BRICS NS-05-4 Smolka & Srba (eds.): INFINITY '05 Preliminary Proceedings

Basic Research in Computer Science

Preliminary Proceedings of the 7th International Workshop on

Verification of Infinite-State Systems

INFINITY '05

San Francisco, USA, August 27, 2005

Scott A. Smolka
Jiří Srba
(editors)

BRICS Notes Series **NS-05-4**

ISSN 0909-3206 **June 2005**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/05/4/`

Preliminary proceedings of

INFINITY 2005

Scott A. Smolka and Jiří Srba (Editors)

INFINITY 2005
7th International Workshop on Verification of Infinite-State Systems
A Satellite Workshop of CONCUR'05
San Francisco, California, USA, August 27, 2005

Preface

INFINITY 2005, the 7th International Workshop on Verification of Infinite-State Systems was held as a satellite workshop of CONCUR 2005 (the 16th International Conference on Concurrency Theory) in San Francisco, California, USA on August 27, 2005.

The aim of the workshop is to provide a forum for researchers interested in the development of mathematical techniques for the analysis and verification of systems with infinitely many states.

The topics of INFINITY 2005 included the following: techniques for modeling and analysis of infinite-state systems; equivalence checking and model checking infinite-state systems, parameterized systems, probabilistic and timed systems; calculi for mobility and security; finite-state abstractions of infinite-state systems; and data structures for infinite state spaces.

The volume consists of five contributed papers selected by the INFINITY 2005 programme committee. The programme of INFINITY 2005 was further enriched by an invited talk given by Antonín Kučera and by several short presentations.

We would like to thank the programme committee members for their support in composing the INFINITY 2005 programme, the CONCUR'05 Organizing Committee chaired by Luca de Alfaro for arranging all local affairs, and Uffe H. Engberg for his help with the proceedings. We gratefully acknowledge a finantial support from BRICS, Basic Research in Computer Science.

Final proceedings will appear in the ENTCS series published by Elsevier (`http://www.sciencedirect.com/science/journal/15710661`). We thank Michael Mislove, the managing editor of ENTCS, for providing this opportunity.

Aalborg and New York, June 2005

Jiří Srba
Scott A. Smolka

Programme Committee

Samik Basu	Ames (USA)
Petr Jančar	Ostrava (Czech Republic)
Richard Mayr	Raleigh (USA)
Ken McMillan	Berkeley (USA)
Faron Moller	Swansea (UK)
Philippe Schnoebelen	Cachan (France)
Scott Smolka (co-chair)	New York (USA)
Jiří Srba (co-chair)	Aalborg (Denmark)
Willem Visser	Moffett Field (USA)
Igor Walukiewicz	Talence Cedex (France)

Content

Invited Talk

Methods for Quantitative Analysis of Probabilistic Pushdown Automata 1
A. Kučera

Contributed Papers

Refining Undecidability Border of Weak Bisimilarity 3
M. Křetínský, V. Řehák, and J. Strejček

Abstract Regular Tree Model Checking 15
A. Boujjani, P. Habermehl, A. Rogalewicz and T. Vojnar

Automatic Verification of Fault-Tolerant Register Emulations 25
P.C. Attie and H. Chockler

Algorithmic Algebraic Model Checking III: Approximate Methods 37
V. Mysore and B. Mishra

Liveness Checking as Safety Checking for Infinite State Spaces 53
V. Schuppan and A. Biere

Methods for Quantitative Analysis of Probabilistic Pushdown Automata

Antonín Kučera [1,2]

Faculty of Informatics, Masaryk University in Brno
Botanická 68a, 60200 Brno, Czech Republic

Abstract

We present several constructions and techniques which have recently been used to tackle the problems of qualitative/quantitative analysis of probabilistic pushdown automata.

Key words: Probabilistic systems. Probabilistic temporal logics. Pushdown automata.

Probabilistic pushdown automata (pPDA) are a natural model for probabilistic programs with recursive procedure calls. Recently, various decidability/complexity questions about the problems of qualitative/quantitative analysis of pPDA were studied in [2,5,4,1,3]. Typical problems of interest include the following:

- *Reachability.* Given two pPDA configurations s, t, what is the probability of reaching t from s? In particular,
  - is this probability equal to one? (the qualitative reachability problem);
  - is this probability bounded by a given constant? (the quantitative reachability problem).
- *LTL model-checking.* Given a pPDA configuration s and an LTL formula φ, what is the probability that a run initiated in s satisfies φ?

  Similarly as above, one can formulate the qualitative/quantitative variant of the corresponding decision problem. Moreover, one can reformulate the problem for general ω-regular properties.
- *PCTL and PCTL* model-checking.* Given a pPDA configuration s and a formula φ of the probabilistic CTL or the probabilistic CTL*, does s satisfy φ?

[1] Supported by the research centre Institute for Theoretical Computer Science (ITI), project No. 1M0021620808.
[2] Email: kucera@fi.muni.cz

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- *Expectations and variances.* Given a pPDA configuration s, what is the expected length of a terminating run initiated in s? What is the corresponding variance?
- *Long-run properties.* A long-run property is a property defined for each run w separately by considering longer and longer prefixes of w and taking the limit of the corresponding sequence of approximations. A typical example is the average service time—if a system repeatedly services certain requests, then each run can be seen as an infinite sequence of finite services. The average service time (i.e., the average number of transitions needed to service a request) for a given run w is then defined by taking the limit of averages computed from longer and longer prefixes of w.

  Thus, one can formulate questions like "what is the probability that the average service time for a run initiated in a given configuration s does not exceed twenty transitions?"

Interestingly, most of the above given problems turned out to be decidable for pPDA, and the complexity bounds are relatively reasonable (some of these problems are even solvable in polynomial time). In this paper we present selected techniques and constructions which have been found useful when dealing with these problems.

References

[1] T. Brázdil, A. Kučera, and O. Stražovský. On the decidability of temporal properties of probabilistic pushdown automata. In *Proceedings of STACS'2005*, vol. 3404 of *Lecture Notes in Computer Science*, pp. 145–157. Springer, 2005.

[2] J. Esparza, A. Kučera, and R. Mayr. Model-checking probabilistic pushdown automata. In *Proceedings of LICS 2004*, pp. 12–21. IEEE Computer Society Press, 2004.

[3] J. Esparza, A. Kučera, and R. Mayr. Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In *Proceedings of LICS 2005*. IEEE Computer Society Press, 2005. To appear.

[4] K. Etessami and M. Yannakakis. Algorithmic verification of recursive probabilistic systems. In *Proceedings of TACAS 2005*, vol. 3440 of *Lecture Notes in Computer Science*, pp. 253–270. Springer, 2005.

[5] K. Etessami and M. Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of non-linear equations. In *Proceedings of STACS'2005*, vol. 3404 of *Lecture Notes in Computer Science*, pp. 340–352. Springer, 2005.

Refining the Undecidability Border of Weak Bisimilarity

Mojmír Křetínský [1] Vojtěch Řehák [2] Jan Strejček [3]

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{kretinsky,rehak,strejcek}@fi.muni.cz

Abstract

Weak bisimilarity is one of the most studied behavioural equivalences. This equivalence is undecidable for *pushdown processes* (PDA), *process algebras* (PA), and *multiset automata* (MSA, also known as *parallel pushdown processes*, PPDA). Its decidability is an open question for *basic process algebras* (BPA) and *basic parallel processes* (BPP). We move the undecidability border towards these classes by showing that the equivalence remains undecidable for weakly extended versions of BPA and BPP.

Key words: weak bisimulation, infinite-state systems, decidability

1 Introduction

Equivalence checking is one of the main streams in verification of concurrent systems. It aims at demonstrating some semantic equivalence between two systems, one of which is usually considered as representing the specification, the other its implementation or refinement. The semantic equivalences are designed to correspond to the system behaviours at the desired level of abstraction; the most prominent ones being strong and weak bisimulations.

Current software systems often exhibit an evolving structure and/or operate on unbounded data types. Hence automatic verification of such systems usually requires modeling them as infinite-state ones. Various specification formalisms have been developed with their respective advantages and limitations. Petri nets (PN), pushdown processes (PDA), and process algebras like BPA, BPP, or PA all serve to exemplify this. Here we employ the classes

[1] Supported by the Grant Agency of the Czech Republic, grant No. 201/03/1161.
[2] Supported by the research centre "Institute for Theoretical Computer Science (ITI)", project No. 1M0021620808.
[3] Supported by the Academy of Sciences of the Czech Republic, grant No. 1ET408050503.

of infinite-state systems defined by term rewrite systems and called *Process Rewrite Systems* (PRS) as introduced by Mayr [13]. PRS subsume a variety of the formalisms studied in the context of formal verification (e.g. all the models mentioned above). The relevance of various subclasses of PRS for modelling and analysing programs is shown, for example, in [5]; for automatic verification we refer to surveys [2,22].

The relative expressive power of various process classes has been studied, especially with respect to strong bisimulation; see [3,17], also [13] showing the hierarchy of PRS classes. Adding a finite-state control unit to the PRS rewriting mechanism results in so-called state-extended PRS (sePRS) classes, see for example [8]. We have extended the PRS hierarchy by sePRS classes and refined this extended hierarchy by introducing restricted state extensions of two types: PRS equipped with a weak finite-state unit (wPRS, inspired by weak automata [18]) [11,10] and PRS with finite constraint unit (fcPRS) [23].

Research on the expressive power of process classes has been accompanied by exploring algorithmic boundaries of various verification problems. In this paper we focus on the equivalence checking problem taking weak bisimilarity as the notion of behavioral equivalence.

The state of the art: Regarding sequential systems, i.e. those without parallel composition, the weak bisimilarity problem is undecidable for PDA even for the normed case [19]. However, it is conjectured [14] that weak bisimilarity is decidable for BPA; the best known lower bound is *EXPTIME*-hardness [14].

Considering parallel systems, even strong bisimilarity is undecidable for MSA [17] using the technique as introduced in [6]. However, it is conjectured [7] that the weak bisimilarity problem is decidable for BPP; the best known lower bound is *PSPACE*-hardness [20].

For the simplest systems combining both parallel and sequential operators, called PA processes [1], the weak bisimilarity problem is undecidable [21]. It is an open question for the normed PA; the best known lower bound is *EXPTIME*-hardness [14].

Our contribution: We move the undecidability border of the weak bisimilarity problem towards the classes of BPA and BPP, where the problem is conjectured to be decidable. We show that the problem remains undecidable for the weakly extended versions of both BPA (wPBA) and BPP (wBPP). In fact, the result is not new for wBPA: Mayr [14] has shown that adding a finite-state unit of the minimal non-trivial size 2 to the BPA process already makes weak bisimilarity undecidable. By inspection of his proof, we note that the result is valid for wBPA as well - see Sections 2 and 3 for the definition of wBPA and more detailed discussion.

2 Preliminaries

We recall the definitions of labelled transition system and weak bisimilarity. Then we define the syntax of process rewrite systems and (weak) finite-state unit extensions of PRS. Their semantics is given in terms of labelled transition systems.

Let $Act = \{a, b, \ldots\}$ be a set of *actions* such that Act contains a distinguished *silent action* τ. A *labelled transition system* is a pair $(S, \longrightarrow)$, where S is a set of *states* and $\longrightarrow \subseteq S \times Act \times S$ is a *transition relation*. We write $s_1 \xrightarrow{a} s_2$ instead of $(s_1, a, s_2) \in \longrightarrow$. The transition relation is extended to finite words over Act in the standard way. Further, we extend the relation to language $L \subseteq Act^*$ such that $s_1 \xrightarrow{L} s_2$ if $s_1 \xrightarrow{w} s_2$ for some $w \in L$. Moreover, we write $s_1 \longrightarrow^* s_2$ instead of $s_1 \xrightarrow{Act^*} s_2$. The *weak transition relation* $\Longrightarrow \subseteq S \times Act \times S$ is defined as $\stackrel{\tau}{\Longrightarrow} = \xrightarrow{\tau^*}$ and $\stackrel{a}{\Longrightarrow} = \xrightarrow{\tau^* a \tau^*}$ for all $a \neq \tau$.

A binary relation R on states of a labelled transition system is a *weak bisimulation* iff whenever $(s_1, s_2) \in R$ then for any $a \in Act$:

- if $s_1 \xrightarrow{a} s_1'$ then $s_2 \stackrel{a}{\Longrightarrow} s_2'$ for some s_2' such that $(s_1', s_2') \in R$ and
- if $s_2 \xrightarrow{a} s_2'$ then $s_1 \stackrel{a}{\Longrightarrow} s_1'$ for some s_1' such that $(s_1', s_2') \in R$.

States s_1 and s_2 are *weakly bisimilar*, written $s_1 \approx s_2$, iff $(s_1, s_2) \in R$ for some weak bisimulation R.

We use a characterization of weak bisimilarity in terms of a *bisimulation game*. This is a two-player game between an *attacker* and a *defender* played in rounds on pairs of states of a considered labelled transition system. In a round starting at a pair of states (s_1, s_2), the attacker first chooses $i \in \{1, 2\}$, an action $a \in Act$, and a state s_i' such that $s_i \xrightarrow{a} s_i'$. The defender then has to choose a state s_{3-i}' such that $s_{3-i} \stackrel{a}{\Longrightarrow} s_{3-i}'$. The states s_1', s_2' form a pair of starting states for the next round. A *play* is a maximal sequence of pairs of states chosen by players in the given way. The defender is the winner of every infinite play. A finite game is lost by the player who cannot make any choice satisfying the given conditions. It can be shown that two states s_1, s_2 of a labelled transition system are not weakly bisimilar if and only if the attacker has a winning strategy for the bisimulation game starting in these states.

Let $Const = \{X, \ldots\}$ be a set of *process constants*. The set of *process terms* (ranged over by $t, \ldots$) is defined by the abstract syntax $t ::= \varepsilon \mid X \mid t.t \mid t \| t$, where ε is the *empty term*, $X \in Const$ is a process constant; and '.' and '$\|$' mean *sequential* and *parallel composition* respectively. We always work with equivalence classes of terms modulo commutativity and associativity of '$\|$', associativity of '.', and neutrality of ε, i.e. $\varepsilon.t = t = t.\varepsilon$ and $t \| \varepsilon = t$. We distinguish four *classes of process terms* as:

1 – terms consisting of a single process constant only, in particular $\varepsilon \notin 1$,

S – *sequential* terms - terms without parallel composition, e.g. $X.Y.Z$,

P – *parallel* terms - terms without sequential composition, e.g. $X\|Y\|Z$,

G – *general* terms - terms with arbitrarily nested sequential and parallel compositions, e.g. $(X.(Y\|Z))\|W$.

Let α, β be classes of process terms $\alpha, \beta \in \{1, S, P, G\}$ such that $\alpha \subseteq \beta$. An *(α, β)-PRS (process rewrite system)* Δ is a finite set of *rewrite rules* of the form $t_1 \xrightarrow{a} t_2$, where $t_1 \in \alpha \setminus \{\varepsilon\}$, $t_2 \in \beta$ are process terms and $a \in Act$ is an action. Given a PRS Δ, let $Const(\Delta)$ and $Act(\Delta)$ be the respective (finite) sets of all constants and all actions which occur in the rewrite rules of Δ.

An (α, β)-PRS Δ determines a labelled transition system where states are process terms $t \in \beta$ over $Const(\Delta)$. The transition relation $\longrightarrow$ is the least relation satisfying the following inference rules (recall that '$\|$' is commutative):

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Delta}{t_1 \xrightarrow{a} t_2} \qquad \frac{t_1 \xrightarrow{a} t_2}{t_1\|t_1' \xrightarrow{a} t_2\|t_1'} \qquad \frac{t_1 \xrightarrow{a} t_2}{t_1.t_1' \xrightarrow{a} t_2.t_1'}$$

The formalism of process rewrite systems can be extended to include a finite-state control unit in the following way. Let $M = \{m, n, \ldots\}$ be a set of *control states*. Let $\alpha, \beta \in \{1, S, P, G\}, \alpha \subseteq \beta$ be the classes of process terms. An *(α, β)-sePRS (state extended process rewrite system)* Δ is a finite set of *rewrite rules* of the form $(m, t_1) \xrightarrow{a} (n, t_2)$, where $t_1 \in \alpha \setminus \{\varepsilon\}$, $t_2 \in \beta$, $m, n \in M$, and $a \in Act$. $M(\Delta)$ denotes the finite set of control states which occur in Δ.

An (α, β)-sePRS Δ determines a labelled transition system where states are the pairs of the form (m, t) such that $m \in M(\Delta)$ and $t \in \beta$ is a process term over $Const(\Delta)$. The transition relation $\longrightarrow$ is the least relation satisfying the following inference rules:

$$\frac{((m, t_1) \xrightarrow{a} (n, t_2)) \in \Delta}{(m, t_1) \xrightarrow{a} (n, t_2)} \qquad \frac{(m, t_1) \xrightarrow{a} (n, t_2)}{(m, t_1\|t_1') \xrightarrow{a} (n, t_2\|t_1')} \qquad \frac{(m, t_1) \xrightarrow{a} (n, t_2)}{(m, t_1.t_1') \xrightarrow{a} (n, t_2.t_1')}$$

To shorten our notation we write mt in lieu of (m, t).

An (α, β)-sePRS Δ is called a *process rewrite system with weak finite-state control unit* or just a *weakly extended process rewrite system*, written (α, β)-wPRS, if there exists a partial order $\leq$ on $M(\Delta)$ such that every rule $(m, t_1) \xrightarrow{a} (n, t_2)$ of Δ satisfies $m \leq n$.

Some classes of (α, β)-PRS correspond to widely known models as finite-state systems (FS), basic process algebras (BPA), basic parallel processes (BPP), process algebras (PA), pushdown processes (PDA, see [4] for justification), and Petri nets (PN). The other (α, β)-PRS classes were introduced and named as PAD, PAN, and PRS by Mayr [13]. The correspondence between (α, β)-PRS classes and the acronyms is given in Figure 1. Instead of (α, β)-sePRS or (α, β)-wPRS we use the prefixes 'se-' and 'w-' in connection with the acronym for the corresponding (α, β)-PRS class. For example, we use wBPA and wBPP rather than $(1, S)$-wPRS and $(1, P)$-wPRS, respectively.

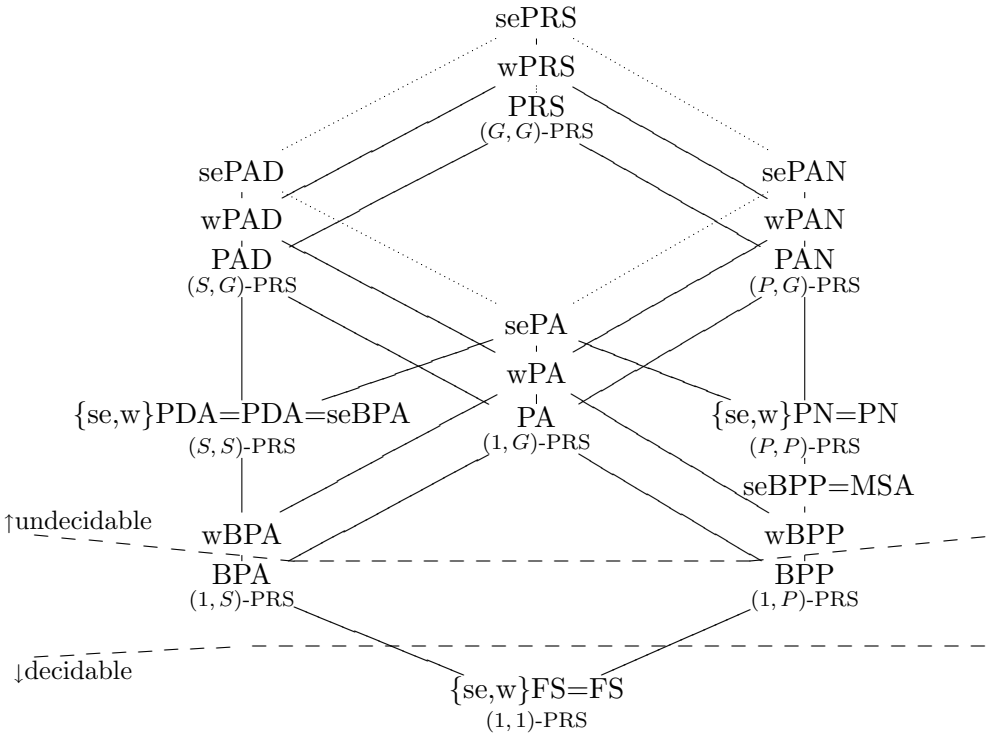


Fig. 1. The hierarchy with (un)decidability boundaries of $\approx$.

Finally, we note that seBPP are also known as multiset automata (MSA) or parallel pushdown processes (PPDA).

Figure 1 depicts relations between the expressive power of the considered classes. The expressive power of a class is measured by the set of labelled transition systems that are definable (up to strong bisimulation equivalence) by the class. A solid line between two classes means that the upper class is strictly more expressive than the lower one. A dotted line means that the upper class is at least as expressive as the lower class (and the strictness is just our conjecture). Details can be found in [11,10].

3 Undecidability of weak bisimilarity

In this section we show that weak bisimilarity is undecidable for the classes wBPA and wBPP. More precisely, we study the following two problems.

Problem: *Weak bisimilarity problem for wBPA (or wBPP respectively)*
Instance: A wBPA (or wBPP) system Δ and two of its states $mt, m't'$
Question: Are the two states weakly bisimilar?

3.1 wBPA

In [14] Mayr studied the question of how many control states are needed in PDA to make weak bisimilarity undecidable.

Theorem 3.1 ([14], Theorem 29) *Weak bisimilarity is undecidable for pushdown automata with only 2 control states.*

The proof is done by a reduction of Post's correspondence problem to the weak bisimilarity problem for PDA. The constructed PDA has only two control states, p and q. Quick inspection of the construction shows that the resulting pushdown automata are in fact wBPA systems as there is no transition rule changing q to p and each rule has only one process constant on the left hand side. Hence Mayr's theorem can be reformulated as follows.

Theorem 3.2 *Weak bisimilarity is undecidable for wBPA systems.*

3.2 wBPP

We show that the non-halting problem for Minsky 2-counter machines can be reduced to the weak bisimilarity problem for wBPP. First, let us recall the notions of Minsky 2-counter machine and the non-halting problem.

