Parosh Aziz Abdulla
Igor Potapov (Eds.)

LNCS 8169

Reachability Problems

7th International Workshop, RP 2013
Uppsala, Sweden, September 2013
Proceedings

Lecture Notes in Computer Science 8169

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

Parosh Aziz Abdulla Igor Potapov (Eds.)

Reachability Problems

7th International Workshop, RP 2013
Uppsala, Sweden, September 24-26, 2013
Proceedings

Springer

Volume Editors

Parosh Aziz Abdulla
Uppsala University
Department of Information Technology
Box 337, 751 05 Uppsala, Sweden
E-mail: parosh@it.uu.se

Igor Potapov
University of Liverpool
Department of Computer Science
Ashton Building, Ashton Street, Liverpool L69 3BX, UK
E-mail: potapov@liverpool.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-41035-2 e-ISBN 978-3-642-41036-9
DOI 10.1007/978-3-642-41036-9
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013948100

CR Subject Classification (1998): F.3, D.2, F.2, D.3, F.4, F.4.1, F.1

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at the 7th International Workshop on Reachability Problems (RP 2013) held during September 24–26, 2013, at Norrlands nation in the heart of central Uppsala. The workshop was organized by the Department of Information Technology, Uppsala University, Sweden.

RP 2013 was the seventh in the series of workshops following six successful meetings at the University of Bordeaux, France, in 2012, the University of Genoa, Italy, in 2011, Masaryk University of Brno, Czech Republic, in 2010, Ecole Polytechnique, France, in 2009, at the University of Liverpool, UK, in 2008, and at Turku University, Finland, in 2007.

The workshop is specifically aimed at gathering together scholars from diverse disciplines and backgrounds interested in reachability problems that appear in algebraic structures, computational models, hybrid systems, logic, and verification.

Reachability is a fundamental problem that appears in several different contexts: finite- and infinite-state concurrent systems, computational models like cellular automata and Petri nets, decision procedures for classical, modal, and temporal logic, program analysis, discrete and continuous systems, time critical systems, hybrid systems, rewriting systems, algebraic structures (groups, semigroups and rings), deterministic or non-deterministic iterative maps, probabilistic and parametric systems, and open systems modelled as games.

Typically, for a fixed system description given in some form (rewriting rules, transformations by computable functions, systems of equations, logical formulas, etc.) a reachability problem consists in checking whether a given set of target states can be reached starting from a fixed set of initial states. The set of target states can be represented explicitly or via some implicit representation (e.g., a system of equations, a set of minimal elements with respect to some ordering on the states). Sophisticated quantitative and qualitative properties can often be reduced to basic reachability questions. Decidability and complexity boundaries, algorithmic solutions, and efficient heuristics are all important aspects to be considered in this context. Algorithmic solutions are often based on different combinations of exploration strategies, symbolic manipulations of sets of states, decomposition properties, reduction to linear programming problems, and they often benefit from approximations, abstractions, accelerations, and extrapolation heurisitics. Ad hoc solutions as well as solutions based on general-purpose constraint solvers and deduction engines are often combined in order to balance efficiency and flexibility.

The purpose of the conference is to promote exploration of new approaches for the predictability of computational processes by merging mathematical, algorithmic, and computational techniques. Topics of interest include (but are not limited to): reachability for infinite state systems; rewriting systems;

reachability analysis in counter/timed/cellular/communicating automata; Petri-nets; computational aspects of semigroups, groups, and rings; reachability in dynamical and hybrid systems; frontiers between decidable and undecidable reachability problems; complexity and decidability aspects; predictability in iterative maps and new computational paradigms.

All these aspects were discussed in the presentations of the seventh edition of the RP workshop. The proceedings of the previous editions of the workshop appeared in the following volumes:

Mika Hirvensalo, Vesa Halava, Igor Potapov, Jarkko Kari (Eds.): Proceedings of the Satellite Workshops of DLT 2007. TUCS General Publication No 45, June 2007. ISBN: 978-952-12-1921-4.

Vesa Halava and Igor Potapov (Eds.): Proceedings of the Second Workshop on Reachability Problems in Computational Models (RP 2008). Electronic Notes in Theoretical Computer Science. Volume 223, Pages 1-264 (26 December 2008).

Olivier Bournez and Igor Potapov (Eds.): Reachability Problems, Third International Workshop, RP 2009, Palaiseau, France, September 2325, 2009, Lecture Notes in Computer Science, 5797, Springer 2009.

Antonin Kucera and Igor Potapov (Eds.): Reachability Problems, 4th International Workshop, RP 2010, Brno, Czech Republic, August 2829, 2010, Lecture Notes in Computer Science, 6227, Springer 2010.

Giorgio Delzanno, Igor Potapov (Eds.): Reachability Problems, 5th International Workshop, RP 2011, Genoa, Italy, September 2830, 2011, Lecture Notes in Computer Science, 6945, Springer 2011.

Alain Finkel, Jerome Leroux, Igor Potapov (Eds.): Reachability Problems, 6th International Workshop, RP 2012, Bordeaux, France, September 17-19, 2012. Lecture Notes in Computer Science 7550, Springer 2012.

The five keynote speakers at the 2013 conference were:

- **Patricia Bouyer**, CNRS Cachan, "Robustness in Timed Automata"
- **Daniel Kroening**, Oxford University, "Automated Verification of Concurrent Software"
- **Rupak Majumdar**, MPI-SWS, "Provenance Verification"
- **Shaz Qadeer**, Microsoft Research Redmond, "Reachability Modulo Theories"
- **Thomas Schwentick**, TU Dortmund University, "The Dynamic Complexity of the Reachability Problem on Graphs"

There were 24 submissions. Each submission was reviewed by at least three Program Committee members. The full list of the members of the Program Committee and the list of external reviewers can be found on the next two

pages. The Program Committee is grateful for the highly appreciated and high-quality work produced by these external reviewers. Based on these reviews, the Program Committee decided to accept 14 papers, in addition to the five invited talks. The workshop also provided the opportunity to researchers to give informal presentations that are prepared very shortly before the event and inform the participants about current research and work in progress.

We gratefully acknowledge the organization team for their help and especially Mohamed Faouzi Atig for effective team management.

It is also a great pleasure to acknowledge the team of the EasyChair system, and the fine cooperation with the Lecture Notes in Computer Science team of Springer, which made the production of this volume possible in time for the conference. Finally, we thank all the authors for their high-quality contributions, and the participants for making this edition of RP 2013 a success.

September 2013

Parosh Aziz Abdulla
Igor Potapov

Organization

Program Committee

Parosh Aziz Abdulla	Uppsala University, Sweden
Rajeev Alur	University of Pennsylvania, USA
Mohamed Faouzi Atig	Uppsala University, Sweden
Bernard Boigelot	University of Liege, Belgium
Ahmed Bouajjani	LIAFA, University Paris Diderot, France
Krishnendu Chatterjee	Institute of Science and Technology (IST), Austria
Giorgio Delzanno	DIBRIS, Università di Genova, Italy
Javier Esparza	Technische Universität München, Germany
Alain Finkel	ENS Cachan, France
Pierre Ganty	IMDEA Software Institute, Spain
Kim Guldstrand Larsen	Aalborg University, Denmark
Jerome Leroux	CNRS, France
Richard Mayr	University of Edinburgh, UK
Markus Müller-Olm	Westfälische Wilhelms-Universität Münster, Germany
K. Narayan Kumar	Chennai Mathematical Institute, India
Andreas Podelski	University of Freiburg, Germany
Igor Potapov	University of Liverpool, UK
Jean-Francois Raskin	Université Libre de Bruxelles, Belgium
Ahmed Rezine	Linköping University, Sweden
James Worrell	Oxford University, UK
Hsu-Chun Yen	National Taiwan University, China
Gianluigi Zavattaro	Dep. Computer Science Bologna, Italy

Additional Reviewers

Bogomolov, Sergiy
Bournez, Olivier
Fijalkow, Nathanael
Geeraerts, Gilles
Haase, Christoph
Hoenicke, Jochen
Prabhakar, Pavithra
Saivasan, Prakash
Sassolas, Mathieu

Table of Contents

Robustness in Timed Automata*

Patricia Bouyer[1], Nicolas Markey[1], and Ocan Sankur[1,2]

[1] LSV, CNRS & ENS Cachan, France
[2] Université Libre de Bruxelles, Belgium
{bouyer,markey,sankur}@lsv.ens-cachan.fr

Abstract. In this paper we survey several approaches to the robustness of timed automata, that is, the ability of a system to resist to slight perturbations or errors. We will concentrate on robustness against timing errors which can be due to measuring errors, imprecise clocks, and unexpected runtime behaviors such as execution times that are longer or shorter than expected.

We consider the perturbation model of guard enlargement and formulate several robust verification problems that have been studied recently, including robustness analysis, robust implementation, and robust control.

1 Introduction

Timed automata are an extension of finite automata with analog clock variables, and a convenient formalism for modelling real-time systems [AD94]. The theory allows the model-checking against a rich set of properties, including temporal logics and their timed extensions. These algorithms have been implemented in several tools (*e.g.* Uppaal, IF2.0) and applied to many case studies.

The idea behind timed automata is to consider clocks with continuous values. The resulting abstraction is appealing in terms of modelling and allows for efficient symbolic algorithms. However, this formalism only allows validating designs under the assumptions that the clock variables are perfectly continuous, their values can be measured instantly and exactly, etc. Because concrete implementations cannot always be assumed to satisfy these assumptions, there is a need to study verification methodologies for timed automata where these idealistic assumptions are relaxed. A comparison of the semantics of timed automata and real-world systems is given in Table 1.

In this paper we survey several approaches to the robustness of timed automata. What we mean by robustness is the ability of a system to resist to slight perturbations or errors. We will concentrate on robustness against timing errors which can be due to measuring errors, imprecise clocks, and unexpected runtime behaviors such as execution times that are longer or shorter than expected. In particular, robustness in timed automata consists in relaxing the idealistic assumptions behind its semantics.

* Partly supported by ANR project ImpRo (ANR-10-BLAN-0317), by ERC Starting grants EQualIS (FP7-308087) and inVEST (FP7-279499), and by European project Cassting (FP7-601148).

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 1–18, 2013.

Table 1. A comparison between the abstract semantics of timed automata and real-world systems

	Timed automata	**Real-world system**
Frequency	Infinite	Finite
Precision	Arbitrary	Bounded
Synchronization	Perfect	Delayed

The benefits of studying robustness for timed systems are twofold. First, robustness is a desired property of real-time systems since it requires the system to tolerate errors upto a given bound. Hence, the properties proven "robustly" will hold in the system even when the environment assumptions change slightly. This property is crucial for critical systems.

Second, a "robust theory" of timed automata will reconcile abstract models and real-world systems, in the sense that the behaviors of the design would more closely correspond to the behaviors of a real system. As a consequence, the analysis made on the abstract models can be transferred to the real-world system. Robustness is thus closely related to implementability.

Robustness in timed automata was first considered in [GHJ97] where a topological semantics was defined with the idea of excluding isolated behaviors, but the emphasis was rather on obtaining the decidability of some hard verification problems. Timed automata with clock drifts were considered in [Pur98, Pur00], where algorithms for safety analysis in presence of clock drifts were given. This work triggered a series of results on robust model-checking timed automata with guard enlargement and clock drifts, e.g. [DDMR08]. See Section 2.4 for more on related work.

In this paper, we formulate several robustness problems in timed automata by considering the perturbation models of guard enlargement, which models time measurement errors and jitter. These perturbations can both be considered as syntactic transformations, or as reactive semantics which we model as games. In all cases, we consider an unknown parameter which expresses the magnitude of the perturbations. We formulate several robust verification problems that have been studied recently, including robustness analysis, robust implementation, and robust control.

