds in X (so X is finitely compatible and cannot be empty). An element $e \in D$ is said to be *finite* iff for all directed sets $X \subseteq D$, if $e \sqsubseteq \bigsqcup X$ then $e \sqsubseteq x$ for some $x \in X$,
- D is *consistently complete* iff every finitely compatible subset $X \subseteq D$ has a least upper bound $\bigsqcup X$ (thus, D has the least element $\bot = \bigsqcup \varnothing$),
- D is *coherent* iff all subsets $X \subseteq D$ such that $\forall d, d' \in X \diamond d \uparrow d'$ have least upper bounds $\bigsqcup X$,
- A consistently complete partial order is *algebraic* iff for every element $d \in D$ holds: $d = \bigsqcup \{e \sqsubseteq d \mid e \text{ is finite}\}$; ω-algebraic iff it is algebraic and the set $\{e \in D \mid e \text{ is finite}\}$ is countable.

We call a consistently complete algebraic partial order a *Scott domain* (or simply a domain). In this paper, only ω-algebraic Scott domains are considered. A *finitary* Scott domain is one in which every finite element e dominates only a finite number of elements, i.e. $\{d' \in D \mid d' \sqsubseteq e\}$ is finite.

Let $(D, \sqsubseteq)$ be a consistently complete partial order. A *(complete) prime* of D is an element $p \in D$ such that $p \sqsubseteq \bigsqcup X \Rightarrow \exists x \in X \diamond p \sqsubseteq x$ for every compatible set $X \subseteq D$. Let P denote the set of prime elements. D is a *prime algebraic domain* iff for every $d \in D$ holds: $d = \bigsqcup\{p \sqsubseteq d \mid p \in P\}$.

Let $(D, \sqsubseteq)$ be a partial order and $d, d' \in D$. Say d is *covered* by d' and write $d \prec d'$ iff there is no $c \in D$ such that $d \sqsubset c \sqsubset d'$ (i.e. $\prec = \sqsubseteq \setminus \sqsubseteq^2$). A *prime interval* is a pair $[d, d']$ such that $d \prec d'$. In this paper, we use I to indicate the set of prime intervals of D. Given two prime intervals $[c, c'], [d, d'] \in I$ define $[c, c'] \leqslant [d, d']$ iff $c = c' \sqcap d \wedge d' = c' \sqcup d$. Define the equivalence relation $\sim$ as the symmetric, transitive closure of the relation $\leqslant$, and write $[d, d']_\sim$ to denote the equivalence class of $[d, d']$ w.r.t $\sim$. A *covering chain* σ is a sequence $\{d_0, d_1, \ldots, d_n, \ldots\}$, which may be empty, finite or infinite, in which $\bot = d_0 \prec d_1 \prec \cdots \prec d_n \prec \cdots$. A *covering chain for an element* $d \in D$ is a covering chain σ such that $d \in \sigma$ and $\bigsqcup \sigma = d$.

Let $(D_0, \sqsubseteq_0)$ and $(D_1, \sqsubseteq_1)$ be partial orders and f be a mapping $f : D_0 \to D_1$. Say f is *additive* iff $\forall X \subseteq D_0 \diamond X\!\uparrow \Rightarrow f(\bigsqcup X) = \bigsqcup fX$, *stable* iff $\forall X \subseteq D_0 \diamond X \neq \varnothing \wedge X\!\uparrow \Rightarrow f(\bigsqcap X) = \bigsqcap fX$, *$\preccurlyeq$-preserving* iff $\forall x, x' \in D_0 \diamond x \prec x' \Rightarrow f(x) \preccurlyeq f(x')$ (here $f(x) \preccurlyeq f(x')$ iff $f(x) \prec f(x') \vee f(x) = f(x')$).

Streaming Networks for Coordinating Data-Parallel Programs

Clemens Grelck[1], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] Inst. of Software Technology and Programming Languages, University of Lübeck, Germany
Grelck@isp.uni-luebeck.de
[2] Compiler Technology and Computer Architecture Group, University of Hertfordshire, United Kingdom
{S.Scholz,A.Shafarenko}@herts.ac.uk

Abstract. A new coordination language for distributed data-parallel programs is presented, call SNet. The intention of SNet is to introduce advanced structuring techniques into a coordination language: stream processing and various forms of subtyping. The talk will present the organisation of SNet, its major type inferencing algorithms and will briefly discuss the current state of implementation and possible applications.

Data-parallel programming languages such as Nesl, Zpl, Sisal, or Single-Assignment C (known primarily as SaC) are known to be suitable for creating highly efficiently executable concurrent code for numerical applications. Instead of relying on programmer-specified explicit annotations as required for Hpf or a library extension as is the case with MPI-based or OpenMP-based solutions, these programming languages are designed in a way that allows compilers to derive concurrency implicitly from homogeneous operations on large arrays.

More recent work in the context of SaC demonstrates that not only does the data-parallel approach raise the level of abstraction in specifying numerical applications, but it also permits compiler technology to be developed that produces code competitive with that of low-level Fortran code. SaC programs can be written in a highly abstract style similar to specialised array programming languages such as Apl or J. This level of abstraction improves code reuse and maintainability of programs since the programmer can focus on the functionality of individual components of a program rather than being constantly driven by performance considerations [14]. Despite being less aware of the performance issue, the programmer's functional specifications can automatically be compiled into code whose sequential runtime is competitive with that of low-level Fortran programs where the programmer has to be aware of the performance issues continually. Thanks to the sophisticated compiler technology developed within the SaC project, SaC programs can be compiled into multithreaded code without any source modification [6]. As sequential runtimes are competitive with sequential low-level code, so too the multithreaded code produced by our compiler is very effective: the speed up for SMP platforms is almost linear in the number of processors up to $p = 8$ and in many cases beyond.

One has to admit, however, that the concurrency that can be exploited by the data parallel approach is limited to homogeneous operations on arrays. Although these prevail within the component code for numerical applications, when components are joined together, other forms of concurrency become prevalent, which are better captured by pipelining, process farming and arbitrary message passing. These forms of concurrency can not always be derived from a sequential application code, since they tend to be strongly application-dependent. There is a whole host of "parallel algorithms" capturing problem-specific data migration and load balancing, of which perhaps the most convincing example is provided by molecular dynamics and plasma particle simulations, see, for example, papers [10,3]. The methods being used rely upon the knowledge of computational properties, such as relative cost of various components, and co-location requirements. For instance, the designer of a particle-in-cell simulation is naturally aware that the main computational cost is in pushing particles under the influence of electromagnetic forces, hence load balancing should focus on that, while field calculations are always assumed to be much cheaper. It is hardly realistic at present time to expect any compilation system to be able to derive this kind of information from sequential (or even purely functional) code.

Process concurrency is difficult to deal with in the framework of a programming language. If properly integrated into the language semantics, it complicates and often completely destroys the properties that enable the kind of profound optimisations that make compilation of array programs so efficient. One solution to this problem, which is the solution that we align ourselves with, is the use of so-called coordination languages. A coordination language uses a readily-available computation language as a basis, and extends it with a certain communication/synchronisation mechanism thus allowing a distributed program to be written in a purely extensional manner. The first coordination language proposed was Linda [5,4], which extended C with a few primitives that looked like function calls and could even be implemented directly as such. However an advanced implementation of Linda would involve program analysis and transformation in order to optimise communication and synchronisation patterns beyond the obvious semantics of the primitives. Further coordination languages have been proposed, many on them extensional in the same way, some not; for the state of the art, see a survey in [12] and the latest Coordination conference [8].

The emphasis of coordination languages is usually on event management, while the data aspect of distributed computations is not ordinarily focused on. This has a disadvantage in that the structuring aspect, software reuse and component technology are not primary goals of coordination. It is our contention that structuring is key in making coordination-based distributed programming practically useful. In this paper we propose several structuring solutions, which have been laid in the foundation of the coordination language SNet. The language was introduced as a concept in [16]; the complete definition, including semantics and the type system, is available as a technical report [17].

The approach proposed in SNet is based on streaming networks as introduced in foundation work [9,1,7], see also more recent work on stream network semantics [2] and language design [11]. The application as a whole is represented as a set of self-contained components, called "boxes" (SNetis not extensional) written in the data-parallel language SaC. SNet deals with boxes by combining them into networks which can be encapsulated as further boxes. The structuring instruments used are as follows:

- Streams. Instead of arbitrary communication, data is packaged into typed variant records that flow in a sequence from their producer to a single consumer.
- Single-Input, Single-Output(SISO) box and network configuration. Multiple connections are, of course, possible and necessary. The unique feature of SNet is that the multiplicity of connection is handled by SNet combinators so that a box sees a single stream of records coming in. The records are properly attributed to their sources by using types (which include algebraic types, or tagged, disjoint unions). Similarly, the production of a single stream of typed records by a box does not preclude the output separation into several streams according to the type outside the box perimeter.
- Network construction using structural combinators. The network is presented as an expression in the algebra of four major combinators (and a small variety of ancillary constructs): serial (pipelined) composition, parallel composition, infinite serial replication (closure) and infinite parallel replication (called index splitter, as the input is split between the replicas according to an "index" contained in data records). We will show that this small nomenclature of tools is sufficient to construct an arbitrary streaming network.
- Record subtyping. Data streams consist of flat records, whose fields are drawn from a linear hierarchy of array subtypes [15,18]. The records as wholes are subtyped since the boxes accept records with extra fields and allow the producer to supply fewer variants than the consumer has the ability to recognise.
- Flow inheritance. Due to subtyping, the boxes may receive more fields in a record than they recognise. In such circumstances flow inheritance causes the extra fields to be saved and then appended to all output records produced in response to a given input one[1]. Flow inheritance enables very flexible pipelining since, on the one hand, a component does not need to be aware of the exact composition of data records that it receives as long as it receives sufficient fields for the processing it is supposed to do; and on the other, the extra data are not lost but passed further down the pipeline that the components may be connected by.
- Record synchronizers. These are similar to I-structures known from dataflow programming. SNet synchronisers are typed SISO boxes that expect two records of certain types and produce a joint record. No other synchronisation mechanism exists in SNet, and no synchronisation capability is required of the user-defined boxes.

[1] This is a conceptual view; in practice the data fields are routed directly to their consumers, thanks to the complete inferability of type in SNet.

– The concept of network feedback in the form of a closure operator. This connects replicas of a box in a (conceptually) infinite chain, with the input data flowing to the head of the chain and the output data being extracted on the basis of fixed-point recognition. The main innovation here is the proposal of a type-defined fixed point (using flow inheritance as a statically recognisable mechanism), and the provision of an efficient type-inference algorithm. As a result, SNet has no named channels (in fact, no explicit channels at all) and the whole network can be defined as a single expression in a certain combinator algebra.

The authors acknowledge the support of the EU-sponsored Integrated Project “EATHER” [13], which is part of the Framework VI Advanced Computing Architecture Initiative.

References

1. E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.
2. M Broy and G Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, (258):99–129, 2001.
3. P M Campbell, E A Carmona, and D W Walker. Hierarchical domain decomposition with unitary load balancing for electromagnetic particle-in-cell codes. In *Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, South Carolina, April, 9-12, 1990*. IEEE Computer Society Press, 1990.
4. D Gelernter and N Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, Feb. 1992.
5. David Gelernter. Generative communication in linda. *ACM Trans Program. Lang Syst.*, 1(7):80–112, 1985.
6. C. Grelck and S.-B. Scholz. A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming*, 2006. to appear.
7. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
8. Jean-Marie Jacquet and Gian Pietro Picco, editors. *Coordination Models and Languages. 7th International Conference, COORDINATION 2005, Namur, Belgium, April 20-23, 2005*, volume 3454 of *Proceedings Series: Lecture Notes in Computer Science, Vol. 3454 Jacquet, Jean-Marie; Picco, Gian Pietro (Eds.) 2005, X, 299 p., Softcover Lecture Notes in Computer Science*. Springer Verlag, 2005.
9. G Kahn. The semantics of a simple language for parallel programming. In L Rosenfeld, editor, *Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden*, pages 471–475. North-Holland, 1974.
10. L V Kale, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Proceedings of the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel, Berkeley, California, USA, August 9-11, 1998. Solving Irregularly Structured Problems in Parallel, Springer Verlag LNCS 1457*, 1988.