A *Minsky 2-counter machine*, or a *machine* for short, is a finite sequence

$$N = l_1 : i_1,\ l_2 : i_2,\ \ \ldots,\ l_{n-1} : i_{n-1},\ l_n : \texttt{halt}$$

where $n \geq 1$, $l_1, l_2, \ldots, l_n$ are *labels*, and each i_j is an instruction for

- *increment*: $\texttt{c}_k\texttt{:=c}_k\texttt{+1; goto } l_r$, or
- *test-and-decrement*: $\texttt{if c}_k\texttt{>0 then c}_k\texttt{:=c}_k\texttt{-1; goto } l_r \texttt{ else goto } l_s$

where $k \in \{1, 2\}$ and $1 \leq r, s \leq n$.

The semantics of a machine N is given by a labelled transition system the states of which are *configurations* of the form (l_j, v_1, v_2) where l_j is a label of an instruction to be executed and v_1, v_2 are nonnegative integers representing current values of counters $\texttt{c}_1$ and $\texttt{c}_2$, respectively. The transition relation is the smallest relation satisfying the following conditions: if i_j is an instruction of the form

- $\texttt{c}_1\texttt{:=c}_1\texttt{+1; goto } l_r$, then $(l_j, v_1, v_2) \xrightarrow{inc} (l_r, v_1 + 1, v_2)$ for all $v_1, v_2 \geq 0$;
- $\texttt{if c}_1\texttt{>0 then c}_1\texttt{:=c}_1\texttt{-1; goto } l_r \texttt{ else goto } l_s$, then $(l_j, v_1 + 1, v_2) \xrightarrow{dec} (l_s, v_1, v_2)$ and $(l_j, 0, v_2) \xrightarrow{zero} (l_r, 0, v_2)$ for all $v_1, v_2 \geq 0$;

and the analogous condition for instructions manipulating $\texttt{c}_2$. We say that the (computation of) machine N *halts* if there are numbers $v_1, v_2 \geq 0$ such that $(l_1, 0, 0) \longrightarrow^* (l_n, v_1, v_2)$. Let us note that the system is deterministic, i.e. for every configuration there is at most one transition leading from the configuration.

The *non-halting problem* is to decide whether a given machine N does not halt. The problem is undecidable [16].

Let us fix a machine $N = l_1 : i_1,\ l_2 : i_2,\ \ \ldots,\ l_{n-1} : i_{n-1},\ l_n : \texttt{halt}$. We construct a wBPP system Δ such that its states $simL_1$ and $simL'_1$ are weakly bisimilar if and only if N does not halt. Roughly speaking, we create a set

of wBPP rules allowing us to simulate the computation of N by two separate sets of terms. If the instruction `halt` is reached in the computation of N, the corresponding term from one set can perform the action *halt*, while the corresponding term from the other set can perform the action *halt*$'$. Therefore, the starting terms are weakly bisimilar if and only if the machine does not halt.

The wBPP system Δ we are going to construct uses five control states, namely $sim, check_1, check_1', check_2, check_2'$. We associate each label l_j and each counter $\mathtt{c}_k$ with process constants L_j, L_j' and X_k, Y_k respectively. A parallel composition of x copies of X_k and y copies of Y_k, written $X_k^x \| Y_k^y$, represents the fact that the counter $\mathtt{c}_k$ has the value $x - y$. Hence, terms $simL_j \| X_1^{x_1} \| Y_1^{y_1} \| X_2^{x_2} \| Y_2^{y_2}$ and $simL_j' \| X_1^{x_1} \| Y_1^{y_1} \| X_2^{x_2} \| Y_2^{y_2}$ are associated with a configuration $(l_j, x_1 - y_1, x_2 - y_2)$ of the machine N. Some rules contain auxiliary process constants. In what follows, β stands for a term of the form $\beta = X_1^{x_1} \| Y_1^{y_1} \| X_2^{x_2} \| Y_2^{y_2}$. The control states $check_k, check_k'$ for $k \in \{1, 2\}$ are intended for testing emptiness of the counter $\mathtt{c}_k$. The only rules applicable in these control states are:

$$check_1 X_1 \xrightarrow{chk_1} check_1 \varepsilon \qquad check_2 X_2 \xrightarrow{chk_2} check_2 \varepsilon$$
$$check_1' Y_1 \xrightarrow{chk_1} check_1' \varepsilon \qquad check_2' Y_2 \xrightarrow{chk_2} check_2' \varepsilon$$

One can readily confirm that $check_k \beta \approx check_k' \beta$ if and only if the value of $\mathtt{c}_k$ represented by β equals zero.

In what follows we construct a set of wBPP rules for each instruction of the machine N. At the same time we argue that the only chance for the attacker to win is to simulate the machine without cheating as every cheating can be punished by the defender's victory. This attacker's strategy is winning if and only if the machine halts.

Halt: l_n : `halt`

Halt instruction is translated into the following two rules:

$$simL_n \xrightarrow{halt} sim\varepsilon \qquad simL_n' \xrightarrow{halt'} sim\varepsilon$$

Clearly, *the states* $simL_n \| \beta$ *and* $simL_n' \| \beta$ *are not weakly bisimilar.*

Increment: l_j : `c`$_k$`:=c`$_k$`+1; goto` l_r

For each such instruction of the machine N we add the following rules to Δ:

$$simL_j \xrightarrow{inc} simL_r \| X_k \qquad simL_j' \xrightarrow{inc} simL_r' \| X_k$$

Obviously, *every round of the bisimulation game starting at states* $simL_j \| \beta$ *and* $simL_j' \| \beta$ *ends up in states* $simL_r \| X_k \| \beta$ *and* $simL_r' \| X_k \| \beta$.

Test-and-decrement: l_j: `if` $\mathtt{c}_k$`>0 then` $\mathtt{c}_k$`:=` $\mathtt{c}_k$`-1; goto` l_r `else goto` l_s

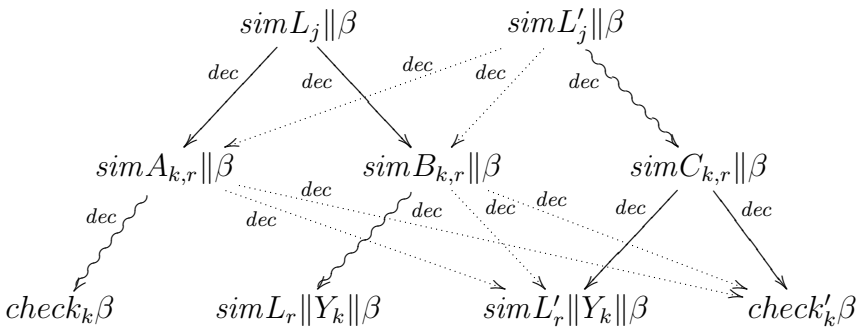
For any such instruction of the machine N we add two sets of rules to Δ, one for the $\mathtt{c}_k > 0$ case and the other for the $\mathtt{c}_k = 0$ case. The wBPP formalism has no power to rewrite a process constant corresponding to a label l_j and to check whether $\mathtt{c}_k > 0$ at the same time. Therefore, in the bisimulation game it is the attacker who has to decide whether $\mathtt{c}_k > 0$ holds or not, i.e. whether he will play an action *dec* or an action *zero*. We show that whenever the attacker tries to cheat, the defender can win the game.

At this point our construction of wBPP rules uses a variant of the technique called *defender's choice* [9]. In a round starting at the pair of states s_1, s_2, the attacker is forced to choose one specific transition (indicated by a wavy arrow henceforth). If he chooses a different transition, say $s_k \xrightarrow{a} s$ where $k \in \{1, 2\}$, then there exists a transition $s_{3-k} \xrightarrow{a} s$ that enables the defender to reach the same state and win the play. The name of this technique refers to the fact that after the attacker chooses the specific transition, the defender can choose an arbitrary transition with the same label. These transitions are indicated by solid arrows. The dotted arrows stands for auxiliary transitions which compel the attacker to play the specific transition.

First, we discuss the following rules designed for the $\mathtt{c}_k > 0$ case:

$$
\begin{array}{lll}
simL_j \xrightarrow{dec} simA_{k,r} & simA_{k,r} \xrightarrow{dec} check_k\varepsilon & simB_{k,r} \xrightarrow{dec} simL_r \| Y_k \\
simL_j \xrightarrow{dec} simB_{k,r} & simA_{k,r} \xrightarrow{dec} simL'_r \| Y_k & simB_{k,r} \xrightarrow{dec} simL'_r \| Y_k \\
simL'_j \xrightarrow{dec} simA_{k,r} & simA_{k,r} \xrightarrow{dec} check'_k\varepsilon & simB_{k,r} \xrightarrow{dec} check'_k\varepsilon \\
simL'_j \xrightarrow{dec} simB_{k,r} & & simC_{k,r} \xrightarrow{dec} simL'_r \| Y_k \\
simL'_j \xrightarrow{dec} simC_{k,r} & & simC_{k,r} \xrightarrow{dec} check'_k\varepsilon
\end{array}
$$

The situation can be depicted as follows.



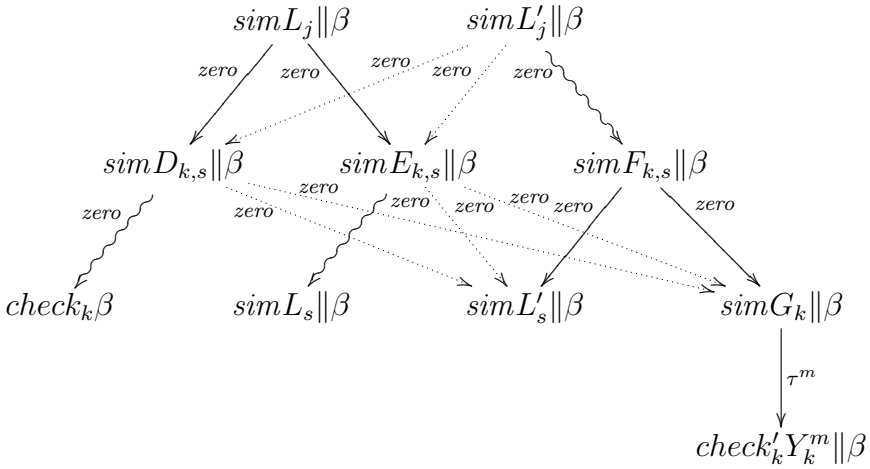
Let us assume that in a round starting at states $simL_j\|\beta$, $simL'_j\|\beta$ the attacker decides to perform the action *dec*. Due to the principle of defender's choice employed here, the attacker has to play the transition $simL'_j\|\beta \xrightarrow{dec} simC_{k,r}\|\beta$, while the defender can choose between the transitions leading from $simL_j\|\beta$ either to $simA_{k,r}\|\beta$ or to $simB_{k,r}\|\beta$. Thus, the round will finish either in states

$simA_{k,r}\|\beta, simC_{k,r}\|\beta$ or in states $simB_{k,r}\|\beta, simC_{k,r}\|\beta$. Using the defender's choice again, one can easily see that the next round ends up in $check_k\beta$ or $simL_r\|Y_k\|\beta$, and $simL'_r\|Y_k\|\beta$ or $check'_k\beta$. The exact combination is chosen by the defender. The defender will not choose any pair of states where one control state is *sim* and the other is not as such states are clearly not weakly bisimilar. Hence, the two considered rounds of the bisimulation game end up in a pair of states either $simL_r\|Y_k\|\beta, simL'_r\|Y_k\|\beta$ or $check_k\beta, check'_k\beta$. The latter pair is weakly bisimilar iff the value of c_k represented by β is zero, i.e. iff the attacker cheats when he decides to play an action *dec*. This means that *if the attacker cheats, the defender wins. If the attacker plays the action dec correctly, the only chance for either player to force a win is to finish these two rounds in states $simL_r\|Y_k\|\beta, simL'_r\|Y_k\|\beta$ corresponding to the correct simulation of an test-and-decrement instruction with a label l_j.*

Now, we focus on the following rules designed for the $\mathsf{c}_k = 0$ case:

$$
\begin{array}{lll}
simL_j \xrightarrow{zero} simD_{k,s} & simD_{k,s} \xrightarrow{zero} check_k\varepsilon & simE_{k,s} \xrightarrow{zero} simL_s \\
simL_j \xrightarrow{zero} simE_{k,s} & simD_{k,s} \xrightarrow{zero} simL'_s & simE_{k,s} \xrightarrow{zero} simL'_s \\
simL'_j \xrightarrow{zero} simD_{k,s} & simD_{k,s} \xrightarrow{zero} simG_k & simE_{k,s} \xrightarrow{zero} simG_k \\
simL'_j \xrightarrow{zero} simE_{k,s} & simF_{k,s} \xrightarrow{zero} simL'_s & simG_k \xrightarrow{\tau} simG_k\|Y_k \\
simL'_j \xrightarrow{zero} simF_{k,s} & simF_{k,s} \xrightarrow{zero} simG_k & simG_k \xrightarrow{\tau} check'_kY_k
\end{array}
$$

The situation can be depicted as follows.



Let us assume that the attacker decides to play the action *zero*. The defender's choice technique allows the defender to control the two rounds of the bisimulation game starting at states $simL_j\|\beta$ and $simL'_j\|\beta$. The two rounds end up in a pair of states $simL_s\|\beta, simL'_s\|\beta$ or in a pair of the form $check_k\beta, check'_kY_k^m\|\beta$ where $m \geq 1$; all the other choices of the defender lead to his loss. As in the previous case, the latter possibility is designed to punish any possible at-

tacker's cheating. The attacker is cheating if he plays the action *zero* and the value of c_k represented by β, say v_k, is positive.[4] In such a case, the defender can control the two rounds to end up in states $check_k\beta, check'_k Y_k^{v_k} \| \beta$ which are weakly bisimilar. If the attacker plays correctly, i.e. the value of c_k represented by β is zero, then the defender has to control the two discussed rounds to end up in states $simL_s \| \beta, simL'_s \| \beta$ as the states $check_k\beta, check'_k Y_k^m \| \beta$ are not weakly bisimilar for any $m \geq 1$. To sum up, *the attacker's cheating can be punished by defender's victory. If the attacker plays correctly, the only chance for both players to win is to end up after the two rounds in states* $simL_s \| \beta, simL'_s \| \beta$ *corresponding to the correct simulation of the considered instruction.*

It has been argued that if each of the two players wants to win, then both players will correctly simulate the computation of the machine N. The computation is finite if and only if the machine halts. The states $simL_1$ and $simL'_1$ are not weakly bisimilar in this case. If the machine does not halt, the play is infinite and the defender wins. Hence, the two states are weakly bisimilar in this case. In other words, *the states* $simL_1$ *and* $simL'_1$ *of the constructed wBPP* Δ *are weakly bisimilar if and only if the Minsky 2-counter machine* N *does not halt.*

Theorem 3.3 *Weak bisimilarity is undecidable for wBPP systems.*

4 Conclusion

We have shown that the weak bisimilarity problem remains undecidable for weakly extended versions of BPP (wBPP) and BPA (wBPA) process classes.

We note that the result for wBPA is just our interpretation (in terms of weakly extended systems) of Mayr's proof showing that the problem is undecidable for PDA with two control states ([14], Theorem 29).

In terms of parallel systems, our technique used for wBPP is new. To mimic the computation of a Minsky 2-counter machine, one has to be able to maintain its state information – the label of a current instruction and the values of the counters c_1 and c_2. As the finite-state unit of wBPP is weak, it cannot be used to store even a part of such often changing information. Hence, contrary to the constructions in more expressive systems (PN [6] and MSA [17]) we have made the term part to manage it as follows. In a test-and-decrement instruction the process constant L_j has to be changed and moreover one of the counters has to be decreased at the same time. As two process constants cannot be rewritten by one wBPP rewrite rule, we introduce new process constants Y_1 and Y_2 to represent inverse elements to X_1 and X_2 respectively and we make a term $X_k^x \| Y_k^y$ to represent the counter c_k the value

[4] We do not have to consider the case when β represents a negative value of c_k as such a state is reachable in the game starting in states $simL_1, simL'_1$ only by unpunished cheating.

of which is $x - y$. We note that the weak state unit allows for controlling the correct order of the successive stages in the progress of a bisimulation game.

In fact, our results hold for even a bit more restricted classes fcBPA and fcBPP (see [23] for the definitions of fcBPA and fcBPP) and remain valid for the normed subclasses of fcBPP and fcBPA [12]. Hence, they hold for normed wBPP and normed wBPA as well. Due to the technical nature of the presentation we have demonstrated the results for (unnormed) wBPP and (unnormed) wBPA only.

We recall that the decidability of weak bisimilarity is an open question for BPA and BPP. We note that these problems are conjectured to be decidable (see [14] and [7] respectively) in which case our results would establish a fine undecidability border of weak bisimilarity.

Acknowledgements. We would like to thank Jiří Srba for valuable suggestions and comments.

References

[1] Baeten, J. and W. Weijland, "Process Algebra," Number 18 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[2] Burkart, O., D. Caucal, F. Moller and B. Steffen, *Verification on infinite structures*, in: *Handbook of Process Algebra* (2001), pp. 545–623.

[3] Burkart, O., D. Caucal and B. Steffen, *Bisimulation collapse and the process taxonomy*, in: *Proc. of CONCUR'96*, LNCS **1119** (1996), pp. 247–262.

[4] Caucal, D., *On the regular structure of prefix rewriting*, Theoretical Computer Science **106** (1992), pp. 61–86.

[5] Esparza, J., *Grammars as processes*, in: *Formal and Natural Computing*, LNCS **2300** (2002).

[6] Jančar, P., *Undecidability of bisimilarity for Petri nets and some related problems*, Theoretical Computer Science **148** (1995), pp. 281–301.

[7] Jančar, P., *Strong bisimilarity on basic parallel processes is PSPACE-complete*, in: *Proc. of 18th IEEE Symposium on Logic in Computer Science (LICS'03)* (2003), pp. 218–227.

[8] Jančar, P., A. Kučera and R. Mayr, *Deciding bisimulation-like equivalences with finite-state processes*, Theoretical Computer Science **258** (2001), pp. 409–433.

[9] Jančar, P. and J. Srba, *Highly undecidable questions for process algebras*, in: *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science (TCS'04)*, Exploring New Frontiers of Theoretical Informatics (2004), pp. 507–520.

[10] Křetínský, M., V. Řehák and J. Strejček, *Extended process rewrite systems: Expressiveness and reachability*, in: P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, LNCS **3170** (2004), pp. 355–370.

[11] Křetínský, M., V. Řehák and J. Strejček, *On extensions of process rewrite systems: Rewrite systems with weak finite-state unit*, in: P. Schnoebelen, editor, *INFINITY 2003: 5th International Workshop on Verification of Infinite-State Systems*, Electronic Notes in Theoret. Computer Science **98** (2004), pp. 75–88.

[12] Křetínský, M., V. Řehák and J. Strejček, *Refining the undecidability border of weak bisimilarity*, Technical Report FIMU-RS-2005-06, Faculty of Informatics, Masaryk University (2005), a full version of this paper.

[13] Mayr, R., *Process rewrite systems*, Information and Computation **156** (2000), pp. 264–286.

[14] Mayr, R., *Weak bisimilarity and regularity of context-free processes is EXPTIME-hard*, Theoretical Computer Science **330** (2005), pp. 553–575.

[15] Milner, R., "Communication and Concurrency," Prentice-Hall, 1989.

[16] Minsky, M. L., "Computation: Finite and Infinite Machines," Prentice-Hall, 1967.

[17] Moller, F., *Infinite results*, in: *Proc. of CONCUR'96*, LNCS **1119** (1996), pp. 195–216.

[18] Muller, D., A. Saoudi and P. Schupp, *Alternating automata, the weak monadic theory of trees and its complexity*, Theoret. Computer Science **97** (1992), pp. 233–244.

[19] Srba, J., *Undecidability of weak bisimilarity for pushdown processes*, in: *Proceedings of 13th International Conference on Concurrency Theory (CONCUR'02)*, LNCS **2421** (2002), pp. 579–593.

[20] Srba, J., *Complexity of weak bisimilarity and regularity for BPA and BPP*, Mathematical Structures in Computer Science **13** (2003), pp. 567–587.

[21] Srba, J., *Undecidability of weak bisimilarity for PA-processes*, in: *Proceedings of the 6th International Conference on Developments in Laguage Theory (DLT'02)*, LNCS **2450** (2003), pp. 197–208.

[22] Srba, J., *Roadmap of infinite results*, in: *Current Trends In Theoretical Computer Science, The Challenge of the New Century, Vol 2: Formal Models and Semantics* (2004), pp. 337–350, `http://www.brics.dk/~srba/roadmap/`.

[23] Strejček, J., *Rewrite systems with constraints, EXPRESS'01*, Electronic Notes in Theoretical Computer Science **52** (2002).

Abstract Regular Tree Model Checking

Ahmed Bouajjani, Peter Habermehl [1,2]

LIAFA, University Paris 7, Case 7014, 2, place Jussieu, F-75251 Paris Cedex 05, France

Adam Rogalewicz, Tomáš Vojnar [3,4]

FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic

Abstract

Regular (tree) model checking (RMC) is a promising generic method for formal verification of infinite-state systems. It encodes configurations of systems as words or trees over a suitable alphabet, possibly infinite sets of configurations as finite word or tree automata, and operations of the systems being examined as finite word or tree transducers. The reachability set is then computed by a repeated application of the transducers on the automata representing the currently known set of reachable configurations. In order to facilitate termination of RMC, various acceleration schemas have been proposed. One of them is a combination of RMC with the abstract-check-refine paradigm yielding the so-called abstract regular model checking (ARMC). ARMC has originally been proposed for word automata and transducers only and thus for dealing with systems with linear (or easily linearisable) structure. In this paper, we propose a generalisation of ARMC to the case of dealing with trees which arise naturally in a lot of modelling and verification contexts. In particular, we first propose abstractions of tree automata based on collapsing their states having an equal language of trees up to some bounded height. Then, we propose an abstraction based on collapsing states having a non-empty intersection (and thus "satisfying") the same bottom-up tree "predicate" languages. Finally, we show on several examples that the methods we propose give us very encouraging verification results.

1 Introduction

Regular model checking [14,4,5] is a general method for formal verification of infinite-state systems. Configurations of systems are encoded as finite words over a finite alphabet Σ and transitions are encoded as relations over words. Then, word automata over Σ can naturally be used to represent and manipulate (infinite) sets of configurations and transducers over $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ are used to represent the transition relation. To verify safety properties, a reachability analysis is performed by calculating transitive closures of transducers or images of automata by iteration of transducers. Termination is usually not guaranteed and therefore various acceleration methods have been proposed.