2 Definition

2.1 Timed Automata

Given a finite set of clocks $\mathcal{C}$, we call *valuations* the elements of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$. For a subset $R \subseteq \mathcal{C}$ and a valuation ν, $\nu[R \leftarrow 0]$ is the valuation defined by $\nu[R \leftarrow 0](x) = \nu(x)$ for $x \in \mathcal{C} \setminus R$ and $\nu[R \leftarrow 0](x) = 0$ for $x \in R$. Given $d \in \mathbb{R}_{\geq 0}$ and a valuation ν, the valuation $\nu + d$ is defined by $(\nu + d)(x) = \nu(x) + d$ for all $x \in \mathcal{C}$. We extend these operations to sets of valuations in the obvious way. We write $\mathbf{0}$ for the valuation that assigns 0 to every clock.

An atomic clock constraint is a formula of the form $k \preceq x \preceq' l$ where $x \in \mathcal{C}$, $k, l \in \mathbb{Z} \cup \{-\infty, \infty\}$ and $\preceq, \preceq' \in \{<, \leq\}$. A *guard* is a conjunction of atomic clock constraints. A valuation ν satisfies a guard g, denoted $\nu \models g$, if all constraints are satisfied when each $x \in \mathcal{C}$ is replaced with $\nu(x)$. We write $\Phi_{\mathcal{C}}$ for the set of guards built on $\mathcal{C}$.

Definition 1. *A* timed automaton $\mathcal{A}$ *is a tuple* $(\mathcal{L}, \mathcal{C}, \ell_0, E)$*, where* $\mathcal{L}$ *is a finite set of locations,* $\mathcal{C}$ *is a finite set of clocks,* $E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times 2^{\mathcal{C}} \times \mathcal{L}$ *is a set of edges, and* $\ell_0 \in \mathcal{L}$ *is the initial location. An edge* $e = (\ell, g, R, \ell')$ *is also written as* $\ell \xrightarrow{g,R} \ell'$*.*

A closed timed automaton is a timed automaton whose all guards have closed atomic clock constraints.

A configuration of a timed automaton is a pair (ℓ, ν) where $\ell \in \mathcal{L}$ is the current location and ν is a clock valuation, assigning a value to each clock. At first sight it might look natural to assume that clocks take integer values, as timed automata will be used to model digital systems (using a cycle of the CPU as the unit of time). This would have several drawbacks: first, our results would only be valid for this given frequency of the CPU; more importantly, when the model consists of several components, a discrete semantics forces the components to have synchronoous transitions; finally, this would not be convenient for modelling the external environment, which should not be restricted to have finite frequency. We refer to [BS91, Alu91] for a longer discussion on this question.

In the rest of this paper, following [AD94], we thus assume that clocks take real values. The set of configurations (or states) of a timed automaton $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \ell_0, E)$ is then $\mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$. A *run* of $\mathcal{A}$ is a sequence $s_1 e_1 s_2 e_2 \ldots$ where $s_i \in \mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$, and writing $s_i = (\ell_i, \nu_i)$, either $e_i \in \mathbb{R}_{>0}$, in which case $s_{i+1} = (\ell_i, \nu_i + e_i)$, or $e_i = (\ell_i, g, R, \ell') \in E$, in which case $s_{i+1} = (\ell', \nu[R \leftarrow 0])$. We denote by $\mathsf{state}_i(\rho)$ the i-th state of any run ρ, by $\mathsf{first}(\rho)$ its first state, and, if ρ is finite, $\mathsf{last}(\rho)$ denotes its last state of ρ.

Example 1. Fig. 1 displays an example of a timed automaton, representing the behavior of a computer mouse: it uses one clock to measure the delays between pushes on the buttons, and translates them into single- or double-clicks, depending on these delays.

2.2 Timed-Automata-Based Model Checking

Theorem 1 ([AD94]). *Reachability and Büchi properties can be decided in polynomial space in timed automata. Both problems are PSPACE-complete.*

The proof of this important theorem relies on the construction of the *region abstraction*: intuitively, if two clock valuations are *close enough*, in the sense that they give to each clock the same integral part and define the same order w.r.t. their fractional parts, then they will give rise to similar behaviours: the same transitions are available, and the valuations reached by letting time elapse can still be made *close enough*. After noticing that we do not need to care

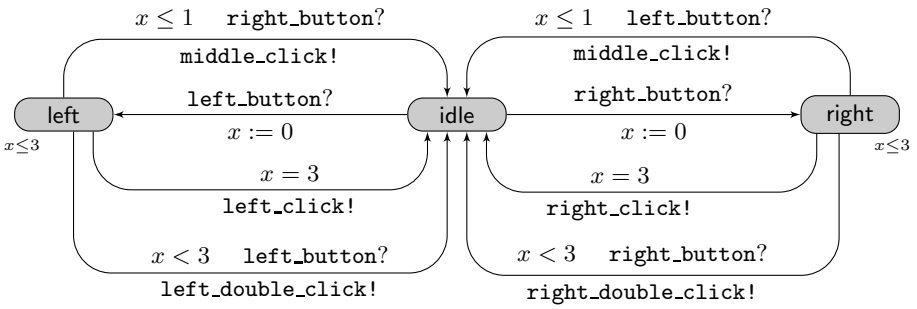


Fig. 1. *Clicks* and *double-clicks* of a mouse

about the value of a clock when it is larger than the maximal integer appearing in the automaton, this gives rise to an equivalence relation (called *region equivalence*) having finite index. Quotienting the automaton with the region equivalence provides us with the *region automaton*, which can be used to verify ω-regular properties. Though the region automaton has size exponential, the algorithm for checking ω-regular properties can be made to run on-the-fly, thus using only polynomial space.

Symbolic approaches based on coarser equivalence relations and adequate data structures have been developed, which make the verification of timed automata usable in practice. Several tools implement optimised versions of these algorithms (*e.g.* Uppaal [BDL+06], Kronos [BDM+98], ...), and have been successfully applied on industrial case studies.

Example 2. We illustrate the region equivalence on the two-clock timed automaton depicted on Fig. 2. The maximal constant appearing in this automaton is 2, so that the set of classes of the region-equivalence is as depicted on Fig. 3. Its region automaton is depicted on Fig. 4, in which we can observe that the rightmost location of the automaton is not reachable.

2.3 Discussion on the Semantics: Are We Doing the Right Job?

We opted for the dense-time semantics to overcome the drawbacks of discrete-time, but is dense-time *really* better?

Obviously, the continuous-time semantics can be used to mimick the discrete-time semantics (by forcing transitions to occur only at integer dates), which in some sense shows that it is not worse. But this semantics is mostly appropriate for abstract designs and high-level analysis of timed systems, and suffers from several inaccuracies when used for more concrete analyses. In particular:

– by assuming infinite frequency (*i.e.*, zero-delay transitions), infinite precision, and immediate communications between components, the continuous-time semantics is not adequate for implementation. Indeed, in practice, clocks do

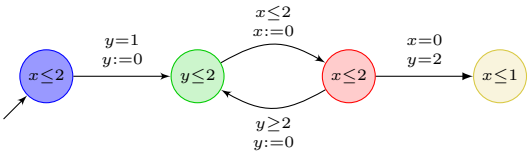


Fig. 2. A two-clock timed automaton $\mathcal{A}$

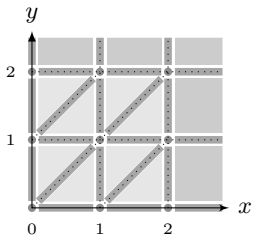


Fig. 3. Region equivalence for $\mathcal{A}$

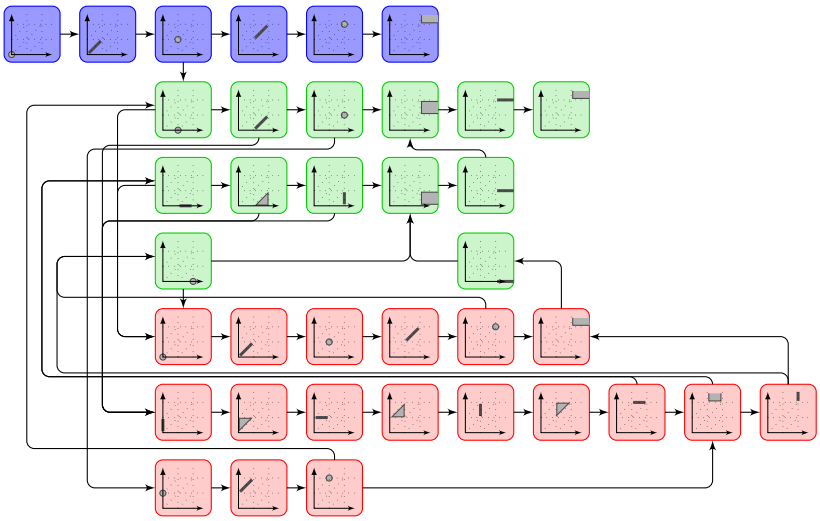


Fig. 4. The (reachable part of the) region automaton for $\mathcal{A}$

not all run at the exact same speed. Moreover, since computers are digital, the value of the clock is updated periodically. Hence a clock constraint might be evaluated to true at a time when it is not true anymore. Both phenomena give rise to emergent behaviors that are not taken into account during verification.

– because of the infinite frequency and precision, timed automata may exhibit non-realistic behaviors, especially convergence phenomena. The best-known example of convergence phenomena are *Zeno runs*, which are infinite runs (in terms of their number of transitions) having finite duration. More complex convergence phenomena can be hidden, which may be difficult to detect: such unrealistic behaviors can nevertheless be used by a standard verification process, witnessing in an inappropriate way the truth of some property.

Example 3. One realizes that any infinite behavior in the automaton of Fig. 2 is such that the value of clock x when entering the green location is non-decreasing and bounded by 2, hence converging (whereas globally time diverges).

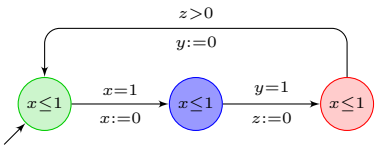


Fig. 5. An automaton with a (non-Zeno) convergence phenomenon

Another example (from [CHR02]) where such a phenomenon occurs is depicted at Fig. 5: because of the constraint on clock z, the time elapsed in the right-most state is always positive, and it can easily be proven that the value of x when entering the left-most state is the accumulated delay spent in the right-most state. Hence infinite runs exist in this automaton, but the delays in the right-most state will have to be arbitrarily small.

The remarks above raise several important questions on the real impact of standard verification processes applied to timed systems. First, given a model that has been proven to satisfy some properties, is it possible to implement it while preserving these properties? Also, to what extent does the result of the verification transfer to real-world systems?

This advocates for the development of a theory of robustness for timed systems. An important issue is to understand what is the real system behind the mathematical model, and also which implementation of a system we have in mind, if any. Also, it should be clear under which model of perturbations the theory applies (perturbations can arise from multiple sources). Finally, we should provide the user with methodologies to develop correct real-world (or implemented) systems.

The aim of this paper is to present a possible approach to the robustness of timed systems. It consists in developping refined techniques to check properties of timed automata in a robust setting: we do not want to model the execution platform explicitly, nor do we want to involve extra formalisms. Our algorithms will take timed automata as inputs, and they will verify properties under a semantics that takes timing imprecisions into account.

2.4 Related Work

An early attempt in defining a robustness notion for timed automata is [GHJ97], where a topological definition is introduced with the hope that the language inclusion problem could become decidable under this semantics; but the undecidability was later shown to hold even in the robust setting [HR00].

Robustness analysis algorithms in presence of parameterized guard enlargement have been studied extensively starting with safety properties in [DDMR08]. These results will be summarized in Section 3.

Following [Pur00, DDMR08], clock drifts have been studied for general timed automata (rather than closed timed automata) in [Dim07, SFK09]. These have also been considered in the context of distributed timed automata in [ABG+08] where one distinguishes existential languages, that is, those behaviors that can be achieved under some evolution of the clocks, and universal languages, which is, the behaviors that are present whatever the drifts are. The emptiness of the former is decidable, while the latter is shown undecidable.