11. Michael I. Gordon *et al.* A stream compiler for communication-exposed architectures. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002*, 2002.
12. G A Papadopoulos and F Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.
13. The AETHER Project. http://aetherist.free.fr/Joomla/index.php.
14. A. Shafarenko, S.-B. Scholz, S. Herhut, C. Grelck, and K. Trojahner. Implementing a numerical solution for the KPI equation using Single Assignment C: lessons and experience. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05*, volume ??? of *LNCS*. Springer, 2006. to appear.
15. Alex Shafarenko. Coercion as homomorphism: type inference in a system with subtyping and overloading. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 14–25, 2002.
16. Alex Shafarenko. Stream processing on the grid: an array stream transforming language. In *SNPD*, pages 268–276, 2003.
17. Alex Shafarenko. Snet: definition and the main algorithms. Technical report, Department of Computer Science, 2006.
18. Alex Shafarenko and Sven-Bodo Scholz. General homomorphic overloading. In *Implemntation and Application of Functional Languages. 16th International Workshop, IFL 2004, Lübeck, Germany, September 2004. Revised Selected Papers.*, LNCS'3474, pages 195–210. Springer Verlag, 2004.

Formal Methods in Industrial Software Standards Enforcement

Alexey Grinevich, Alexey Khoroshilov, Victor Kuliamin, Denis Markovtsev, Alexander Petrenko, and Vladimir Rubanov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{tanur,hed,kuliamin,day,petrenko,vrub}@ispras.ru

Abstract. The article presents an approach to development of software standards usage infrastructure. The approach is based on formalization of standards and automated conformance test derivation from the resulting formal specifications. Strong technological support of such a process in its engineering aspects makes it applicable to software standards of real-life complexity. This is illustrated by its application to Linux Standard Base. The work stands in line with goals of international initiative Grand Challenge 6: Dependable Systems Evolution [1].

1 Introduction

The needs of economical and social development make current software systems very complex. Such a system usually consists of many components of different vendors, and a lot of individual software engineers take part in its construction. Due to this fact, interoperability of those parts and reliability of the system as a whole become problematic. The well-known way to solve these problems is enforcement of *software interface standards*.

The idea of interface standards is rather clear – we make possible for different software systems to work together through standardized interfaces without hard restrictions on the internal implementations of them. So, interoperability is ensured without damping individual creativity and corporate innovation potential. This approach works well if the standard defines the functionality of the corresponding interfaces clearly, unambiguously, and precisely.

However, looking at the current state of software standards one can see that many of them are not so clear. Historically, they were developed under the market pressure to ensure some interoperability taking into account conflicting interests of many software vendors. In such a situation, only the basic functionality can be defined unambiguously. And each group of developers usually has its own solution for peculiar and complex cases. Since such a solution is already invested, it is hard for vendors to throw it away and take the point of a competitor, which preserves its investments.

Usual compromise is to agree with some minor changes for each of competing vendors and to make the standard ambiguous in cases where serious elaboration is needed. Thus solutions in use can be declared as conforming to it.

This helps vendors to spend acceptable money to standard enforcement in their systems, but also compromise the desired interoperability of different implementations. Some of current software and telecommunication standards appeared about 20-25 years ago, and, of course, with each revision they become more strict and more consistent. Most of ambiguities of the first versions were removed, but newer additions and elaborations required by technological progress still may have unclear and equivocal statements.

The way out of this situation proposed by many software engineering researchers and practitioners is *formalization of standards.* Formal re-statement of standard's requirements discloses contradictions and ambiguities, but exacts a lot of hard work. Nobody expects that inaccuracy and inconsistency will be removed from standards at once. But by starting formalization, we at least will be aware of real problems and can make practical suggestions on their solutions.

Formalization definitely makes standards more useful, but alone it does not give instruments of their enforcement in practice. Developers of real-life systems need tools that help them to ensure conformance to standards. So, practically useful formal description of standard requirements should be supplemented with test suites checking conformance to these requirements. The approach described in this article uses one of the most elaborated frameworks for conformance testing based on formal descriptions presented in telecommunication standards ISO 9646 [2] and ITU-T Z.500 [3].

Solid formal framework makes conformance test construction more rigorous, and therefore helps to enforce quality of the systems implementing the standard. However, practical use of conformance test suites brings into account additional engineering issues.

– *Requirements traceability.* For software engineers and their managers the real source of requirements is the standard text. Formal specification is an additional document that should clearly demonstrate its correspondence to the standard. The same holds for tests derived from them – they should be traced to some requirements stated in some sections of the standard. This really helps conscientious development of standard implementations.
– *Component-wise treatment of standard.* Real-life standards are complex, as well as real-life software. Some decomposition techniques are required for adequate treatment of them. Specification formalism used should support definition of separate components of the standard, should allow their consideration in isolation, but also provide means for describing them as a whole.
– *Change management.* Standards and their implementations are not fixed entities – they are changing. These changes should have adequate support in the process of specification development, test derivation and translation. Lack of this support quickly makes the specifications and tests useless.
– *Configurations.* Real-life standards describe parameterized systems, having a lot of configuration options that significantly influence their functionality. Such options should be also supported in specifications and tests.

All those problems should have suitable solutions in the technology that aspires to provide an adequate infrastructure for standard enforcement. This article presents a candidate approach based on UniTesK technology of automated test construction developed in ISPRAS. Main ideas and techniques of the approach are considered in the next two sections. The fourth section describes preliminary results of its application to Linux Standard Base [4], the industrial standard on interfaces of Linux operating system core libraries. The last two sections of the article contain brief comparison of the approach presented here with other methods and a conclusion stating main results and directions of future development.

2 Standard Formalization

The main difficulties of standard formalization are concerned with informal and compromise nature of the industrial standards. Their main goals are to fix the consensus of main vendors on functionality of related systems and to provide reference and programming guide for developers of both the standard implementations and the systems using them. They use the language and the structure that seem to be suitable for these goals.

Interface standards often consist of two parts – rationale, presenting the main concepts and features in their integrity, and reference, describing each interface element separately. In addition to interface elements (data types, operations, constants, global data elements) reference can describe complex entities – large subsystems, header files, etc. Reference sections can refer to each other and contain parts of each other.

Standard formalization consists of several activities.

1. *Standard decomposition.* Since the number of interface elements described in the standard can be very large, on the first step they are partitioned into logically related groups. Such a group usually is closed under operation inversion and consists of operations and types concerned with one feature. For example, operations to open and close files, to create and destroy objects should be put in one group. All further work is performed mostly inside such groups.
2. *Requirements arrangement.* The next task is to extract all standard requirements to interface elements that can be interpreted in formal way and can be checked. Since one thing can be described in several places, all the corresponding sections should be read attentively, phrase by phrase, and all the constraints found should be marked. Then, these constraints are united in some consistent set.

   It is rather tedious work, but not a mechanical or trivial one. Transition from informal to formal is essentially informal itself (M. R. Shura-Bura).

   Standard statements in different parts can be inconsistent, ambiguous, represent similar, but not the same ideas. The standard itself is not enough to make consistent conclusions. One should use for that common knowledge, related books and articles, solutions in use, and communication with authors of the standard, experts in the domain or experienced developers.

   Some aspects are made intentionally unspecified to make implementations of different vendors conforming to the standard. Amusing example of such statement is in the current version of POSIX standard [5]. Description of `fstatvfs()` and `statvfs()` functions from `sys/statvfs.h` header says: "It is unspecified whether all members of the `statvfs` structure have meaningful values on all file systems."

There are two possible ways to resolve such situations.

– Do not check anything. In this case no constraints are extracted from the corresponding text of the standard.
– If there are few possible implementations, they all can be presented as a parameterized constraint. A configuration option corresponding to the kind of implementation used is added. The constraint to check is selected depending on this option.

  Example of this case from POSIX description of `basename()` function: "If the string pointed to by path is exactly "//", it is implementation-defined whether '/' or "//" is returned."

This activity is performed until all the text of the standard concerning the chosen group of interface elements is partitioned into requirements that can be checked and other phrases that do not contain testable restrictions. Main results of this work are the following.

– *Catalogue of requirements.* It lists the requirements imposed by the standard and maps each requirement to the corresponding piece of standard's text. One requirement usually corresponds to logically complete piece of text expressing one constraint on an interface element or data element. Further this catalogue helps to ensure test adequacy in terms of original text of the standard.
– *Defects of the standard and notes* presenting found ambiguities, inconsistencies, unintentionally unclear and imprecise statements, incomplete descriptions of functionality, etc.

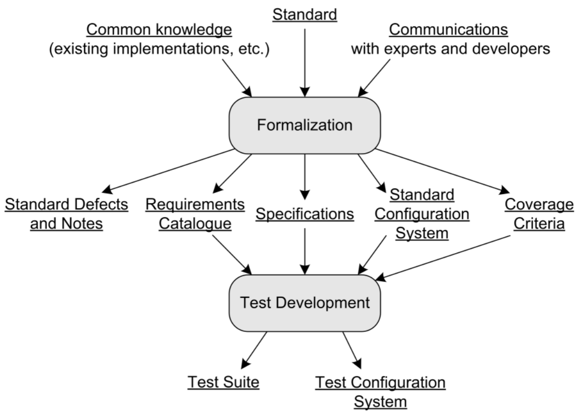

Standard
Common knowledge (existing implementations, etc.)
Communications with experts and developers
Formalization
Standard Defects and Notes
Requirements Catalogue
Specifications
Standard Configuration System
Coverage Criteria
Test Development
Test Suite
Test Configuration System

Fig. 1. Ins and outs of formalization and conformance test development process

3. *Specification development.* This work is usually performed in some mix with the previous one. They are separated only for clarity reasons.

   All the constraints found are recorded in the form of *contract specifications* of operations, data types, and interface data elements. Each operation is described with its precondition and postcondition. *Precondition* of an operation defines its domain. *Postcondition* defines the constraints on the operation results depending on values of parameters and internal system state. Data types and data elements are described with their integrity constraints called *invariants.*

   This activity produces two results.
   – *Specifications* of all interface elements. Code of specifications is marked out to map the formal constraints specified to the corresponding requirements from requirements catalogue.
   – *Configuration system of the standard.* This system consists of a set of configuration parameters declared in the standard and additional ones, dependencies between them, and their links with interface elements and constraints. Some configuration options represented as values of the parameters can govern the set of constraints that should be checked. Others can say that some functionality is absent and the corresponding operations should not be called at all. Third can influence possible error codes returned by operations.

     Example of the first case is given by POSIX description of `pthread_create()` function: "If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero."
4. *Coverage criteria definition.* The last activity is to define coverage criteria that can be used to measure the adequacy of the conformance testing of standard's implementation. Basic criterion is to cover all the standard's requirements applicable to the current configuration of the system under test. More elaborate criteria can specify additional situations to be tested. All these criteria are based on the structure of specifications representing standard requirements.

   Coverage criteria can be in complex relations with configuration parameters. Possibility to cover some situation depends on values of configuration parameters and this dependency should be recorded to prevent confusion and burdensome reviews during analysis of test results.
   This work results in *a set of coverage criteria* correlated with the configuration system.

The process of standard formalization and further test construction is illustrated by Fig. 1.

3 Conformance Test Construction

The base of the conformance test construction is UniTesK technology developed in ISPRAS. It uses almost the same general formal testing framework that was developed in works of Bernot [6], Brinksma, and Tretmans [7,8] and described in the standards ISO 9646 [2] and ITU-T Z.500 [3]. Main elements of this framework can be formulated as follows (Fig. 2 illustrates relations between them).

– The standard requirements are represented as a model in some formalism. This model is called *a specification.*
– The software system, which conformance with the standard we need to check (called *system under test*, or SUT), is supposed to be adequately modeled in the same formalism. The corresponding model is called *an implementation.* We do not know it, but assume that it exists. 'Adequate' modeling here means that we cannot observe any difference between the real behavior of the SUT and the model behavior of the implementation.

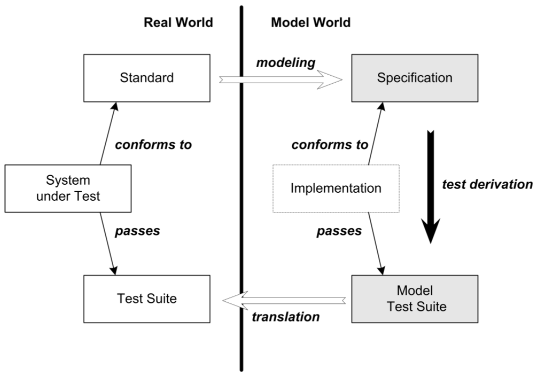


Fig. 2. Relations between real-life and model entities

– The fact that the SUT conforms to the standard is modeled by formal *implementation relation* or *conformance relation* between the implementation and the specification.
– Model tests are derived from the specification. *A model test* is a model that provides boolean verdict as the result of interaction with other models. Implementation *passes* a test, if the latter provides the verdict **true** after this interaction. It is reasonable to derive only *sound tests,* which are passed by any implementation conforming to the specification. One wish to construct a set of model tests, or *a model test suite* that is *complete* in the sense that a model passes it if and only if this model conforms to the specification.
– Model tests are represented as test programs interacting with the SUT. Since this interaction is supposed to be modeled by the interaction between model tests and the implementation, one can conclude that the SUT passing a complete test suite conforms to the standard.