[1] Supported by the French ministry of research (ACI project Securité Informatique).

[2] Email: abou@liafa.jussieu.fr, Peter.Habermehl@liafa.jussieu.fr

[3] Supported by the Czech Grant Agency projects 102/05/H050, 102/03/D211, and 102/04/0780.

[4] Email: rogalew@fit.vutbr.cz, vojnar@fit.vutbr.cz

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

As one of the most successful acceleration methods and also as a way to cope with the problem of state space explosion in automata representing configurations, *abstract regular model checking* (ARMC) [8] has been introduced recently. This generic method uses the well known *abstract-check-refine* paradigm within regular model checking. Abstractions are defined on word automata representing configurations. Then, an abstract reachability analysis which is guaranteed to terminate is performed. Suitable refinements of abstractions are defined for the case a spurious counter-example is encountered. In this way, an abstraction detailed just enough to answer a particular verification question is computed. ARMC has been successfully applied to a lot of different systems, like counter automata, parameterised networks of processes, and programs with lists [7].

To handle other structures than linear (or easily linearisable) ones, regular *tree* model checking [14,6,1,19,2] has been proposed. Instead of words, configurations are finite trees and instead of word automata, tree automata are used to represent sets of configurations. Then, tree transducers model transitions. Like in the word case, several acceleration approaches for reachability analysis exist.

Tree like structures are very common and appear naturally in many modelling and verification contexts. For example, in the case of parameterized tree networks, labelled trees of arbitrary height represent a configuration of the network: each process is a node of the tree and the label its control state. Trees also arise naturally, e.g., as a representation of configurations of multithreaded recursive programs [12,17], as a representation structure of heaps [15], or when representing structured data such as XML documents [9].

In this paper, we extend the framework of ARMC from words to trees. We use bottom-up tree automata and transducers. Like in ARMC, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets. To achieve this, we define techniques that systematically map any tree automaton M to a tree automaton M' from some finite domain such that M' recognises a superset of the language of M. For the case that the computed overapproximation is too coarse and a spurious counter-example is detected, we give effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counter-example.

We, in particular, propose two abstractions for tree automata. Similarly to ARMC, both of them are based on collapsing automata states according to a suitable equivalence relation. The first is based on considering two tree automata states equivalent if their *languages of trees up to a certain fixed height* are equal. The second abstraction is defined by a set of regular *predicate languages* L_P. We consider a state q of a tree automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the tree language $L(M, q)$ accepted from the state q is not empty. Then, two states are equivalent if they satisfy the same predicates.

We have implemented the above abstractions in a prototype tool using the Timbuk [13] tree automata library. We have experimented with the tool on various parameterized tree network protocols. The results are very encouraging and com-

pare very well with other tools, which gives us a very good basis and motivation for a further development of the method.

2 Regular Tree Languages and Transducers

This section is a brief introduction to regular tree languages and transducers. A more detailed description can be found, e.g., in [10,11].

An *alphabet* Σ is a finite set of symbols. Σ is called *ranked* if there exists a *rank* function $\rho : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank k. Symbols of Σ_0 are called *constants*. Let χ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over Σ and χ. The set $T_\Sigma[\emptyset]$ is denoted by T_Σ, and its elements are called *ground terms*. A term t from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in t. Terms in $T_\Sigma[\chi]$ can be viewed as trees—leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k.

A *bottom-up tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, \Sigma, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, and δ is a set of transitions of the following types: (i) $f(q_1, \ldots, q_n) \to_\delta q$, (ii) $a \to_\delta q$, and (iii) $q \to_\delta q'$ where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: Below, we call a bottom-up tree automaton simply a tree automaton.

Let t be a ground term. A run of a tree automaton A on t is defined as follows. First, leaves are labelled with states. If a leave is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leave is labelled by q. An internal node $f \in \Sigma_k$ is labelled by q if there exists a rule $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label q_1, the second one q_2, ..., and the last one q_k. Rules of the type $q \to_\delta q'$ are called *ε-steps* and allow us to change a state label from q to q'. If the top symbol is labelled with a state from the set of final states F, the term t is accepted by the automaton A.

A set of ground terms accepted by a tree automaton A is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state q*—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is an input ranked alphabet, Σ' is an output ranked alphabet, and δ is a set of transition rules of the following types: (i) $f(q_1(x_1), \ldots, q_n(x_n)) \to_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, (ii) $q(x) \to_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$, and (iii) $a \to_\delta q(u)$, $u \in T_{\Sigma'}$ where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer τ on a ground term t is similar to a run of a tree automaton on this term. First, rules of type *(iii)* are used. If a leaf is labelled by a symbol a and there is a rule $a \to_\delta q(u) \in \delta$, the leaf is replaced by the term

u and labelled by the state q. If a node is labelled by a symbol f, there is a rule $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow_\delta q(u) \in \delta$, the first subtree of the node has the state label q_1, the second one q_2, ..., and the last one q_n, then the symbol f and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \ldots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label q is assigned to the new tree. Rules of type *(ii)* are called *ε-steps*. They allow us to replace a q-state-labelled tree by the right hand side of the rule and assign the state label q' to this new tree with the variable x in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from F.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. By abuse of notation, we identify a transducer τ with the relation $\{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t') \text{ for some } q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $R \subseteq T_\Sigma \times T_\Sigma$, we denote $R(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in R\}$ and $R^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in R\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\tau(L)$ and $\tau^{-1}(L)$ are regular and effectively constructible [11,10].

Let $id \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

3 Abstract Regular Tree Model Checking

In this section, we first recall the notion of regular tree model checking. Then, we introduce abstract regular tree model checking by defining several abstractions on tree automata.

3.1 Regular Tree Model Checking

Regular tree model checking [1,6,14] is a generalisation of regular model checking [5] to trees. A configuration of a system is encoded as a term (tree) over a ranked alphabet and a set of such terms as a regular tree automaton. The transition relation of a system is encoded as a linear tree transducer τ. We are given a tree automaton $Init$ encoding the set of initial states. For safety properties, a set of bad states (represented by a tree automaton Bad) is given. Then, the basic verification problem consists in deciding whether

$$\tau^*(L(Init)) \cap L(Bad) = \emptyset \tag{1}$$

This problem is in general undecidable (an iterative computation of $\tau^*(L(Init))$ does not terminate). Several methods [1,2,6] have been proposed to calculate in some cases τ^* or $\tau^*(L(Init))$. These techniques all compute exact sets or relations. We tackle the model-checking problem by generalising the abstract regular model

checking method [8] to tree automata. This method computes an overapproximation of $\tau^*(L(Init))$ with a precision just sufficient to safely solve the verification problem (1).

3.2 Abstract Regular Tree Model Checking

Abstract regular tree model checking (ARTMC) combines regular tree model checking with automatic abstraction. The main idea of ARTMC is a generalisation of abstract regular model checking [8] to regular tree languages. For this, the abstraction techniques designed for word automata have to be adapted to tree automata.

We start by recalling the basic framework of abstract regular model checking (here phrased directly for trees).

Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer τ and abstraction α, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is a minimal automaton describing the language $\tau(L(M))$. An abstraction α is *finite range* if the set $\mathbb{A}_\Sigma$ is finite.

Let $Init$ be a tree automaton representing the set of initial configurations and Bad be a tree automaton representing the set of bad configurations. Now, we may iteratively compute the sequence $(\tau_\alpha^i(Init))_{i\geq 0}$. Since we suppose $id \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of α implies $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the verification problem (1) has a positive answer. Otherwise, the answer to the problem (1) is not necessarily negative since during the computation of $\tau_\alpha^*(L(Init))$, the abstraction α may introduce extra behaviours leading to $L(Bad)$. Let us examine this case. Assume that $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{2}$$

such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$. We analyse this path by computing the sets $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$, and for every $k \geq 0$, $X_k = L(\tau_\alpha^k(Init)) \cap \tau^{-1}(X_{k+1})$. Two cases may occur: (i) either $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the problem (1) has a *negative answer*, or (ii) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the symbolic path (2) is actually a *spurious counter-example* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, our approach is based on the definition of abstraction schemas allowing to compute families of (automatically) refinable abstractions.

3.3 Abstraction Based on Automata State Equivalence

Below, we discuss two possible tree automata abstraction schemas which are based on tree automata state equivalence. First, tree automata states are split into sev-

eral equivalence classes by an equivalence relation. Then, the abstraction function collapses states from each equivalence class into one state. Formally, a tree automata state equivalence schema $\mathbb{E}$ is defined as follows: To each tree automaton $M = (Q, \Sigma, F, \delta) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_{\mathbb{E}}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ finitary if $\alpha_{\mathbb{E}}$ is finitary (i.e. there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^{\mathbb{E}}$ finer.

3.4 Abstraction Based on Languages of Finite Height

We now present the possibility of defining automata state equivalence schemas based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalisation of a similar schema proposed for word automata in [8]. This schema defines two states of a tree automaton M as equivalent if their languages up to the given height n are identical.

Formally, for a tree automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{H}_n$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

There is a finite number of languages of trees with a maximal height n, and so this abstraction is finite range. Refining of the abstraction can be done by increasing the value of n.

The abstraction schema $\mathbb{H}_n$ can be implemented in a similar way as minimisation of tree automata. Just the main loop of the minimisation procedure is stopped after n iterations.

3.5 Abstraction Based on Predicate Languages

We next introduce a predicate-based abstraction schema $\mathbb{P}_{\mathcal{P}}$, which was inspired by the predicate based abstraction on words [8].

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton, then two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

Formally, for an automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{P}_{\mathcal{P}}$ defines the state equivalence as the equivalence $\sim_M^{\mathcal{P}}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, since $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{P}_{\mathcal{P}}$ is *finitary*. This schema can be refined by adding new predicates into the set $\mathcal{P}$. The following theorem shows that we may eliminate a spurious counter-example by extending the predicate set $\mathcal{P}$ by the languages of all states of the tree automaton representing X_{k+1} in the analysis of the spurious counter-example (recall that $X_k = \emptyset$) as presented in Section 3.2.

Theorem 3.1 *Let us have any two tree automata $M = (Q_M, \Sigma, F_M, \delta_M)$ and $X = (Q_X, \Sigma, F_X, \delta_X)$ and a finite set of predicate automata $\mathcal{P}$ s.t. $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof. The proof is a generalisation of the proof [8] for word automata. We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $t \in L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$. As t is accepted by $\alpha_{\mathbb{P}_\mathcal{P}}(M)$, M must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim_M^\mathcal{P}$—after accepting a subtree of t and getting to some $q \in Q_M$, M is allowed to jump to any $q' \in Q_M$ such that $q \sim_M^\mathcal{P} q'$ and go on accepting from there (with or without further jumps).

Let $i > 0$ be the minimum number of jumps needed for accepting a tree from $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ in M and let t' be such a tree. When looking at the acceptance of t' in M (with some jumps allowed), we can identify maximum subtrees of t' that may be accepted without jumps—in the worst case, they are just the leaves. Let us take any of such subtrees. Such a subtree t_1 is accepted in some q_1, from which M jumps to some q_2 and goes on accepting the rest of the input. Suppose that t_1 is accepted in some $q_X \in Q_X$ in X. As $t_1 \in L(M, q_1)$, $L(M, q_1) \cap L(P) \neq \emptyset$ for the predicate $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^\mathcal{P} q_2$, $L(M, q_2) \cap L(P) \neq \emptyset$ too. This implies there exists $t_2 \in L(P)$ such that $t_2 \in L(M, q_2)$ and $t_2 \in L(X, q_X)$. However, this means that the tree t'' that we obtain from t' by replacing its subtree t_1 with t_2 and that clearly belongs to $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in M with $i - 1$ jumps, which is a contradiction to the assumption of i being the minimum number of jumps needed. □

The abstraction of an automaton M wrt. the state equivalence based on predicate languages $\mathbb{P}_\mathcal{P}$ can be implemented as labelling each state of M by the predicates with which its language has a non-empty intersection, and then collapsing states with an equal labelling. Here, let us stress that when refining $\mathbb{P}_\mathcal{P}$, it is not necessary to store each of the newly introduced predicates corresponding to the states of X_{k+1} independently and then perform the labelling independently for each of them. We may keep just X_{k+1} and then perform labelling not by just X_{k+1} but by each of its states. Moreover, this labelling may be implemented by one simultaneous run through M and X_{k+1}, which corresponds to an efficient simultaneous labelling by all the predicates contained in X_{k+1}.

4 Experiments with ARTMC

In order to be able to practically evaluate the proposed methods of ARTMC, we have implemented them in a prototype tool. We have based our prototype tool on the *Timbuk* library [13] written in Ocaml. Timbuk provided us with the basic operations over tree automata needed in ARTMC (such as union, intersection, complementation, etc.). However, we had to extend Timbuk with a support for tree transducers. We added two implementations of tree transducers—a simpler and more efficient for structure-preserving transducers and a more complex for general transducers. The latter implementation exploits a decomposition of a tree trans-

ducer into three less complicated ones as described in [11]. This decomposition can be performed automatically for any tree transducer.

We have tested our verification methods on several examples of protocols using a parameterised tree-shaped network cited in the literature [14,3,1,2] where the necessity to cover all possible values of the parameters leads to dealing with infinite state spaces:

- *Simple Token Protocol*. A token is being passed in a tree-shaped network from a leave to the root. We check that the token does not disappear nor replicate.
- *Two-Way Token Protocol*. An analogy to the previous example, but we allow the token to be passed upwards as well as downwards.
- *Percolate Protocol*. A tree-shaped network of processors computes the logical disjunction of the boolean values that appear in the leave nodes. We check that the computed value is always correct.
- *Tree Arbiter Protocol*. A tree-shaped network is used to implement mutual exclusion among the leave processors. A request to enter the critical section is propagated upwards till a node is found which has a token allowing one to enter the critical section or which knows where the token is (because it granted the token to one of its children). A node with the token can always send the token upwards or grant it to any of its children. We check the mutual exclusion property.
- *Leader Election Protocol*. One of a set of processors is to be elected a leader and a tree-shaped network is used for this purpose. The leaves are divided into candidates and non-candidates. The information about the existence of candidates is propagated upwards. In the subsequent downward phase, a path leading from the root to one of the candidate nodes is non-deterministically selected and thus a leader is established. We check that exactly one leader is chosen.

All the above examples work with a tree-shaped network of a fixed structure. In order to test the ability of our method to work with non-structure-preserving systems, we have considered a *simple broadcast protocol*. In the protocol, the root sends a message to all leave nodes. They answer and the answers are combined when travelling upwards. An intermediate node may decide to resend the message downwards and wait for new data. New nodes may dynamically join the network at leaves and also leave the network in a suitable moment. We check that there is at most one active message on each path from the root to the leaves.

The results of our experiments are summarised in Table 1. We performed experiments with both the finite-height abstraction as well as with the predicate-based abstraction. We considered both forward as well as backward verification—i.e. starting with the set of initial states and checking that the bad states cannot be reached or vice versa. In the table, we always present the better result of these two approaches. For the finite-height abstraction, we considered the initial height one (and increased it by one if necessary—in the cases presented in Table 1, this was not necessary). For the predicate-based abstraction, we considered the automaton describing the set of bad states as the only initial predicate (or—more precisely—all the automata that can be obtained from it by considering each of its states as the only accepting one; in the cases presented in Table 1, no refinement was necessary

when using these initial predicates). We experimented with the empty initial set of predicates too—this turned out to be the fastest option for the Percolate protocol (one refinement was necessary in this case).

Table 1
Some results of experimenting with ARTMC

Protocol	$\mathbb{H}_n$	$\mathbb{P}_{\mathcal{P}}$
Token passing	backwards: 0.08s	forwards: 0.06s
Two-way token passing	backwards: 1.0s	forwards: 0.09s
Percolate	backwards: 20.8s	forwards: 2.4s
Tree arbiter	backwards: 0.31s	backwards: 0.34s
Leader election	backwards: 2.0s	forwards: 1.74s
Broadcasting	backwards: 9.1s	forwards: 1.0s

Notice that the predicate-based abstraction is almost always better than the finite-height abstraction. This is different from the word case where the results differ. An explanation of this phenomenon is a part of our future work. The verification times presented in Table 1 were obtained on an Intel Centrino 1.6GHz machine with 768MB of memory. We consider these results very encouraging and we are now working on a new version of our tool that will be based on the Mona library [16]. This gives us hope of even better results and an expectation of a successful applicability of the tool on real-life case studies (including, e.g., verification of programs with dynamic linked data structures).

5 Conclusions

We have proposed abstract regular tree model checking as a generalisation of the successful approach of abstract regular model checking. In particular, we have proposed two kinds of abstractions over tree automata based on collapsing in some sense equivalent states of these automata. One of the abstractions decides which states are equivalent by comparing their languages of trees of a bounded height while the second one compares the states wrt. whether their languages satisfy (i.e. are not disjoint with) a set of predicates having the form of regular tree languages. Both of these abstractions are automatically refinable when a spurious counterexample is found and allow one to deal with an overapproximation of the state space precise just enough to verify a given property of interest. In this way, the state explosion in automata representing the reachability set is fought. The above abstractions were inspired by some of the schemas used in the original ARMC.

We have implemented the proposed methods in a prototype tool and evaluated them on multiple verification examples with very encouraging results. Currently, we are building a new and much more elaborate version of our tool based on the tree libraries of Mona [16]. This tool promises even better results and a high potential for a successful application on real-life verification problems.

Apart from finishing the new version of our tool, our future work includes a research on the various application domains of ARTMC. They include, e.g. ver-

ification of programs with dynamic linked data structures. ARMC has already been shown useful for verification of programs with 1-selector linked dynamic data structures [7]. The use of ARTMC could allow us to handle much more general structures. To encode data structures with a graph shape, we plan to use trees with some special symbols placed in their nodes to describe additional edges over the tree. Another promising application area is the domain of XML manipulations. Indeed, XML documents have a tree structure and most of XML parsers are based on the tree automata theory—in particular, on hedge automata [9]. Furthermore, we intend to use our approach for programs with abstract data structures and cryptographic protocols along the lines of [18]. For all these applications we plan to study the encoding in tree automata and transducers and the possibility of defining application dependent abstractions.

References

[1] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, *LNCS* 2404. Springer, 2002.

[2] P.A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. *Simulation-Based Iteration of Tree Transducers.* In *Proc. of TACAS'05*, *LNCS* 3440. Springer, 2005.

[3] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, S. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proc. of CAV'97*, *LNCS* 1254. Springer, 1997.

[4] B. Boigelot and P. Wolper. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, *LNCS* 1427. Springer, 1998.

[5] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, *LNCS* 1855. Springer, 2000.

[6] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, *LNCS* 2404. Springer, 2002.

[7] A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar. *Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking*. In *TACAS'05*, *LNCS* 3440. Springer, 2005.

[8] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking, In *Proc. CAV'04*, *LNCS* 3114. Springer, 2004.

[9] A. Bruggemann-Klein, M. Murata, and D. Wood. *Regular tree and regular hedge languages over unranked alphabets: Version 1*. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[10] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005. `http://www.grappa.univ-lille3.fr/tata`

[11] J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison, *Mathematical System Theory*, 9:198–231, 1975.

[12] J. Esparza. Grammars as Processes. In *Formal and Natural Computing*, *LNCS* 2300, 2002.

[13] T. Genet. Timbuk, a tree automata library, 2005. `http://www.irisa.fr/lande/genet/timbuk`

[14] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. CAV'97*, *LNCS* 1254. Springer, 1997.

[15] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*, ACM, 1993.

[16] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.

[17] M. Křetínský, V. Řehák, and J. Strejček. Extended Process Rewrite Systems: Expressiveness and Reachability. In *Proc. of Concur'04*, *LNCS* 3170. Springer, 2004.

[18] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Science of Computer Programming*, Volume 47, Issue 2-3 (May 2003).

[19] A. Pnueli and E. Shahar. *Acceleration in Verification of Parameterized Tree Networks*. Technical report MCS02-12, Weizmann Institute of Science, Israel, 2004.

Automatic Verification of Fault-Tolerant Register Emulations

Paul C. Attie [1]

College of Computer Science, Northeastern University,
360 Huntington Avenue, Boston, Massachusetts 02115.
and
MIT CSAIL, 32 Vassar street,
Cambridge, MA, 02139, USA.

Hana Chockler [2]

MIT CSAIL, 32 Vassar street,
Cambridge, MA, 02139, USA.
and
Department of Computer Science, WPI,
100 Institute Road, Worcester, MA 01609, USA.

Abstract

The design and verification of fault-tolerant distributed algorithms is a complicated task. Usually, the proof of correctness is done manually, and thus depends on the skill of the prover. Using automated verification methods, such as model checking, can greatly simplify the verification. However, model checking of distributed algorithms is often intractable because of the state-explosion problem. In this paper we present a novel approach to verification of quorum-based distributed register emulation algorithms with undetectable crash failures of processes. Our approach is based on projection and abstraction and allows us to reduce the task of model-checking the whole system to fair model-checking of subsystems consisting of a constant number of processes. Our method is highly scalable and can be applied to a large class of algorithms. Aside from efficient verification, it can also be used for finding redundancies in existing algorithms.

Key words: Distributed algorithms, parametrized systems, fault-tolerance, crash failures, register emulation, automatic verification, model checking

[1] Email: attie@ccs.neu.edu
[2] Email: hanac@theory.lcs.mit.edu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Formal verification is widely recognized as a key means for assuring the correct behavior of large and complex software systems. Such systems are usually distributed, and contain a large number of components, or processes, which are subject to a variety of failures. Thus, distributed algorithms for such system must be fault-tolerant. The design of such algorithms is a complex task, and manual proofs of correctness of such algorithms are usually very complicated. Automatic verification of such systems is limited by the *state-explosion* problem, which becomes acute even for medium-size systems. Resent research in the area of *parametrized systems*, that is, systems that consist of potentially numerous instantiations of (usually) small and simple modules, uses techniques such as abstraction and deductive verification (see [9] for a survey). None of these techniques deal with fault-tolerance.

In this paper, we consider asynchronous distributed systems with undetectable crash failures of processes. We assume that there is a large number of processes, which can be placed into a small number of "equivalence classes," e.g., readers, writers, and servers (that store the value of a shared data object), so these systems can be regarded as parametrized systems. In a system with undetectable crash failures, a failed process stops communicating with the other processes, but the failure cannot be detected by the others. That is, a failed process is not distinguishable from one that is "very slow" [5]. We do not consider Byzantine failures, where a failed process can behave arbitrarily, and in particular, can deviate from its algorithm.