The semantics under an unknown regular sampling of time was considered in [CHR02, KP05, AKY10]. Reachability under unknown sampling rate was shown to be undecidable in [CHR02], while safety is decidable [KP05]. Here, the goal is to synthesize a sampling parameter under which a reachability or safety objective holds, or the untimed language is preserved. This problem is related to the implementability of timed automata with discrete clocks. Such discrete time approaches are interesting when the model is a low-level designs, that is, timed automata where one time unit is comparable to the clock period in the target hardware platform.

The work [AT05] considers encoding in timed automata models several perturbations including clock drifts, enlargement, sampling. The resulting approach allows analyzing systems with fixed perturbation parameters, and often with a fixed granularity (e.g. for clock drifts), but not applying parameter synthesis. Many cases from the literature do in fact use such encodings since robustness algorithms are not yet available in verification tools.

The robustness of timed models against decreases in execution times have been considered in [ACS10] using simulations, and an application in a multimedia system was discussed.

3 Robustness Analysis

Standard analysis of the timed-automaton model might not be satisfactory if we are interested in the implementation of this model. In this section, we describe a framework in which one implements directly the system which is designed. Unfortunately, as mentioned earlier, the implemented system might not behave precisely according to the model, since imprecisions might occur while executing. The idea is then to understand how one should adapt the analysis process in order to capture and analyze the real behavior of the implemented system.

Robustness analysis. We write $\mathcal{A}_\delta$ for the timed automaton obtained from $\mathcal{A}$ by enlarging its guards by a parameter δ: that is, every upper-bounded constraint $x \leq b$ or $x < b$ is replaced by $x \leq b+\delta$, and every lower-bounded constraint $x \geq a$ or $x > a$ is replaced by $x \geq a-\delta$. Obviously every behavior in $\mathcal{A}$ is also a behavior in $\mathcal{A}_\delta$. Also quite obviously, there are behaviors in $\mathcal{A}_\delta$ that are absent in $\mathcal{A}$ (since some delays cannot be exactly matched), but maybe more surprisingly, there are *qualitative behaviors* that can be found in any $\mathcal{A}_\delta$ (however small $\delta > 0$ may be) but that cannot be found in $\mathcal{A}$. In particular, a simple (untimed safety) property proven in $\mathcal{A}$ can be violated in any $\mathcal{A}_\delta$, however small $\delta > 0$ may be. We illustrate this fact in the example below.

Example 4. Consider the automaton of Fig. 2. When enlarged by δ, the first transition of the loop can be anticipated by δ, hence taken when $x = 2-\delta$, while the second transition of the loop can be delayed until $y = 2+\delta$. Starting from a configuration where $x = 1$ and $y = 0$, we end up with $x = 1-2\delta$ and $y = 0$ after applying these two transitions. This can be repeated, until we come back to the left-most state of the loop with x being close to zero, which will then give access to the right-most state of the automaton.

The idea of *robustness analysis* (or *robust model-checking*) is to verify the correctness of the enlarged timed automaton. We define it formally below.

Definition 2 (Robustness analysis). *Given a linear-time property φ[1] and a timed automaton $\mathcal{A}$, decide whether there is some $\delta_0 > 0$ such that for all $\delta \in [0, \delta_0]$, all executions in $\mathcal{A}_\delta$ satisfy property φ. If this is the case, we say that* $\mathcal{A}$ robustly satisfies *φ, and that δ_0 witnesses this robust satisfaction.*

Why robustness analysis? The idea behind robustness analysis is that the enlargement by δ of all the guards should capture the imprecisions of the executed (or implemented) system. The accurateness of the approach then relies on the assumption that for some $\delta > 0$, $\mathcal{A}_\delta$ over-approximates the real behavior of the system (with δ depending on the characteristics of the processor). To support this assumption, program semantics are given in [DDR05] and in [SBM11]. They take into account various delay parameters of a processor, and are shown to satisfy this assumption.

This yields the following methodology for the development of correct implemented real-time systems.

1. Design $\mathcal{A}$;
2. Verify the robust satisfaction of φ by $\mathcal{A}$, that is verify $\mathcal{A}_\delta$, where δ is a parameter;
3. Implement $\mathcal{A}$: its correctness will be implied by that of $\mathcal{A}_\delta$.

Let us comment the robust satisfaction. It is easy to get convinced that if δ_0 witnesses the robust satisfaction of φ by $\mathcal{A}$, then so does any $0 < \delta \leq \delta_0$. This "faster-is-better" property is a desirable property when studying implementability: if a system is correct when implemented on a processor, then it will remain correct on a faster processor.

Decidability and complexity results. When $\delta_0 > 0$ is fixed, verifying that $\mathcal{A}_{\delta_0}$ satisfies φ can be done using standard model-checking techniques. However, robust model-checking is rather interested in a paramaterized analysis, and in the synthesis of a positive value for δ_0. In the theorem below, by timed automata with *progress cycles*, we refer to timed automata whose all cycles reset each clock at least once.

[1] That can be any ω-regular or LTL property, or even some timed property.

Theorem 2 ([DDMR08, BMR06, BMS11, BMR08]). *Robust model-checking of safety, Büchi, LTL properties for closed timed automata is PSPACE-complete. Robust model-checking of coFlatMTL (a fragment of MTL) for closed timed automata with progress cycles is EXPSPACE-complete.*

One can notice that the theoretical complexity of robust model-checking is the same as that of standard model-checking, which is quite surprising but very positive. Unfortunately no efficient symbolic algorithms have been developed so far, and a big effort is required in this direction, so that the practical impact of these results is consolidated.

4 Robust Implementation

In this section, we will explore techniques that allow generating implementations that are robust to perturbations by construction. These techniques also ensure that some desired properties of the initial model are preserved in the implementation. Hence, this approach has the advantage of allowing the system designer to concentrate on the initial model, and use existing tools to prove correctnes, while leaving the implementation effort to a computer.

More precisely, we describe results that allow one to automatically transform a given timed automaton into a robust one, while preserving at least its time-abstract behaviors. Thus, our target implementation formalism is again timed automata subject to guard enlargement. We will present two ways of achieving robust implementation, each corresponding to a different point of view on the models.

4.1 Shrinking

Assume that we are interested in strictly respecting the timing constraints described by the guards of given timed automata, that is, we do not tolerate any enlargement of the guards. To motivate this assumption, consider the following scenario. We are designing a system that communicates with another component which only receives signals in a given time interval, say $x \in [l, u]$; any signal sent outside this interval is lost. While an abstract model of the system assuming instantaneous communication might simply require the signals to be sent inside the same interval, that is, $x \in [l, u]$, the system would not be correct under timing imprecisions, say, due to communication delays. A natural idea is to obtain a correct implementation by *shrinking* the guard into $x \in [l+\delta, u-\delta]$, where $\delta > 0$ is an appropriate parameter. In fact, assuming that timing imprecisions are modelled by an enlargement of Δ, we have $[l+\delta-\Delta, u-\delta+\Delta] \subseteq [l, u]$. In other terms, the behaviors induced by the implementation of the shrunk guard is included in the behaviors of the original abstract model.

While shrinking can offer an easy way to implement timed automata while preserving behaviors, it can also remove some significant behaviors from the model, such as liveness. In fact, some timed automata become blocking under the slightest shrinking of their guards. Such an example is given in Fig. 6.

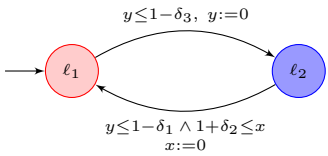


Fig. 6. A shrunk timed automaton that is blocking whenever $\delta_2 > 0$ or $\delta_3 > 0$. To see this, consider any infinite execution and let $d_1, d_3, \ldots$ denote the delays at location l_1, and $d_2, d_4, \ldots$ those at l_2. One can show that $1 \le d_{2i-1} + d_{2i}$ for all $i \ge 1$, which means that time diverges, but also $\delta_2 + \delta_3 \le d_{2i+2} - d_{2i}$ and $d_{2i} \le 1$. The latter means that the sequence $(d_{2i})_i$ increases at least by $\delta_2 + \delta_3$ at each step and is bounded above by 1, which is possible only when $\delta_2 = \delta_3 = 0$. Note that even if $\delta_2 = \delta_3 = 0$, non-blockingness requires consecutive delays to be equal, which is not realistic for digital systems.

We are interested in obtaining an implementation by shrinking the guards of a timed automaton such that some desired properties are preserved in the resulting model. We will consider a possibly different shrinking parameter for each atomic guard in order to increase the chance of success. Formally, given a timed automaton $\mathcal{A}$, let I denote the set of its atomic guards. Each atomic guard will be shrunk by $k_i\delta$ where $\delta > 0$ is a uniform parameter, and k_i's are possibly different natural numbers. Notice that any vector of rational numbers can be written as $\boldsymbol{k}\delta$ where $\boldsymbol{k}$ is the vector of k_i's. The resulting shrunk timed automaton is denoted $\mathcal{A}_{-\boldsymbol{k}\delta}$ (Note that this is simply enlargement by a negative amount). The *shrinkability* problem asks for property preservation under possible shrinking; its two variants are defined formally as follows.

A timed automaton is said to be *non-blocking* if from any state some discrete transition is enabled after some time delay.

Definition 3 (Shrinkability). *Given a timed automaton $\mathcal{A}$, decide whether for some $\delta_0 > 0$, and some positive integer vector $\boldsymbol{k}$,*

1. *$\mathcal{A}_{-\boldsymbol{k}\delta}$ is non-blocking,*
2. *and $\mathcal{A}_{-\boldsymbol{k}\delta}$ time-abstract-simulates $\mathcal{A}$,*

for all $\delta \in [0, \delta_0]$.

We also consider variants of the shrinkability problem. *Non-blocking-shrinkability* only asks for condition 1 to be satisfied in Definition 3, while *simulation-shrinkability* asks only for condition 2. The latter can be refined by only requiring the shrunk timed automaton to time-abstract simulate a given finite automaton $\mathcal{F} \sqsubseteq_{\text{t.a.}} \mathcal{A}$. In this case, only some part $\mathcal{F}$ of the time-abstract behavior of $\mathcal{A}$ is preserved in $\mathcal{A}_{-\boldsymbol{k}\delta}$.

Theorem 3 ([SBM11]). *For closed non-blocking timed automata, non-blocking-shrinkability is decidable in polynomial space, and in NP if the maximum degree of the locations is fixed.*

For closed timed automata, under some technical assumptions, simulation-shrinkability is decidable in pseudopolynomial time in the sizes of $\mathcal{A}$ and $\mathcal{F}$.

Algorithms of Theorem 3 allow computing the parameters $\boldsymbol{k}$ and the maximum $\delta > 0$. This enables the following design methodology which we sketched at the beginning of this section.

1. Design and verify timed automaton $\mathcal{A}$;
2. Apply shrinkability, which yields $\mathcal{A}_{-\boldsymbol{k}\delta}$;
3. Implement $\mathcal{A}_{-\boldsymbol{k}\delta}$ which is a safe refinement even under imprecisions: $\mathcal{A}_{-\boldsymbol{k}\delta+\Delta} \sqsubseteq \mathcal{A}$, and moreover it is non-blocking and it preserves all time-abstract behaviors of $\mathcal{A}$.

Tool support. The simulation-shrinkability algorithm described above was implemented in a software tool called Shrinktech [San13]. The tool is integrated with the Kronos model-checker: it takes as input (network of) timed automata and checks for shrinkability with respect to a given finite automaton $\mathcal{F}$. If shrinkability succeeds, then the tool outputs the computed parameters δ and $\boldsymbol{k}$, and the witnessing simulator sets. Otherwise, a counter-example is output, which is a finite automaton that cannot be simluated by any shrinking of the timed automaton. Some benchmarks are given in [San13], and the tool is available at `http://www.lsv.ens-cachan.fr/Software/shrinktech`.

4.2 Approximately Bisimilar Implementation

We now assume that the implementation of the timed automaton design can tolerate small timings errors, but we would like to avoid the accumulation of these over time as in Example 4. Under this "relaxation", we will show that we can achieve more than with the shrinking technique we described in the previous subsection: under some assumptions, all timed automata can be "approximately" implemented.