Real-life standards usually describe rather complex systems. Specification of such a standard is also complex and huge, so any complete test suite for it contains infinitely many tests. To make formal testing possible in this setting, we introduce additional element of the framework.

– *A coverage criterion* is an equivalence relation on model tests. We use coverage criterion with the corresponding hypothesis stating that an implementation either passes any two pair of equivalent tests or does not pass both of them. The criterion is chosen in such a way that it is reasonable to consider only such implementations. A test suite is called *complete according to the coverage criterion* if the union of equivalence classes of tests of this suite is a complete test suite in the traditional sense.

Note that coverage criterion can be taken from different sources. The only desired property is existence of finite test suite complete according to the criterion used. In conformance test construction we use criteria derived from the structure of specifications.

Here we present only basic ideas of UniTesK, details can be found in [9,10,11,12].

– Functional requirements to the SUT's behavior are stated in the form of *contract specifications.* Contract of a component consists of pre- and postconditions of all its operations and asynchronous events provided and invariants of its internal data.
– *Test coverage criteria* are defined on the base of postcondition structure. Examples of such criteria are functional branch coverage and multiple condition coverage of postconditions. One can add user-defined criteria.
– *Test scenarios* are targeted to achieve certain level of coverage of some group of operations. Such test scenario describes a finite state machine (FSM) modeling the component's behavior in such a way that its transition tour ensures 100% coverage according to the criterion. It defines state calculation procedure and a set of applicable actions for an arbitrary state. Each action corresponds to call of some operation with some arguments.
– Tests are generated during test execution by on-the-fly automatic construction of paths on the FSM described by a test scenario.

Conformance test construction for real-life software standards also requires development of *a test configuration system.* This system includes standard configuration system and additional parameters that influence test execution process. They correspond to different target coverage criteria, different sets of test scenarios, and so on. Powerful configuration system makes the resulting test suite useful in any settings where an implementation of the corresponding standard can operate in practice.

4 Applications of the Approach

The approach described in the previous sections was used for formalization and test construction for parts of IPv6 and IPMP2 protocol standards [10,13].

More serious case study is provided by the project of Linux Verification Center on formalization of Linux Standard Base (LSB) [4]. The goal of this project is to create formal specifications of LSB requirements and corresponding conformance test suite for core libraries of Linux operating system described in the sections

III and IV of LSB 3.1 and including 1532 functions. The requirements stated there in many cases (but not always!) coincide with POSIX [5] requirements to the same functions.

Now the project is in progress. Its results will be accessible at the end of 2006 as open source and will include the following.

– The set of notes to the text of LSB standard, pointing out unclear, inconsistent, or ambiguous statements.
– Parameterized conformance test suite (including formal specifications with mapping from them to the standard requirements). The parameters will control configuration of the SUT, testing quality level, test execution time, and so on.

On the first step, 1532 functions were partitioned into 170 groups, which in turn are grouped into larger subsystems according to the features concerned – threads, interprocess communication, file system, memory management, mathematical functions, etc. By the end of August 2006 tests and specifications for 1245 functions of 140 groups is developed. 14400 separate standard requirements are extracted for them. Some functions have just a few corresponding requirements, while others – several dozens (maximum is 134).

The productivity achieved shows that the project will require about 15 man-years for complete formalization (with development of basic level tests) of LSB. An experienced programmer (not an average tester!) can be trained to a level needed by this project in about a month. These intermediate results give hope that the approach presented is able to cope with tasks of such a size.

The preliminary results obtained also demonstrate rather high quality of the tests developed. Tests are targeted mostly to cover all the requirements extracted during standard formalization (more presicely, those of them that are achievable on the configuration under test). At the same time they achieve high levels of source code coverage. For example, tests for 24 functions working with threads get 72% coverage of GLIBC source code lines, more than most analogous test suites (LTP test suite [14] gets 48%, LSB binary conformance test suite [15] gets 71%). Only specialized GLIBC test suite [16] gets more (78%), since it is written by the developers knowing peculiarities of the library and paying no attention to check strict conformance to the standard. For 41 functions working with strings the numbers are similar: our test suite gets 91%, LTP – 53%, LSB conformance test suite – 67%, GLIBC test suite – 84%. For 13 utility search functions (located in `search.h`) our test suite gets 65%, LTP – 28%, LSB conformance test suite – 33%, GLIBC test suite – 70%.

5 Related Work

Most advances in standard formalization and conformance test construction techniques were made in telecommunication domain. The general framework [3,6,8] (see also above) was developed in this area.

Our approach has a lot in common with it. The differences are related with larger size of OS interface standards such as POSIX or LSB in comparison with typical protocol standards. Scalability considerations lead to introduction of coverage criteria in the general formal testing framework and coverage-oriented test generation. Use of contract specifications makes possible more suitable decomposition then automata or labeled transition systems used by most research and industrial development in telecommunications.

Article [17] presents another attempt of standard formalization on the example of IEEE 1003.5 – POSIX Ada Language Interfaces. Standard requirements were transformed there into formally described tests directly, without intermediate specifications. This method seems to us only slightly different from manual test development.

More close to our approach is the paper [18] presenting case studies in model based testing with GOTCHA-TCBeans toolkit. One of those case studies is concerned with testing POSIX function `fcntl()`. The methodology of standard formalization used in this work is focused more on getting effective formal model for testing than on traceability to standard requirements, as in our case. Nevertheless, the approach presented in [18] uses very similar ideas with our approach, although it is based on other kind of formalism – FSMs described in Murphy language.

There are a number of similar activities of conformance test suite development for operating system interface standards. Most well-known standards in this field are IEEE Std 1003.1 (POSIX) [5] and Linux Standard Base (LSB) [4]. The main conformance test suite development projects dealing with them are the Open POSIX Test Suite [19], an open source project, and official certification test suite for LSB conformance [15] developed by Free Standards Group [20].

Both these projects use similar techniques for requirements extraction from standard text and requirements catalogue creation. Then, tests are developed manually using traditional approaches, one test per requirement. They do not perform formalization of requirements and do not use automated test construction techniques.

Note that the approach 'one test per requirement' tempts to consolidate many separate constraints into one requirement to be able to check it within one test. In Open POSIX Test Suite we found examples of requirements that correspond to dozen of the requirements extracted in our project. This ends up in situations when sub-constraints of big requirement are passed over and not tested while the whole requirement is reported as tested. So, the resulting reports can be too optimistic on the coverage of the standard requirements.

Automated test derivation from specifications used in our approach makes test suite much more manageable in case of changes in the standard. It forces to record one piece of knowledge in one place.

6 Conclusion

Enforcement of software interface standards usage is a complex task having technical, economical, and social aspects. Nevertheless, all experts agree on its necessity for stable development of software industry. Standard formalization was proposed a lot of times as a possible way to solve at least numerous technical problems related to this activity. However, formalization requires a lot of effort and many doubts were expressed on its applicability to software standards of real-life size and complexity. The project mentioned in the Case Studies section seems to be one of the first attempts to formalize a significant part of industrial software standard and to taste the fruits of this work.

We believe that the approach presented allows successful accomplishment of projects of such a scale. Strong arguments in favor of this opinion is given by the stable progress of the project, the history of its technological background (see [11,13]), and choice of good engineering practices as a main guide [12]. Taking into account real-life engineering and organizational issues helps to avoid producing ungainly and useless results usually pertinent to pioneering use of advanced methods in practice.

We consider this work as a part of effort concerned with Dependable System Evolution Grand Challenge [1]. Tony Hoar suggested two tracks of activities dealing with this problem: development of methods and tools capable to help in resolving it and development of "challenge codes" – realistic examples of usage of those methods and tools. The last track is necessary to demonstrate ways from state of the art in software engineering to state of the practice, to find the scope of systems, for which advanced the methods developed in research community are applicable.

To stimulate activities of the second kind, ISPRAS donates the results of the project and all the tools needed to deal with them to open source community. We hope that members of this community can be attracted to more challenging projects such as formalization of the huge set of Carrier Grade Linux standards [21], embedded and real-time versions of Linux, and some widely used programming languages.

Acknowledgements. We thank Federal Agency for Science and Innovations of Russia for its financial support of the Linux Verification Center creation and LSB formalization project.

References

1. http://www.fmnet.info/gc6/
2. *ISO 9646. Information Theory – Open System Interconnection – Conformance Testing Methodology and Framework.* ISO, Geneve, 1991.
3. *ITU-T. Recommendation Z.500. Framework on formal methods in conformance testing.* International Telecommunications Union, Geneve, Switzerland, 1997.
4. http://www.linuxbase.org/spec
5. http://www.unix.org/version3/ieee_std.html
6. G. Bernot. *Testing against Formal Specifications: A Theoretical View.* In Proc. of TAPSOFT'91, Vol. 2. S. Abramsky and T. S. E. Maibaum, eds. LNCS 494, pp. 99–119, Springer-Verlag, 1991.

7. E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. *A formal approach to conformance testing.* In J. de Meer, L. Mackert, and W. Effelsberg, eds. 2-nd Int. Workshop on Protocol Test Systems, pp. 349–363. North-Holland, 1990.
8. J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.
9. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. *UniTesK Test Suite Architecture.* In Proc. of FME 2002. LNCS 2391, pp. 77–88, Springer-Verlag, 2002.
10. V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, I. Bourdonov. *Integration of Functional and Timed Testing of Real-time and Concurrent Systems.* In Proc. of PSI 2003, LNCS 2890, pp. 450–461, Springer-Verlag, 2003.
11. V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. *UniTesK: Model Based Testing in Industrial Practice.* In Proc. of 1-st European Conference on Model-Driven Software Engineering, Nurnberg, December 2003, pp. 55–63.
12. V. Kuliamin. *Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System.* In Proc. of 1-st International Symposium on Leveraging Applications of Formal Methods, Cyprus, October 2004, pp. 311–316.
13. V. Kuliamin, A. Petrenko, N. Pakoulin. *Practical Approach to Specification and Conformance Testing of Distributed Network Applications.* In M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68–83, Springer-Verlag, 2005.
14. http://ltp.sourceforge.net/
15. http://www.linuxbase.org/download/#test_suites
16. ftp://ftp.gnu.org/gnu/glibc/
17. J. F. Leathrum and K. A. Liburdy. *A Formal Approach to Requirements Based Testing in Open Systems Standards.* In Proc. of 2-d International Conference on Requirements Engineering, 1996, pp. 94–100.
18. E. Farchi, A. Hartman, and S. S. Pinter. *Using a model-based test generator to test for standard conformance.* IBM Systems Journal, 41:89–110, 2002.
19. http://posixtest.sourceforge.net/
20. http://freestandards.org/
21. http://www.osdl.org/lab_activities/carrier_grade_linux

Visual Rules Modeling

Sergey Lukichev and Gerd Wagner

Brandenburg University of Technology at Cottbus
{Lukichev, G.Wagner}@tu-cottbus.de

1 Introduction

Rules are widely recognized to play an important role in the Semantic Web. They are a critical technology component for the early adoption and applications of knowledge-based techniques in e-business, especially enterprize integration and B2B e-commerce. This includes, in particular, markup languages for integrity and derivation rules, such as the Semantic Web Rule Language (SWRL)[5] that has recently been proposed as an extension of the Web ontology language OWL[4]. Rules also play an important role in information systems engineering, especially in the specification of functional requirements where business rules are the foundation for capturing and modeling business application logic.

A lot of work has been conducted in the area of visual representation of business vocabularies. The mainstream technology is MOF[9]/UML[10], which allows visualization of domain concepts by means of, for instance, UML class diagrams.

On the other hand, relatively few research has been done in the area of visual rules modeling. The emerging technologies for the Semantic Web, where rules play an important role, experience lack of modeling tools for visual representation of ontologies and rules. The request for a UML-based rule modeling tool for the Semantic Web comes from the industry. Many companies claim that even if they understand benefits of using Semantic Web technologies like ontologies and rule languages, it is difficult for them to start since ontology architects and rule experts are quite expensive. A UML-based rule modeling approach for the Semantic Web will facilitate the use of the Semantic Web technologies by traditional UML modelers.

The actuality of the proposed research also comes from the rules standardization efforts of W3C (`http://www.w3.org/2005/rules`) and OMG (`http://www.omg.org`), which need rules modeling methodologies and tools.

This paper gives a quick overview of existing rule modeling solutions (Section 3) and presents visual rules modeling approach, based on MOF/UML, using examples (Section 4). In the conclusion part we formulate main advantages of our research in progress (Section 5).