The distributed algorithms that we consider emulate shared registers and are based on quorum systems. In quorum systems, every broadcast operation awaits acknowledgments from only a quorum of processes, rather than from all processes. There exist different types of quorum systems. The common feature of all quorum systems is that an intersection of every two quorums is nonempty. Taking this feature as an axiom, we abstract away a particular quorum system. We propose a method for modeling and verification of quorum-based concurrent algorithms with crash failures by means of fully automatic model-checking procedures. Informally, our method is as follows. We express correctness (safety and liveness) properties of systems by quantified LTL formulae. We then show that the quantification over processes can be replaced by quantification over subsystems, each of which contains a constant number of processes and executes a constant number of requests, while maintaining semantic equivalence of the formulae. Model checking these small subsystems then suffices to conclude the correctness of the whole system. In general systems, we have to verify all possible subsystems, and since the number of subsystems is exponential in the number of processes, the verification cannot be done automatically for an arbitrary number of processes. The key idea, which allows us to check only a small number of subsystems, is that the systems we consider consist of a small number of "equivalence classes" of similar processes. Two processes are placed in the same equivalence class if they

run the same algorithm and are in the same state at the present time in the execution. Since server processes store the value of the register, the current value stored in a server is also taken into account when checking equivalence (in fact, we avoid considering the data values separately by storing them in states of a server).

Algorithms that we study in this paper use unbounded time-stamps. Thus, the straightforward implementation of each process yields a structure with an infinite number of states. However, since we only deal with subsystems with a constant number of processes, and quorum-based decisions are based solely on comparison between time-stamps (and not on their absolute values), we are able to abstract the absolute values of time-stamps away and to use only their relative values. Since the number of requests in subsystems is constant, the number of time-stamps is also constant, and thus we are able to express their comparative values by a constant number of boolean variables.

When we construct subsystems, we *project* the processes onto the subsystem, abstracting away all states that are unreachable and all variables that are unaccessible in this subsystem. The main problem in projecting the processes is that in a large system, processes perform transitions depending on the quorum-based decisions. Since in a constant-size subsystem a quorum of processes is inaccessible, we replace quorum-based decisions by non-deterministic decisions. This replacement creates illegal executions which do not exist in the original system. The first type of illegal executions is executions in which the client process non-deterministically moves to a state where the output is chosen before receiving the most updated reply. We filter such executions away by always including a process in the intersection of the current quorum with all other quorums in the small subsystem that we model-check. The second type is executions in which processes are stuck in the "waiting-for-replies" state and never terminate. We filter such executions away by replacing regular model-checking with fair model-checking, where the fairness constraint effectively expresses the condition "there is always a live quorum of processes".

An additional advantage of model-checking small subsystems is didactic. That is, the necessity of a part of the algorithm can be easily verified by removing this part from the model and model-checking the altered subsystem. If the altered subsystem still satisfies the desired property (non-vacuously), it means that the removed part was not necessary for the correctness of the algorithm.

2 Preliminaries

2.1 Temporal logic and Kripke structures

In model checking, we check whether a system given as a Kripke structure (labeled state-transition graph) satisfies a specification given as a temporal logic formula (or a finite automaton). In this paper we assume that the specifications are given in the linear temporal logic LTL [8,4]. The semantics of

temporal logic is defined with respect to Kripke structures. We use Büchi fairness constraints, where an execution is *fair* if it visits a *fair* state an infinite number of times.

2.2 Registers

A *read/write* register (or simply, register) type supports an arbitrary set $Vals$ of values with an arbitrary initial state $v_o \in Vals$. Its invocations are $read$ and $write(v)$, for some $v \in Vals$. Its responses are v and ack. Its sequential specification f requires that every $write$ operation overwrites the last value written and returns ack (that is, $f(write(v), w) = (ack, v)$); and every $read$ operation returns the last value written (that is, $f(read, v) = (v, v)$). In a system consisting of processes $P_1, P_2, \ldots$, a process P_i interacts with a shared register by means of input actions of the form $read_i$ and $write_i(v)$ and output actions of the form v_i and ack. A read/write register is called k-reader/m-writer if only k processes are allowed to read, and m processes are allowed to write the register. We use the term multi-reader (multi-writer) when the number of readers (writers) is unrestricted. We now define several register properties. In our definitions, we talk about $read$ and $write$ operations. As opposed to I/O actions (which are atomic by definition), operations start when the request is placed in the system and terminate when the result is returned: the result of a $read$ operation is a value and the result of a $write$ operation is an indication that a $write$ has terminated successfully.

When expressing the register properties in quantified LTL, we denote by r_i the i-th read operation, by w_i the i-th write operation, and by s_i the i-th server. In order to express the order between processes, we introduce new boolean variables for beginning and end of each operation: $op_i.b$ ($op_i.e$) changes from **false** to **true** when the first (last) action of op_i is executed. We use Lamport's notation of arrows to express order between processes [7], where $op_1 \rightarrow op_2$ (op_1 strictly precedes op_2) is a shortcut for $\mathsf{G}(op_2.b \Rightarrow op_1.e)$, and $op_1 \leftrightarrow op_2$ (op_1 is concurrent with op_2) is a shortcut for $\neg(op_1 \rightarrow op_2) \wedge \neg(op_2 \rightarrow op_1)$, where $\Rightarrow$ denotes boolean implication. We also use variables $w_i.val$ for a value written by w_i, $r_i.val$ for a value read by r_i, and $s_i.val$ for a value that s_i holds. The values are pairs of the data value and a time-stamp, that is, $op_i.val = \langle v, ts \rangle$. When comparing between values, we say that $op_i.val = op'_j.val$ iff $op_i.val.v = op'_j.val.v$ and $op_i.val.ts = op'_j.val.ts$. Let x be a register, and let σ be a sequence of invocations and responses of x. The following definitions of registers in a single-writer multi-reader system are taken from [7].

- **Safe register:** σ is *safe* [7] if every complete $read$ operation that does not overlap any $write$ operation returns the latest written value or the initial value if no value has been written yet. A register x is *safe* iff all its traces σ are safe. Formally,

$$\forall w_i, r : \mathsf{G}[\ (w_i \rightarrow r) \wedge (\neg\exists w_j : w_i \rightarrow w_j \rightarrow r) \wedge (\neg\exists w_k : w_k \leftrightarrow r) \Rightarrow$$

$$\mathsf{F}(r.val = w_i.val)\ ]. \tag{1}$$

We note that we only express the condition of correct value returned when there are previous writes. When there are no previous writes, the requirement is expressed trivially.

- **Regular register:** σ is *regular* [7] if it is safe and in addition every *read* operation that overlaps some *write* operations returns either one of the values written by overlapping *writes* or the latest non-overlapping value. A register x is *regular* iff all its traces σ are regular. Formally, this additional requirement is expressed as

$$\forall w_i, r : \mathsf{G}[\ (w_i \rightarrow r) \wedge (\neg\exists w_j : w_i \rightarrow w_j \rightarrow r) \Rightarrow$$

$$\mathsf{F}(r.val = w_i.val)\ \vee\ \exists w_k : (w_k \leftrightarrow r \wedge \mathsf{F}(r.val = w_k.val))\ ]. \tag{2}$$

- **Atomic register:** σ is *atomic* [7] if it is regular and in addition all invocations of σ are *linearizable* (see [6] for a definition) to a sequential register (that is, a register in which there are no overlapping operations). A register x is *atomic* iff all its traces σ are atomic. Formally, a register is atomic if it satisfies Equation 2 and in addition

$$\forall w_1, w_2, r_1, r_2 : \mathsf{G}[\ ((r_1 \rightarrow r_2) \wedge \mathsf{F}(w_2.val = r_1.val) \wedge \mathsf{F}(w_1.val = r_2.val)) \Rightarrow$$
$$\neg\mathsf{F}(w_1 \rightarrow w_2)\ ]. \tag{3}$$

For the purposes of this work, we consider an operation to be *live* if the terminating state is eventually reached (that is, $\mathsf{F}op.e$ holds).

2.3 Characterization of algorithms

We start by characterizing the class of algorithms $\mathcal{R}$ to which our method applies. The characterization of $\mathcal{R}$ is as follows.

(i) Algorithms in $\mathcal{R}$ emulate a shared register by numerous instances of *server* processes.

(ii) There is a single writer process [3].

(iii) Servers can crash and stop responding to requests. However, there is always a live *quorum* of servers.

(iv) Clients (writers and readers) are non-faulty.

(v) Each server can hold exactly one value at each point.

(vi) All values are accompanied by (unique) time-stamps.

(vii) Upon receiving different replies from servers, the output answer is chosen based on the time-stamps of the replies.

[3] Our work can be extended to the multi-writer case. For multiple writer processes, additional work should be performed in order to prove uniqueness of the time-stamp. Here we assume a single writer, because we rely on the time-stamp as the unique identification of register values.

We do not restrict the number of rounds each client performs in order to read/write a value. We do assume, however, that this number is constant.

Each invocation of the algorithm (whether a read or a write) can be verified in isolation from the previous invocations. This observation allows us to model invocations by *micro-processes* [2]. A micro-process implements a single operation (e.g., a single read or write), after which it is destroyed. It has a very few states, and thus systems consisting of micro-processes can be easily model checked. The only problem with micro-processes is that a writer should store the current time-stamp. We circumvent it by assuming that the time-stamp is given to the writer micro-process as an input together with the input value. In single-writer algorithms, we can safely assume that this input time-stamp is unique and increases with each subsequent request. In multi-writer algorithms, more subtle reasoning is needed.

3 Automatic Verification of Quorum-Based Register Emulations

In this section, we present our main result, namely, that the desired safety and liveness properties can be automatically verified in small subsystems and the correctness of the whole system deduced from the correctness of these subsystems. This result follows from Lemmas 3.1 and 3.2.

Lemma 3.1 *Safety properties of algorithms from $\mathcal{R}$ can be verified by model checking of a finite number of subsystems which contain a constant number of readers and writers.*

Essentially, we prove that the temporal formulas that express the correctness properties of the register can be rewritten in the prenex way with universal quantifiers only. The pure LTL part of the formula involves at most 4 clients. The resulting pure LTL formulas are

$$\varphi_{safe} = \mathsf{G}[\ (w_i \rightarrow r) \wedge \neg(w_i \rightarrow w_j \rightarrow r) \wedge \neg(w_k \leftrightarrow r) \Rightarrow \mathsf{F}(r.val = w_i.val)\ ] \quad (4)$$

for safe register,

$$\varphi_{regular} = \mathsf{G}[\ (w_i \rightarrow r) \wedge \neg(w_i \rightarrow w_j \rightarrow r) \Rightarrow$$
$$\mathsf{F}(r.val = w_i.val) \vee (w_k \leftrightarrow r \wedge \mathsf{F}(r.val = w_k.val))\ ] \quad (5)$$

for regular register, and

$$\varphi_{at} = \mathsf{G}[\ ((r_1 \rightarrow r_2) \wedge \mathsf{F}(w_2.val = r_1.val) \wedge \mathsf{F}(w_1.val = r_2.val)) \Rightarrow \neg\mathsf{F}(w_1 \rightarrow w_2)\ ] \quad (6)$$

for atomic register. Since we model clients by micro-processes, each of which executes exactly one request, the number of requests in a subsystem is equal to the number of clients in this subsystem. We are not done yet, since it remains to prove that we are able to consider a constant number of servers as well.

Lemma 3.2 *The subsystems in Lemma 3.1 can be constructed with a constant number of servers. The number of servers participating in a single subsystem is bounded by 2^m, where m is the number of communication rounds between a client (reader or writer) and servers in this subsystem.*

The proof is based on the observation that at any given time during the execution there is a constant number of equivalence classes of servers. Since the decision is made based on the time-stamp and not the number of replies with the same value, we need only one representative from each equivalence class to be included in a subsystem. Since each communication round divides the set of servers by 2 (into servers that responded and servers that did not respond in this round), m rounds divide the set of servers by 2^m.

We note that Lemma 3.2 assumes that the current value is stored in a state of a server. This results in an unbounded number of states in the large system. In the constant-size subsystems that we consider, however, the number of values is is constant, and thus only a finite number of states of each server is reachable. We abstract away unreachable states. The number of rounds depends on the algorithms for readers and writers. For example, a reader micro-process can perform two communication rounds with the servers, where in the first one it reads the value and in the second one it writes it to the servers. We further note that the set of servers that did not reply to any of the clients can be ignored, as it does not play any role in the correctness of the system.

Combining Lemma 3.1 and Lemma 3.2 we are able to verify safety and liveness properties of algorithms we consider by model-checking a constant number of subsystems with a constant number of (small) processes. It remains to show how quorum-based decisions are projected onto small subsystems. We do so by replacing quorum-based transitions by non-deterministic transitions. Clearly, such abstraction creates executions that are illegal in the global system. First, a process can now loop indefinitely in the state where it awaits replies from a quorum, thus creating a non-terminating execution. Second, it is now allowed to move to the state where it chooses the output value regardless of the number of replies it received.

We deal with the first problem by replacing the standard model checking procedure with fair model checking, where the fairness constraint allows only executions in which the state where a process awaits replies from a quorum occurs only a finite number of times. This is equivalent to the assumption that there is always a live quorum of servers. We deal with the second problem by a careful construction of the product Kripke structure. In our product Kripke structure, we want to eliminate all executions which are not consistent with receiving a quorum of replies. We rely on the following feature of quorums: every two quorums intersect. Thus, we construct the small subsystems so that for every pair of communication rounds in the subsystem, we include a single server process that lies in the intersection of the quorums of these two rounds. This ensures that the correct safety properties for this pair of rounds

are verified, since they are enforced by this server.

Time-stamps Algorithms for register emulation usually use unbounded time-stamps. Fortunately, since our subsystems consists of a constant number of processes and a constant number of communication rounds is used, we are able to abstract away unbounded time-stamps. For correctness properties of safe, regular, and atomic registers in the single-writer systems it suffices to use 2 boolean variables for encoding the time-stamps. Indeed, the number of values written in the small subsystems does not exceed 4 for any of these properties.

Fine tuning Our previous reasoning allows us to reduce verification of a parametrized system to model-checking subsystems with a constant number of processes. We showed that it suffices to consider a subsystem with m communication rounds and 2^m servers. In fact, we can reduce the number of servers in the subsystem to $m(m-1)/2$ (the number of pairs of rounds). Indeed, the only servers that are essential for the proof of correctness of the algorithms in $\mathcal{R}$ are the ones in the intersections of quorums of a read and a write communication round.

4 Examples

In this section we illustrate our method on examples of algorithms that emulate a shared read/write register. We start with a simple example of a safe register in a single-writer multi-reader system. Then we extend it to the algorithm presented by Attiya et al. in [3].

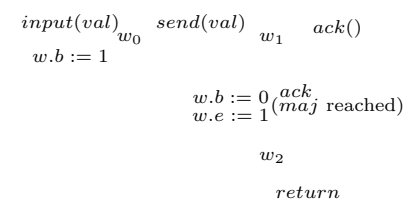
The following codes of the client and server processes implement the safe register under the single writer assumption in a system in which there always exists a majority of live processes [7].

writer($\langle$**M**$\rangle$):
increase $TimeStamp$;
$count := 0$;
send $\langle M, TimeStamp\rangle$ to all;
upon receiving a reply ack do
$count := count + 1$;
until $count > n/2$,
where n is the number of processes;
return;

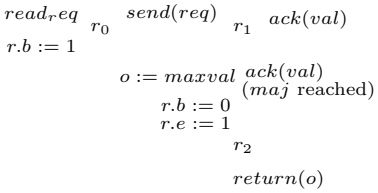
server($\langle$**M**$\rangle$):
upon receiving $\langle M\rangle$ do:
case $\langle M\rangle$:
$\langle M\rangle = ReadRequest$:
send $\langle M, TimeStamp\rangle$;
$\langle M\rangle = \langle m, TimeStamp\rangle$:
if $TimeStamp > LocalTimeStamp$,
then update M;
send ack;

reader():
$count := 0$;
send $\langle ReadRequest\rangle$ to all;
upon receiving a reply
$\langle M, TimeStamp\rangle$ do
$count := count + 1$;
save $\langle M, TimeStamp\rangle$;
until $count > n/2$,
where n is the number of processes;
choose the message M
with the maximal
time-stamp $MaxTimeStamp$;
return $\langle M, MaxTimeStamp\rangle$;

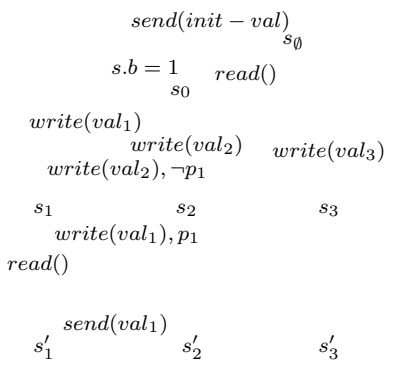
Recall that safe registers satisfy Equation 4, which is checked in subsystems of three writer processes and one reader process. Since each writer process invokes one write request and the reader process invokes one read request, there are 3 pairwise intersections of quorums of a write and a read requests, and thus 3 server processes in the subsystem. Moreover, there are three values written in this subsystem. Thus, the order between the three time-stamps can be expressed by three boolean predicates: $p_1 \equiv (w_i.val.ts > w_j.val.ts)$ and $p_2 \equiv (w_i.val.ts > w_k.val.ts)$, and $p_3 \equiv (w_j.val.ts > w_k.val.ts)$. Since the system has one (global) writer process, in the product subsystem the three write requests should strictly precede each other. Also, we do not have to consider the order in which $w_i \rightarrow w_j \rightarrow r$. The projection of a single write request (that is, a micro-process) and the projection of a single read request onto a subsystem with a constant number of processes are presented in the figures below. We assume that the correct time-stamp is given to the writer micro-process together with the input value (that is, val is a tuple $\langle v, ts \rangle$). The projection is obtained by abstracting away all variables that are unaccessible in the subsystem and computing the quotient abstraction of the resulting structure. The transitions $w_1 \rightarrow w_2$ and $r_1 \rightarrow r_2$ are non-deterministic. Recall, that we ensure liveness of the subsystem by replacing regular model-checking with fair model-checking and safety by including the server that lies in the intersection of quorums in the subsystem. The server process stores the current value and its time-stamp in its state. Since in our subsystem there are three write requests, the projection of a server process onto a subsystem has a finite number of states. The projection of a server process on the subsystem with three write operations is presented in Figure 1.



Projection of $write(val)$ on the subsystem



Projection of $read()$ on the subsystem
($maxval$ is the value with the maximal time-stamp)



Projection of a server on the subsystem

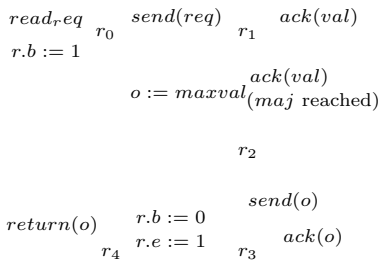
Fig. 1. Projection of processes in the safe register emulation

To avoid cluttering the figure, we omitted transitions similar to the ones between s_1 and s_2 and between s_1 and s'_1. All projections end in the $idle$ state

(omitted from the figure), in which they loop forever. The resulting subsystem is a product of 7 processes and has 3 write requests. The largest process is, thus, a server, whose size depends on the number of write requests. In this case, it has 8 states (2 states for each value, an initial state, and a state in which the initial value is returned). Thus, the straightforward implementation results in a subsystem that can be encoded with 21 variables. Filtering away interleavings that result in a vacuous satisfaction of the property and noting that replies from servers are assumed to be received in a known order results in a subsystem, which is the cross-product of a sequential composition of 3 clients and 2 servers (which has in total 25 states), one client (the writer w_k) and one server (which lies in the intersection of the quorums of w_k and r). Together with time-stamps, this subsystem can be encoded using 12 boolean variables, which is well within the reach of modern model-checkers.

The algorithm of Attiya et al. [3] for emulating a regular register in single-writer multi-reader asynchronous message-passing systems with crash failures differs from the example we studied above in the code of the reader process. In this algorithm, the reader process performs two rounds: one read request and then one write request with the value received from the read request. The projection of the reader micro-process onto a subsystem is presented in the figure below. This difference leads to the atomicity of the register. Recall, that atomicity is expressed by Equations 5 and 6.

Equation 5 is checked in a subsystem of three writer micro-processes and one reader micro-process. Note that the reader issues one read request and one write request, therefore the subsystem contains 4 server processes. The largest process in the subsystem is a server, which has 10 states, and thus the resulting subsystem can be encoded using at most 30 boolean variables.



Projection of $read()$ on the subsystem ($maxval$ is the value with the maximal time-stamp)

Equation 6 is checked in a subsystem of two writer and two reader micro-processes. The subsystem contains 8 servers, thus the resulting subsystem can be encoded using at most 44 boolean variables. For both properties, applying the reasoning above further reduces the size of the subsystem.

Remark 4.1 [The neccessity of the second round of read] By abstracting away the second round of read and model-checking the resulting subsystem, we can show that in order to ensure the property of always returning the last preceding write value or a concurrent write value we only need one round of read. On the other hand, two rounds of read are essential for proving atomicity. Indeed, we can construct a subsystem with two writes combined sequentially and a read concurrent with the last write, in which Equation 5 holds, with automatically generated interesting witnesses to exhibit both types

of satisfaction: there is an execution in which the read returns the value written by the first write, and another execution in which the read returns the value written by the second write. Since this is true for both reads regardless of their order, there exists an execution in which the first read returns the value of the second write and the second read returns the value of the first write.

5 Conclusions and Future Work

We proposed a method for modeling and verification of distributed algorithms for register emulation that allow crash failures of less than a quorum of servers. We argued that correctness (safety and liveness) properties of the whole system can be automatically model-checked in subsystems of constant size and then extrapolated to the whole system. We avoided examining quorums of processes by replacing quorum-based transitions with non-deterministic transitions, and we showed how to filter away illegal executions that are created by using non-determinism. Modeling and automatically verifying distributed algorithms by means of small abstracted systems may help to determine what parts of the algorithm are really essential for its correctness by abstracting away a part in question and model-checking the resulting system. While the examples we considered in this paper are fairly simple, we believe that applying these methods to more complex algorithms may lead to interesting insights and even improvements of existing algorithms.