In order to compare the behaviors of two timed automata allowing bounded errors in timings, we use ε-bisimilarity, an extension of timed bisimilarity. Formally, an ε-bisimulation R is a relation between the states of a timed automaton defined as follows. For any s and t with $s\ R\ t$, whenever $s \xrightarrow{\sigma} s'$, there exists t' such that $t \xrightarrow{\sigma} t'$, and $t\ R\ t'$, and whenever $s \xrightarrow{d} s'$ for some $d \geq 0$, then there exists $d' \in [\max(0, d-\varepsilon), d+\varepsilon]$ such that $s'\ R\ t+d'$. We denote $s \sim_\varepsilon t$, if there is an ε-bisimulation between states s and t. Note that $\sim_0$ is the usual timed bisimulation, also written $\sim$.

Now, given timed automata $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}'$ is an *ε-implementation*, if $\mathcal{A} \sim \mathcal{A}'$ and $\mathcal{A}' \sim_\varepsilon \mathcal{A}'_\delta$ for some $\delta, \varepsilon > 0$. Here, the former condition means that the alternative timed automaton $\mathcal{A}'$ is "equivalent" to $\mathcal{A}$ in the exact semantics, while the latter condition requires the robustness of $\mathcal{A}'$: under enlargement, all behaviors are approximately included in those of $\mathcal{A}$.

We also consider the simpler notion of *safe implementation* by requiring that $\mathcal{A} \sim \mathcal{A}'$ and that $\mathcal{A}'$ and $\mathcal{A}'_\delta$ have the same set of reachable locations for some $\delta > 0$.

Theorem 4. *For any timed automaton $\mathcal{A}$, and any $\varepsilon > 0$, one can construct an ε-implementation, and a safe implementation, both in exponential time.*

Although one can use ε-implementations to obtain safe implementations, the latter are smaller in size in practice.

Methodology. The above theorem allows one to completely separate design and implementation of timed automata models. In fact, since the implementation can be ensured for all models, one can only concentrate on designing the model and proving properties in the exact semantics; imprecisions do not need to be taken into account. One can then specify the precision ε depending on the required accuracy.

1. Design and verify timed automaton $\mathcal{A}$;
2. Automatically generate an implementation.

5 Robust Realisability and Control

In this section, we are interested in checking whether a given desired behavior is realisable in a timed automaton when the time delays are subject to perturbations. Our goal is to detect behaviors (such as defined by Büchi conditions) that can be realised even when the delays can only be chosen upto a bounded error. In fact, although the theory of timed automata allows characterizing the existence of infinite runs accepting for a Büchi condition, not all such runs are realisable with finite precision (hence, not in hardware neither); we already saw in Example 3 a timed automaton where any infinite run contains delay and clock value sequences that are convergent, that is, requires infinite precision. If the input model is timed games, then this problem is closely related to that of *robust controller synthesis*, where the goal is to find winning strategies that resist to systematic perturbations in the time delays.

We thus present the problem both as a realisability problem for timed automata, and as robust controller synthesis in timed games. We formalise this problem as a turn-based game between a controller that chooses delays and edges and an environment that perturbs the chosen delays. Two variants of this game semantics were studied in [CHP11, SBMR13] and [BMS12] respectively. We first present the two variants and related results, and compare them at the end of this section.

5.1 Conservative Semantics

We start with the *conservative game semantics*. Intuitively, the semantics is a turn-based two-player game parameterized by $\delta > 0$, where Player 1, also called *Controller* chooses a delay $d > \delta$ and an edge whose guard is satisfied after any delay in $d + [-\delta, \delta]$. Then, Player 2, also called *Perturbator* chooses an actual delay $d' \in d + [-\delta, \delta]$ after which the edge is taken. Hence, the delays suggested by Controller are perturbed by Perturbator by an amount from $[-\delta, \delta]$, but the guard is required to be satisfied whatever the perturbation is.

We also consider two-player timed games where, in addition to being able to suggest delays and actions, Perturbator can perturb by an amount in $[-\delta, \delta]$ the delays chosen by Controller.

While for $\delta = 0$, a strategy for Controller in this game semantics merely yields a run, for $\delta > 0$, it describes how Controller reacts to perturbations in the past. It may, for instance, choose the delays so as to correct the effect of the earlier perturbations.

We are interested in deciding whether for some $\delta > 0$, there exists a strategy of Controller whose all outcomes satisfy a given ω-regular condition. This is the *parameterized robust controller synthesis for Büchi objectives*. If the parameter is given as part of the input, then we will call the problem *fixed-parameter robust controller synthesis*. While the timed automaton of Fig. 6 is uncontrollable in this sense (which follows from the fact that it is not shrinkable), Fig. 7 shows example of a controllable timed automaton.

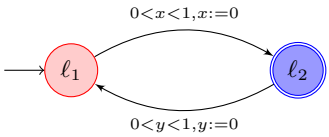


Fig. 7. A timed automaton (from [BA11]) that is robustly controllable for the Büchi objective $\{\ell_2\}$. In fact, perturbations at a given transition do not affect the rest of the run; they are *forgotten*.

For a fixed parameter $\delta > 0$, we have the following result for parity conditions.

Theorem 5 ([CHP11]). *The fixed-parameter robust controller synthesis for timed games and parity conditions in the conservative game semantics can be solved in exponential time.*

The above algorithm is obtained by reduction to timed games; in fact, when δ is known, it is possible to encode the game semantics as a timed game, and apply known algorithms.

For Büchi objectives, the parameterized version of the problem can be solved in polynomial space:

Theorem 6 ([SBMR13]). *Parameterized robust controller synthesis for timed automata and Büchi conditions in the conservative game semantics is PSPACE-complete.*

The above theorem is established by characterising those cycles that can be repeated infinitely by Controller against any strategy of Perturbator. The characterisation is expressed in terms of the reachability graph between the vertices of the visited regions (a.k.a. the orbit graph), and was introduced in [BA11] in a different context. [BA11] characterizes those cycles of a timed automaton along which the clock values are not convergent; these are called *forgetful cycles*.

The contribution of [SBMR13] consists in showing that any long enough run along non-forgetful cycles can be made blocking by an appropriate strategy of Perturbator.

5.2 Excess Semantics

We now present a variant of the previous semantics, similar in definition, but different in terms of usage. The *excess game semantics* is a turn-based two-player game between Controller and Perturbator, and parameterized by $\delta > 0$. The difference is that at each turn, Controller now only has to suggest a delay $d \geq \delta$ and an edge whose guard is satisfied only after the delay d. After such a move, Perturbator can modify the delay by any amount in $[-\delta, \delta]$, and the perturbed delay and the edge chosen by Controller are taken whether or not the guard is satisfied after the perturbation. We refer to Example 5 for an illustration of this semantics.

The following result shows the decidability of the parameterized robust controller synthesis problem for reachability objectives in timed automata and turn-based timed games.

Theorem 7 ([BMS12]). *Parameterized robust controller synthesis for turn-based timed games and reachability objectives in the excess game semantics is EXPTIME-complete.*

5.3 Comparison

We defined here two very similar game semantics. We believe both are meaningful and can appear naturally at different levels of abstraction. For instance, the excess-perturbation game semantics is a natural choice when modelling a real-time system with fixed task execution times, if we also know that these execution times will be subject to perturbation whose magnitude is unknown in advance, which may depend on the implementation platform to be chosen later. Incorporating these perturbations in the model, for instance, by replacing equality constraints by non-punctual intervals, requires a choice of a suitable interval length which may not be known. Moreover this will increase the state space in general. Hence the excess-perturbation game semantics allows one to keep the design abstract and still apply robustness analysis.

On the other hand, in some applications, specifying distinct lower and upper bounds in timings may be natural. For instance, if one is interested in modelling an embedded system that should send a signal to another component which accepts input signals in a time interval $[l, u]$, then it is natural to look for a controller that strictly respects this interval. Such a model and its semantics would be closer to the actual program. In this case, the conservative-perturbation game semantics is the natural choice.

Note also that while any outcome of the conservative game semantics is a run of the timed automaton (or game) in the usual semantics, this is not the case for the excess semantics. On the other hand, outcomes of the excess game semantics are runs of the enlarged automata.

5.4 Weighted Timed Automata and Games

Weighted timed automata [ALP01, BFH+01] are an extension of timed automata with one cost variable which grows with a possibly different constant derivative at each location. Cost-optimal reachability is decidable in this model, but undecidable in weighted timed games. Robustness problems can naturally be addressed for weighted timed automata and games, following similar motivations as timed automata. Moreover, one could hope that in weighted timed games, a suitable robust semantics could render the cost-optimal reachability problem decidable. We show in this section that undecidability still holds in both game semantics.

We consider conservative and excess perturbation game semantics. As Example 5 below shows, the optimal cost that can be achieved depends on δ. Because we assume that δ is a small parameter whose value can be adjusted as required, we will concentrate on the *limit-cost* of a strategy in a weighted timed automaton under the game semantics. The limit cost is the cost achieved by a strategy, when δ goes to 0.

Example 5. Figure 8 displays an example of a weighted timed game. Plain (resp. dashed) arrows are for Player 1 (resp. Player 2) edges. The slopes are indicated above each state. A strategy for Player 1 is to suggest a delay of 1 and choose

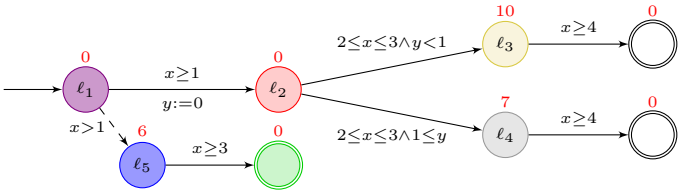


Fig. 8. Example of a WTG

the edge from ℓ_1 to ℓ_2. This prevents Player 2 from going down to location ℓ_5, where the cost of accepting is 12. From location ℓ_2, Player 1 can go to ℓ_4, from where a target location is reached with cost 7.

Under the excess-perturbation semantics, as in the exact case, Controller can suggest a delay of 1 and choose the edge from ℓ_1 to ℓ_2. The location ℓ_5 can thus be avoided. Now, one can see that the move of Perturbator determines the next location to be visited: if Perturbator adds a positive perturbation (*i.e.* if the delay is in $[1, 1+\delta]$), then only location ℓ_3 is reachable. Conversely, a negative perturbation enables only location ℓ_4. To maximize the cost, Perturbator will force the play to ℓ_3, so Controller can only ensure a cost of $10 + \Theta(\delta)$.

We now focus on the conservative semantics. The above strategy is no more valid since in this case, Controller can only suggest delays of at least $1+\delta$. Then Perturbator can force the play to ℓ_5. Here, the cost of winning is $12 + \Theta(\delta)$.

We define the *optimal limit-cost (strong) decision problem* for the conservative or excess semantics as the problem of deciding whether some strategy achieves a limit cost of at least (resp. more than) a given threshold.

Theorem 8 ([BMS13])

- *The optimal limit-cost decision problem is undecidable for weighted timed automata under the excess game semantics.*
- *The optimal limit-cost strong decision problem is PSPACE-complete for weighted timed automata under the conservative game semantics, and undecidable for weighted timed games.*

Hence, the decision problems on optimal limit-cost in weighted timed games remain undecidable. Moreover, the problem even becomes undecidable for weighted timed *automata* in the excess game semantics. This is rather surprising since it is often believed that introducing imprecisions render problems easier to solve [AB01].

6 Conclusion

In this paper we have presented a recent approach to the theory of robustness for timed systems. The model of perturbation that is considered is that of guard enlargement, which captures time measurement errors and jitter. Other models of perturbation could be considered, which would more adequately take into account other sources of errors in the execution of timed systems.

Under this model of perturbation, we have described several robust verification problems which have been formulated and partly solved recently. Complexities of these problems are quite standard in the domain of timed systems, and symbolic technics need now to be developed for solving the various problems, that would support usability of the approach.