2 Research Description

There is a general problem of interaction between domain experts and technicians, who formalize a business domain and business requirements. To contribute to the solution of this problem we work on methodologies for visual representation of rules, which intend to help capturing business rules from a natural language to the visual/formal representation.

Our present and future work intends to give appropriate answers, for instance, to the following questions:

1. How extensible is UML to support rules in diagrams?
2. How can we integrate the visual modeling of rules with existent modeling tools?
3. Is it possible to adopt component SE and aspect SE to ontology and rules modeling in order to deal with large ontologies and different rule systems? This question address a well-known problem of business rules management and business rules validation in large rule-based systems.
4. What are the relations between UML/OCL and OWL/SWRL? Can OWL and SWRL be transferred into UML/OCL and vise versa in order to exchange rules between two communities of UML modelers and ontology architects?

3 Current Knowledge of the Problem Domain

Existing UML modeling tools usually provide facilities for class and relationship modeling. These models have a static and declarative nature and cannot be used for modeling reactive nature of the Semantic Web in particular and rules-driven business processes in general. The Object Constraint Language (OCL [1])is used for expressing rules in UML class diagrams. Existing tools support serialization of OCL constraints to XMI and there are efforts of Java code generation directly from UML class diagrams with OCL constraints. The latest is supported by Fujaba Tool Suit (`http://www.fujaba.de`).

German company Visual Rules (`http://www.visual-rules.com`) provides a tool for visual modeling of rules in block-schema like style, which may cover some types of business rules.

Market leaders in business rules solutions, ILOG (`http://www.ilog.com`) and LibRT (`http://www.librt.com`), provide flexible tools for rules modeling and deployment, but contain no visual modeling components, which complicates development of rule-based applications.

Concerning Semantic Web technologies, there are several methods for rules modeling.

The Protege tool (`http://protege.stanford.edu/`) provides facilities for ontology and rules modeling. In particular, it supports modeling in RDF and OWL as well as modeling of SWRL rules. In conjunction with reasoning engine, the tool can be used for consistency check of ontologies and serialization to the rule markup. Protege is not a visual tool and requires a significant knowledge of ontology modeling. Moreover it is doubtful that it can be easily adopted in enterprizes, which already use UML technologies for software engineering.

There are ontology language specific tools for visual representation of ontologies, for instance, SemTalk from Semtation GmbH (`http://www.semtation.de`), which provides a visual language for modeling of OWL ontologies. The approach of defining visual language for a particular ontology language has a lack of flexibility and scalability, while our UML-based approach has a power of MDA and allows obtain rules in language-independent manner.

In general, our main activities are focused on development of new visual notations for vocabularies and rules. We consider rules on top of UML class diagrams because they are widely used in software development and such rule modeling principles can be easily adopted by large community of UML modelers.

According to Business Rules Manifesto[6], rules are build on top of vocabularies. This is why extending UML, which is used to express business vocabularies, with a concept of a rule is natural.

4 The State of Art

Main classes of rules at three different abstraction levels are depicted in Fig. 1.

More detailed description of rules classification is provided in [7] and defines, in particular, derivation rules, production rules, reaction rules and integrity constraints.

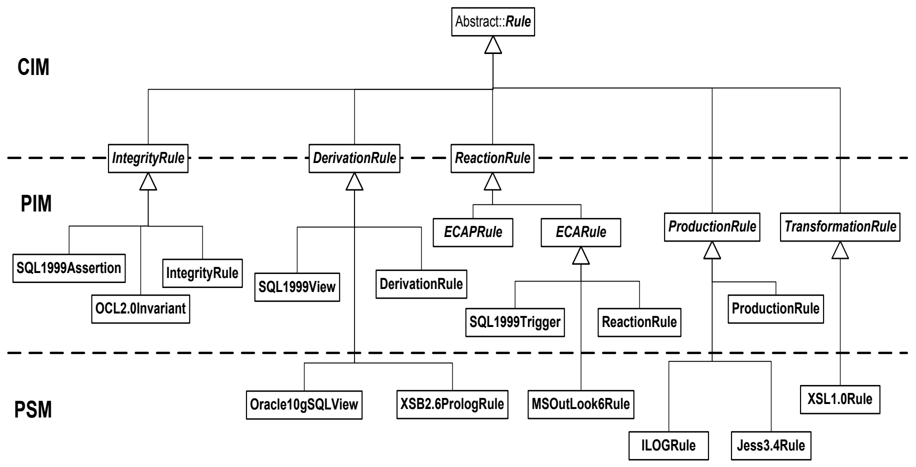


Fig. 1. Rule concepts at three different abstraction levels: computation-independent (CIM), platform-independent (PIM) and platform-specific (PSM) modeling

In order to support modeling of these rules in UML, a *UML-Based Rule Modeling Language (URML)* [1] has been developed, which extends UML metamodel with a notion of a rule and defines a visual notation for rules.

[1] The URML on I1 website http://www.rewerse.net/I1 or in REWERSE I1 deliverable D8.

In order to exchange rules between communities of UML modelers and ontology architects, the *rule markup framework R2ML* ([8]) has been developed. The R2ML accommodates main concepts of UML/OCL and OWL/SWRL, which allows rules capturing, expressed in datalog-like languages (f.e. SWRL) and functional languages (OCL). The visual rules modeling tool "Strelka" for derivation rules, production rules and reaction rules is currently under development[2]. The tool supports URML as a visual language for rules and serializes rule models into R2ML, which allows rules deployment into rule systems and rule reasoners.

As an example of the visual modeling of derivation rules, let's consider the rule formulation by domain expert in a natural language and rule visualization in a case tool.

Let's consider a rule example from the EU-Rent case study[2]:

If return branch of rental is different from pickup branch, then the rental is one way rental.

This is a *derivation rule*, which specifies how *one way rental* class is derived. The part of a business vocabulary, visualized by means of a class diagram is denoted in Fig. 2.

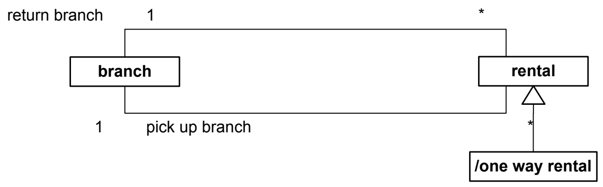


Fig. 2. Part of the EU-Rent business vocabulary

The OCL expression of this rule is:

```
context Rental inv:
if self.returnBranch<>self.pickupBranch
then self.oclIsKindOf(OneWayRental) endif
```

For visual representation of rules we introduce the following URML constructs:

Derivation rule expressed graphically as a circle with a rule identifier, "DR" in the circle stands for "Derivation Rule".

Condition arrow defines a relationship between a model element that is conditioned and the rule. An example of conditioned model element is a class or an association.

[2] Strelka tool description: http://rewerse.net/events/annual-meeting-2006/demos/i1-demos.html, Strelka home page on I1 website: http://www.rewerse.net/I1

Conclusion arrow defines a relationship between the rule and a derived model element. An example of derived model element is a class, an association or an attribute.

Using this modeling notation we may visualize the rule as depicted in Fig. 3. The boolean expression *returnBranch <> pickupBranch* at the beginning of

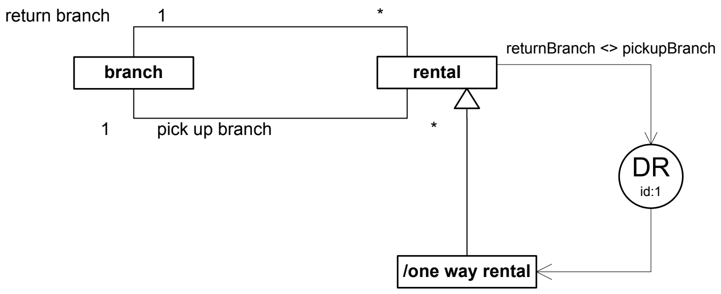


Fig. 3. Visualization of a derivation rule

the condition arrow is a filter. It filters rental objects with different return and pickup branches. Using this approach we may visualize different derivation rules, where classes, associations or attributes are derived. This visual representation corresponds to the following logical formula, where "." is a function, which return attribute value for an object:

$$x \in OneWayRental \longleftarrow x \in Rental \ and \ x.returnBranch <> x.pickupBranch$$

As can be seen from the example, the visualization of the rule is vivid and simple. For detailed description of the rules metamodel and more examples we refer to the website of the REWERSE Working Group I1.

Another important class of rules under consideration is production rules. These rules are widely used in business process automation, supported by several commercial tools and under standardization procedure of W3C.

As an example let's consider the following rule:

> *If the total amount of shopping cart of a customer is more than 100 give customer a voucher with value 10.*

The rule is represented as a circle with "PR" inside, which stands for "Production Rule", and an identifier. The voucher is created by means of so called *CreateAction* (denoted by character "C" near the arrow head) with a set of initialization parameters (Fig 4). In this example, parameter is a voucher value, which is set to 10 (*voucher := 10*).

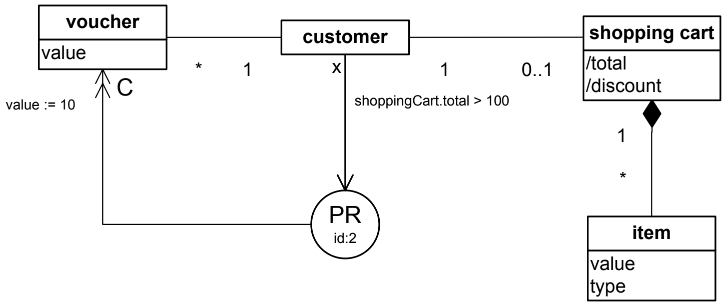


Fig. 4. Visualization of a production rule

5 Conclusion

Main advantages of the introduced rules visualization approach against technologies, described in Section 3, are:

Simplicity — with relatively small and simple extension of UML metamodel, visual modeling of main rule types can be implemented.

Visualization — visual representation of rules facilitates the use of rule-based technologies.

W3C Semantic Web and OMG MOF technologies — the solution under development for visual rules modeling may connect widely used OMG MOF methodologies with emerging Semantic Web technologies. For instance, UML case tools with support of rules may be used for modeling of Semantic Web applications, which include ontologies and rules.

Potential — the proposed method for visual rules modeling and verbalization introduces the possibility for rule–based software development, which is a powerful paradigm for special classes of software applications (for example: insurance, mortgage, business automation).

In this paper we have described a UML-based rule modeling approach. We have provided a quick overview of existing rule modeling technologies for business automation, software development and the Semantic Web and argue that our approach can be used for formalization of business requirements, which is demonstrated by means of two examples. The full specifications of R2ML and URML are available in the D8 deliverable of the REWERSE Working Group I1.

The modeling approach has been evaluated on several business case studies. The future work in this area is towards the approach evaluation on real business applications. The issue of modeling of reaction rules-driven Semantic Web Services is currently under consideration.

Acknowledgment

This research has been funded by the European Commission and by the Swiss State Secretariat for Education and Research within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. Object Constraint Language (OCL), Version 2.0, `http://www.omg.org/docs/ptc/03-10-14.eps`
2. EU-Rent case study on I1 website `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/12`
3. G. Klyne and J.J. Caroll (Eds.), Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C, 2004
4. Patel-Schneider, Peter F., Horroks I., OWL Web Ontology Language Semantic and Abstract Syntax, `http://www.w3.org/2004/OWL`
5. Patel-Schneider, Peter F., Horroks I., Boley H., Tabet S., Grosof B, Dean M., SWRL: A Semantic Web Rule Language Combining OWL and RuleML, `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521`
6. Ross, R. G., Principles of the Business Rule Approach. Addison-Wesley Information Technology Series (2003).
7. Wagner G., Antoniou G., Tabet S., Boley H.: The Abstract Syntax of RuleML — Towards a General Web Rule Language Framework
8. Wagner, G., Giurca, A., Lukichev, S. (2006). A Usable Interchange Format for Rich Syntax Rules. Integrating OCL, RuleML and SWRL. Will appear in proceedings of Reasoning on the Web Workshop at WWW2006, May 2006
9. Meta Object Facility (MOF) Core Specification, Version 2.0, January 2006, `http://www.omg.org/docs/formal/06-01-01.eps`
10. Unified Modeling Language (UML), version 2.0, `http://www.omg.org/technology/documents/formal/uml.htm`

Security for Multithreaded Programs Under Cooperative Scheduling

Alejandro Russo and Andrei Sabelfeld

Dept. of Computer Science and Engineering, Chalmers University of Technology
412 96 Göteborg, Sweden
Fax: +46 31 772 3663

Abstract. Information flow exhibited by multithreaded programs is subtle because the attacker may exploit scheduler properties when deducing secret information from publicly observable outputs. Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing sensitive timing behavior of protected commands and therefore prevents undesired information flows. While a useful construct, `protect` is nonstandard and difficult to implement. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

1 Introduction

Information-flow security specifications and enforcement mechanisms for sequential programs have been developed for several years. Unfortunately, they do not naturally generalize to multithreaded programs [17]. Information flow in multithreaded programs remains an important open challenge [12]. Furthermore, otherwise significant efforts (such as Jif [7] and Flow Caml [14]) in extending programming languages (such as Java and Caml) with information flow controls have sidestepped multithreading issues. Nevertheless, concurrency and multithreading are important in the context of security because environments of mutual distrust are often concurrent. As result, the need for controlling information flow in multithreaded programs has become a necessity.