It remains to prove formally that correctness of small subsystems that we constructed implies the correctness of the whole system. The formal framework for these proofs is provided by the pairwise representation of concurrent systems [1]. The method of [1] can be generalized from analyzing products of pairs of processes to analyzing products of small numbers of processes.

References

[1] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, 1998.

[2] P.C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, Northeastern University, 2003.

[3] H. Attiya, Bar-Noy A., and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.

[4] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, B:16, pp. 997–1072. MIT press, 1990.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[6] M. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] L. Lamport. On interprocess communication – part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[8] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science*, pp. 215–273. ILSCS, 1981.

[9] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages - special issue* (to appear).

Algorithmic Algebraic Model Checking III: Approximate Methods

Venkatesh Mysore [1,2]

Computer Science Department, Courant Institute
New York University, New York, USA

Bud Mishra [3]

Departments of Computer Science and Mathematics, Courant Institute
Department of Cell Biology, School of Medicine
New York University, New York, USA

Abstract

We present computationally efficient techniques for approximate model-checking using bisimulation-partitioning, polyhedra, grids and time discretization for *semi-algebraic hybrid systems*, and demonstrate how they relate to and extend other existing techniques.

Key words: Semi-algebraic hybrid systems | Model checking.

1 Introduction

A *semi-algebraic hybrid automaton* [9,8] is a hybrid automaton, whose expressions corresponding to the initial values, state invariants, continuous flows, and the guards and resets of the discrete transitions are all semi-algebraic, i.e., Boolean combinations of polynomial equations and inequalities. They are often used to approximate more general systems, whose flow equations are not polynomial, since truncated Taylor series, polynomial splines and other symbolic integration schemes provide good semi-algebraic local approximations for flows, etc. A *location* of a semi-algebraic hybrid automaton H is a pair $\langle v, X \rangle$, where $v \in V$ is a state and $X \in \mathbb{R}^k$ is an assignment of values to the k system variables. The *transition relation* $\langle u, X \rangle \xrightarrow[T]{h} \langle v, X' \rangle$ of H connects all

[1] The work reported in this paper was supported by grants from NSF's ITR program and DARPA's BioCOMP program.
[2] Email: mysore@cs.nyu.edu
[3] Email: mishra@nyu.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

possible values of the system variables before and after *one step*; namely, it is either a discrete step $\langle u, X\rangle \xrightarrow[\mathcal{D}]{0} \langle v, X'\rangle$ for a time $h = 0$ or a continuous evolution $\langle u, X\rangle \xrightarrow[\mathcal{C}]{h} \langle v(= u), X'\rangle$ for a time period $h > 0$. (For detailed definitions and summary of results, please refer to the *Appendix* .)

Earlier, in the first paper in this series [9], we introduced this class and demonstrated the use of real algebraic methods for solving the bounded reachability problem. In the second paper [8], we examined the *single-step until* operator $p \triangleright q$ of the dense-time logic *Timed Computation Tree Logic* (TCTL), which is defined as $p \vee q$ holding all along "one step" of the hybrid system and q being true at the end of the transition. *Since quantifier elimination over semi-algebraic sets is decidable [10], $p \triangleright q$ was shown [8] to be decidable for semi-algebraic hybrid systems if p and q were also semi-algebraic.* It was further proved [8] that the "existential" segment of TCTL (including reachability) and the negation of the "universal" segment are semi-decidable over semi-algebraic hybrid automata. Further, all subscripted TCTL operators become decidable in the absence of zeno-paths.

Effectively, the symbolic algebraic model checking problem was reduced to a series of quantifier elimination problems which could be solved by a software tool such as Qepcad [5]. The only source of error (if any) arose in approximating non-polynomial systems. However, the computational complexity (double exponential) of the cylindrical algebraic decomposition severely limited the applicability of the method. In this paper, we discuss the applicability of the different approximation approaches involving space- and time- discretization to semi-algebraic hybrid automata. Approximate methods have been very successful in timed automata and linear hybrid systems, yielding efficient decidable algorithms in many cases [3,4,2,1]. However, these methods rely on computational techniques that exploit the low dimensionality and other restrictions, of the dynamics of these subclasses of hybrid systems. In other words, the techniques are seldom applicable to more complex systems. Our first goal in this paper is to show that many existing ideas can be made applicable to semi-algebraic hybrid systems, by using quantifier elimination in place of the original efficient-but-restrictive computational method. The second goal is to develop these ideas to obtain new optimizations and techniques. Further, we seek to identify well-behaved sublcasses that are more general than timed or linear automata.

By suitably relaxing accuracy requirements, we aim to model-check the vast semi-algebraic class, without being severely computationally hindered. Quantifier elimination will still remain our engine of computation, though it will be used differently; namely, it will be invoked to solve many simple problems instead of a few complex problems. In this paper, we will present the modified versions of existing techniques and understand their behavior over the semi-algebraic class. Clearly, different techniques will prove to be effective in different scenarios. In this paper, we do not delve into this aspect, but

instead focus on generalizing and optimizing existing techniques. In this sense, it is the first effort to catalogue the algorithms for approximate verification (reachability) for the vast semi-algebraic class.

2 Approximate Methods

In this section, we develop new approximation methods applicable to the semi-algebraic class, based on the existing literature for much simpler subclasses of hybrid automata. For brevity, all proofs are provided in the *Appendix*.

2.1 Bisimulation Partitioning

The bisimulation idea is to convert the given hybrid automaton into a simpler one, which only preserves the properties of interest to us (in the query). The conventional bisimulation partitioning algorithm [4] involves splitting the discrete states based on the out-going discrete transitions. The source state of a transition is split so that, each new state (its partitions) has the minimal number of out-going transitions (ideally, each new partition will have only one possible successor discrete state). The rationale is that one expects only some of the guards (of the different out-going transitions) to be satisfiable, from different parts of the continuous space representing the discrete state (its state invariant).

We first prove that these partitions are *computable* for semi-algebraic hybrid systems by expressing the task as a quantifier elimination problem.

Theorem 2.1 *The standard bisimulation partitions are computable for semi-algebraic hybrid automata.* □

Having proved that the existing idea becomes applicable via quantifier elimination, we now suggest an improvement of the technique. This approach is founded on the observation that only a portion of the destination state may ever be accessed, after a specific discrete transition. Thus, by *splitting the destination state* as well, based on what fraction of it is accessible from the source state, we can refine the partitions. This simple extension was not necessary in linear systems, as the destination state space was typically entirely reachable from the resetted region (after a discrete transition). Since semi-algebraic sets have innate complexity, it is very unlikely that continuous evolution from different resetted regions will all envelope the entire state invariant. Clearly, since we chop off the region of the state invariant that is not reachable, the state invariants represent smaller sets. Hence, the extended-bisimulation-partitioning is likely to be a sharper one than the standard approach. The second advantage to this extended algorithm is that well-behaved subclasses can be characterized (see *Convergent Deterministic Automata* below). We now enumerate the complete series of computations:

Algorithm 1 ***[Extended Partitioning For Semi-Algebraic Automata]***

(i) *Pick a state s (source) with a discrete transition to state d (destination);*

(ii) *Split s into two states s_d and $s_{\bar{d}}$ thus:* $Inv_{s_d}(X) \equiv \exists h, X' \ \langle s, X\rangle \xrightarrow[C]{h} \langle s, X'\rangle \wedge Guard_{s,d}(X')$ *and* $Inv_{s_{\bar{d}}}(X) \equiv Inv_s(X) \wedge \neg Inv_{s_d}(X)$;

(iii) *Split d is into d_s and $d_{\bar{s}}$ thus:* $Inv_{d_s}(X) \equiv \exists X'', X' \ Inv_s(X'') \wedge \langle s, X''\rangle \xrightarrow[D]{0} \langle d, X'\rangle \wedge \{\exists h \ \langle d, X'\rangle \xrightarrow[C]{h} \langle d, X\rangle\}$ *and* $Inv_{d_{\bar{s}}}(X) \equiv Inv_d(X) \wedge \neg Inv_{d_s}(X)$;

(iv) *The states $s_d, s_{\bar{d}}, d_s, d_{\bar{s}}$ replace s and d. The transition from s to d is replaced by the one from s_d to d_s. All other transitions to (or from) s (d) are each replaced by two transitions to (or from) s_d and $s_{\bar{d}}$ (d_s and $d_{\bar{s}}$);*

(v) *Repeat steps $(i) - (iv)$ until no transition from any state s to any state d can be found which splits s or d.* □

It is to be recalled that convergence of the partitioning does not imply decidability of reachability for general hybrid automata. As in the standard bisimulation case, the *over-approximated* set of points reachable from $\langle s_0, X_0\rangle$ in the original hybrid system is given by the *union of the invariants* of all the states along all the trajectories starting at the state d_0 of the partitioned system corresponding to the partition of s_0 containing X_0. Even in the non-convergent case, this procedure yields an estimate of the reachable set if we roll-out H for a reasonable number of steps. Similarly, we can check if a specific X_f is reachable from a specific X_0. We iteratively partition until the partition containing X_f is not in any trajectory starting from the partition containing X_0. We can then conclude guaranteed unreachability, or approximate reachability otherwise (counterexample-guided abstraction refinement).

Having generalized and extended an existing technique, we now characterize the broadest subclasses of hybrid systems where this new technique is well-behaved.

Convergent Deterministic Automata

In *deterministic* hybrid automata, a discrete transition is taken the moment its guard is satisfied (with no two guards ever holding simultaneously). Hence there is a unique future trajectory for every initial system state. If the *extended partitioning* procedure converges for a deterministic hybrid automaton, the original automaton will now correspond to a set of disconnected trajectories. Each of these will be a cycle of discrete states, with each state possibly preceded by a linear path of unique discrete states (all other topologies get excluded because there is no "future-branching" in deterministic automata). The extended partitioning *unlike the standard bisimulation partitioning*, produces exactly onto maps between successive states when convergent. We now show how many of their mathematical properties can be fruitfully exploited to address the reachability problem, for broad subclasses of convergent deterministic semi-algebraic hybrid automata.

In *linear convergent deterministic semi-algebraic automata*, all flows and

reset maps are linear. Thus, infinite cycles are ruled out, since there are only a finite number of exactly onto maps possible (except when the sets have infinite axes of symmetry as does a circle). Thus:

Theorem 2.2 *There are only a finite number of 1-to-1 linear maps* $f(X) = \Sigma AX + B,\ A_i, B_i \in \mathbb{R}^d$ *possible, between two* d*-dimensional sets with finite axes of symmetry.* □

Corollary 2.3 *Given a cycle* $S_1, \cdots, S_n, S_1$ *of* n d*-dimensional sets with* s_l *axes of linear symmetry and* s_r *axes of rotational symmetry each, where each set maps exactly onto its successor* ($S_{i+1} = f(S_i)$)*, the number of unique successors of any point is at most* $ns_r 2^{s_l}$*.* □

Theorem 2.4 *Reachability over a deterministic semi-algebraic hybrid system with linear resets* $Reset_{u,v}(X, X') \equiv (X' = \Sigma AX + B),\ A_i, B_i \in \mathbb{R}^d$ *and linear flows* $Flow_u(X, X') \equiv (X' = \Sigma AX + B),\ A_i, B_i \in \mathbb{R}^d$ *is decidable, if the extended partitioning algorithm converges into states with finite axes of symmetry.* □

The more general notion of monotonicity has recently been identified as a useful restriction in characterizing hybrid systems [6]. A function is said to be monotonic (with respect to its arguments), if it is always increasing or always decreasing or constant in the specified interval. For *monotonic convergent deterministic semi-algebraic automata*, monotonic flow and reset maps guarantee that the system has to eventually converge to a fixed point or a limit cycle (chaotic behavior can be ruled out). We now show that, unlike linearity which ensures decidability of reachability, monotonicity only guarantees that approximate reachability can be decided upto any specified accuracy. Thus:

Theorem 2.5 *If there exist 1-to-1 monotonic maps between two sets, all points converge to one of a finite number of fixed points or limit cycles.* □

Theorem 2.6 *Reachability over a deterministic semi-algebraic hybrid system with resets and flows monotonic (with respect to all system variables), that converges upon extended partitioning, is decidable up to an arbitrary degree of accuracy.* □

2.2 Approximating as a Polytope

The bisimulation approach produced a new hybrid system more amenable to approximate temporal analysis. A completely different approach, very popular for the reachability problem, is to approximate from the first step. This involves assuming a mathematically convenient geometrical shape for the initial set—the simplest being a polytope (bounded polyhedron), which can be written as a boolean combination of linear inequalities [3]. At each iteration, we compute the successor polyhedron by expanding it using the (one-step) transition relation of the hybrid system. Also, we need to ensure that the

successor is also a polyhedron. At each iteration, the mathematics involves keeping track of the movement of the vertices and computing their new convex hull, or keeping track of the faces and moving them based on their maximum outward growth along the normal.

Clearly, a polyhedron can serve as a complexity restricting approximation of a semi-algebraic set as well. However, the conventional computational techniques are not applicable for two reasons. First, the *convex hull* of the successors of vertices of a polyhedron cannot be guaranteed to over-approximate the successor of the polyhedron. This is because, unlike linear systems, the flows cannot be assumed to be convexity preserving in semi-algebraic systems. Secondly, the *face-lifting* approach is not applicable in its basic from. This difficulty arises because, there is no straightforward way of calculating the maximum outward component of the flow along the normal to each face, of a polyhedron evolving with arbitrary polynomial dynamics.

In this section, we develop two new approaches that circumvent this problem. Instead of directly computing the approximated successor, we calculate the accurate complex successor (of the polyderon), and *then* approximate it with a new polyhedron. Though the accurate successor computation slows us down, it is still better than the entirely exact computation. This is because the quantified semi-algebraic expression for the successor is relatively simple (polyhedron). We first describe a very coarse over-approximation which merely keeps track of the extremities along each dimension. This simple over-approximation can be obtained by calculating the maximum and minimum value along each dimension and bounding by one hyper rectangle. (We denote the value of the i-th dimension of X by X_i.)

Algorithm 2 ***[Over-Approximating as One Hyper-Rectangle]***

(i) *Initialize the current over-approximation of the reachable set $\mathcal{R}$ with the starting hyper-rectangle $\bigwedge_i(i_{min} \leq X_i \leq i_{max})$;*

(ii) *Calculate the exact successor of $\mathcal{R}$ thus:*
$\mathcal{R}_E(\langle s, X\rangle) \equiv \exists s', X', h\ R(\langle s, X'\rangle) \wedge \langle s', X'\rangle \xrightarrow[T]{h} \langle s, X\rangle$;

(iii) *Calculate the maximum value of each dimension X_i in $\mathcal{R}_E$ thus:*
$\{\exists s, X\ (X_i = i'_{max}) \wedge \mathcal{R}_E(\langle s, X\rangle)\} \bigwedge \{\forall s, X\ \mathcal{R}_E(\langle s, X\rangle) \Rightarrow X_i \leq i'_{max}\}$;

(iv) *Calculate the minimum value of each dimension X_i in $\mathcal{R}_E$ thus:*
$\{\exists s, X\ (X_i = i'_{min}) \wedge \mathcal{R}_E(\langle s, X\rangle)\} \bigwedge \{\forall s, X\ \mathcal{R}_E(\langle s, X\rangle) \Rightarrow X_i \geq i'_{min}\}$;

(v) *For each dimension, $i'_{min} \equiv min(i_{min}, i'_{min})$, $i'_{max} \equiv max(i_{max}, i'_{max})$;*

(vi) *If $j'_{max} \neq j_{max}$ or $j'_{min} \neq j_{min}$ for some dimension X_j, repeat the steps $(ii) - (v)$ with $\mathcal{R} \equiv \bigwedge_i(i'_{min} \leq X_i \leq i'_{max})$; else, the procedure has converged.* □

While the utility of such a gross over-approximation is questionable, it is nevertheless a technique one can resort to when the complexity of the problem is very high. If we want to approximate with a general polyhedron (more

than just a hyper-rectangle), we have to resort to the convex-hull or face-lifting approaches. As arbitrary face-lifting is not known to be amenable to computational analysis, we suggest a convex-hull-based approach. Since the new positions of the vertices cannot capture the new convex-hull, we move them by the maximum possible increments and decrements in one step of the hybrid system. In other words, we compute the maximum (and minimum) displacement (along each dimension) of any point in the polyhedron; and then assume that all the vertices could have moved by these amounts. The convex hull of the vertices, moved by these maximal amounts, is clearly guaranteed to be an over-approximation of the original polyhedron.

Algorithm 3 ***[Over-Approximating as One Hyper-Polygon]***

(i) *Initialize the current over-approximation of the reachable set $\mathcal{R}$ with the starting hyper-polygon, composed of the initial set of n vertices $v_1, \cdots, v_n$;*

(ii) *Calculate the exact successor of $\mathcal{R}$ thus:*
$\mathcal{R}_E(\langle s, X\rangle) \equiv \exists s', X', h\ \mathcal{R}(\langle s', X'\rangle) \wedge \langle s', x'\rangle \xrightarrow[T]{h} \langle s, X\rangle$;

(iii) *Calculate the maximum increment δ_{inc} in each dimension X_i thus:*
$\{\exists s, X, s', X'\ \mathcal{R}(\langle s, X\rangle) \wedge \mathcal{R}_E(\langle s', X'\rangle) \wedge (X_i' - X_i = \delta_{inc})\} \bigwedge$
$\{\forall s, X, s', X'\ (\mathcal{R}(\langle s, X\rangle) \wedge \mathcal{R}_E(\langle s', X'\rangle)) \Rightarrow (X_i' - X_i \leq \delta_{inc})\}$;

(iv) *Calculate the maximum decrement δ_{dec} in each dimension X_i thus:*
$\{\exists s, X, s', X'\ \mathcal{R}(\langle s, X\rangle) \wedge \mathcal{R}_E(\langle s', X'\rangle) \wedge (X_i - X_i' = \delta_{dec})\} \bigwedge$
$\{\forall s, X, s', X'\ (\mathcal{R}(\langle s, X\rangle) \wedge \mathcal{R}_E(\langle s', X'\rangle)) \Rightarrow (X_i - X_i' \leq \delta_{dec})\}$;

(v) *Each vertex contributes 2^d new points, with each dimension being increased or decreased by the maximum amounts. R is assigned the convex hull of these $n2^d$ points;*

(vi) *Iterate $(ii) - (vii)$ until $\delta_{inc} = \delta_{dec} = 0$.* □

2.3 Rectangular Grid Abstraction

Instead of using one large polytope, the *grid abstraction* approach relies on keeping track of a number of small simple hyper-rectangles. Rectangular grids admit canonical representations and the number of faces grows linearly with the dimension, as opposed to convex polyhedra which become intractable in higher dimensions [2]. Two common simplifying strategies are restricting the vertices to be integers ("griddy") and the edges to be axis-parallel ("isothetic") [1].

We first show that the extension to semi-algebraic hybrid automata of the standard procedure is possible, because quantifier elimination can be used to compute the transitions between hyper-rectangles. One can partition the entire space into N^d hyper-rectangles, where N is the number of the $\mathcal{A}$-sized partitions [4] of each of the d dimensions. We use $B(X)$ to denote the k-

[4] $\mathcal{A}$ should be fixed in relation to the error in the $\xrightarrow[C]{h}$.

dimensional grid unit $\bigwedge_i(B_i \leq X_i < (B_i + \mathcal{A}))$ of size $\mathcal{A}^d$. States will be connected to some of their 3^d-1 immediate neighbors, which differ by $+\mathcal{A}$,$-\mathcal{A}$ or 0 units in each dimension (with the identity-case alone excluded), and to some farther ones resulting from discrete resets. We now list the series of computations necessary to calculate the reachable region starting from a specific grid unit:

Algorithm 4 ***[Reachability Over Numerical Grids]***

(i) *Given one hyper-rectangle $F(X)$ corresponding to the source;*

(ii) *Initialize "frontier" set $\mathcal{F}$ with $\{F(X)\}$, and "reachable set" $\mathcal{R}$ with* null*;*

(iii) *For each new hyper-rectangle $P(X) \in \mathcal{F}$*

(a) *Compile the set of neighbors:*
$\mathcal{N} \equiv \{Q(X)|(|Q_i - P_i| = \mathcal{A} \vee 0) \wedge \bigvee_i(|Q_i - P_i| \neq 0)\}$*;*

(b) *For each neighbor $Q(X)$ of $P(X)$ not already in the reachable set, test if it is reachable i.e.* $\exists X,\ P(X) \wedge \bigvee_{\forall v}\{\exists Y\ \bigvee_{\forall u}\langle v, X\rangle \xrightarrow[D]{0} \langle u, Y\rangle \wedge Q(Y)\}\ \bigvee\{\exists Y, h\ (0 < h \leq \mathcal{A}) \wedge \langle v, X\rangle \xrightarrow[C]{h} \langle v, Y\rangle \wedge Q(Y)$*;*

(c) *All candidate non-adjoint cells $Q(X)$ that can be reached by discrete state transitions can be tested thus:*
$\exists X,\ P(X) \wedge \bigvee_{\forall v}\{\exists Y\ \bigvee_{\forall u}\langle v, X\rangle \xrightarrow[D]{0} \langle u, Y\rangle \wedge Q(Y)\}$*.*

(d) *Add all reachable cells to both the reachable set $\mathcal{R}$ and the frontier set $\mathcal{F}$ and remove $P(X)$ from $\mathcal{F}$;*

(iv) *Iterate until there are no more new-hyper-rectangles.* □

Having shown that the standard procedure is applicable, we now develop a new approach for computing a sharper over-approximation (successor set of small hyper-rectangles) of the given hyper-rectangle. The idea is to compute the exact successor of a hyper-rectangle, and then over-approximate the region outside the initial hyper-rectangle (the "spill") by hyper-rectangles. In the previous case, we considered each of the $3^d - 1$ non-overlapping neighboring zones, and tested the transition to each. To simplify the expressions further, we suggest considering fewer overlapping neighbors; in particular, the zones with exactly one of the d dimensions increased or decreased i.e., $2d$ in all. To summarize, the standard method (previous case) accumulates the hyper-rectangles reachable from the given hyper-rectangle by testing transition to each of the 3^d-1 non-overlapping neighbors. The size of each neighbor is fixed ("griddy") forcing the approximation error to be at least that big. In the new technique, the hyper-rectangles continue to be axis-parallel ("isothetic"), but their vertices are not fixed. As a result, the approximation is guaranteed to be much better than the "griddy" case. The additional trick of considering fewer overlapping rectangles cannot be applied to the standard method, as the approximation will become too coarse.