Robustness in timed systems is an important issue, the effort to develop a full theory of robustness needs therefore to be continued.

References

[AB01] Asarin, E., Bouajjani, A.: Perturbed turing machines and hybrid systems. In: Proc. 16th Annual Symposium on Logic in Computer Science (LICS 2001), pp. 269–278. IEEE Computer Society Press (2001)

[ABG+08] Akshay, S., Bollig, B., Gastin, P., Mukund, M., Narayan Kumar, K.: Distributed Timed Automata with Independently Evolving Clocks. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 82–97. Springer, Heidelberg (2008)

[ACS10] Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Proc. 10th International Workshop on Embedded Software (EMSOFT 2010), pp. 229–238. ACM (2010)

[AD94] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

[AKY10] Abdulla, P.A., Krčál, P., Yi, W.: Sampled semantics of timed automata. Logical Methods in Computer Science 6(3:14) (2010)

[ALP01] Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)

[Alu91] Alur, R.: Techniques for Automatic Verification of Real-Time Systems. PhD thesis, Stanford University, Stanford, CA, USA (1991)

[AT05] Altisen, K., Tripakis, S.: Implementation of timed automata: An issue of semantics or modeling? In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 273–288. Springer, Heidelberg (2005)

[BA11] Basset, N., Asarin, E.: Thin and thick timed regular languages. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 113–128. Springer, Heidelberg (2011)

[BDL+06] Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST 2006), pp. 125–126. IEEE Computer Society Press (2006)

[BDM+98] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)

[BFH+01] Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.M.T., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)

[BMR06] Bouyer, P., Markey, N., Reynier, P.-A.: Robust model-checking of linear-time properties in timed automata. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 238–249. Springer, Heidelberg (2006)

[BMR08] Bouyer, P., Markey, N., Reynier, P.-A.: Robust analysis of timed automata *via* channel machines. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 157–171. Springer, Heidelberg (2008)

[BMS11] Bouyer, P., Markey, N., Sankur, O.: Robust model-checking of timed automata via pumping in channel machines. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 97–112. Springer, Heidelberg (2011)

[BMS12] Bouyer, P., Markey, N., Sankur, O.: Robust reachability in timed automata: A game-based approach. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 128–140. Springer, Heidelberg (2012)

[BMS13] Bouyer, P., Markey, N., Sankur, O.: Robust weighted timed automata and games. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 31–46. Springer, Heidelberg (2013)

[BS91] Brzozowski, J.A., Seger, C.-J.H.: Advances in asynchronous circuit theory. Bulletin of the European Association of Theoretical Computer Science, EATCS (1991)

[CHP11] Chatterjee, K., Henzinger, T.A., Prabhu, V.S.: Timed parity games: Complexity and robustness. Logical Methods in Computer Science 7(4) (2011)

[CHR02] Cassez, F., Henzinger, T.A., Raskin, J.-F.: A comparison of control problems for timed and hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)

[DDMR08] De Wulf, M., Doyen, L., Markey, N., Raskin, J.F.: Robust safety of timed automata. Formal Methods in System Design 33(1-3), 45–84 (2008)

[DDR05] De Wulf, M., Doyen, L., Raskin, J.-F.: Systematic Implementation of Real-Time Models. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 139–156. Springer, Heidelberg (2005)

[Dim07] Dima, C.: Dynamical properties of timed automata revisited. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 130–146. Springer, Heidelberg (2007)

[GHJ97] Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)

[HR00] Henzinger, T.A., Raskin, J.-F.: Robust undecidability of timed and hybrid systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 145–159. Springer, Heidelberg (2000)

[KP05] Krčál, P., Pelánek, R.: On sampled semantics of timed systems. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 310–321. Springer, Heidelberg (2005)

[Pur98] Puri, A.: Dynamical properties of timed automata. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 210–227. Springer, Heidelberg (1998)

[Pur00] Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)

[San13] Sankur, O.: Shrinktech: A tool for the robustness analysis of timed automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1006–1012. Springer, Heidelberg (2013)

[SBM11] Sankur, O., Bouyer, P., Markey, N.: Shrinking timed automata. In: Proc. 30th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011). LIPIcs, vol. 13, pp. 375–386. Leibniz-Zentrum für Informatik (2011)

[SBMR13] Sankur, O., Bouyer, P., Markey, N., Reynier, P.-A.: Robust controller synthesis in timed automata. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 546–560. Springer, Heidelberg (2013)

[SFK09] Swaminathan, M., Fränzle, M., Katoen, J.-P.: The surprising robustness of (closed) timed automata against clock-drift. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Proc. 5th IFIP International Conference on Theoretical Computer Science (TCS 2008). IFIP, vol. 273, pp. 537–553. Springer, Boston (2009)

Automated Verification of Concurrent Software*

Daniel Kroening

University of Oxford

Abstract. Effective use of concurrency is key to accelerating computations in a post frequency-scaling era. We review a research programme aimed at automated formal verification of a broad variety of concurrent systems. We briefly survey different forms of asynchronous concurrent computations, with a focus on multi-threaded, multi-core computation. We then highlight semantic and scalability challenges that arise when applying automated reasoning technology to this class of software.

We then discuss two very different techniques to address the challenges in this domain. The key insight behind the first technique is to exploit the symmetry that is inherent in many concurrent software programs: the programs execute a parametric number of identical threads, operating on different input data. Awareness of this design principle enables the application of symmetry reduction techniques such as counter abstraction, and encodings as Petri net coverability problems [2,6,4,3].

The second technique exploits the observation that asynchronous concurrent systems are frequently only very loosely synchronised. This gives rise to an encoding of the system using a set of constraints over partial orders. The constraints can be passed using a modern SAT/SMT solver, which gives rise to an effective bounded verification technique for asynchronous concurrent systems [1,5].

The research presented is joint work with Jade Alglave, Gerard Basler, Alastair Donaldson, Jim Grundy, Alexander Horn, Alexander Kaiser, Lihao Liang, Michele Mazzucchi, Tom Melham, Michael Tautschnig, Celina Val and Thomas Wahl.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
2. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Context-aware counter abstraction. Formal Methods in System Design (FMSD) 36(3), 223–245 (2010)
3. Donaldson, A., Kaiser, A., Kroening, D., Tautschnig, M., Wahl, T.: Counterexample-guided abstraction refinement for symmetric concurrent programs. Formal Methods in System Design (FMSD) 41(1), 25–44 (2012)

* Supported by ERC project 280053, EPSRC project EP/G026254/1 and the Semiconductor Research Corporation (SRC) under task 2269.002.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 19–20, 2013.

4. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 356–371. Springer, Heidelberg (2011)
5. Horn, A., Tautschnig, M., Val, C., Liang, L., Melham, T., Grundy, J., Kroening, D.: Formal co-validation of low-level hardware/software interfaces. In: Formal Methods in Computer-Aided Design, FMCAD (2013)
6. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)

Provenance Verification

Rupak Majumdar[1], Roland Meyer[2], and Zilong Wang[1]

[1] MPI-SWS
[2] University of Kaiserslautern

The *provenance* of an object is the history of its origin and derivation. Provenance tracking records the provenance of an object as it evolves. In computer science, provenance tracking has been studied in many different settings, such as databases [7,3,2], scientific workflows [13,5], and program analysis [4,12,9], often under different names (lineage, dependence analysis, taint analysis) and with varying degrees of (in)formality. Provenance information can be used in many ways, for example, to identify which sources of data led to a result, to ensure reproducibility of a scientific workflow, or to check security properties such as information flow.

We study provenances tracking in the context of distributed message-passing programs. These programs consist of *principals*, who communicate with each other, and associate additional information with messages — the *provenance*. In a simple setting, the provenance records the sequence of principals that accessed the message in the past (with principals potentially appearing multiple times). We study the *provenance verification problem*: the problem of statically checking whether the provenances of all messages belong to a specified regular set of provenances along all possible executions of the program.

We give a unifying view of provenance tracking for distributed message-passing programs. Following Souilah, Francalanza, and Sassone [14], we model distributed systems in the π-calculus and give a provenance-carrying semantics. This semantics is relative to a domain of provenance annotations. Besides the regular word-languages mentioned above, we use the domains of provenance sets and regular tree-languages.

We focus on the algorithmic verification of provenances. Since the provenance-verification problem is undecidable for the full π-calculus, we consider restricted classes of programs. Our main result shows that provenance verification is decidable for the class of *depth-bounded* π-calculus processes [11], an expressive class that subsumes most known decidable subclasses of the π-calculus. Intuitively, depth-boundedness is a restriction on the communication topologies which limits the length of acyclic paths. Depth-bounded systems strictly generalize Petri nets, and are expressive enough to capture common programming models, such as asynchronous programs [6], actor-like programs [15], and some further generalizations.

We show a reduction from provenance verification to coverability of depth-bounded processes, a problem shown to be decidable [15]. Our proof uses well-structuredness arguments [1] with symbolic representations of automata. Interestingly, the general method is strong enough to recover the decidability of provenance verification for asynchronous message passing programs with finite data domains [10].

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 21–22, 2013.

As an application of our results, we formulate privilege escalation problems in abstractions of browser extensions and Android programs as instances of provenance verification. Thus, our formalization gives a uniform static analysis algorithm for information flow issues in distributed message passing programs.

We also consider more expressive linear-temporal logic specifications for provenance policies. While provenance verification for linear-temporal logic is not decidable for all depth-bounded systems, we show that it is decidable for a recently studied subclass, the *name-bounded* processes [8]. Intuitively, name-bounded processes model distributed message-passing programs operating under resource bounds. Formally, there is a bound on the number of channels that holds for all reachable processes. We show that asynchronous programs can be modeled by name-bounded processes. Thus, model checking linear-temporal logic provenance specifications for asynchronous programs is decidable as well.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321. IEEE (1996)
2. Benjelloun, O., Sarma, A., Halevy, A.Y., Widom, J.: ULDBs: Databases with uncertainty and lineage. In: VLDB, pp. 953–964. ACM (2006)
3. Buneman, P., Khanna, S., Tan, W.-C.: Why and where: A characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2000)
4. Cheney, J., Ahmed, A., Acar, U.: Provenance as dependency analysis. Math. Struct. in Computer Science 21, 1301–1337 (2011)
5. Cui, Y., Widom, J., Wiener, J.: Tracing the lineage of view data in a warehousing environment. ACM TODS 25, 179–227 (2000)
6. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. Submitted for publication, TOPLAS (2011)
7. Green, T., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS, pp. 31–40. ACM (2007)
8. Hüchting, R., Majumdar, R., Meyer, R.: A theory of name boundedness. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 182–196. Springer, Heidelberg (2013)
9. Livshits, B., Lam, M.: Finding security errors in Java programs with static analysis. In: Usenix Security Symposium, pp. 271–286 (2005)
10. Majumdar, R., Meyer, R., Wang, Z.: Static provenance verification for message passing programs. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 366–387. Springer, Heidelberg (2013)
11. Meyer, R.: On boundedness in depth in the π-calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) IFIP TCS. IFIP, vol. 273, pp. 477–489. Springer, Boston (2008)
12. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21, 5–19 (2003)
13. Simmhan, Y., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Record 34(3), 31–36 (2005)
14. Souilah, I., Francalanza, A., Sassone, V.: A formal model of provenance in distributed systems. In: Theory and Practice of Provenance (2009)
15. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

Reachability Modulo Theories

Akash Lal and Shaz Qadeer

Microsoft Research
{akashl,qadeer}@microsoft.com

Abstract. Program verifiers that attempt to verify programs automatically pose the verification problem as the decision problem: *Does there exist a proof that establishes the absence of errors*? In this paper, we argue that program verification should instead be posed as the following decision problem: *Does there exist an execution that establishes the presence of an error*? We formalize the latter problem as Reachability Modulo Theories (RMT) using an imperative programming language parameterized by a multi-sorted first-order signature. We present complexity results, algorithms, and the Corral solver for the RMT problem. We present our experience using Corral on problems from a variety of application domains.