This paper is focused on preventing attacks that exploit scheduler properties to deduce secret information from publicly observable outputs. Suppose h is a secret (or *high*) variable and l is a public (or *low*) one. Consider threads c_1 and c_2:

$$c_1 : (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\ l := 1$$
$$c_2 : \texttt{sleep}(50);\ l := 0$$

Although these threads do not exhibit insecure information flow in isolation (because 1 is always the outcome for l in c_1, and 0 is always the outcome for l in c_2), there is a race between assignments $l := 1$ and $l := 0$, whose outcome depends on secret h. If h is originally positive, then—under many schedulers—it is likely that the final value of l is 1. If h is not positive, then it is likely that the final value of l is 0. It is the timing behavior of thread c_1 that leaks—via the scheduler—secret information into l. This

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\alpha} \langle\!| c_i', m' |\!\rangle \qquad \alpha \in \{\epsilon, \vec{d}\} \qquad \sigma = i}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma, \langle c_1 \dots c_{i-1} c_i' \alpha c_{i+1} \dots c_n \rangle, m' |\!\rangle}$$

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\alpha} \langle\!| \mathit{stop}, m' |\!\rangle \qquad \sigma = i}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma, \langle c_1 \dots c_{i-1} c_{i+1} \dots c_n \rangle, m' |\!\rangle}$$

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\not\rightarrowtail} \langle\!| c_i', m |\!\rangle \qquad \sigma = i \qquad \sigma' = (i \bmod n) + 1 \qquad c_i' \neq \mathit{stop}}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma', \langle c_1 \dots c_{i-1} c_i' c_{i+1} \dots c_n \rangle, m |\!\rangle}$$

Fig. 1. Semantics for threadpools

phenomenon is due to *internal timing*, i.e., timing that is observable to the scheduler. As in [17, 18, 15, 1, 16, 8], we do not consider *external timing*, i.e., timing behavior visible to an attacker with a stopwatch.

Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing the timing behavior of the protected command and therefore prevents undesired information flows. A protected command is executed atomically *by definition*. Although it has been acknowledged [13, 8] that `protect` is hard to implement, no implementation of `protect` has been discussed by approaches that rely on it [18, 15, 16]. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. This transformation can be integrated into source-to-source translation that introduces `yield` commands for cooperative schedulers. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

2 Language

We consider a simple imperative language that includes `skip`, assignment, sequential composition, conditionals, and `while`-loops. Its sequential semantics is standard [20]. The language also includes dynamic thread creation and a `yield` command. A *command configuration* $\langle\!| c, m |\!\rangle$ consists of a command c and memory m. Memories $m : \mathit{IDs} \rightarrow \mathit{Vals}$ are finite maps from identifier names IDs to values Vals. Transitions between configurations have form $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$ where α is either ϵ (empty label), $\vec{d}$ (indicating a sequence of newly spawned threads), or $\not\rightarrowtail$. The latter label is used in the transition rule for `yield`:

$$\langle\!| \texttt{yield}, m |\!\rangle \xrightarrow{\not\rightarrowtail} \langle\!| \mathit{stop}, m |\!\rangle$$

Labels are propagated through sequential composition to the threadpool-semantics level. Dynamic thread creation is performed by command `fork`:

$$\langle\!| \texttt{fork}(c, \vec{d}), m |\!\rangle \xrightarrow{\vec{d}} \langle\!| c, m |\!\rangle$$

This has the effect of continuing with thread c while spawning a sequence of fresh threads $\vec{d}$. *Threadpool configurations* have form $\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle$ where σ is the scheduler's running thread number, $\langle c_1 \dots c_n \rangle$ is a threadpool, and m is a shared memory.

Threadpool semantics, describing the behavior of threadpools and their interaction with the scheduler, are displayed in Figure 1. The rules correspond to normal execution of thread i from the threadpool, termination of thread i, and yielding by thread i. Note that due to cooperative scheduling, only termination or a `yield` by a thread may change the decision of the scheduler which thread to run next. Although these semantics model a round-robin scheduler, our approach can be generalized to a wide class of schedulers.

Let $cfg \rightarrow^0 cfg$, for any configuration cfg, and $cfg \rightarrow^v cfg'$, for $v > 0$, if there is a configuration cfg'' such that $cfg \rightarrow cfg''$ and $cfg'' \rightarrow^{v-1} cfg'$. Then, $cfg \rightarrow^* cfg'$ if $cfg \rightarrow^v cfg'$ for some $v \geq 0$. Threadpool configuration cfg *terminates* in memory m (written $cfg \Downarrow m$) if $cfg \rightarrow^* \langle\!| \sigma, \langle\rangle, m |\!\rangle$ for some σ. In particular, $cfg \Downarrow^v m$ is written when $cfg \rightarrow^v \langle\!| \sigma, \langle\rangle, m |\!\rangle$. If $\langle\rangle$ is not finitely reachable from cfg, then cfg *diverges* (written $cfg \Uparrow$). Termination $\Downarrow$ and divergence $\Uparrow$ are defined similarly for command configurations.

3 Security Specification

We define two security conditions, termination-insensitive and termination-sensitive security, both based on *noninterference* [4]. Suppose *security environment* $\Gamma : IDs \rightarrow \{high, low\}$ specifies a partitioning of variables into high and low ones. Two memories m_1 and m_2 are *low-equal* ($m_1 =_L m_2$) if they agree on low variables, i.e., $\forall x \in IDs.\, \Gamma(x) = low \Longrightarrow m_1(x) = m_2(x)$.

Command c satisfies termination-insensitive noninterference if c's terminating executions on low-equal inputs produce low-equal results.

Definition 1. *Command c satisfies* termination-insensitive security *if*

$$\forall m_1, m_2. m_1 =_L m_2 \;\;\&\;\; \langle\!| 1, \langle c \rangle, m_1 |\!\rangle \Downarrow m_1' \;\;\&\;\; \langle\!| 1, \langle c \rangle, m_2 |\!\rangle \Downarrow m_2' \Longrightarrow m_1' =_L m_2'$$

Command c satisfies termination-sensitive noninterference if c's executions on any two low-equal inputs either both diverge or both terminate in low-equal results.

Definition 2. *Command c satisfies* termination-sensitive security *if*

$$\forall m_1, m_2. m_1 =_L m_2 \Longrightarrow$$
$$\langle\!| 1, \langle c \rangle, m_1 |\!\rangle \Downarrow m_1' \;\&\; \langle\!| 1, \langle c \rangle, m_2 |\!\rangle \Downarrow m_2' \;\&\; m_1' =_L m_2' \vee \langle\!| 1, \langle c \rangle, m_1 |\!\rangle \Uparrow \& \; \langle\!| 1, \langle c \rangle, m_2 |\!\rangle \Uparrow$$

4 Transformation

By performing a simple analysis while injecting `yield` commands, we are able to automatically enforce both termination-insensitive and termination-sensitive security. The transformation rules are presented in Figure 2. They have form $\Gamma \vdash c \hookrightarrow c'$, where command c is transformed into c' under Γ. In order to rule out *explicit flows* [2] via assignment, we ensure that expressions assigned to low variables may not depend on high data. This is enforced by demanding the type of the assigned variable to be at least as restrictive as the type of the expression that is to be assigned. Restrictiveness relation $\sqsubseteq$ on security levels is defined by $low \sqsubseteq low$, $high \sqsubseteq high$, $low \sqsubseteq high$ and $high \not\sqsubseteq low$.

$$\frac{\forall v \in \mathit{Vars}(e).\, \Gamma(v) = \mathit{low}}{\Gamma \vdash e : \mathit{low}} \qquad \frac{\exists v \in \mathit{Vars}(e).\, \Gamma(v) = \mathit{high}}{\Gamma \vdash e : \mathit{high}}$$

$$(\text{HCTX})\,\frac{\text{No } \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash c : \mathit{high}}$$

$$\frac{}{\Gamma \vdash \texttt{skip} \hookrightarrow \texttt{skip}; \texttt{yield}} \qquad \frac{}{\Gamma \vdash \texttt{yield} \hookrightarrow \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \sqsubseteq \Gamma(v)}{\Gamma \vdash v := e \hookrightarrow v := e; \texttt{yield}} \qquad \frac{\Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$\frac{\Gamma \vdash e : \mathit{low} \quad \Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1')\ \texttt{else}\ (\texttt{yield}; c_2')}$$

$$(\text{H-IF})\,\frac{\Gamma \vdash e : \mathit{high} \quad \Gamma \vdash c_1 : \mathit{high} \quad \Gamma \vdash c_2 : \mathit{high}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2); \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \mathit{low} \quad \Gamma \vdash c \hookrightarrow c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ (\texttt{yield}; c')); \texttt{yield}}$$

$$(\text{H-W})\,\frac{\Gamma \vdash e : \mathit{high} \quad \Gamma \vdash c : \mathit{high}}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ c); \texttt{yield}}$$

$$\frac{\Gamma \vdash c \hookrightarrow c' \quad \Gamma \vdash d_1 \hookrightarrow d_1' \quad \ldots \quad \Gamma \vdash d_n \hookrightarrow d_n'}{\Gamma \vdash \texttt{fork}(c, d_1 \ldots d_n) \hookrightarrow \texttt{fork}(c', d_1' \ldots d_n')}$$

Fig. 2. Transformation rules

In order to reject *implicit flows* [2] via control flow, we guarantee that `if`'s and `while`'s with high guards may not have assignments to low variables in their bodies. These two techniques are well known [2, 19] and do not require code transformation.

The transformation injects `yield` commands in such a way that threads may not yield whenever their timing information depends on secret data. This is achieved by a requirement that `if`'s and `while`'s with high guards may not contain `yield` commands. In addition, such control flow statements may not contain `fork`. The rationale is that if secrets influence the number of threads, then it is possible for some schedulers to leak this difference via races of publicly-observable assignments [13, 10]. Rules H-IF and H-W enforce the above requirements. The rest of the transformation injects `yield` commands without significant restrictions (but with some obvious liveness guarantees for commands that do not branch on secrets).

The first lemma shows that commands typed under rule HCTX do not affect the low-security variables.

Lemma 1. *Given a command c and memories m_1 and m_2 so that $\Gamma \vdash c : \mathit{high}$, $m_1 =_L m_2$, $\langle\!| c, m_1 |\!\rangle \Downarrow m_1'$, and $\langle\!| c, m_2 |\!\rangle \Downarrow m_2'$, then $m_1' =_L m_2'$.*

The following theorem states that pools of transformed threads preserve low-equality on memories:

Theorem 1. *Given two (possibly empty) threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_1|\!\rangle \Downarrow^v m_1'$, and $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_2|\!\rangle \Downarrow^w m_2'$, then $m_1' =_L m_2'$.*

Proof. The proof is done by induction on $v + w$. □

As desired, the transformation enforces termination-insensitive security:

Corollary 1. *If $\Gamma \vdash c \hookrightarrow c'$ then c' satisfies termination-insensitive security.*

Proof. By applying Theorem 1 with $\vec{c} = \langle c \rangle$, $\vec{c}\,' = \langle c' \rangle$, and $\sigma = 1$. □

The transformation can be adopted to termination-sensitive security in a straightforward way. We write $\Gamma \vdash_{\text{TS}} c \hookrightarrow c'$ whenever $\Gamma \vdash c \hookrightarrow c'$ with the modifications that (i) rule H-W is not used, and (ii) rule HCTX is replaced by:

$$\text{(HCTX')}\ \frac{\text{No } \texttt{while}, \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash_{\text{TS}} c : \mathit{high}}$$

These modifications ensure that loops have low guards and that no loop may appear in an `if` statement with a high guard. These requirements are similar to those of Volpano and Smith [18] (except for the requirement on `fork`, which Volpano and Smith lack):

Lemma 2. *Given a command c so that $\Gamma \vdash c : \mathit{high\ cmd}$ for some security environment Γ in Volpano and Smith's type system [18]; and given command c' obtained from c by erasing occurrences of* `protect`*, we have $\Gamma \vdash_{TS} c' : \mathit{high}$.*

Proof. By structural induction on the type derivation of c. □

This allows us to connect the transformation to Volpano and Smith's type system:

Theorem 2. *If command c is typable under security environment Γ in Volpano and Smith's type system [18], then there exists command c'' such that $\Gamma \vdash_{TS} c' \hookrightarrow c''$, where c' is obtained from c by erasing occurrences of* `protect`.