We now present the details of the method. We estimate the spill in each neighboring zone by calculating the extremities in that zone, along the lines of

the scheme for over-approximating the entire set as a single-hyper-rectangle. We use $B(X)$ to denote the k-dimensional grid unit $\bigwedge_i (B_i^l \leq X_i < B_i^r)$ (side of the hyper-rectangles are no longer fixed at $\mathcal{A}$). Further, $B_{\neg j,k}(X)$ denotes $\bigwedge_{i \neq j \vee k} (B_i^l \leq X_i < B_i^r)$.

Algorithm 5 ***[Approximating with Many Hyper-Rectangles]***

(i) *As before, maintain the set of reachable hyper-rectangles $\mathcal{R}$ and the set of new hyper-rectangles $\mathcal{F}$ just added to the reachable set, representing the expanding frontier;*

(ii) *For each $P(X) \in \mathcal{F}$, compute the exact successor set of $\mathcal{R}$ thus:*
$\mathcal{R}_E \equiv \bigvee_{\forall v} \{\exists Y \bigvee_{\forall u} \langle v, X \rangle \xrightarrow[D]{0} \langle u, Y \rangle \wedge P(Y) \bigvee \{\exists Y, h \ (0 < h \leq \mathcal{A}) \wedge \langle v, X \rangle \xrightarrow[C]{h} \langle v, Y \rangle \wedge P(Y)$*;*

(iii) *For each dimension X_i:*
   (a) *For the neighbor $Q(X)$ where $Q_i^l = N_i^r$, calculate Q_i^r: $\{\exists X \ (P_{\neg i}(X) \wedge X_i = Q_i^r) \wedge \mathcal{R}_E(X)\} \bigwedge \{\forall X \ (P_{\neg i}(X) \wedge X_i > Q_i^r) \Rightarrow \neg \mathcal{R}_E(X)\}$. If $Q_i^r < P_i^r$, skip the next two steps;*
   (b) *We now need to calculate the extremities l_j^{i+}, r_j^{i+} in each of the other dimensions X_j where $j \neq i$: $\{\exists X \ (P_{\neg i,j}(X) \wedge X_i > P_i^r \wedge X_i < Q_i^r \wedge X_j = l_j^{i+}) \wedge \mathcal{R}_E(X)\} \bigwedge \{\forall X \ (P_{\neg i,j}(X) \wedge X_i > P_i^r \wedge X_i < Q_i^r \wedge X_j < l_j^{i+}) \Rightarrow \neg \mathcal{R}_E(x)\}$ and $\{\exists X \ (P_{\neg i,j}(X) \wedge X_i > P_i^r \wedge X_i < Q_i^r \wedge X_j = r_j^{i+}) \wedge \mathcal{R}_E(X)\} \bigwedge \forall X \ (P_{\neg i,j}(X) \wedge X_i > P_i^r \wedge X_i < Q_i^r \wedge X_j > r_j^{i+}) \Rightarrow \neg \mathcal{R}_E(X)$.*
   (c) *The hyper-rectangle defined by $Q_i^l < X_i < Q_i^r \bigwedge_{j \neq i} l_j^{i+} < X_j < r_j^{i+}$ is added to the list of new hypercubes and also to the reachable set R;*
   (d) *Repeat the above three steps for the neighbor where $Q_i^r = P_i^l$ and $Q_i^l, l_j^{i-} (< X_j), (X_j <) r_j^{i-}$ need to be calculated;*

(iv) *Repeat $(ii) - (iii)$ until the procedure converges.* □

In the "griddy" case, we inspect every possible neighbor and test transition. Alternately, we could have computed the exact successor of the entire set, and then extracted the component hyper-rectangles. Such an approach would require a procedure for converting a semi-algebraic set (the exact successor) into an over- (or under-) approximating union of hyper-rectangles of fixed dimension.

In the "isothetic" case, we over-approximated the "spill" outside the hyper-rectangle with a hyper-rectangle in each neighboring zone (with substantial overlap). Alternatively, we could compute the best non-overlapping but non-griddy hyper-rectangles that cover the newly reachable points, without having to compute the maximum and minimum values of each dimension in each neighboring zone. This approach again requires a general procedure for converting the exact successor into a union of hyper-rectangles of arbitrary dimension.

We solve this problem by actually testing if potential vertices (from a

griddy or isothetic grid) are included in the exact reachable set. We then use the resulting set of present and absent points to pick candidate hyper-rectangles. Quantifier elimination is still necessary, since we may wish to guarantee that the hyper-rectangles we have picked are wholely inside (under-approximation) or that the hyper-rectangles we have omitted are wholely outside (over-approximation). Hence the approach we have suggested addresses the problem of minimizing the number of quantifier-elimination queries. We now provide the details of this new technique, which can be used in conjunction with both the algorithms presented before.

Algorithm 6 ***[Over-Approximating using Hyper-Rectangles]***

(i) *Calculate* i_{max} *and* i_{min}*, the maximum and minimum values of* X_i *in the given set* $\mathcal{R}$*:* $\{\exists X\ (X_i = i_{max}) \wedge \mathcal{R}(X)\} \bigwedge \{\forall X\ \mathcal{R}(X) \Rightarrow X_i \leq i_{max}\}$ *and* $\{\exists X\ (X_i = i_{min}) \wedge R(X)\} \bigwedge \{\forall X\ R(X) \Rightarrow X_i \geq i_{min}\}$*;*

(ii) *Split each dimension into equidistant points of the desired resolution;*

(iii) *Evaluate membership in* R *for each grid point* g *by substitution:* $\mathcal{R}(g)$*;*

(iv) *The small hyper-rectangles created by the grid points which contain at least one vertex in* $\mathcal{R}$ *are immediately included in the over-approximation;*

(v) *Hyper-rectangles where none of the vertices are in* $\mathcal{R}$ *are included only if* $\exists x \in G\ \mathcal{R}(x)$ *returns* true. □

In the under-approximation case, hyper-rectangles with at least one vertex not in R can be safely omitted. The hyper-rectangles with all vertices in R are the contenders for quantifier elimination. In both cases, one could use a "proof-by-example" approach, where one verifies the feasibility at some randomly selected points (center being the first choice) to see if quantifier elimination can be avoided. By randomizing or biasing the grid points, one can obtain non-griddy vertices. If, in addition, high-dimensional convex hull algorithms are used, one could build upon this method to derive general polyhedral representations as well.

2.4 Time Discretization

For the sake of completeness, we also note that time discretization can be employed (in conjunction with most techniques) to approximate the hybrid system dynamics. Conventionally, the most restricted transition relation enforces continuous evolution for a fixed time-step Δ followed by one optional discrete transition. The typical "improvement" over the previous case could be allowing the discrete jump anywhere during the continuous evolution, as opposed to only at the end of it. This model could be made even more realistic by allowing N jumps anywhere during the continuous evolution. Clearly, the only paths that get excluded here are those that involve more than N jumps in Δ time. All the restrictions described above are "fixed step" i.e. the system progresses in timesteps of Δ. Each of them could be relaxed by allowing the time-step to be in the range $[0, \Delta]$ to capture many other behaviors. Such re-

strictive transition relations greatly simplify fixpoint evaluations of temporal logic operators.

A completely different time-discretization-based under-approximating approach would be to ignore the behavior of the system *during* the continuous evolution. We simply use the end-points to verify the temporal query. For example, the TCTL one-step until operator for semi-algebraic hybrid systems [8] can be simplified as: $p \triangleright q = q \bigvee_{\forall v}\{\exists s \bigvee_{\forall u}\langle v,r\rangle \rightarrow_D^0 \langle u,s\rangle \wedge q(s)\} \bigvee\{\exists s,h\ (0<h\leq\Delta)\wedge\langle v,r\rangle \rightarrow_C^h \langle v,s\rangle \wedge q(s) \wedge p(r)\}$. Another simplifying over-approximation would be to assume that the state invariant needs to be true only at the beginning of (and not all along) the Δ time units of continuous evolution. This heuristic could prove particularly useful if we combine time discretization with the partitioning algorithm discussed earlier (which will accumulate complex state-invariants).

3 Discussion

In this paper, we have extended the theory of approximate verification of hybrid systems from the linear to the more expansive semi-algebraic domain. The algebraic model checking method presented in [8] was made more computationally practicable by extended bisimulation partitioning, approximation with general polyhedra and unions of simple polyhedra, and time discretization. For the extended bisimulation procedure suggested, we identified well-behaved subclasses based on some novel critical observations about the behavior of exactly onto linear and monotonic maps between arbitrary sets. For polyhedral approximations, we used the maximum and minimum values of the system variables and their possible growth in one step to expand the convex hull. We demonstrated how these same metrices (maximal growth along each dimension in one step) could be used to obtain a hyper-rectangular approximation of semi-algebraic sets. We also introduced a practical strategy to identify candidate hyper-rectangles, for which the quantifier elimination needs to be invoked. Time discretization was seen to simplify the problem by allowing fewer discrete jumps, excluding zeno paths, and by verifying the temporal property at certain sampling times rather than everywhere.

All these methods need to be refined to better handle discrete resets and symbolic approximations. More crucial is their actual implementation and performance analysis. On the purely algebraic side, approximate quantifier elimination and direct maximum-minimum estimation of a semi-algebraic set are those mathematical techniques, that need to be developed to further accelerate these methods. There are several approximating methods that are yet to be extended to semi-algebraic systems. These include: (1) piece-wise approximations of continuous dynamics; (2) problem domain transformation: optimal control using Pontryagin Maximum Principle, level sets of solutions to Hamilton-Jacobi-Bellman equations, sum of squares decomposition (semidefinite programming) and geometric programming; (3) predicate abstraction and

qualitative simulation; (4) other geometric approximations: oriented rectangular hulls, zonotopes and ellipsoids. Next, we wish to determine practical applicability of these methods, trade-offs among them and suitable combinations of these approximations that work best with the available tools.

References

[1] E. Asarin, T. Dang, O. Maler, and O. Bournez. Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems. In B. Krogh and N. Lynch, editors, *Hybrid Systems: Computation and Control (HSCC'00)*, volume 1790 of *LNCS*, pages 20–31. Springer-Verlag, 2000.

[2] O. Bournez, O. Maler, and A. Pnueli. Orthogonal Polyhedra: Representation and Computation. In F. Vaadrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC 1999)*, volume 1596 of *LNCS*, pages 19–30. Springer-Verlag, 1999.

[3] A. Chutinan and B. Krogh. Verification of Polyhedral-Invariant Hybrid Automata Using Polygonal Flow Pipe Approximations. In F. W. Vaandrager and J. H. van Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *LNCS*, pages 76–90. Springer-Verlag, 1999.

[4] Ronojoy Ghosh and Claire Tomlin. Symbolic reachable set computation of piecewise affine hybrid automata and its application to biological modelling: Delta-notch protein signalling. *Systems Biology*, 1(1):170–183, June 2004.

[5] H. Hong. Quantifier elimination in elementary algebra and geometry by partial cylindrical algebraic decomposition, version 13. *WWW site* `www.eecis.udel.edu/~saclib`, 1995.

[6] R. Lanotte and A. Maggiolo-Schettini. Monotonic hybrid systems. *Journal of Computer and System Sciences*, 2004.

[7] B. Mishra. *Computational Real Algebraic Geometry.* CRC Press, Boca Raton, FL, 2004.

[8] V. Mysore, C. Piazza, and B. Mishra. Algorithmic Algebraic Model Checking II: Decidability of Semi-Algebraic Model Checking and its Applications to Systems Biology. In *Automated Technology for Verification and Analysis (ATVA) (submitted)*, 2005.

[9] C. Piazza, M. Antoniotti, V. Mysore, A. Policriti, F. Winkler, and B. Mishra. Algorithmic Algebraic Model Checking I: The Case of Biochemical Systems and their Reachability Analysis. In *Computer Aided Verification (CAV)*, 2005.

[10] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, second edition, 1948.

4 Appendix

4.1 Semi-Algebraic Hybrid Automata: Definitions and Decidability

Definition 4.1 Semi-Algebraic Set[7] Every quantifier-free boolean formula composed of polynomial equations and inequalities defines a semialgebraic set (i.e., unquantified first-order formulæ over the reals —$(\mathbb{R}, +, \times, =, <$)). □

Definition 4.2 Semi-Algebraic Hybrid Automata [9] A k-dimensional *hybrid automaton* is a 7-tuple, $H = (Z, V, E, Init, Inv, Flow, Jump)$, consisting of the following components:

- $Z = \{Z_1, \ldots, Z_k\}$ and $Z' = \{Z'_1, \ldots, Z'_k\}$ are two finite sets of variables ranging over the reals $\mathbb{R}$
- (V, E) is a directed graph of discrete states and transitions
- Each vertex $v \in V$ is labeled by "Init"(initial), "Inv"(invariant) and "Flow" labels of the form $Init_v[Z]$, $Inv_v[Z]$, and $Flow_v[Z, Z', t, h]$
- Each edge $e \in E$ is labeled by a "Jump" condition of the form $Jump_e[Z, Z'] \equiv Guard_e(Z) \wedge Reset_e(Z, Z')$
- $Init$, Inv, $Flow$, and $Jump$ are semi-algebraic. □

Definition 4.3 Semantics of Hybrid Automata[8] Let $H = (Z, V, E, Init, Inv, Flow, Jump)$ be a hybrid automaton of dimension k.

- A *location* ℓ of H is a pair $\langle v, R\rangle$, where $v \in V$ is a state and $R \in \mathbb{R}^k$ is an assignment of values to the variables of Z. A location $\langle v, R\rangle$ is said to be *admissible*, if $Inv_v(R)$ is satisfied.
- The *continuous reachability transition relation* $\xrightarrow[\mathcal{C}]{h}$ forces the state invariant to hold at every point except the end-point along the evolution curve determined by the flow equations during the $h(> 0)$ time units from the current time t_0:

$$\langle v, R\rangle \xrightarrow[\mathcal{C}]{h} \langle v, S\rangle \quad \text{iff}$$
$$\Big(Flow_v(R, S, t_0, h) \ \wedge \ \forall Z', h' \in [0, h) \ Flow_v(R, Z', t_0, h') \Rightarrow Inv_v(Z') \Big),$$

  where $Flow_v(Z, Z', T, h)$ is the flow label of v.
- The *discrete reachability transition relation* $\xrightarrow[\mathcal{D}]{0}$ ensures that both parts of the *zero-time* jump – the guard condition which needs to be satisfied just before the transition is taken, and the reset condition which determines the values after the transition, are satisfied.

$$\langle v, R\rangle \xrightarrow[\mathcal{D}]{0} \langle u, S\rangle \quad \text{iff} \quad \langle v, u\rangle \in E \ \wedge \ Jump_{v,u}(R, S).$$

- The *transition relation* $\mathcal{T}$ of H connects the possible values of the system

variables before and after *one step*—a discrete step for a time $h = 0$ or a continuous evolution for any time period $h > 0$:

$$\mathcal{T}(\ell \xrightarrow{h} \ell') = \{h = 0 \wedge \ell \xrightarrow[\mathcal{D}]{0} \ell'\} \vee \{h > 0 \wedge \ell \xrightarrow[\mathcal{C}]{h} \ell'\}.$$

- A *trace* of H is a sequence $\ell_0,\ell_1, \ldots, \ell_n, \ldots$ of admissible locations such that

$$\forall i \geq 0,\ \exists h_i \geq 0,\ \mathcal{T}(\ell_i \xrightarrow{h_i} \ell_{i+1}). \quad \square$$

Remark 4.4 When a semi-algebraic relation $Flow_v(R, S, t, h)$ is used between the continuous states R at time t and S at time $t + h$ in a discrete state v, it may have been "derived" in two ways: (1) *Solution Is A Polynomial*: The equation describing the continuous evolution of the variables in a discrete state is a polynomial, say $Y(t)$, and $Flow_v(Z, Z', t, h) \equiv \{\ Z = Y(t)\ \wedge\ Z' = Y(t + h)\ \}$. Or, (2) *Differential Equation Is A Polynomial*: Differential equations describing the continuous evolution are *approximated* in $Flow_v$ using one of the symbolic integration schemes (e.g., the *Taylor* series in [9] or based on a direct integration scheme such as the linear *Euler* or the higher degree *Runge-Kutta*). The error is controlled by an upper bound (say Δ) on the time spent in one continuous step as we aim for over- or under-approximating the flow equations. The Lagrange Remainder Theorem can be used to estimate errors.

Definition 4.5 $\triangleright$ for Semi-Algebraic Hybrid Systems. The expression $p \triangleright q$ is *True* at the current continuous state R if q is true now, *OR*

- For one of the possible current discrete states v, there exists at least one state u to which a transition can be taken such that q holds at the end, *OR*
- For one of the possible current discrete states v, there exists a continuous transition (of at most Δ time units when we need to upper-bound the flow-approximation error) all along which $p \vee q$ holds, with q being true at the end.

$$\begin{aligned} p \triangleright q \ =\ & q(R) \bigvee_{\forall v} \\ & \Big(\{\exists S\ \bigvee_{\forall u} \langle v, R\rangle \xrightarrow[\mathcal{D}]{0} \langle u, S\rangle \wedge q(S)\}\ \bigvee \\ & \{\exists S, h\ (0 < h \leq \Delta) \wedge \langle v, R\rangle \xrightarrow[\mathcal{C}]{h} \langle v, S\rangle \wedge q(S)\ \wedge \\ & \quad \forall S', h'\ ((0 \leq h' < h) \wedge \langle v, R\rangle \xrightarrow[\mathcal{C}]{h'} \langle u, S'\rangle) \Rightarrow (p(S') \vee q(S'))\} \Big) \quad \square \end{aligned}$$

Remark 4.6 The last term in the formula, $p(S') \vee q(S')$, can be replaced with just $p(S')$ for evaluating $\exists \mathcal{U}$ over semi-algebraic hybrid systems. Also, the upperbound Δ on h should be *omitted* if there is no error in the $Flow_v$ expression.

Theorem 4.7 *[8] The* one-step-until *operator $p \triangleright q$ is decidable for semi-algebraic hybrid systems if p and q are also semi-algebraic.*

Corollary 4.8 *For semi-algebraic hybrid systems:*

(i) $\exists\mathcal{U}$, $\exists\mathcal{F}$, $\exists\mathcal{G}$ *and their subscripted versions* $\exists\mathcal{U}_{\leq z}$, $\exists\mathcal{F}_{\leq z}$ *and* $\exists\mathcal{G}_{\leq z}$ *are* semi-decidable.

(ii) *The* negations *of* $\forall\mathcal{U}$, $\forall\mathcal{F}$, $\forall\mathcal{G}$ *and their subscripted versions* $\forall\mathcal{U}_{\leq z}$, $\forall\mathcal{F}_{\leq z}$ *and* $\forall\mathcal{G}_{\leq z}$ *are* semi-decidable.

(iii) *All subscripted operators become* decidable *in the absence of* zeno *paths.*

4.2 Details of Proofs

Proof Of Theorem 2.1 Each state s (source) needs to be split into two states s_d and $s_{\bar{d}}$ depending on whether or not the guard of the transition to each d (destination) can ever be satisfied. Since real quantifier elimination is decidable [10,5], these partitions can be computed thus: $Inv_{s_d}(X) \equiv \exists h, X' \langle s, X\rangle \xrightarrow[C]{h} \langle s, X'\rangle \wedge Guard_{s,d}(X')$ and $Inv_{s_{\bar{d}}}(X) \equiv Inv_s(X) \wedge \neg Inv_{s_d}(X)$.□

Proof Of Theorem 2.2 Consider two sets S_1 and S_2 between which *onto* linear maps exist. Maps of the form $x' = \Sigma a_i x_i + a_0$ correspond to a rotation, stretch and shift of the coordinate axes. In other words, S_2 *has* to be a stretched, rotated and shifted image of S_1 for such an onto map to exist. There are 2^{s_l} maps possible because of the s_l axes of linear symmetry, on *each* of the s_r axes of rotational symmetry. Hence the total number of coupled linear onto maps is $s_r 2^{s_l}$. □

Proof of Corollary 2.3 Let $m(= s_r 2^{s_l})$ be the number of possible onto maps between S_1 and S_2. Let $x_1 \in S_1$ map to one of $y_1, \cdots, y_m \in S_2$. Let y_1 map to one of $x_1, \cdots, x_m \in S_1$ (x_1 has to appear because the inverse of a linear map is linear). Suppose x_2 maps to y_{m+1}. Since linear maps are closed under composition and retain their ontoness property, by following the linear maps from $x_1 \rightarrow y_{1\leq i\leq m} \rightarrow x_{1\leq i\leq m} \rightarrow y_{m+1}$, we get a new linear map that takes x_1 to y_{m+1}. However, we know from symmetry arguments that only m linear maps can exist. This contradiction proves that no matter how many times we compose the two given onto linear maps, we remain within the set of $2n$ points (for example, rectangles have 8 possible linear onto maps while cubes have 24). Extending this argument to a cycle of n states, each point can have only one of m different successors in each of the n states. Hence, the length of the biggest cycle is nm. □

Proof of Theorem 2.4 The continuous evolution can be treated as a linear map from the initial value to the final value that first satisfies the guard. Further, *time* does not appear in the equation as in deterministic systems, an initial value corresponds to a unique final value. No restriction on the guard is necessary as we assume there is only 1 successor to each discrete state. Thus a cycle of n states corresponds to a cycle of $2n$ linear 1-to-1 maps with only the values before and after a reset sufficing to capture the dynamics. If m is the maximum number of possible onto maps between any two consecutive sets, the number of unique successors is $\leq 2nm$. Since x_0 has a finite number

of successors, if the target x_f is not reached before the system begins to cycle, we conclude exact unreachability. If x_f is eventually reached, then it is indeed exactly reachable. □

Proof of Theorem 2.5 Let the sequence be X_0, $X_1 = f(X_0)$, $X_0' = g(X_1) = g(f(X_0)), \cdots$. Since f and g are monotonic, X will continue to move in the same direction. The process can continue ad infinitum if X approaches a fixed point ($g(f(X)) = X$). The other mathematical alternative is that it approaches a limit cycle ($f(g(f(g(..(X)..)))) = X$). Monotonicty ensures progress while ontoness ensures finiteness of the number of iterations required to reach the neighborhood of a fixed point or a limit cycle. □

Proof of Theorem 2.6 Just as in the linear case, the continuous flow can also be thought of as a monotonic function from the initial value (from a reset that brought the system to this state) to the final value (when a guard is first satisfied). Thus any cycle of n discrete states corresponds to a cycle of $2n$ monotonic maps (n flow-maps and n reset maps). Further, from the previous theorem, we know that iterative evolution along such a cycle of exactly onto maps has to approach a fixed point or a limit cycle. We can stop iterating when $\bigwedge(|X_i - X_i'| < \epsilon_i$, where X is the d-dimensional value of the system variables, ϵ_i is the desired accuracy in the i-th dimension and X' is the value after one cycle (reached the neighborhood of a fixed point) or after $2, 3, \cdots, d$ cycles (reached the neighborhood of a limit cycle) . The monotonic resets guarantee that this will happen in a finite number of steps and that once this happens, the system cannot escape out of it. □

Liveness Checking as Safety Checking for Infinite State Spaces

Viktor Schuppan [1]

ETH Zürich, Computer Systems Institute, CH-8092 Zürich, Switzerland

Armin Biere [2]

Johannes Kepler University, Institute for Formal Models and Verification
Altenbergerstrasse 69, A-4040 Linz, Austria

Abstract

In previous work we have developed a syntactic reduction of repeated reachability to reachability for finite state systems. This may lead to simpler and more uniform proofs for model checking of liveness properties, help to find shortest counterexamples, and overcome limitations of closed-source model-checking tools. In this paper we show that a similar reduction can be applied to a number of infinite state systems, namely, ($\omega-$)regular model checking, push-down systems, and timed automata.