1 Introduction

Practical program verifiers are difficult to design and implement. To be useful, the verification must be automated as much as possible. At the same time, the verifier must be able to model precisely the complex features and abstractions used in real-world programming languages. First-order provers based on satisfiability-modulo-theories (SMT) satisfy these conflicting requirements simultaneously by providing both a rich modeling framework for encoding language semantics and high-degree of automation for deciding satisfiability of expressions. Consequently, many program verifiers use SMT solvers in their core.

Efficiently decidable satisfiability checking of expressions is necessary but insufficient for building practical program verifiers. The reason is that while verification of finite executions can be encoded precisely as satisfiability checking, verification of unbounded executions cannot be similarly encoded. The latter problem is undecidable for the standard theories, e.g., linear arithmetic, uninterpreted functions, arrays, etc. used in modeling program behaviors.

Program verifiers that attempt to verify programs automatically pose the verification problem as the decision problem: *Does there exist a proof that establishes the absence of errors*? As mentioned above, this problem is undecidable; in fact, it is not even recursively enumerable. Intuitively, enumerating program proofs is so difficult because it requires a complete proof system whose assertions come from a statically known and decidable language. Practical verifiers use the Floyd-Hoare proof system and the language of expressions in the programming langauge as the assertion language. Our experience building verifiers for sequential programs indicates that while the Floyd-Hoare proof system is often complete enough, expressions in the programming language are inadequate

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 23–44, 2013.

for capturing proofs of even simple properties. As an example, the theory of arrays which naturally encodes program expressions for looking up and updating the heap is inadequate for expressing a Floyd-Hoare proof of correctness of a heap-manipulating program. Such a proof would typically require a richer theory capable of expressing both quantified facts as well as data abstractions such as objects, lists, and trees. We have encountered this difficulty in practice while deploying the HAVOC verifier [22] to verify simple type-state properties on device drivers [23].

In this paper, we argue that program verification should instead be posed as the following decision problem: *Does there exist an execution that establishes the presence of an error*? This problem is undecidable but recursively enumerable for a programming language with decidable expression language. A prototypical semi-decision procedure for this problem would enumerate program executions in a fair manner to provide complete search in the limit. There are several advantages of this problem formulation. First, it directly matches the most important and common uses of automatic program verification—bug-finding and debugging. When program verification is posed as a proof discovery problem, a counterexample is a by-product of the failure of proof discovery. A direct search for counterexamples could potentially be more efficient at uncovering bugs. Second, it naturally allows the formulation of bounded and decidable versions of the problem. It is possible that as we develop better techniques for solving the bounded problem, we will get incrementally better at solving the harder unbounded problem. As anecdotal evidence from the literature on hardware verification, success in solving the bounded problem (NP-complete) via *Boolean* satisfiability solvers has led to increasing success in solving the harder (PSPACE-complete) unbounded problem. Finally, we note that stating the problem as a search for counterexamples does not preclude the use of proof techniques for pruning search; proofs simply become an opportunity for optimization rather than a goal by themselves.

We present *reachability modulo theories* (RMT), a parameterized framework for modeling program executions and stating verification problems. RMT emphasizes reachability of an error state as opposed to unreachability of all error states. A RMT problem is specified by picking points along two orthogonal axes defining a programming language—control and data. Control is specified using a control-flow graph with an appropriately restricted set of features. For example, in this paper we specifically address sequential and potentially-recursive control flow but it is just as easy to restrict recursion or generalize to allow concurrency by allowing asynchronous and parallel calls. Data is specified by using a multi-sorted first-order signature, much like in the definition of satisfiability modulo theories. Given such a signature, we allow each program variable to be associated with a sort and assignments from well-sorted expression to a variable of a matching sort. In other words, RMT exposes the full power of first-order modeling provided by the satisfiability modulo theories framework and strengthens it with a control flow graph, allowing us to define bounded and unbounded operational semantics over rich data domains.

In addition to presenting the basic definition of RMT in Section 2, this paper also contains the following contributions:

- To improve our understanding of the RMT problem, we studied its complexity for loop-free and recursion-free programs. We call the problem for such programs (with acyclic call graphs) the hierarchical RMT problem. In Section 3, we present complexity results for various expression languages that are relevant for modeling practical problems. Restricting attention to quantifier-free expressions, we show that if the expression language is decidable in NP, then hierarchical RMT is decidable in NEXPTIME; if the only sort is *Boolean*, then hierarchical RMT is PSPACE-complete; if, in addition, uninterpreted functions are available, then hierarchical RMT is NEXPTIME-complete.
- We use the Boogie [9] language as the concrete representation for an RMT problem. We have developed translators from C and .NET bytecode into Boogie. In Section 4, we give examples of how the operational semantics and verification problems for different source languages are encoded into Boogie. Our work enables both sequential and concurrent Boogie programs, regardless of the source language from which they are derived, to be verified without requiring contracts such as preconditions, postconditions, and loop invariants.
- We have implemented Corral [25], a solver for the RMT problem. Corral will replace Slam [6] as the solver inside Static Driver Verifier in the next release of the Microsoft Windows operating system. In Sections 5 and 6, we describe the core techniques and architecture of Corral and our experience applying it to solve reachability problems on device drivers. We also describe other applications, such as analyzing sequentializations of concurrent programs, detecting security vulnerabilities in web applications, and solving debugging queries for .NET bytecode.

Related Work. Software model checkers [7, 21, 29] based on predicate abstraction [19] are the best known examples of program verification as proof search, as opposed to our work which is founded in counterexample search. In addition to this foundational difference, another difference is the programming language on which the problem is typically stated. Software model checking has traditionally been defined for the C programming language, whose expression language makes implicit reference to the heap and and is consequently not directly amenable to logical reasoning. On the other hand, we define our programming language abstractly using a multi-sorted first-order signature; consequently, the well-understood techniques of weakest preconditions and verification-condition generation are immediately available to us.

Recently, another attempt to state the proof search problem using first-order signatures has been made using a formulation based on Horn clauses [12, 13]. The main difference from our work is the focus on proof discovery as opposed to counterexample discovery; in this regard, their approach is similar to software model checking. However, similar to our formulation, their approach is independent of the syntax and semantics of source-level programming languages. While

```
// Identifiers
Id

// Sorts
Sort

// Expressions
Expr

// Variable declarations
VarDecl ::=  var Id: Sort

// Commands
Cmd ::=  assume Expr  |  Id := Expr  |  call Id* := Id(Id*)  |  havoc Id

// Basic Blocks
Block ::=  Id: Cmd goto Id*  |  Id: return

// Procedures
Proc ::=  procedure Id (VarDecl*) returns VarDecl* { VarDecl* Block+ }

// Program
Program ::=  Proc*
```

Fig. 1. The grammar of programs

RMT uses control flow and variables whose values can be updated, the Horn clause formulation uses side-effect free logical expressions. The Horn clause formulation has the advantage that the representation can encode not just program semantics but various proof systems such as Floyd-Hoare for sequential programs and Owicki-Gries for concurrent programs. Our formulation has the advantage that we can reason directly about sequential or concurrent program executions.

2 Reachability Modulo Theories

We define the RMT problem over a simple imperative programming language. The syntax of the language is shown in Fig. 1. A program (Program) is a list of procedure declarations. A procedure (Proc) can have any number of input and output parameters. The procedure body is a list of local variable declarations followed by a list of basic blocks; the first block is the one where execution of the procedure starts. The output variables of a procedure (if any) act like any other local variable, except that their value at a **return** command is the tuple of values returned on a call to the procedure. A basic block (Block) is a label followed by a list of commands. A command (Cmd) is either an assume command, or an assignment, or a **havoc** command, or a procedure call. The command **havoc** `x` non-deterministically assigns an arbitrary value (of the right type) to

x. The rest of the commands have the standard meaning. We disallow loops in our programs (they can be encoded using tail-recursion). Thus, the control-flow graph of a procedure is always acyclic. We refer to a recursion-free program as a *hierarchical* program.

We leave the syntax of expressions and sorts unspecified in the syntax and require only the following two properties:

- Expr is generated from a multi-sorted first-order signature containing the *Boolean* sort and the equality relation $=$.
- It is decidable to check satisfiability of *Boolean*-valued expressions in Expr.

The smallest expression language that satisfies the above properties is quantifier-free and contains only *Boolean* sort. For this expression language, the satisfiability problem is NP-complete.

$$\frac{P \vdash (l : \textbf{assume } e;\ \textbf{goto } ls') \qquad l' \in ls' \qquad e(M,\sigma) = \mathit{true}}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma, l') \cdot ss}$$

$$\frac{P \vdash (l : x := e;\ \textbf{goto } ls') \qquad l' \in ls' \qquad e(M,\sigma) = v}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma[x := v], l') \cdot ss}$$

$$\frac{P \vdash (l : \textbf{havoc } x;\ \textbf{goto } ls') \qquad l' \in ls' \qquad v \in M(\mathit{Sort}(x))}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma[x := v], l') \cdot ss}$$

$$\frac{\begin{array}{c} P \vdash (l : \textbf{call } \boldsymbol{y} := p(\boldsymbol{x});\ \textbf{goto } ls') \\ \boldsymbol{a} = \mathit{Ins}(p) \qquad \boldsymbol{b} = \mathit{Outs}(p) \qquad \boldsymbol{c} = \mathit{Locals}(p) \qquad \forall i.\ \sigma'(\boldsymbol{a}_i) = \sigma(\boldsymbol{x}_i) \\ \forall j.\ \sigma'(\boldsymbol{b}_j) = M(\mathit{Sort}(\boldsymbol{b}_j)) \qquad \forall k.\ \sigma'(\boldsymbol{c}_k) = M(\mathit{Sort}(\boldsymbol{c}_k)) \qquad \mathit{Count}((\sigma, l) \cdot ss, p) < b \end{array}}{P, M \vdash (\sigma, l) \cdot ss \rightarrow_b (\sigma', \mathit{Start}(p)) \cdot (\sigma, l) \cdot ss}$$

$$\frac{P \vdash (l : \textbf{return}) \qquad P \vdash (l' : \textbf{call } \boldsymbol{y} := p(\boldsymbol{x});\ \textbf{goto } ls') \qquad l'' \in ls' \qquad \boldsymbol{b} = \mathit{Outs}(p)}{P, M \vdash (\sigma, l) \cdot (\sigma', l') \cdot ss \rightarrow_b (\sigma'[\boldsymbol{y} := \sigma(\boldsymbol{b})], l'') \cdot ss}$$

Fig. 2. Operational semantics

Figure 2 presents the operational semantics of our programming language. The semantics is given using the derivation $P, M \vdash ss \rightarrow_b ss'$ that refers to the following elements:

- P is a program.
- M is a model for the first-order signature of program P.
- Each of ss and ss' is a stack of activation records, essentially a list of pairs, with each pair comprising a label and a valuation to program variables.
- $\rightarrow_b$ is the transition relation for an integer bound $b > 0$.

The semantics also use an auxiliary derivation of the form $P \vdash l : c; \textbf{goto } ls$ or $P \vdash l : \textbf{return}$. This derivation indicates that the program P contains an appropriately labeled basic block. We assume that all labels in the program are distinct.

The first rule in Figure 2 is for the assume statement; it allows execution to proceed only if the expression e evaluates to true. The next rule is for the assignment statement. The rule for havoc updates the variable x to an arbitrary value belonging to the interpretation of $Sort(x)$ in the model M. Next are the rules for procedure call and return. The activation record of the called procedure gets arbitrary initial values for the output and local variables. Upon return, actual output parameters in the caller are updated by looking up the appropriate variables in the callee. For any stack ss and procedure p, $Count(ss, p)$ returns the total number of activation records of procedure p on the stack. The call rule ensures that the number of activation records for any procedure does not go beyond b. We further define

$$\rightarrow = \bigcup_{b>0} \rightarrow_b$$

as the full transition relation of the program.