Proof. By structural induction on the type derivation of c and Lemma 2. □

We also achieve termination-sensitive security with the above modifications of the transformation. We firstly present some auxiliaries lemmas. The following lemma states that commands typed as *high* terminate and do not affect the low part of the memory:

Lemma 3. *Given a command c and memory m so that $\Gamma \vdash_{TS} c : \mathit{high}$, then $\langle\!|c, m|\!\rangle \Downarrow m'$ and $m =_L m'$.*

Proof. By induction on the size of c. □

In order to show termination-sensitive security, we track the behavior of threadpools after executing some number of `yield` and `fork` commands. We capture this by relation $\rightarrow^*_{y,f}$ so that $\mathit{cfg} \rightarrow^*_{1,0} \mathit{cfg}'$ if there is cfg'' such that $\mathit{cfg} \rightarrow^* \mathit{cfg}''$ where no `yield`'s have been executed, $\mathit{cfg}'' \rightarrow \mathit{cfg}'$ results from executing a `yield` command; and $\mathit{cfg} \rightarrow^*_{y,f} \mathit{cfg}'$ if there is cfg'' such that $\mathit{cfg} \rightarrow^*_{y-1,f} \mathit{cfg}''$ (resp. $\mathit{cfg} \rightarrow^*_{y,f-1} \mathit{cfg}''$) and $\mathit{cfg}'' \rightarrow \mathit{cfg}'$ results from executing a `yield` (resp. `fork`) command.

The next two lemmas state that low-equivalence between memories is preserved after executing some number of `yield` and `fork` commands:

Lemma 4. *Given two non-empty threadpools $\vec{c}$ and $\vec{c}'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash_{TS} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}' \rangle, m_1 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}' \rangle, m_2 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.*

Proof. By simple induction on the number of steps of $\rightarrow^*_{1,0}$. □

Lemma 5 (`yield`/`fork` *lock-step execution*)**.** *Given two non-empty threadpools $\vec{c}$ and $\vec{c}'$ of equal size, memories m_1 and m_2, numbers σ, y, and f so that $\Gamma \vdash_{TS} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}' \rangle, m_1 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}' \rangle, m_2 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.*

Proof. By induction on $y + f$ and by applying Lemmas 3 and 4 when necessary. □

The final theorem shows that the transformation eliminates the need for `protect`:

Theorem 3. *If $\Gamma \vdash_{TS} c \hookrightarrow c'$ then c' satisfies termination-sensitive security.*

Proof. By applying Lemma 5 with $\vec{c} = \langle c \rangle$, $\vec{c}' = \langle c' \rangle$, and $\sigma = 1$ and observing that a divergent configuration (originating from c') performs an infinite number of `yield`'s. □

5 Related Work

An overview of information flow controls for concurrent programs can be found in [12]. We briefly mention most closely related work. External timing-sensitive information-flow policies have been addressed for a multithreaded language [13], and extended with synchronization [9], message passing [11], and declassification [6]. Type systems have been investigated for termination-sensitive flows in possibilistic [1] and probabilistic [18, 15, 16] settings. Recently, we have presented a type system that guarantees termination-insensitive security with respect to a class of deterministic schedulers [8]. Information flow via low determinism, prohibiting races on low variables from the outset, has been addressed in [21, 5].

6 Conclusion

We have presented a transformation that prevents timing leaks via cooperative schedulers. We argue that this technique is general: it applies to a wide class of schedulers (although only a round-robin scheduler has been considered here for simplicity).

We have experimented with the GNU Pth [3], a portable thread library for threads in user space. We have modified this library to allow the round-robin scheduling policy from Section 2. We have successfully applied the transformation for source-to-source translation of multithreaded programs without `yield`'s into GNU Pth programs. The security of this translation is ensured by Theorems 1 and 3.

Acknowledgment. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 Mobius project.

References

[1] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[2] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[3] R. S. Engelschall. Gnu pth - the gnu portable threads. `http://www.gnu.org/software/pth/`, Nov. 2005.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[5] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

[6] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[7] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. `http://www.cs.cornell.edu/jif`, July 2001–2006.

[8] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[9] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[10] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 260–273. Springer-Verlag, July 2003.

[11] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[13] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[14] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[15] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[16] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.

[17] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[18] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[20] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[21] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

A Fully Dynamic Algorithm for Recognizing and Representing Chordal Graphs

Yrysgul Tursunbay kyzy

A.P. Ershov Institute of Informatics Systems
Russian Academy of Sciences, Siberian Branch
6, Acad. Lavrentjev pr., Novosibirsk, 630090, Russia
ryskulya@gmail.com

Abstract. This paper considers the problem of recognition and representation of dynamically changing chordal graphs. The input to the problem consists of a series of modifications to be performed on a graph, where modifications can be additions or deletions of complete r-vertex graphs. The purpose is to maintain a representation of the graph as long as it remains a chordal graph and to detect when it ceases to be so.

1 Introduction

A graph G is said to be *chordal* if every cycle of length 4 or more contains a chord (an edge between two non-consecutive vertices in a cycle). From the practical point of view, chordal graphs have numerous applications in, for example, sparse matrix computation (e.g., see [1]), relational databases [2], and computational biology [3].

Several authors have studied the problem of dynamically recognizing and representing various graph families. [4] devises a fully dynamic recognition algorithm for chordal graphs which handles edge operations in $O(n)$ time. The authors of [5] improve the current complexities for maintaining a chordal graph by starting with an empty graph and repeatedly adding or deleting edges. They use their result to ameliorate the time bound for the biology-based problem of improving the matrix representation of an evolutionary tree (phylogeny) which contains errors. For proper interval graphs [6], each update can be supported in $O(d+\log n)$ time where d is the number of edges involved in the operation.

Unlike the existing works, we develop an algorithm for maintaining representations of chordal graphs under complete r-vertex graph insertions or deletions, where cliques have at least 3 vertices. Since a clique tree of a chordal graph has at most n nodes, each operation performs in $O(n)$ time.

2 Preliminaries

In this article, we assume that the reader has a moderate familiarity with graph theory. This section aims at defining notions and notations related to chordal graphs.

Let $G = (V(G), E(G)) = (V, E)$ be a finite undirected and simple graph with $|V| = n$ vertices and $|E| = m$ edges. The subgraph of G *induced by* S is $G[S] = (S, E[S])$, where $E[S] = \{uv \in E \mid u, v \in S\}$. Let K_r be a complete r-vertex graph, where $r \geq 3$[1]. We define the following:

$$V(G) \cap V(K_r) = p, E(G) \cap E(K_r) = q,$$

$$G + K_r = \{V(G) + (V(K_r) \setminus p), E(G) + (E(K_r) \setminus q)\},$$

$$G - K_r = \{V(G) - (V(K_r) \setminus p), E(G) - E(K_r)\}.$$

A subset S of V is called a *separator* if $G[V \setminus S]$ is disconnected. S is a *uv-separator* if vertices u and v from $G[V \setminus S]$ are in different connected components of $G[V \setminus S]$; S is a *minimal uv-separator* if none of S subsets is a uv-separator. S is a *minimal separator* if S is a minimal uv-separator for all u and v from $G[V \setminus S]$.

A *clique* of a chordal graph G is a non-empty subset $C \subseteq V$ such that all the vertices of C are mutually adjacent. A clique K is *maximal* if K is not properly contained in another clique. A *clique tree* of G is a tree T such that its nodes have a 1-1 correspondence with maximal cliques of G, edges correspond to non-empty intersections of pairs of maximal cliques, and for all vertices v in G, the set of maximal cliques which contain v induces a subtree of T. It is worth remarking that a graph G is *chordal* iff it has a clique tree (see [3],[8],[9] for detailed explanation). We use u, v as vertex names of G and x, y as node names of T. The nodes x and y correspond to maximal cliques K_x and K_y of G. A clique tree has n nodes and each edge xy of T has the weight $w(xy) = |K_x \cap K_y|$. There are known algorithms to find a clique tree of a chordal graph in $O(m + n)$ time (see, e.g.,[1]).

We use $I_j = K_j \cap N(K_j)$ to denote a minimal separator of a graph G, where $N(K_j)$ is a set of all nodes of a tree T adjacent with node j.

3 Algorithm

We consider how to implement modification operations. Some of these operations are identical to those in [4], but are repeated here so that the reader can have easy access to the full algorithm.

Our algorithm supports the following operations: **Insert Query** and **Delete Query** which return "yes" if a modified graph ($G+K_r$ and $G-K_r$, respectively) is chordal and "no" otherwise; **Insert** and **Delete** modify the clique tree T according to made modification.

We first deal with the insertion of a complete r-vertex graph K_r.

Lemma 1. ([10], [11]) *Let G be a connected chordal graph with its clique tree T. Then*

[1] For $r = 1$ an incremental dynamic algorithm which considers addition of vertices is presented in [7], for $r = 2$ a dynamic algorithm which deals with addition and deletion of edges is proposed in [4].

(i) a set S is a minimal vertex separator of G iff $S = K_x \cap K_y$ for some edge $xy \in T$,
(ii) if $S = K_x \cap K_y$ for $xy \in T$, then S is a minimal uv-separator for any $u \in K_x \setminus S$ and $v \in K_y \setminus S$.

Theorem 1. *Let G be a chordal graph without a complete r-vertex graph K_r. Then $G + K_r$ is chordal iff the following conditions are satisfied:*

(i) G has a clique tree T with $u \in K_x, v \in K_y$ such that $u, v \in K_r$ for some edge xy in T,
(ii) there is a path from x to y in T such that $K_r \cap I_j \neq \varnothing$, where I_j is a set of vertices contained in this path.

Proof. (i) Let $I = K_x \cap K_y \neq \varnothing$. Since uv is not an edge of G, we have $u \notin K_y, v \notin K_x$ and hence $u \in K_x - I, v \in K_y - I$. By Lemma 1, I is a uv-separator.

Let C be any cycle in $G + K_r$ with length ≥ 4 that contains uv where $uv \in q$. Let $P = C - uv$, so that P is a path from u to v of length ≥ 3. Since I is a uv-separator, P must contain a vertex $s \in I$. Then either su or sv is a chord of P, which means C has a chord. Hence, $G + K_r$ is a chordal graph.

(ii) Let an edge $xy \notin T$ where $u \in K_x, v \in K_y$ for $u, v \in p$, then there exists the path P from x to y in T and a minimal separator I_j of G, containing in this path. Suppose to the contrary that there exists any node z in a clique tree T, such that $K_r \cap I_z = \varnothing$, which is contained in the path P. Let $I_z = \{u', v'\}$ and $K_z \cap K_x = u', K_z \cap K_y = v'$, then $u', v' \notin K_r$. Since $uv \in G + K_r$ and there exist the edges $uu' \in K_x, u'v' \in K_z, v'v \in K_y$, $G + K_r$ has a chordless cycle (u, u', v', v). We get a contradiction. □

Insert Query(K_r)

If the conditions of Theorem 1 are satisfied, return "yes", otherwise return "no".

End Insert Query

We next show how to update a clique tree for $G + K_r$.

Insert(K_r)

1. Consider such edges of $G + K_r$ that $uv \notin G$ with $u \in K_x, v \in K_y$ such that $u, v \in K_r$ for any $xy \in T$. If such edges do not exist in $G + K_r$, we pass to item 2. Otherwise, let $I = K_x \cap K_y$ then $K = I \cup \{u, v\}$ is a clique in $G + K_r$. As K is not a clique of G, we must add to a new node z with $K_z = K$. We must consider whether cliques K_x, K_y are maximal in $G + K_r$. Since $v \notin K_x$, then $K_x \subset K_z$ iff $K_x = I \cup \{u\}$ iff $|K_x| = |I| + 1$. Similarly, $u \notin K_y$, then $K_y \subset K_z$ iff $K_y = I \cup \{v\}$ iff $|K_y| = |I| + 1$. Thus, comparing $|K_x|, |K_y|$ and $w(x, y) = |I|$ we determine, whether cliques K_x and K_y is maximal in $G + K_r$.

   Replace xy in T with a new node z representing $K_z = I \cup \{u, v\}$ and add xz, yz, each with weight $|I| + 1$. Determine whether cliques K_x, K_y are

maximal in $G+K_r$. If K_x, K_y are maximal then we pass to item 2. Otherwise, if K_x is not the maximal clique, remove xz and replace x with z, if K_y is not the maximal clique, remove yz and replace y with z.
2. Add a new node r to T corresponding to K_r. Connect r with other nodes i such that $K_i \cap K_r \neq \varnothing$, attribute to it weights $w(i,r)$. Moreover, if $K_w \subset K_i$ for some $w \in T$ then K_w is not maximal clique in $G + K_r$ and we must remove w from T.