Key words: liveness, safety, linear temporal logic, model checking, infinite state space

1 Introduction

While model checking of safety properties can be reduced to computing the set of reachable states of a system [16], verification of general LTL properties is typically performed by searching for infinite paths in the product of the system and an automaton representing the property [22].

In [19] we have developed a syntactic reduction from computing repeated reachability to computing reachability for finite state systems. This reduction has been used to develop a BDD-based method to find shortest counterexamples [20]. On selected examples a significant speed up compared to traditional liveness checking can be observed [19,21]. It can also help to simplify proofs if a proof for safety properties is easier than the corresponding proof for general LTL properties. It may finally discourage tool vendors from charging

[1] Email: Viktor.Schuppan@inf.ethz.ch
[2] Email: biere@jku.at

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

separately for liveness-enabled versions of their verification tools. For further motivation and related work on finite state systems see [19].

In this paper we develop similar reductions for a number of infinite state systems. Classes of infinite state systems, which have received considerable attention in the past and for which verification tools are available (e.g., [1,12,17]), are ($\omega-$)regular model checking [15,23,7,5], pushdown systems [6,13,11], and timed automata [3].

Early work on liveness for regular model checking includes [7,18]. Pnueli and Shahar [18] also use a copy of a current state to detect bad cycles in parameterized systems. However, this is not performed as syntactic transformation of a model but as part of a dedicated liveness checking algorithm. A variant of LTL geared towards parameterized systems is proposed in [2]. [8] gives details on how to encode a broader set of properties than [2] for ($\omega-$)regular model checking, which can be used in conjunction with our reduction. Algorithms to compute repeated reachability, on which we also base our reductions, can be found for pushdown systems, e.g., in [6], and for timed automata in [3].

After some notation common to all classes of systems in Sect. 2, Sect. 3 presents the basic idea of our reduction using finite state systems as an example. It is extended to ($\omega-$)regular model checking in Sect. 4 and to pushdown systems in Sect. 5. Due to space constraints the construction for timed automata can only be sketched in Sect. 6. The last section concludes.

2 Common Notation

The set of Booleans is denoted by $\mathbb{B} = \{0, 1\}$; $\mathbb{N}$ and $\mathbb{R}$ are naturals and reals, respectively. Elements of a tuple are separated by commas. Elements of a sequence typically have no operator between them, $\circ$ is used only if ambiguity might arise. For a sequence ρ, $\rho(i)$ denotes the i-th element of the sequence (starting with $\rho(0)$). The length of a sequence, $|\rho|$, is defined as the number of its elements. If S is a set, S^* and S^ω are the sets of finite and infinite sequences of elements of S.

We introduce an operator μ, which forms a sequence of tuples from a tuple of sequences. Given two words $v, w \in \Sigma^*$ with $|v| \leq |w|$ we define $\mu(v, w) = (v(0), w(0)) \ldots (v(|v| - 1), w(|v| - 1)) \in (\Sigma \times \Sigma)^*$.

3 Liveness Checking as Safety Checking – Finite Case

In this section we briefly restate the main result from [19] to explain the basic idea and notation of our reduction.

3.1 Preliminaries

Let AP be a finite set of atomic propositions. A *Kripke structure*, see, e.g., [9], is a four tuple $M = (S, S_0, R, L)$ where S is a finite set of *states*, $S_0 \subseteq S$

Definition 3.1 Let $M = (S, S_0, R, L)$ be a Kripke structure. Then $M^{\mathbf{S}} = (S^{\mathbf{S}}, {S_0}^{\mathbf{S}}, R^{\mathbf{S}}, L^{\mathbf{S}})$ is defined as:

$$
\begin{aligned}
S^{\mathbf{S}} &= S \times S \times \mathbb{B} \times \mathbb{B} \\
{S_0}^{\mathbf{S}} &= \{(s_0, \hat{s}_0, 0, 0) \mid s_0 \in S_0\} \cup \{(s_0, s_0, 1, 0) \mid s_0 \in S_0\} \\
R^{\mathbf{S}} &= \{((s, \hat{s}, lb, lc), (s', \hat{s}', lb', lc')) \mid (s, s') \in R \wedge \\
&\quad ((\neg lb \wedge \neg lb' \wedge \neg lc \wedge \neg lc' \wedge \hat{s} = \hat{s}') \vee && (1) \\
&\quad\ (\neg lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge s' = \hat{s}') \vee && (2) \\
&\quad\ (lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge \hat{s} = \hat{s}') \vee && (3) \\
&\quad\ (lb \wedge lb' \wedge \neg lc \wedge lc' \wedge \hat{s} = s' = \hat{s}') \vee && (4) \\
&\quad\ (lb \wedge lb' \wedge lc \wedge lc' \wedge \hat{s} = \hat{s}'))\} && (5) \\
L^{\mathbf{S}}((s, \hat{s}, lb, lc)) &= L(s)
\end{aligned}
$$

is the set of *initial states*, $R \subseteq S \times S$ is a *transition relation*, and $L : S \mapsto 2^{AP}$ is a *labeling* of the states.

A *run* is a (finite or infinite) sequence of states $\rho = \rho(0)\rho(1)\ldots$ where $\forall 0 \leq i < |\rho| \;.\; (\rho(i), \rho(i+1)) \in R$. ρ is initialized if $\rho(0) \in S_0$, $Runs(M)$ denotes the set of runs of M.

3.2 Reduction

A liveness property $\mathbf{F}p$, where p is propositional, is violated in a finite state system iff there exists a lasso-shaped path where p never holds on that path. Finding such loop is a key ingredient of many model checking algorithms for LTL, e.g., [22,4]. Our reduction integrates the detection of a loop into the model to be verified by nondeterministically saving the current state (i.e., guessing a potential loop start) and then watching for a second occurrence of that state (i.e., detecting closure of the loop). For this purpose, the reduction extends a state s in the original model with a component to store a previously seen state, $\hat{s}$, and two flags lb ($l$oop $b$ody) and lc ($l$oop $c$losed). lb is set to true when a state is saved and prevents future overwriting of the stored state. lc indicates that a second occurrence of $\hat{s}$ has been found.

Definition 3.1 shows the construction. The transitions of $R^{\mathbf{S}}$ are partitioned into subsets. Subset (1) covers the case when no state has been saved so far. Saving happens either at the initial state or via a transition from set (2). Transitions from the third set (3) are taken as long as no second occurrence of the stored state has been seen. A second occurrence is finally detected by a transition in (4). After that only transitions from the last set (5) are taken.

Theorem 3.2 *Let $M = (S, S_0, R, L)$ be a Kripke structure, let $M^{\mathbf{S}}$ be defined*

as above. Assume $k > l \geq 0$.

$$(s_0 \dots s_{l-1})(s_l \dots s_{k-1})^\omega \in Runs(M)$$

$$\Leftrightarrow$$

$$(s_0, \hat{s}_0, 0, 0) \dots (s_{l-1}, \hat{s}_0, 0, 0)(s_l, s_l, 1, 0) \dots (s_{k-1}, s_l, 1, 0)(s_k, s_l, 1, 1) \in Runs(M^{\mathbf{S}})$$

Proof. "⇒": Let $\rho = (s_0 \dots s_{l-1})(s_l \dots s_{k-1})^\omega$ be a run in M. We construct $\rho^{\mathbf{S}}$ as follows. If $l > 0$ choose $\rho^{\mathbf{S}}(0) = (s_0, \hat{s}_0, 0, 0)$ with arbitrary $\hat{s}_0$. Construct $(s_0, \hat{s}_0, 0, 0) \dots (s_{l-1}, \hat{s}_0, 0, 0)$ by taking transitions from subset (1). Proceed to $(s_l, s_l, 1, 0)$ via a transition from (2). Continue to $(s_{k-1}, s_l, 1, 0)$ with $k - l - 1$ transitions from (3). Finally, as $k > l$, there exists $s_k = s_l$, so take a transition from (4) to $(s_k, s_l, 1, 1)$. Otherwise, if $l = 0$, start with $(s_0, s_0, 1, 0)$ and continue with $k - 1$ transitions from (3) and one from (4) as before.

"⇐": Let $\rho^{\mathbf{S}} = (s_0, \hat{s}_0, 0, 0) \dots (s_{l-1}, \hat{s}_0, 0, 0)(s_l, s_l, 1, 0) \dots (s_{k-1}, s_l, 1, 0) \circ (s_k, s_l, 1, 1)$ be a run in $M^{\mathbf{S}}$ such that $k > l$. From the construction of $M^{\mathbf{S}}$, $\rho' = s_0 \dots s_{l-1} s_l \dots s_{k-1} s_k$ is a finite run in M with $s_k = s_l$. Hence, $\rho = (s_0 \dots s_{l-1})(s_l \dots s_{k-1})^\omega$ is a run in M as desired. □

Remark 3.3 Checking properties given as Büchi automata requires finding *fair loops*. This can be achieved by adding a flag for each fairness constraint, for details see [19]. The infinite cases can be handled similarly.

3.3 Complexity

Intuitively, $M^{\mathbf{S}}$ consists of $|S|$ parallel copies of M. Hence, we immediately have the following result.

Proposition 3.4 *Let* $M = (S, S_0, R, L)$ *be a Kripke structure.* $M^{\mathbf{S}}$ *has* $\mathbf{O}(|S|^2)$ *states and* $\mathbf{O}(|S||R|)$ *transitions.*

Proof. Each state in $S^{\mathbf{S}}$ stores, in addition to the original state s, another state $\hat{s}$ and flags *lb*, *lc*. $R^{\mathbf{S}}$ contains $\mathbf{O}(|S|)$ transitions $t^{\mathbf{S}}$ per transition $t \in R$ in subsets (1), (3), and (5), and $\mathbf{O}(1)$ $t^{\mathbf{S}}$ per $t \in R$ in subsets (2) and (4). □

As reachability in a Kripke structure can be determined in $\mathbf{O}(|S| + |R|)$ time and $\mathbf{O}(|S|)$ space, $S^{\mathbf{S}}$ can be checked in $\mathbf{O}(|S|^2 + |S||R|)$ time and $\mathbf{O}(|S|^2)$ space. For results on other parameters that are important when using BDD-based symbolic model checking, e.g., radius, diameter, or BDD size, see [19].

4 Regular Model Checking

4.1 Preliminaries

The notation in this section is mostly borrowed from [7]. Let Σ be a finite alphabet. Regular sets (respectively relations) can be represented as finite-state automata (resp. transducers). These are given as four tuple (Q, q_0, δ, F)

Definition 4.1 Let $\mathcal{P} = (\Sigma, \Phi_I, R)$ be a program. Then $\mathcal{P}^{\mathbf{S}} = (\Sigma^{\mathbf{S}}, {\Phi_I}^{\mathbf{S}}, R^{\mathbf{S}})$ is defined as

$$
\begin{aligned}
\Sigma^{\mathbf{S}} &= \mathbb{B} \cup (\Sigma \times \Sigma) \\
{\Phi_I}^{\mathbf{S}} &= \{0\} \circ \{0\} \circ \{\mu(w, \hat{w}) \in (\Sigma \times \Sigma)^* \mid |w| = |\hat{w}| \wedge w \in \Phi_I\} \cup \\
&\quad \{1\} \circ \{0\} \circ \{\mu(w, w) \in (\Sigma \times \Sigma)^* \mid w \in \Phi_I\} \\
R^{\mathbf{S}} &= \{((lb\ lc\ \mu(w, \hat{w})), (lb'\ lc'\ \mu(w', \hat{w}'))) \subseteq (\mathbb{B} \circ \mathbb{B} \circ (\Sigma \times \Sigma)^*)^2 \mid \\
&\quad |w| = |\hat{w}| = |w'| = |\hat{w}'| \wedge (w, w') \in R \wedge \\
&\quad ((\neg lb \wedge \neg lb' \wedge \neg lc \wedge \neg lc' \wedge \hat{w} = \hat{w}') \vee && (1) \\
&\quad\ (\neg lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge w' = \hat{w}') \vee && (2) \\
&\quad\ (lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge \hat{w} = \hat{w}') \vee && (3) \\
&\quad\ (lb \wedge lb' \wedge \neg lc \wedge lc' \wedge \hat{w} = w' = \hat{w}') \vee && (4) \\
&\quad\ (lb \wedge lb' \wedge lc \wedge lc' \wedge \hat{w} = \hat{w}'))\} && (5)
\end{aligned}
$$

where Q is a finite set of states, q_0 is the initial state, $\delta : (Q \times \Sigma) \mapsto 2^Q$ (resp. $\delta : (Q \times (\Sigma \times \Sigma)) \mapsto 2^Q$) is the transition function, and $F \subseteq Q$ is the set of accepting states.

A relation $R \subseteq \Sigma^* \times \Sigma^*$ is *length-preserving* iff $\forall (w, w') \in R\ .\ |w| = |w'|$. A *program* is a triple $\mathcal{P} = (\Sigma, \Phi_I, R)$ where $\Phi_I \subseteq \Sigma^*$ is a regular set of *initial configurations* and $R \subseteq \Sigma^* \times \Sigma^*$ is a regular, length-preserving *transition relation*.

A *configuration* of a program $\mathcal{P}$ is a word w over Σ. *Runs* are finite or infinite sequences of configurations $\rho = \rho(0)\rho(1)\ldots$, such that $\forall 0 \leq i < |\rho|\ .\ (\rho(i), \rho(i+1)) \in R$. A run is *initialized* if $\rho(0) \in \Phi_I$. $Runs(\mathcal{P})$ is the set of runs of $\mathcal{P}$.

4.2 Reduction

In the finite case the state to be saved was simply added as a separate component to the state of the transformed system. A finite automaton can only remember a finite amount of information. Hence, in order to apply the reduction to regular model checking it is not possible to construct an automaton that first reads a state of the original program and compares that with a saved copy. Instead, we extend the alphabet of the program to tuples of letters to store and compare states position by position of a word. Other than that the construction in Def. 4.1 is exactly the same as in the finite case.

Lemma 4.2 *If $\mathcal{P} = (\Sigma, \Phi_I, R)$ is a program, so is $\mathcal{P}^{\mathbf{S}} = (\Sigma^{\mathbf{S}}, {\Phi_I}^{\mathbf{S}}, R^{\mathbf{S}})$.*

Proof. Assume that Φ_I is given by $(Q_I, q_{0I}, \delta_I, F_I)$. To represent an automaton (not) saving the initial state we use separate copies of $(Q_I, q_{0I}, \delta_I, F_I)$, $(Q_I^{\neq}, q_{0I}^{\neq}, \delta_I^{\neq}, F_I^{\neq})$ and $(Q_I^{=}, q_{0I}^{=}, \delta_I^{=}, f_I^{=})$. Then $({Q_I}^{\mathbf{S}}, {q_{0I}}^{\mathbf{S}}, {\delta_I}^{\mathbf{S}}, {F_I}^{\mathbf{S}})$ with

$$
\begin{aligned}
{Q_I}^{\mathbf{S}} &= Q_I^{\neq} \cup Q_I^{=} \cup \{q_{lb}, q_{lc}^{\neq}, q_{lc}^{=}\},\\
{q_{0I}}^{\mathbf{S}} &= q_{lb},\\
{\delta_I}^{\mathbf{S}} &= \{(q_{lb}, 0, q_{lc}^{\neq}), (q_{lc}^{\neq}, 0, q_{0I}^{\neq})\} \cup \{(q^{\neq}, (a, \hat{a}), q^{\neq\prime}) \mid (q^{\neq}, a, q^{\neq\prime}) \in \delta_I^{\neq}\} \cup\\
&\quad \{(q_{lb}, 1, q_{lc}^{=}), (q_{lc}^{=}, 0, q_{0I}^{=})\} \cup \{(q^{=}, (a, a), q^{=\prime}) \mid (q^{=}, a, q^{=\prime}) \in \delta_I^{=}\}, \text{ and}\\
{F_I}^{\mathbf{S}} &= F_I^{\neq} \cup F_I^{=},
\end{aligned}
$$

is a finite automaton accepting ${\Phi_I}^{\mathbf{S}}$.

Similarly, if R is given by $(Q_R, q_{0R}, \delta_R, F_R)$, we construct a finite transducer $({Q_R}^{\mathbf{S}}, {q_{0R}}^{\mathbf{S}}, {\delta_R}^{\mathbf{S}}, {F_R}^{\mathbf{S}})$ to accept $R^{\mathbf{S}}$. We use separate copies of $(Q_R, q_{0R}, \delta_R, F_R)$ to leave the saved word unchanged (superscript ^{135}, corresponding to disjuncts 1, 3, and 5 in Def. 4.1), save a word (sup. ^{2}, corr. to subset (2)), and compare current and stored word (sup. ^{4}, corr. to subset (4)).

$$
\begin{aligned}
{Q_R}^{\mathbf{S}} &= Q_R^{135} \cup Q_R^{2} \cup Q_R^{4} \cup \{q_{lb}, q_{lc}^{1}, q_{lc}^{2}, q_{lc}^{345}\},\\
{q_{0R}}^{\mathbf{S}} &= q_{lb},\\
{\delta_R}^{\mathbf{S}} &= \{(q_{lb}, (0,0), q_{lc}^{1}), (q_{lb}, (0,1), q_{lc}^{2}), (q_{lb}, (1,1), q_{lc}^{345})\} \cup\\
&\quad \{(q_{lc}^{1}, (0,0), q_0^{135}), (q_{lc}^{2}, (0,0), q_0^{2}), (q_{lc}^{345}, (0,0), q_0^{135}),\\
&\quad\ (q_{lc}^{345}, (0,1), q_0^{4}), (q_{lc}^{345}, (1,1), q_0^{135})\} \cup\\
&\quad \{(q^{135}, ((a, \hat{a}), (a', \hat{a})), q^{135\prime}) \mid (q^{135}, (a, a'), q^{135\prime}) \in \delta_R^{135}\} \cup\\
&\quad \{(q^{2}, ((a, \hat{a}), (a', a')), q^{2\prime}) \mid (q^{2}, (a, a'), q^{2\prime}) \in \delta_R^{2}\} \cup\\
&\quad \{(q^{4}, ((a, a'), (a', a')), q^{4\prime}) \mid (q^{4}, (a, a'), q^{4\prime}) \in \delta_R^{4}\}, \text{ and}\\
{F_R}^{\mathbf{S}} &= F_R^{135} \cup F_R^{2} \cup F_R^{4}
\end{aligned}
$$

□

Theorem 4.3 *Let $\mathcal{P} = (\Sigma, \Phi_I, R)$ be a program, $\mathcal{P}^{\mathbf{S}}$ be defined as above, and $\hat{w}_I \in \Sigma^*$ with $|\hat{w}_I| = |w_0|$. Assume $k > l \geq 0$.*

$$
\begin{gathered}
(w_0 \dots w_{l-1})(w_l \dots w_{k-1})^{\omega} \in \mathit{Runs}(\mathcal{P})\\
\Leftrightarrow\\
(0\ 0\ \mu(w_0, \hat{w}_I)) \dots (0\ 0\ \mu(w_{l-1}, \hat{w}_I))(1\ 0\ \mu(w_l, w_l)) \dots\\
\dots (1\ 0\ \mu(w_{k-1}, w_l))(1\ 1\ \mu(w_k, w_l)) \ \in \mathit{Runs}(\mathcal{P}^{\mathbf{S}})
\end{gathered}
$$

Proof. Analogous to the proof of Thm. 3.2. □

Remark 4.4 Bouajjani et al. developed a technique to compute the transitive closure of a regular relation R [7,14]. A sufficient criterion for termination of that computation is *bounded local depth* [7,14] of R. Our construction preserves that property. Intuitively, a relation has local depth k if for any $(w, w') \in R^+$ each position in w needs to be rewritten no more than k times. Note that in any run $\rho^{\mathbf{S}}$ of $\mathcal{P}^{\mathbf{S}}$ the projection of $\rho^{\mathbf{S}}$ onto (lb, lc) will be a prefix of $(0,0)^*\ (1,0)^+\ (1,1)^+$. Furthermore, $\hat{w}$ changes its value in $\rho^{\mathbf{S}}$ at most once

at the transition of (lb, lc) from $(0, 0)$ to $(1, 0)$. Hence, with similar reasoning as for radius and diameter in [19] we can infer that, if R has local depth k, $R^{\mathbf{S}}$ has local depth $\leq 3k + 2$.

Remark 4.5 The ideas of regular model checking have been extended to infinite words [5] by regarding the finite automata used to represent sets of states and the transition relation as Büchi automata on infinite words. The techniques of [5] require the Büchi automata to be *weakly deterministic*. A Büchi automaton is weak (1) if each of its strongly connected components contains either only accepting or only non-accepting states and (2) if the set of states can be partitioned into an ordered set of subsets such that each path in the automaton progresses in descending order through these subsets. From the proof of Lemma 4.2 it's easy to see that, if B is a weakly deterministic Büchi automaton (for the set of initial configurations) or transducer (for the transition relation), so is $B^{\mathbf{S}}$. Clearly, repeated reachability may not be sufficient to verify general LTL properties for $\omega-$regular programs.