Let p be a procedure with no input or output parameters. We say that p has a *terminating execution* if and only if there is a model M, label l, and variable evaluations σ, σ' such that $P \vdash (l : \textbf{return})$ and $P, M \vdash (Start(p), \sigma) \rightarrow^* (l, \sigma')$. We say that p has a *b-terminating execution* for some $b > 0$ if and only if there is a model M, label l, and variable evaluations σ, σ' such that $P \vdash (l : \textbf{return})$ and $P, M \vdash (Start(p), \sigma) \rightarrow_b^* (l, \sigma')$. Using these definitions, we can define the following two decision problems.

Reachability Modulo Theories. Given a program P and a procedure p in the program with no input or output parameters, return YES if p has a terminating execution and return NO otherwise. We use $\text{RMT}(P, p)$ to denote an instance of this problem. If P is hierarchical, the problem is referred to as the hierarchical reachability modulo theories problem.

Bounded Reachability Modulo Theories. Given a program P, a procedure p in the program with no input or output parameters, and a bound $b > 0$, return YES if p has a b-terminating execution and return NO otherwise. We use $\text{RMT}(P, p, b)$ to denote an instance of this problem.

Fig. 3 shows a simple program P for which $\text{RMT}(P, main)$ and $\text{RMT}(P, main, 100)$ holds, but $\text{RMT}(P, main, b)$ does not hold for $b < 100$.

3 Complexity of the RMT Problem

The RMT problem is undecidable in general. The presence of recursion along with say, linear arithemetic in expressions, is enough to encode Turing-powerful computations. However, the bounded and hierarchical RMT problems are decidable. This section first gives an algorithm for deciding bounded RMT and then

```
procedure main() {
   call bar(0);
}

procedure bar(i: int) {
  if (i < 100) {
    i := i + 1;
    call bar(i);
  }
}
```

Fig. 3. An example program over *Boolean* and *Integer* sorts. Structured command `if` is used for convenience and can be easily compiled to labeled blocks.

refines the complexity analysis depending on the choice of the expression language. We first consider the special case of call-free single-procedure programs. Deciding RMT in this case can be reduced to the satisfiability of a single expression (which is decidable) through a process called Verification Condition (VC) generation.

3.1 Verification-Condition Generation

Let Expr be the expression language and P be a program consisting of a single call-free procedure f. The VC generation algorithm converts the body of f to a *Boolean* expression $VC(f)(\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{t})$ such that $\boldsymbol{i}$ is the list of all input parameters to the procedure, $\boldsymbol{o}$ is the list of all output parameters, and $\boldsymbol{t}$ are some temporary variables. We call the tuple $(\boldsymbol{i}, \boldsymbol{o})$ the interface variables of f. The expression $VC(f)$ satisfies two important properties. First, $VC(f)$ is an expression in Expr. Second, $VC(f)$ is satisfiable if and only if some execution of f starting in state $\boldsymbol{i}$ can return with state $\boldsymbol{o}$. Therefore, $\mathrm{RMT}(P, f)$ can be decided by checking the satisfiability of $VC(f)$.

We now describe a standard VC generation algorithm [10]. The first step is to *passify* f by converting all commands to **assume** statements. Consider the single procedure shown on the left of Fig. 4. Its passified version is shown on the right of the figure. Passification can be done in two steps: first, do *single-static assignment* (SSA renaming) [28] by introducing fresh variable incarnations so that each variable has at most one assignment. Next, push the ϕ functions to their definitions. For instance, the SSA renaming of Fig. 4 would create the statement `x4 := ϕ(x2,x3)` at label `l3` where control-flow merges. This statement can be realized by instead having the statements **assume** `x4 = x2` and **assume** `x4 = x3` right after the definitions of `x2` and `x3`, respectively. Next, as is standard, an assignment `x := e` after SSA renaming can be replaced by **assume** `x = e`, and **havoc** statements can be dropped (assuming that uninitialized variables are unconstrained). Finally, we add an **assume** statement at each **return** stating that the output parameters are constrained to the appropriate incarnation. This results in the passified version shown in Fig. 4 (right). Note that passification requires Expr to be closed under $=$.

Once the body of procedure f is passified, it can be converted to $VC(f)$ as follows. Let l be an arbitrary block label in f. We define $C(l)$ to be the

```
procedure f(w: int)
   returns (x: int, y: int, z: int)
{
start:
   havoc x;
   y := x + w;
   goto l1, l2;

l1:
   x := x + 1;
   goto l3;

l2:
   x := x + 2;
   goto l3;

l3:
   assume !(x > y);
   return;
}
```

```
procedure f(w: int)
   returns (x: int, y: int, z: int)
{
   var x0, x1, x2, x3, x4: int;
   var y0, y1: int;
   var z0: int;

start:
   assume y1 = x1 + w;
   goto l1, l2;

l1:
   assume x2 = x1 + 1;
   assume x4 = x2;
   goto l3;

l2:
   assume x3 = x1 + 2;
   assume x4 = x3;
   goto l3;

l3:
   assume !(x4 > y1);
   assume (x = x4 && y = y1 && z = z0);
   return;
}
```

Fig. 4. A program and its passified version

conjunction of the expressions in the assume statements in the block. We also define $\texttt{Succ}(l)$ to be the set of successor labels of block l if l ends with a **goto** and the empty set otherwise. We create a set of fresh *Boolean* variables $\{\ldots, b_l, \ldots\}$, one for each block l in f. We define the equation $E(l)$ of a block l as $b_l = C(l)$ if the block ends with a **return** and $b_l = c(l) \wedge \bigvee_{n \in \texttt{Succ}(l)} b_n$ otherwise. Then we get the following expression for $VC(f)$:

$$VC(f) = b_e \wedge \bigwedge_l E(l)$$

The expression $VC(f)$ refers to input and output variables of f; the temporary variables are the incarnation variables created for the SSA renaming. As an example, for the program in Figure 4, the block equations are as follows:

$$\begin{aligned} b_{start} &\equiv y1 = x1 + w \wedge (b_{l1} \vee b_{l2}) \\ b_{l1} &\equiv x2 = x1 + 1 \wedge x4 = x2 \wedge b_{l3} \\ b_{l2} &\equiv x3 = x1 + 2 \wedge x4 = x3 \wedge b_{l3} \\ b_{l3} &\equiv \neg(x4 > y1) \wedge x = x4 \wedge y = y1 \wedge z = z0 \end{aligned}$$

We capture the correctness of VC generation in the following lemma.

Lemma 1. *Let P be a call-free program and p be a procedure in P. Then the answer to* $\mathrm{RMT}(P, p)$ *is Yes iff $VC(p)$ is satisfiable.*

The above VC generation algorithm shows that RMT for call-free programs is decidable. This result can be extended to the hierarchical RMT problem for

arbitrary (non-recursive) programs: one can simply inline all procedures to obtain a single call-free procedure (because there is no recursion) and then decide reachability using VC generation. We call this algorithm the *static inlining* algorithm. This algorithm is easily extended to to the bounded RMT problem as well. Given a program P with a procedure p and bound b, it is possible to create a recursion-free program P' with a procedure p' such that $\mathrm{RMT}(P, p, b)$ is equivalent to $\mathrm{RMT}(P', p')$. Therefore, bounded RMT is decidable as well. The decision procedure for bounded RMT also implies that RMT is recursively enumerable: one can start with $b = 1$ and increment b until a witness for RMT is found.

Theorem 1. *Let P be a program and p be a procedure in P. Then* $\mathrm{RMT}(P, p, b)$ *is decidable and* $\mathrm{RMT}(P, p)$ *is recursively enumerable.*

It is worth contrasting the $\mathrm{RMT}(P, p)$ problem with a different $UMT(P, p)$ problem which asks the question where there is no terminating execution of procedure p in program P. Let us consider Expr such that $\mathrm{RMT}(P, p)$ is undecidable (easy as soon as either arithmetic or uninterpreted sorts are introduced). If $UMT(P, p)$ is recursively enumerable, then we can use Theorem 1 to conclude that $\mathrm{RMT}(P, p)$ is decidable which would be a contradiction. Therefore, $UMT(P, p)$ is neither decidable nor recursively enumerable. The $UMT(P, p)$ problem captures the problem definition being solved by software model checkers whose goal is to discover program proofs automatically. Intuitively, it appears that searching for a proof is more difficult than searching for a feasible path and an RMT solver is solving an "easier" problem than the one being solved by a software model checker.

3.2 Complexity of Hierarchical RMT

In this section, we demonstrate certain complexity results for the hierarchical RMT problem. As discussed earlier, the hierarchical RMT problem is decidable. We can refine the complexity analysis further by restricting the sorts in Expr.

Theorem 2. *If checking satisfiability of Boolean expressions in* Expr *is decidable in* NP*, then hierarchical RMT is decidable in* NEXPTIME*.*

Proof. Let D be a non-deterministic machine that does satisfiability of expressions in polynomial time. Let P be a non-recursive program. Then the length of any execution σ of P will be at most exponential in the size of P. Construct a non-deterministic machine M that guesses an execution σ_w of P, then rewrites it as straightline program P_w. (The size of P_w is at most exponential in the size of P.) Next, M does VC generation on P_w to obtain a single expression e_w and feeds it to D. M says that $\mathrm{RMT}(P)$ holds if and only if D says "satisfiable". It is easy to see that M solves $\mathrm{RMT}(P)$ in time at most exponential in the size of P.

The upper bound of NEXPTIME can be tightened if we restrict programs to use only the *Boolean* sort. When we only allow the *Boolean* sort, then we end up with the class of programs commonly known as *Boolean programs* [8, 18] that have been extensively studied in the literature.

Theorem 3. *If* Expr *is quantifier-free and contains only Boolean sort, hierachical RMT is* PSPACE*-complete.*

Proof. It is known that reachability for recursion-free Boolean programs is PSPACE-complete [1]; this result is enough to show membership in PSPACE. We only need to show that the problem is PSPACE-hard. We do that by reduction from the PSPACE-complete problem of checking whether there is a path from an initial state to a bad state in a transition system over a vector $\boldsymbol{x}$ of n *Boolean* variables. Let *Init* be the predicate representing the set of initial states, *Good* the predicate representing the set of good states, and *Trans* the transition relation. We construct a program P with procedure p

```
procedure p() {
  var y;
  assume Init(y);
  call y := p_0(y);
}
```

procedure p_i for $i \in [0, n)$

```
procedure p_i(x) returns (y) {
  y := x;
  call y := p_{i+1}(y);
  ...
  call y := p_{i+1}(y);
}
```

and procedure p_n

```
procedure p_n(x) returns (y) {
  assert Good(x);
  assume Trans(x, y);
}
```

The desired problem is $RMT(P, p)$. A transition system over n *Boolean* variables can have non-repeating paths of length at most 2^n; thus, the executions of p encode all non-repeating paths of the input program. The procedure p_n uses the assert statement; we show in Section 6 that assert statements can be compiled away with at most a linear cost.

Theorem 4. *If* Expr *is quantifier-free and contains only uninterpreted sorts (in addition to Boolean sort), hierarchical RMT is* NEXPTIME*-complete.*

Proof. Checking satisfiability of quantifier-free first-order logic is decidable in NP. Therefore, Theorem 2 gives membership in NEXPTIME. To show hardness, we demonstrate a polynomial-time reduction from the satisfiability problem for the EPR fragment of first-order logic. This fragments is given as $\exists \boldsymbol{x}.\ \forall \boldsymbol{y}.\ \varphi(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} = x_1, \ldots, x_m$ and $\boldsymbol{y} = y_1, \ldots, y_n$ and φ refers only to uninterpreted relation symbols. The problem of checking satisfiability of EPR formulas is known to be NEXPTIME-complete [27]. The decision procedure is straightforward. Skolemize $\boldsymbol{x}$ and then create a ground formula by taking the conjunction of φ

for all possible instantiations of $\boldsymbol{y}$ using only the skolem constants. The resulting ground formula is exponentially larger and equisatisfiable to the original. We show how to encode this decision procedure using a polynomial-size hiearchical RMT problem over a single uninterpreted sort S.