End Insert

We will now examine a deletion of a complete r-vertex graph K_r.

Theorem 2. *Let G be a chordal graph which contains a complete r-vertex graph K_r. Then $G - K_r$ is chordal iff the following conditions are satisfied:*

(i) the edge $uv \in K_r$ is contained exactly in two maximal cliques of G;
(ii) G does not contain any cycle consisting of vertices of the set $I_r = K_r \cap N(K_r)$.

Proof. (i) It is known that $uv \in K_r$, i.e. K_r is one of the maximal cliques containing this edge. Then uv must be contained exactly in the one clique of G except K_r. Suppose to the contrary that $uv \in q$ is contained in two cliques $\{u, v, s\}$ and $\{u, v, t\}$, where $st \notin G$, which are different from K_r. Then these two cliques cannot be contained in one maximal clique of G. In this case the deletion of K_r leads to the appearance of a chordless cycle (u, s, v, t) in $G - K_r$. We get a contradiction.

(ii) Suppose to the contrary that I_r forms a cycle C. Note that I_r forms a cycle iff it contains all vertices of a complete r-vertex graph K_r. Consider a case when $|N(Kr)| \geq 2$. Let $K_x, K_y \in N(K_r), xy \in T$ and $I_r = (K_x \cap K_r, K_y \cap K_r)$. Since I_r forms a cycle C, it is clear that $K_x \cap K_y \neq \varnothing$. By the definition of a chordal graph, all the cycles of G have length 3. The deletion of K_r leads to the disappearance of a third edge for each clique $N(K_r)$. Since these cliques are connected between themselves by the common vertices or edges, the cycle of length $2 \cdot |N(K_r)|$ appears in $G - K_r$. It means that $G - K_r$ has a cycle with length ≥ 4. We get a contradiction. □

Delete Query(K_r)

If the conditions of Theorem 2 are satisfied, return "yes", otherwise return "no".

End Delete Query

We show how to update a clique tree for $G - K_r$.

Delete(K_r)

1. Consider edges uv of G such that $uv \notin G - K_r$, with $u \in K_x, v \in K_y$ and $u, v \in K_r$ for any $xy \in T$. If such edges do not exist in $G - K_r$, pass to item 2. Otherwise, T of G contains a node z corresponding $K_z = K$ (see item 1

of Insert for the definition of K_z). In $G - K_r$, the maximal clique K_z has split into the cliques $K_z^u = K_z - \{v\}$ and $K_z^v = K_z - \{u\}$ which may not be maximal.

Divide the set $N(K_z)$ into $N_u = \{x \in N(z) \mid u \in K_x\}, N_v = \{y \in N(z) \mid v \in K_y\}$ and $N_w = \{w \in N(z) \mid u, v \notin K_w\}$. Then K_z^u is not maximal in $G - K_r$ iff $\exists x \in N_u$ such that $K_z^u \subset K_x$ and $w(x, z) = k - 1$. Similarly, K_z^v is not maximal in $G - K_r$ iff $\exists y \in N_v$ such that $K_z^v \subset K_y$ and $w(y, z) = k - 1$.

Replace z with two nodes z_1 and z_2 respectively representing K_z^u and K_z^v and add the edge $z_1 z_2$ with weight $w(z_1, z_2) = k - 2$. If $x \in N_u$, replace xz with xz_1. If $y \in N_v$, replace yz with yz_2. If $w \in N_w$, replace zw with $z_1 w$ or $z_2 w$.

If K_z^u and K_z^v are maximal cliques then pass to item 2. Otherwise, if K_z^u is not maximal because $K_z^u \subset K_{x_i}$ for some $x_i \in N_u$ then remove $x_i z_1$ and replace z_1 with x_i. Similarly, if K_z^v is not maximal because $K_z^v \subset K_{y_i}$ for some $y_i \in N_v$ then remove $y_i z_2$ and replace z_2 with y_i.

2. Remove r corresponding K_r from T.

End Delete

Corollary 1. *If $I_r = K_r \cap N(K_r)$ forms two or more different paths P_i, then K_r is a separator of G.*

Proof. Let P_1 and P_2 be two paths formed by $I_r = K_r \cap N(K_r)$. Let $K_x, K_y \in N(K_r)$ and $I_{r'} = K_x \cap K_r, I_{r''} = K_y \cap K_r$. Assume that $I_{r'} \subset P_1$ and $I_{r''} \subset P_2$. Then we have $K_x \cap K_y = \varnothing$. It means that deleting K_r leads to the appearance of two connected components, where cliques K_x and K_y are contained in the different connected components. Hence K_r is a separator of graph G. □

We use a clique tree T of a chordal graph G for performing the described operations. Since T has at most n nodes, each operation runs in $O(n)$ time.

4 Conclusions

In this paper, we described a fully dynamic algorithm, which considers new modifications of graphs, i.e. insertions or deletions of complete r-vertex graph, where $r \geq 3$. The proposed algorithm could be a suitable addition to the algorithm of Ibarra [10] for the maintenance of chordal graphs. Also, if it is known that the edges which should be added to the input graph G form a clique, then we are able to implement the algorithm more efficiently than if we were to add or delete the edges one by one.

Acknowledgment. The author would like to thank V.A. Evstegneev for the statement of the problem and I.B. Virbitskaite for her helpful comments and advice.

References

1. J. R. S. Blair and B. Peyton. *An introduction to chordal graphs and clique trees.* In Graph Theory and Sparse Matrix Computation, volume 56 of IMA, pp. 1-29. Ed.A.George and J.R.Gilbert and J.W.H.liu), Springer, 1993.

2. C. Beeri, R. Fagin, D. Maier and M. Yannakasis. *On the desirability of acyclic database schemes.* Journal of the ACM, 30:479-513, 1983.
3. P. Buneman. *A characterization of rigid circuit graphs.* Discrete Mathematics, 9:205-212, 1974.
4. L. Ibarra. *Fully dynamic algorithms for chordal graphs.* In Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), SIAM, Philadelphia, 1999, pp.923-924.
5. A. Berry, A. Sigayret, and J. Spinrad. *Faster dynamic algorithms for chordal graphs, and an application to Phylogeny.* In 31st Int. Workshop on Graph Theoretical Concepts in Computer Science (WG05), number 3787 in Lecture Notes in Computer Science, pp.445-455.
6. P. Hell, R. Shamir, and R. Sharan. *A fully dynamic algorithm for recognizing and representing proper interval graphs.* in Proceedings of t he 7th Annual European Symposium on Algorithms, Lecture Notes in Computer Science 1643, Springer-Verlag, 1999, pp. 527-539.
7. A. Berry, P. Heggernes and Y. Villanger. *A vertex incremental approach for dynamically maintaining chordal graphs.* Discrete Mathematics, 3063 (2006), pages 318-336.
8. F. Gavril. *The intersection graphs of subtrees in trees are exactly the chordal graphs.* J.Combinatorial Theory B, 1974, 16: pp.47-56.
9. J. R. Walter. *Representations of chordal graphs of a tree.* J.Graph Theory, 1978, 2: pp.265-267.
10. C. Ho and R. C. T. Lee. *Counting clique trees and computing perfect elimination schemes in parallel.* Information Processing Letters, 1989, 31: pp.61-68.
11. M. Lundquist. *Zero patterns, chordal graphs, and matrix completions.* PhD thesis, Dept.of Mathematical Sciences, Clemson University, 1990.

A Knowledge Portal for Cultural Information Resources: Towards an Architecture*

Yury Zagorulko[1], Jawed Siddiqi[2], Babak Akhgar[2], and Olesya Borovikova[1]

[1] A.P.Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev avenue, 630090, Novosibirsk, Russia
{zagor, olesya}@iis.nsk.su

[2] Informatics Research Group, Sheffield Hallam University
City Campus, Sheffield, South Yorkshire, UK S1-1WB
United Kingdom
{B.Akhgar, J.I.Siddiqi}@shu.ac.uk

Abstract. The paper presents a concept and architecture of a specialized Internet portal providing semantic access to cultural knowledge and information resources (i.e. electronic collections). The information basis of such a portal is an ontology that supports integration of information resources relevant to the subject domain of a portal into a uniform information space and provides content-based (semantic) access to these resources. The portal is adaptable to meet the needs of a variety of users: it provides multilingual access and supports a user's profile with personal preferences and areas of interests. The portal supports formulation of queries in terms of the chosen subject domain and ontology-driven navigation through the information space of the portal. Extension of the information space is realized by both experts and specialized subsystem searching for and automatically indexing relevant resources.

1 Introduction

A vast amount of information resources related to various fields of knowledge or culture has been accumulated by now. However, access to these resources is rather complicated, both because most of the fields of knowledge, especially in the humanities, are insufficiently formalized, and because these resources are disembodied, ill-structured, distributed over various Internet sites, electronic libraries and archives, and mostly inaccessible to the traditional search engines.

Besides, search engines themselves work inefficiently and quite often return to user a flood of useless information. The reason for this is that the modern search engines use primarily keyword search mechanisms, which are insensitive to the query semantics, and index Web resources with virtually no tools for analysis of the information presented in them.

* The first and last authors are grateful to the Russian Foundation for Basic Research (grant 04-01-00884) for financial support of this work. The other two authors are grateful to the EU IST FP6 for funding the MOSAICA project.

In recent years, there have been attempts to exploit thesauri and ontologies [1] to describe the Web resource semantics. There are many examples of tools currently being developed for semantic annotation of Web-pages and documents, when each document is linked to its semantic content. Using such annotations, the intelligent search agents provide more relevant responses to a user query as compared to existing engines. For example, to do this, the SHOE system [2] supplies HTML documents with a set of special tags for knowledge presentation, and the Semantic Web initiative [3] presumes supplying documents with annotations in the RDF language [4]. There has been certain progress in this direction, however that does not improve the situation in general, since Web-pages annotated in such a manner are an infinitesimal drop in the sea of the Web.

There exists a further problem that cannot be solved with semantic annotation of Web resources. When different groups of users communicate with search engines, they use both their own professional terminology and terms widely used in other communities with other meanings. This leads to decrease in relevancy of information found and in search transparency, because the existing search mechanisms do not take into consideration the context in which the information exists.

One more difficulty is that, as a rule, the user submits queries in his or her own language, while the Internet contains resources in different national languages. Not all search engines and sites provide quality translation of the query into other languages, and many are limited to one or two languages. Hence the user cannot get a large share of information.

To solve most of the problems mentioned above, we suggest to create specialized topic-oriented Internet portals that have to provide multilingual content-based (semantic) access to the knowledge and information resources for any given field of knowledge or culture. Such topic-oriented portal, or knowledge portal, would be useful for both a specialist in certain subject domain and ordinary users interested in getting knowledge and information resources on this topic.

The paper is structured as follows. Chapter 2 presents functions and main components of the knowledge portal. In Chapter 3, proposed architecture of the portal is described in detail. Related works are discussed in Chapter 4. Chapter 5 presents main results and conclusions.

2 The Proposed Knowledge Portal: Functions and Components

2.1 Functions of the Portal

The proposed knowledge Internet portal have to perform the following functions:

- provide access to knowledge and data on various aspects of culture and cultural activity, such as: components of the culture (monument of the culture, works of art, etc), the information about cultural events as well as persons, creative teamworks and organizations involved in the cultural process;

- integrate the multilingual resources on portal's subjects which are located in the Internet or in a local network;
- provide the user with advanced tools for navigating and finding the necessary information in the entire information space of the portal;
- provide the user with tools for multilingual semantic-based search in the Internet;
- provide the user with information support (for example, announcements on various events and actions);
- support a flexible user interface that takes into account the user's preferences with respect to language and the user's work with the resources and services.

2.2 Ontology of a Portal

The information basis of the portal is formed by the ontology [1,5,6] of the portal and descriptions of the network resources associated with it. The ontology is used for presentation of concepts and entities of given subject domain, integration of relevant Internet resources and external data sources in the information space of the portal as well as basis for navigation through this information space.

For a sufficiently complete and systematized representation of knowledge and information resources relating to certain kind of culture, the ontology of a portal includes two domain-independent ontologies such as ontology of cultural activity and ontology of culture as well as ontology of a subject domain.

The ontology of cultural activity includes the concepts related to organization of any cultural activity. For example, Organization, Person, Publication, Event, Exhibition, etc.