5 Pushdown Systems

5.1 Preliminaries

Notation in this section is along the lines of [11]. A *pushdown system* M is a four tuple $M = (P, \Gamma, \Delta, C_I)$ where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*, and $C_I \subseteq P \times \Gamma$ is a finite set of *initial configurations*.

A *configuration* is a pair $\langle p, w \rangle$ with $p \in P$ and $w \in \Gamma^*$. A *run* is a (finite or infinite) sequence of configurations $\rho = \rho(0)\rho(1)\ldots$, where $\rho(i) = \langle p_i, w_i \rangle$, such that $\forall i < |\rho| - 1 \;.\; \exists \gamma_i \in \Gamma, \exists u_i, v_i \in \Gamma^* \;.\; w_i = \gamma_i v_i \wedge w_{i+1} = u_i v_i \wedge ((p_i, \gamma_i), (p_{i+1}, u_i)) \in \Delta$. A run is *initialized* if $\rho(0) \in C_I$. $Runs(M)$ is the set of runs of M.

A *head* is a pair $\langle p, \gamma \rangle$ with $p \in P$ and $\gamma \in \Gamma$. If $c = \langle p, \gamma w \rangle$ is a configuration, $head(c) = \langle p, \gamma \rangle$. A head $\langle p, \gamma \rangle$ is *repeating* if there exist a run ρ in M and $w \in \Gamma^*$ such that $|\rho| > 1$, $\rho(0) = \langle p, \gamma \rangle$, and $\rho(|\rho| - 1) = \langle p, \gamma w \rangle$. $heads(\rho)$ denotes the sequence of heads derived from a run ρ.

Bouajjani et al. proved [6] that (1) every run that ends in a configuration with a repeating head can be extended to an infinite run, and (2) from every infinite run ρ a run $\sigma\tau$ can be derived such that $|\sigma| < \infty$ and $heads(\tau) = (\langle p_0, \gamma_0 \rangle \ldots \langle p_{l-1}, \gamma_{l-1} \rangle)^\omega$. I.e., if there exists an infinite run in M, then there also exists one whose sequence of heads forms a lasso.

5.2 Reduction

Based on the results of [6] it is sufficient to find repeating heads when checking LTL formulae on pushdown systems. Hence, a reduction of repeated reachability to reachability need only store and watch out for a second occurrence

of a repeating head $\langle p, \gamma \rangle$ rather than an entire configuration. However, to infer from the second occurrence of a head that this head is indeed repeating, one has to ensure that the stack height between the first and the second occurrence never fell below the stack height at the first occurrence. To this end the stack alphabet is extended such that each stack symbol has an additional flag *bs* (*b*ottom of *s*tack) to remember a given stack height. When saving a head this flag is set for the bottom element pushed on the stack in the post-configuration. Whenever an element with $bs = 1$ is removed from the stack without being replaced in the same transition a *l*oop *e*rror flag *le* is set.

In the previous examples, *lc* signals a second occurrence of a state immediately at that occurrence. However, the definition of the transition rules for pushdown systems may not give access to the topmost element of the stack in the post-configuration. If no new element is pushed on the stack a comparison with a stored stack element cannot be performed. For this reason we introduce a one-state delay in the case of pushdown systems for *lb*, *lc*, and the stored head. Hence, there is no need for an initial configuration with that configuration already saved.

Definition 5.1 shows the entire reduction. The transition relation is partitioned into 5 sets again. While no state has been saved (subset (1)), flags *lb*, *lc*, and *le* remain false, the initial values for $\hat{p}$ and $\hat{\gamma}$ are just copied, and no stack height need be remembered (bs_0 is false). Saving a state (subset (2)) can only occur if a non-empty word is pushed back on the stack — otherwise, the next transition would immediately violate the above-mentioned condition for the stack height of a repeating head. Taking a transition from subset (2) saves the head (p, γ) (in the pre-configuration) in $\hat{p}$ and $\hat{\gamma}$ (in the post-configuration), sets *lb* to true, and marks the current stack height by setting *bs* to true for the bottom element pushed on the stack. Transitions from subset (3) are taken while a second occurrence of the stored head has not been seen, hence, the flags *lb*, *lc*, as well as $\hat{p}$ and $\hat{\gamma}$ keep their values. In addition, the condition not to fall below the stack height at the time of saving is checked. When this is the case, i.e., when an element with *bs* true is popped from the stack and only an empty word is pushed back, the loop error flag *le* is set to true. This prevents signalling a repeating head when a second occurrence of the stored head could be detected in the future by restricting subsequent transitions to subset (3). When the stack height remains above the required level, *le* keeps its value and the flag *bs* is set in the bottom element of the word pushed onto the stack iff it was set in the symbol popped from the stack. A second occurrence of (p, γ) is signalled by setting *lc* to true when taking a transition from subset (4). *lb*, *le*, $\hat{p}$, and $\hat{\gamma}$ keep their values. Any remembered stack height is discarded. Transitions of the last subset (5) keep all additional location components constant.

In the following we prove correctness of the reduction.

Theorem 5.2 *Let $M = (P, \Gamma, \Delta, c_I)$ be a pushdown system and $M^{\mathbf{S}}$ be defined as above. There exists an initialized run ρ to a repeating head $\langle p_0, \gamma \rangle$*

Definition 5.1 Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system, let $(\hat{p}_I, \hat{\gamma}_I) \in P \times \Gamma$ be arbitrary but fixed. Then, $M^{\mathbf{S}} = (P^{\mathbf{S}}, \Gamma^{\mathbf{S}}, \Delta^{\mathbf{S}}, {C_I}^{\mathbf{S}})$ is defined as

$$
\begin{aligned}
P^{\mathbf{S}} &= P \times P \times \Gamma \times \mathbb{B}^3 \\
\Gamma^{\mathbf{S}} &= \Gamma \times \mathbb{B} \\
\Delta^{\mathbf{S}} &= \{(((p, \hat{p}, \hat{\gamma}, lb, lc, le), (\gamma, bs)), ((p', \hat{p}', \hat{\gamma}', lb', lc', le'), \mu(w', bs'_h \dots bs'_0))) \mid \\
&\quad (((p, \gamma), (p', w')) \in \Delta) \wedge (|w'| > 1 \rightarrow \neg bs'_h \wedge \dots \wedge \neg bs'_1) \wedge \\
&\quad ((\neg lb \wedge \neg lb' \wedge \neg lc \wedge \neg lc' \wedge \neg le \wedge \neg le' \wedge \hat{p} = \hat{p}' \wedge \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow \neg bs'_0)) \vee && (1) \\
&\quad (\neg lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge \neg le \wedge \neg le' \wedge p = \hat{p}' \wedge \gamma = \hat{\gamma}' \wedge (|w'| > 0) \wedge bs'_0) \vee && (2) \\
&\quad (lb \wedge lb' \wedge \neg lc \wedge \neg lc' \wedge ((|w'| = 0 \wedge bs \vee le) \leftrightarrow le') \wedge \\
&\qquad \hat{p} = \hat{p}' \wedge \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow (bs \leftrightarrow bs'_0))) \vee && (3) \\
&\quad (lb \wedge lb' \wedge \neg lc \wedge lc' \wedge \neg le \wedge \neg le' \wedge p = \hat{p} = \hat{p}' \wedge \gamma = \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow \neg bs'_0)) \vee && (4) \\
&\quad (lb \wedge lb' \wedge lc \wedge lc' \wedge \neg le \wedge \neg le' \wedge \hat{p} = \hat{p}' \wedge \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow \neg bs'_0)))\} && (5) \\
{C_I}^{\mathbf{S}} &= \{\langle (p_I, \hat{p}_I, \hat{\gamma}_I, 0, 0, 0), (\gamma_I, 0) \rangle \mid \langle p_I, \gamma_I \rangle \in C_I\}
\end{aligned}
$$

in M if and only if there exists an initialized run $\rho^{\mathbf{S}}$ in $M^{\mathbf{S}}$ with $\rho^{\mathbf{S}}(|\rho^{\mathbf{S}}| - 2) = \langle (p_0, p_0, \gamma, 1, 0, 0), w_{|\rho^{\mathbf{S}}|-2} \rangle$, where $w_{|\rho^{\mathbf{S}}|-2}(0) = \gamma$, and $\rho^{\mathbf{S}}(|\rho^{\mathbf{S}}| - 1) = \langle (p, p_0, \gamma, 1, 1, 0), w_{|\rho^{\mathbf{S}}|-1} \rangle$.

Proof. "⇒": Assume a run ρ to a repeatable head $\langle p_0, \gamma \rangle$. Hence, there exist $l \geq 0$, $q_0, \dots, q_{l-1} \in P$, $w_0, \dots, w_{l-1} \in \Gamma^*$, $v \in \Gamma^*$ where $\forall i < l \,.\, \rho(i) = \langle q_i, w_i \rangle$ and $\rho(l) = \langle p_0, \gamma v \rangle$.

By the definition of a repeating head there are $k > l$, $p_1, \dots, p_{k-l-1} \in P$, $u_0, \dots, u_{k-l} \in \Gamma^+$, where $u_0 = u_{k-l}(0) = \gamma$, such that ρ can be extended to an infinite run $\rho^\infty \in Runs(M)$:

$$
\begin{aligned}
&\forall i < l \,.\, \rho^\infty(i) = \rho(i) \\
&\forall i \geq l \,.\, \rho^\infty(i) = \langle p_{(i-l) \bmod (k-l)}, \\
&\qquad\qquad u_{(i-l) \bmod (k-l)}(u_{k-l}(1) \dots u_{k-l}(|u_{k-l}| - 1))^{(i-l) \operatorname{div} (k-l)} v \rangle
\end{aligned}
$$

From that we construct a finite run $\rho^{\mathbf{S}}$ as follows:

$$
\begin{aligned}
&\forall i < l \,.\, \rho^{\mathbf{S}}(i) = \langle (q_i, \hat{p}_I, \hat{\gamma}_I, 0, 0, 0), \mu(w_i, 0^{|w_i|}) \rangle \\
&\rho^{\mathbf{S}}(l) = \langle (p_0, \hat{p}_I, \hat{\gamma}_I, 0, 0, 0), (\gamma, 0)\ \mu(v, 0^{|v|}) \rangle \\
&\rho^{\mathbf{S}}(l+1) = \langle (p_1, p_0, \gamma, 1, 0, 0), \mu(u_1, 0^{|u_1|-1}1)\ \mu(v, 0^{|v|}) \rangle \\
&\forall l+1 < i < l+k \,.\, \rho^{\mathbf{S}}(i) = \langle (p_{i-l}, p_0, \gamma, 1, 0, 0), \mu(u_{i-l}, 0^{|u_{i-l}|-1}1)\ \mu(v, 0^{|v|}) \rangle
\end{aligned}
$$

if $|u_{k-l}| > 1$

$$
\begin{aligned}
&\rho^{\mathbf{S}}(k) = \langle (p_0, p_0, \gamma, 1, 0, 0), (\gamma, 0)\ \mu(u_{k-l}(1) \dots u_{k-l}(|u_{k-l}| - 1), 0^{|u_{k-l}|-2}\ 1)\ \mu(v, 0^{|v|}) \rangle \\
&\rho^{\mathbf{S}}(k+1) = \langle (p_1, p_0, \gamma, 1, 1, 0), \\
&\qquad\qquad \mu(u_1, 0^{|u_1|})\ \mu(u_{k-l}(1) \dots u_{k-l}(|u_{k-l}| - 1), 0^{|u_{k-l}|-2}\ 1)\ \mu(v, 0^{|v|}) \rangle
\end{aligned}
$$

otherwise

$$
\begin{aligned}
&\rho^{\mathbf{S}}(k) = \langle (p_0, p_0, \gamma, 1, 0, 0), (\gamma, 1) \mu(v, 0^{|v|}) \rangle \\
&\rho^{\mathbf{S}}(k+1) = \langle (p_1, p_0, \gamma, 1, 1, 0), \mu(u_1, 0^{|u_1|})\ \mu(v, 0^{|v|}) \rangle
\end{aligned}
$$

"⇐": Assume an initialized run $\rho^{\mathbf{S}}$ to $\rho^{\mathbf{S}}(|\rho^{\mathbf{S}}|-2) = \langle (p_0, p_0, \gamma, 1, 0, 0), w_{|\rho^{\mathbf{S}}|-2} \rangle$, where $w_{|\rho^{\mathbf{S}}|-2}(0) = \gamma$, and $\rho^{\mathbf{S}}(|\rho^{\mathbf{S}}| - 1) = \langle (p_1, p_0, \gamma, 1, 1, 0), w_{|\rho^{\mathbf{S}}|-1} \rangle$. By Def. 5.1, $\exists 0 < l < |\rho^{\mathbf{S}}| - 2$ such that $\rho^{\mathbf{S}}(l) = \langle (p_0, \hat{p}_I, \hat{\gamma}_I, 0, 0, 0), \mu(w_l, 0^{|w_l|}) \rangle$ and $w_l(0) = \gamma$. Clearly, the projection of $\rho^{\mathbf{S}}(0 \ldots l)$ on the first components of its state and stack is a run in M to a repeatable head. □

5.3 Complexity

Proposition 5.3 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system. $M^{\mathbf{S}}$ has $\mathbf{O}(|P||\Gamma||P|)$ locations and $\mathbf{O}(|P||\Gamma||\Delta|)$ transition rules.*

Proof. The locations of M are extended in $M^{\mathbf{S}}$ to store another location, a stack symbol, and three flags. For $\Delta^{\mathbf{S}}$, there are $\mathbf{O}(|\Delta|)$ transition rules in subsets (1), (2), and (4), and $\mathbf{O}(|P||\Gamma||\Delta|)$ in (3) and (5). □

Algorithm 3 in [11] can be used to check reachability for a pushdown system $M = (P, \Gamma, \Delta, C_I)$ where $(p, \gamma, p', w') \in \Delta \Rightarrow |w'| \leq 2$. It computes the set of reachable configurations in $\mathbf{O}(|P||\Delta|^2 + |\delta|)$ time and space.

Proposition 5.4 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system such that $(p, \gamma, p', w') \in \Delta \Rightarrow |w'| \leq 2$. Algorithm 3 in [11] runs on $M^{\mathbf{S}}$, with $A_{M^{\mathbf{S}}}$ accepting ${C_I}^{\mathbf{S}}$, in time and space*

$$\mathbf{O}(|P||\Gamma|(|P||\Delta|^2) + |\delta|)$$

Proof. See the full version of this paper. □

6 Timed Automata

In this section we briefly give the idea of how to apply our reduction to timed automata [3]. Details can be found in the full version of this paper. In addition to a finite set of control locations, timed automata have a finite set of real-valued clocks. Transitions are labeled with integer clock constraints of the form $c \sim n$ where c is a clock variable, $\sim \in \{<, \leq, =, \geq, >\}$, and $n \in \mathbb{N}$.

Alur and Dill showed [3] that for model checking of LTL the precise value of the clocks is not relevant. Rather, clock valuations fall into a finite number of equivalence classes called *regions*. Model checking is then performed on the abstract *region automaton*.

We use this fact in our reduction as follows. We do not store the precise valuation of the clocks but the clock region. This requires a variable in the range $\{0, \ldots, c_x\}$ and a flag for each clock x, where c_x is the maximal integer x is compared with in a clock constraint. Furthermore, we store the order of the fractional parts of the clocks. This requires k variables of range $0 \ldots k-1$ if there are k clocks and $k - 1$ flags to indicate equality between each pair of successors in the order.

7 Conclusion

We have extended our reduction of repeated reachability to reachability to some popular classes of infinite state systems. For these classes the reductions "pull the original algorithm into the model". To explore the limits of our method we are looking for systems where liveness can still be reduced to repeated reachability, but where our method might not seem applicable. It is clear that the construction for the finite case can not always be lifted to infinite state systems. In general, counterexamples to liveness properties in infinite state systems can not necessarily be restricted to have lasso shape. In some cases, abstractions [18] or simulations [8] might help. Maybe our method can also provide additional insight why liveness is undecidable for some classes of systems. Finally, experiments need to prove the viability of our approach.

References

[1] Abdulla, P., B. Jonsson, M. Nilsson and J. d'Orso, *Algorithmic improvements in regular model checking*, in: W. Hunt and F. Somenzi, editors, *CAV'03*, LNCS **2725** (2003), pp. 236–248.

[2] Abdulla, P., B. Jonsson, M. Nilsson, J. d'Orso and M. Saksena, *Regular model checking for LTL(MSO)*, in: R. Alur and D. Peled, editors, *CAV'04*, LNCS **3114** (2004), pp. 348–360.

[3] Alur, R. and D. Dill, *A theory of timed automata*, Theor. Comput. Sci. **126** (1994), pp. 183–235.

[4] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: R. Cleaveland, editor, *TACAS'99*, LNCS **1579** (1999), pp. 193–207.

[5] Boigelot, B., A. Legay and P. Wolper, *Omega-regular model checking*, in: K. Jensen and A. Podelski, editors, *TACAS'04*, LNCS **2988** (2004), pp. 561–575.

[6] Bouajjani, A., J. Esparza and O. Maler, *Reachability analysis of pushdown automata: Application to model-checking*, in: A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR'97*, LNCS **1243** (1997), pp. 135–150.

[7] Bouajjani, A., B. Jonsson, M. Nilsson and T. Touili, *Regular model checking*, in: Emerson and Sistla [10], pp. 403–418.

[8] Bouajjani, A., A. Legay and P. Wolper, *Handling liveness properties in ($\omega-$)regular model checking*, in: *INFINITY'04*, 2004.

[9] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[10] Emerson, E. and A. Sistla, editors, "Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings," LNCS **1855**, Springer, 2000.

[11] Esparza, J., D. Hansel, P. Rossmanith and S. Schwoon, *Efficient algorithms for model checking pushdown systems*, in: Emerson and Sistla [10], pp. 232–247.

[12] Esparza, J. and S. Schwoon, *A BDD-based model checker for recursive programs*, in: G. Berry, H. Comon and A. Finkel, editors, *CAV'01*, LNCS **2102** (2001), pp. 324–336.

[13] Finkel, A., B. Willems and P. Wolper, *A direct symbolic approach to model checking pushdown systems (extended abstract)*, in: F. Moller, editor, *INFINITY'97*, ENTCS **9** (1997).

[14] Jonsson, B. and M. Nilsson, *Transitive closures of regular relations for verifying infinite-state systems*, in: S. Graf and M. Schwartzbach, editors, *TACAS'00*, LNCS **1785** (2000), pp. 220–234.

[15] Kesten, Y., O. Maler, M. Marcus, A. Pnueli and E. Shahar, *Symbolic model checking with rich assertional languages.*, Theor. Comput. Sci. **256** (2001), pp. 93–112.

[16] Kupferman, O. and M. Vardi, *Model checking of safety properties*, in: N. Halbwachs and D. Peled, editors, *CAV'99*, LNCS **1633** (1999), pp. 172–183.

[17] Larsen, K., P. Pettersson and W. Yi, Uppaal *in a Nutshell*, International Journal on Software Tools for Technology Transfer (STTT) **1** (1997), pp. 134–152.

[18] Pnueli, A. and E. Shahar, *Liveness and acceleration in parameterized verification*, in: Emerson and Sistla [10], pp. 328–343.

[19] Schuppan, V. and A. Biere, *Efficient reduction of finite state model checking to reachability analysis*, International Journal on Software Tools for Technology Transfer (STTT) **5** (2004), pp. 185–204.

[20] Schuppan, V. and A. Biere, *Shortest counterexamples for symbolic model checking of LTL with past*, in: N. Halbwachs and L. Zuck, editors, *TACAS'05*, LNCS **3440** (2005), pp. 493–509.

[21] Sebastiani, R., S. Tonetta and M. Vardi, *Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking*, in: *CAV'05*, 2005, to appear.

[22] Vardi, M. and P. Wolper, *An automata-theoretic approach to automatic program verification*, in: *LICS'86* (1986), pp. 332–344.

[23] Wolper, P. and B. Boigelot, *Verifying systems with infinite but regular state spaces*, in: A. Hu and M. Vardi, editors, *CAV'98*, LNCS **1427** (1998), pp. 88–97.

Recent BRICS Notes Series Publications

NS-05-4 **Scott A. Smolka and Jiří Srba, editors.** ***Preliminary Proceedings of the 7th International Workshop on Verification of Infinite-State Systems, INFINITY '05,*** **(San Francisco, USA, August 27, 2005), June 2005. vi+64 pp.**

NS-05-3 **Luca Aceto and Andrew D. Gordon, editors.** ***Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond, PA '05,*** **(Bertinoro, Forlì, Italy, August 1–5, 2005), June 2005. vi+235 pp.**

NS-05-2 **Luca Aceto and Willem Jan Fokkink.** ***The Quest for Equational Axiomatizations of Parallel Composition: Status and Open Problems.*** **May 2005. 7 pp. To appear in a volume of the BRICS Notes Series devoted to the workshop "Algebraic Process Calculi: The First Twenty Five Years and Beyond", August 1–5, 2005, University of Bologna Residential Center Bertinoro (Forlì), Italy.**

NS-05-1 **Luca Aceto, Magnus Mar Halldorsson, and Anna Ingólfsdóttir.** ***What is Theoretical Computer Science?*** **April 2005. 13 pp.**

NS-04-2 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency and Distributed Computing, GETCO '04,*** **(Amsterdam, The Netherlands, October 4, 2004), September 2004. vi+80.**

NS-04-1 **Luca Aceto, Willem Jan Fokkink, and Irek Ulidowski, editors.** ***Preliminary Proceedings of the Workshop on Structural Operational Semantics, SOS '04,*** **(London, United Kingdom, August 30, 2004), August 2004. vi+56.**

NS-03-4 **Michael I. Schwartzbach, editor.** ***PLAN-X 2004 Informal Proceedings,*** **(Venice, Italy, 13 January, 2004), December 2003. ii+95.**

NS-03-3 **Luca Aceto, Zoltán Ésik, Willem Jan Fokkink, and Anna Ingólfsdóttir, editors.** ***Slide Reprints from the Workshop on Process Algebra: Open Problems and Future Directions, PA '03,*** **(Bologna, Italy, 21–25 July, 2003), November 2003. vi+138.**

NS-03-2 **Luca Aceto.** ***Some of My Favourite Results in Classic Process Algebra.*** **September 2003. 21 pp. Appears in the *Bulletin of the EATCS,* volume 81, pp. 89–108, October 2003.**