Ontology of culture contains both primary terms and concepts in use and the meta-notions which specify the structures for description of some kind of culture. In particular, it includes diverse taxonomies: hierarchy of types of the culture, hierarchy of schools of the culture, hierarchy of objects (artifacts) of the culture, etc. These hierarchies are connected by the relations presenting various associative connections between their elements. Thus ontology of culture serves for ontology of subject domain as meta-ontology.

Ontology of subject domain describes a certain kind of culture (for example, culture of Russia, Western culture, culture of ancient Rome). It is constructed on the base of ontology of culture.

Whilst appreciating that there a variety of approaches in developing a workable solution to understanding and constructing ontologies and there is much to be debated about the relative strengths and weaknesses of these approaches, for our purposes the following working definition will suffice: a system, which consists of a set of concepts associated by binary relations, their definitions and assertions (axioms and rules) allowing one to constrain (restrict) the meaning of concepts within some problem or subject domain. We consider that this presentation of ontology is adequate for our aims, i.e. integration of information resource and provision of semantic access to it.

2.3 Description of Information Resources

The description of information resources is an important component of the information content of a portal. It should be noted that information resource itself is notion of ontology and includes specific attributes and relations that link the resource with the other elements of the ontology.

The set of attributes and relations is based on Dublin Core [7] standard and includes the following units: Title of the resource, Subject of the resource, Resource type, Language, Access permissions, etc. Besides, each resource is linked to Semantic index which consists of a set of objects and relationships presenting the resource content in terms of the portal ontology.

The information space of a portal integrates structured resources (external databases), semi-structured resources (in HTML, XML, RDF) and unstructured resources (text documents).

3 Proposed Architecture

The knowledge portal has traditional three tier architecture: information access layer, information processing layer and the base layer (see Fig. 1).

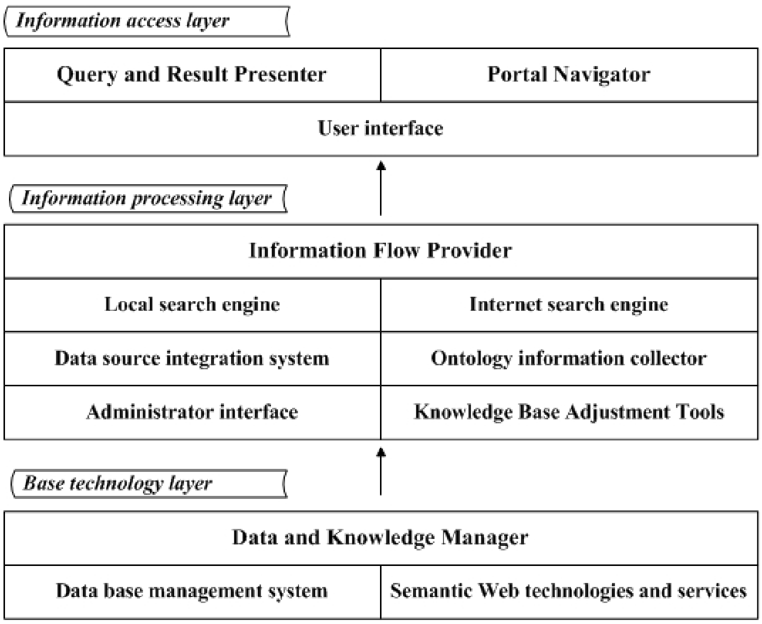

Information access layer
Query and Result Presenter
Portal Navigator
User interface
Information processing layer
Information Flow Provider
Local search engine
Internet search engine
Data source integration system
Ontology information collector
Administrator interface
Knowledge Base Adjustment Tools
Base technology layer
Data and Knowledge Manager
Data base management system
Semantic Web technologies and services

Fig. 1. The high level architecture of the knowledge portal

The information access layer provides front-end of the portal for user. The information processing layer supports all information (data) flows in the portal from construction of ontology to processing of user query. The base layer maintains data and knowledge management using database and Semantic Web technologies and services.

3.1 The Information Access Layer

The information access layer includes user interface which provides the users with presentation of his queries and results of searching as well as navigation through the information space of the portal.

Because of the use of an ontology this interface allows user to formulate queries in terms of the chosen subject domain and supports ontology-driven navigation through information objects, being instances of notions of the portal ontology, and the indexed information resources.

The interface is adaptable to meet the needs of a variety of users. For customization of the interface to a specific user or user group, the model of a user is used. This model contains language and subject preferences, the list of additionally connected/disabled resources, the technique for visualization of pages, etc. Note that the model of the user is updated at each logon and so it always represents the current user model or metaphorically "an information portrait".

3.2 The Information Processing Layer

The information processing layer provides local and Internet searching for information, data source integration, collection of ontology information, adjustment of knowledge base and the portal management.

The Internet search engine provides advanced semantic-based search in the Internet. The local search engine transmits a user query to the data source integration system and/or the location-based search module that perform query processing and information retrieval. In order to provide the user with multilingual semantic access to cultural resources a multilingual thesaurus is included in the portal.

An important component of the information processing layer is data source integration system which integrates accessible relevant external data sources in the information space of a portal and provides advanced information search in them. The specialized multi-agent system intended for semantic-based retrieval of information from a collection of distributed heterogeneous structured and semi-structured data sources is used as such system. In this system an access to each of data source is provided by a special agent. In order to make this system able to process user queries formulated in terms of the subject domain of the portal, the data schema of each external data source is mapped onto the ontology of the portal.

Facilities for adjustment of the portal are administrator interface and knowledge base adjustment tools. The administrator interface is used to set up and manage all subsystems of the portal. It also provides both linking to new information resources (external data sources) and user registration and management. The ontology editor and the thesaurus editor are intended for adjustment of the knowledge base. The data editor serves for creation and edition of information objects, that are instances of notions of the portal ontology, and their linking with other objects. All these tools are Web-applications and provide remote adjustment of the portal.

To solve the laborious problem concerned with a filling of a portal with new knowledge and data we suggest, in addition to the editors described above, to include a subsystem for extraction of knowledge and data from the Internet, which is called the ontology information collector. This subsystem performs search and collection of the relevant Internet resources (documents), their semantic analysis and indexing (annotating) based on thesaurus and ontology.

The ontology information collector comprises two basic modules: the Internet document collection module and the semantic indexing module. The former consists of the following components: search bot, link database, terminology dictionary. The search bot crawls through the Internet by following the links from the link database and searches for documents containing keywords from the terminology dictionary. The link database can be filled with links to new resources relevant to the portal topics both manually (by expert) and automatically (with links found in documents or discovered by the search mechanism of the portal, which runs at defined intervals). The semantic indexing module analyzed the document and builds its semantic index (annotation). Indexes of documents are stored in an internal database of the portal and used for searching. In this manner, the information content of the portal can be automatically updated.

3.3 The Base Technology Layer

The base technology layer performs management of data and knowledge which are stored in the knowledge base and internal data base.

The knowledge base is a key component of the portal. It includes the ontology of the portal and multilingual thesaurus elaborated in ISO 5964 standard. The thesaurus contains terms, i.e. words and phrases of several natural languages by means of which concept of ontology are presented in texts and queries of user. Existing relations between the thesaurus terms and the concepts of ontology create the prerequisites for their combined use in search and information processing.

For holding and manipulation of the knowledge base the Semantic Web technologies (ontologies) and services are used.

Internal database stores all local data, including information objects and their links (relations) with other objects, the indexes of documents, the descriptions of information resources and data sources. This database can be managed by traditional relational DBMS.

4 Related Work

Use of ontologies and other constituents of Semantic Web technologies for development of the knowledge portal allows one to relate it to a sort of Semantic Web Portal [8]. But our portal has particular features which makes it a portal of knowledge. In particular our portal provides content-based access to both information resource and the systematized knowledge of certain subject domain.

There exist several portals that use approaches and methods similar to ours. For example, the Esperonto Portal [9] is a case study of the ODESeW knowledge portal generator developed by the Ontology Group at Facultad de Informatica, Universidad Politecnica de Madrid. The information contained and its reliability makes the Esperonto portal one of the best sources on ontology research. Five different domain ontologies connected through several relations were developed for this portal.

The OntoWeb Portal [10] is a community portal for both academic and industrial partners who share an interest in the Semantic Web. The OntoWeb portal is structured according to an ontology which serves as a shared basis for supporting communication. OntoWeb community members can publish annotated information on the web, which is then crawled by a syndicator and stored in the portal knowledge base. However, the portal is a repository focused on the project and depends on the contribution of users in content.

Besides, a few attempts to build a culture web portals have been made, such as "The portal of culture of Latin American and the Caribbean"[1], Portal of UNESCO Culture Sector[2] and the portal "Culture of Russia"[3]. Some of the portals like this use also ontology-based approach, as for instance the semantic portal "MuseumFinland" that is intended for publishing heterogeneous museum collections on the Semantic Web which is discussed in [11].

5 Conclusion

The paper presents a concept and architecture of specialized Internet portal providing semantic access to the knowledge and information resources for any given field of knowledge or culture.

Let us underline the most important features of the portal that allow us to consider it as an Advanced Information System:

– Use of the common and subject knowledge at all levels and stages of portal operation: user, conceptual, logical and physical.
– Possibility of adjustability at all levels: subject domain, data and knowledge sources, the user's area of interests, and a user as an individual.
– Support of several languages. To support a new language, it is only necessary to add corresponding terms of this language to the thesaurus.
– Simplicity of information space extension. This is provided by the linking of new data sources through mapping the ontology into their data frameworks and the automatic indexing of semi-structured and non-structured resources relevant to the subject domain of the portal.

At present, the core of a portal including the ontology and data editors as well as facilities for search and navigation through information space of the portal

[1] http://www.lacult.org
[2] http://portal.unesco.org/culture
[3] http://www.culture.mincult.ru/

has been implemented. Based on the core, a specialized Internet portal providing semantic access to systematized knowledge and information resources relating to archeology has been successfully developed. This development demonstrates the soundness of the proposed approach and justifies the implementation of the complete portal from the core.

Our immediate goals are to develop a more detailed architecture for knowledge portal and its key components based on the concepts proposed. Further, we want to apply this architecture for development of a specialized Internet portal for multilingual semantic access to information resources relating to vanishing cultures. We plan to supplement the portal with thesaurus for several languages, first of all for Russian and English.

References

1. Benjamins V. R., Fensel D., et. all. "Community is Knowledge! in KA2" // Proc. of the KAW'98, Banff, Canada, — 1998.
2. Heflin J., Hendler J. Searching the Web with SHOE // Artificial Intelligence for Web Search. — Menlo Park: AAAI Press, 2000. — P. 35–40.
3. Berners-Lee T., Hendler J., Lassila O. The Semantic Web // Scientific American. — 2001. — Vol. 284, N 5. — P. 34–43.
4. Brickley D., Guha R. RDF Vocabulary Description Language 1.0: RDF Schema W3C Recommendation 10 February 2004. — http://www.w3.org/TR/2004/REC-rdf-schema-20040210/
5. Gruber T. Towards Principles for the Design of Ontologies Used for Knowledge Sharing // International Journal of Human and Computer Studies.— 1995. — Vol. 43, N 5/6. — P. 907–928.
6. Guariano N., Giaretta P. Ontologies and Knowledge Bases. Towards a Terminological Clarification // Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing / Ed. by N. J. I. Mars. — Amsterdam: IOS Press, 1995.
7. Dublin Core Metadata Initiative, Dublin Core Metadata Element Set, Version 1.1 — http://purl.org/dc/documents/rec-dces-19990702
8. Lausen H., Stollberg M., Hernandez R. L., Ding Y., Han S.-K., Fensel D. Semantic Web Portals — state of the art survey // Technical Report TR-2004-04-03, DERI (www.deri.org), 2004.
9. Corcho O., Gomez-Perez A., Lopez-Cima A., Lopez-Garcia V., Suarez-Figueroa MC. ODESeW. Automatic Generation of Knowledge Portals for Intranets and Extranets // The Semantic Web — ISWC 2003, Second International Semantic Web Conf., Sanibel Island, FL, USA, October 20-23, 2003, Lecture Notes in Computer Science, Vol 2870. — Springer-Verlag, 2003. — P. 802–817.
10. Spyns P., Oberle D., Volz R., Zheng J., Jarrar M., Sure Y., Studer R., Meersman R. OntoWeb a Semantic Web Community Portal // Proc. of the Fourth International Conf. on Practical Aspects of Knowledge Management (PAKM), December 2002, Vienna, Austria, LNAI 2569, Springer Verlag, 2002. — P.189–200.
11. Hyvonen E., Salminen M., Kettula S., Junnila M. A Content Creation Process for the Semantic Web // Proc. of the OntoLex2004: Ontologies and Lexical Resources in Distributed Environments, 2004, May 29, Lisbon, Portugal.

Author Index