Let P be a hierachical program constructed as follows. First, declare m constants named $x_1, \ldots, x_m$ each of sort S. Next, declare procedures $p_0, p_1, \ldots, p_n$ such that procedure p_i has i input parameters each of sort S and no output parameters. Finally, define procedures p_i for $i \in [0, n)$

```
procedure p_i(v) {
  call p_{i+1}(v, x_1);
  ...
  call p_{i+1}(v, x_m);
}
```

and procedure p_n

```
procedure p_n(v) {
  assume φ(x, v);
}
```

The desired RMT problem is $\mathrm{RMT}(P, p_0)$.

4 Encoding Verification Problems

The Boogie language [9] is a concrete instance of an RMT language. In particular, the expression language of Boogie consists of the usual SMT theories (uninterpreted functions, theory of arrays, etc.) whose quantifier-free subset can be decided in NP using SMT solvers. The presence of such an expressive language offers a convenient way to encode many software verification tasks.

In this section, we briefly survey past effort on compiling programs in languages such as C and C# down to Boogie. Instead of rigorously describing the compilers, we illustrate using examples how source-level features are modeled in Boogie using decidable theories. More details can be obtained from the original papers on HAVOC, for C to Boogie [22] and the ByteCode Translator (BCT), for C# to Boogie [11].

Fig. 5 shows the encoding of a simple C program in Boogie, focusing on the treatment given to pointers and the heap in C. The HAVOC tool treats a pointer as simply an `int`. Memory allocation happens through a special variable called `alloc` that monotonically increases, as captured by the procedure `malloc`. It is easy to verify that successive calls to `malloc` will return distinct pointers.

The heap is modeled using arrays. Conceptually, the entire heap can be encoded using a single map `Mem` of type `int` $\rightarrow$ `int`, and each dereference `*x` can be translated to `Mem[x]`. However, for efficiency reasons, HAVOC splits the `Mem` map to multiple maps, one for each type and field, assuming certain type safety conditions on the program [14]. In Fig. 5, the use of two maps, one for field `f` and the other for `g` statically encodes the non-aliasing constraint that `x->f` cannot alias `y->g`, irrespective of the values of `x` and `y`. Such a constraint enables local reasoning.

HAVOC supports the option of encoding pointers using bitvectors as well. In that case, arithmetic operations are compiled to bitvector operations that

```
struct S {
  int f;
  int g;
};

void main() {
   S *x = malloc(sizeof(S));
   S *y = malloc(sizeof(S));
   x->f = 1;
   y->g = 2;
   assert(x->f == 1);
}
```

```
var Mem.f_S: [int]int;
var Mem.g_S: [int]int;

var alloc: int;

procedure malloc(size: int)
       returns (ptr: int) {
  var old_alloc: int;

   assume size >= 0;
   old_alloc := alloc;
   havoc alloc;
   assume alloc >
    old_alloc + size;
   ptr := alloc;
}

procedure main() {
   var x: int;

   assume alloc > 0;

   call x := malloc(8);
   call y := malloc(8);

   Mem.f_S[x] := 1;
   Mem.g_S[y] := 2;

   assert Mem.f_S[x] == 1;
}
```

Fig. 5. A C program (left) and the corresponding compiled Boogie program (right)

still fall under the QFBV theory (quantifier-free bit-vectors) supported by SMT sovlers.

The SMACK compiler [32] is another tool for compiling C programs to Boogie. It has the option of using a memory model where pointers are not `int`s but rather a type that encapsulates the actual pointer value and meta-data such as the block of memory the pointer belongs to and the size of that block. Such a memory model allows asserting of memory safety (i.e., every pointer dereference is inside allocated memory).

The compilation of a C# program to Boogie, using BCT, also encodes the heap using a series of maps, one for each field declared in the program. However, C# being a higher-level language than C, BCT has to model several other features of C#. For instance, the sub-typing relation can be encoded using a series of axioms (written in Boogie syntax):

```
// a type for C# types
type Type;

// an uninterpreted function
function SubType(Type,Type): bool;

// whenever A inherits from B
axiom SubType(A, B);
```

```
// reflexive, transitive, anti-symmetric
axiom forall t: Type :: SubType(t, t);
axiom forall t1, t2, t3: Type :: SubType(t1, t2) && SubType(t2, t3
    ) ==> SubType(t1, t3);
axiom forall t1, t2: Type :: t1 != t2 && SubType(t1, t2) ==> !
    SubType(t2, t1)
```

Axioms are structural constraints that are assumed to hold in any valid program state. Such an encoding of subtyping, even though it uses quantifiers, falls under the *effectively propositional* class of formulas [30] that turns out to be decidable.

The use of SMT theories allows Boogie to capture many verification tasks, but there are still certain important aspects of software that are hard to model. For example, there is no easy way to encode floating-point computation or string operations in a decidable theory. One can use quantifiers or recursive procedures to model string operations, but it remains to be seen if this leads to an effective end-to-end solution for string-manipulating programs.

5 Stratified Inlining

Section 4 shows that even the most common software verification tasks requires the use of linear arithmetic, uninterpreted functions and maps. For these problems, the bounded RMT problem is NEXPTIME-hard (Section 3) and one cannot hope for a better algorithm than static inlining, in the worst case. On one hand, it is unlikely that a polynomially-sized formula captures a bounded RMT problem; on the other hand, static inlining is inefficient on practical problems. In an experiment on safety verification of device drivers [25], we found that static inlining ran out of memory during VC generation. Even when the VC did fit in memory, the SMT solver (Z3) was overwhelmed by its size and timed out in many instances.

This section presents the *stratified inlining* (SI) algorithm for solving the bounded RMT problem with respect to a bound $b > 0$. SI tries to delay the construction of an exponentially-sized formula as much as possible, in hope of efficiently solving RMT for most programs. Instead of inlining all procedures upfront, SI inlines procedures on-demand, in a goal-directed manner. Experiments validated stratified inlining to be much more efficient than static inlining in practice [25].

Overview. SI works as follows: at any point in time, SI maintains a partially-inlined program P, along with the set of call-sites C of P that have not been inlined so far. Initially, P is `main` and C is the set of all call-sites in `main`. Next, it queries the theorem prover to see if P has a valid execution of `main` that does not go through any call-site in C. If so, it returns this execution. If not, then it queries the theorem prover again, this time allowing executions to go through C and simulating the effect that open call-sites can modify state arbitrarily. This query represents an over-approximation of the input program. If no valid

```
procedure main() {
   var i: int;
   i := 0;
   if( * )
     call i := foo(i);
   else
     call i := bar(i);
   assume i >= 5;
}
```

```
procedure foo(i: int)
     returns (i: int) {
   if( * ) {
     i := i + 1;
     call i := foo(i);
     call i := bar(i);
   }
}
```

```
procedure bar(i: int)
     returns (i: int) {
   if( * ) {
    i := 2*i;
    call i := bar(i);
   }
}
```

Fig. 6. An example program

```
procedure main() {
   var i1,i2,i3,i4: int;
   assume i1 = 0;
   if( * )
     call i2 := foo(i1);
     assume i4 = i2;
   else
     call i3 := bar(i1);
     assume i4 = i3;
   assume i4 >= 5;
}
```

```
procedure main() {
   var i1,i2,i3,i4: int;
   var c1,c2: bool;

   assume i1 = 0;
   if( * )
     assume c1;
     assume i4 = i2;
   else
     assume c2;
     assume i4 = i3;
   assume i4 >= 5;
}
```

```
c1 ↦ (foo, (i1, i2))
c2 ↦ (bar, (i1, i3))
```

Fig. 7. The passified version of `main` of Fig. 6; replacing procedure calls with fresh Boolean constants; and the mapping between such constants and the input-output variables of the corresponding procedure call

execution of P is found, then the original program is safe, i.e., RMT does not hold. If there is a valid execution of P, then it must go through some call-sites in C. These call-sites are inlined, provided they are under the recursion bound, and the process continues. We now describe this process in more detail.

The VC generation algorithm used by SI is similar to the one described in Section 3, with slight modifications to handle procedure calls. Given a passified procedure `f`, we replace each procedure call with **assume** c for a fresh Boolean constant c, and then do the VC generation as usual. An example is shown in Fig. 7. In this case: (1) constraining c to **false** blocks executions that go through the call, underapproximating the behaviors of the call, and (2) constraining c to **true** allows executions in which the return values of the call can be arbitrary. For example, in Fig. 7, if `c1` is **true** then there is no constraint between `i1` and `i2`, i.e., the call to `foo` could return any output. This represents an overapproximation to the call. When these Boolean constants are introduced, we also record the mapping between them and the input-output variables of the calls that they replace, as shown on the right of Fig. 7.

For a procedure `f`, let VcGen(`f`) be a tuple $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ such that $\phi(\boldsymbol{io}, \boldsymbol{t})$ is the VC of `f`, $\boldsymbol{io}$ are the interface (input, output) variables of `f`, $\boldsymbol{t}$ are some internal variables and d is a map from Boolean variables to information about

Procedure INIT(`main`)

1: Let $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ = VCGEN(`main`)
2: CallTree := (`main`)
3: Assert $\phi(\boldsymbol{io}, \boldsymbol{t})$
4: **for all** $c \mapsto (\mathtt{f}, \boldsymbol{io'})$ **in** d **do**
5: Create edge $(\mathtt{main}, (\mathtt{f}, \boldsymbol{io'}, c))$ in CallTree
6: **end for**

Procedure INSTANTIATE(Node n)

1: Let $(\mathtt{f}, \boldsymbol{io'}, c) = n$
2: Let $(\phi(\boldsymbol{io}, \boldsymbol{t}), d)$ = VCGEN(`f`)
3: Let $\boldsymbol{t'}$ be fresh variables
4: Assert $c \Rightarrow \phi(\boldsymbol{io'}, \boldsymbol{t'})$
5: Let $d' = d[\boldsymbol{t'}/\boldsymbol{t}][\boldsymbol{io'}/\boldsymbol{io}]$
6: **for all** $c \mapsto (\mathtt{b}, \boldsymbol{v})$ **in** d' **do**
7: Create edge $(n, (\mathtt{b}, \boldsymbol{v}, c))$ in CallTree
8: **end for**

Fig. 8. Procedures for initializing and growing the CallTree

Procedure QUERYUNDER()

1: Push
2: **for all** leaves $l = (\_, \_, c)$ **do**
3: Assert $\neg c$
4: **end for**
5: Check
6: **if** Satisfiable **then**
7: **return** "RMT$_b$ holds"
8: **end if**
9: Pop

Procedure QUERYOVER()

1: Push
2: **for all** leaves $l = (\_, \_, c)$ **do**
3: **if** $\mathtt{RB}(l) > b$ **then**
4: Assert $\neg c$
5: **end if**
6: **end for**
7: Check
8: **if** Unsatisfiable **then**
9: **return** "RMT_b does not hold"
10: **end if**
11: Let τ be the error trace
12: Pop
13: **for all** leaves l on τ **do**
14: INSTANTIATE(l)
15: **end for**

Fig. 9. Querying the theorem prover with under- and over-approximations

the procedure calls that they replaced. For Fig. 7, d(`c1`) = `(foo, (i1, i2))` and d(`c2`) = `(bar, (i1, i3))`.

The SI algorithm maintains a partially-inlined program in the form of a tree, called the *CallTree*. Nodes of the tree represent a dynamic instance of a procedure and children of a node are the procedures called by that node. Thus, the *CallTree* is a partial unrolling of the call graph of the program. Internal nodes of the tree are all the procedure that have been inlined so far by SI, and leaves represent non-inlined procedure calls. We also use the term "open call-sites" to refer to leaves or the non-inlined calls. SI maintains the invariant that at any time, the VCs of all internal nodes are asserted in the theorem prover stack. All nodes in the *CallTree*, except the root node, are a triple $(\mathtt{f}, \boldsymbol{io}, c)$ where `f` is the name of the procedure, $\boldsymbol{io}$ are the input-output variables of this particular dynamic instance of `f`, and c is the unique Boolean variable that substituted the call to

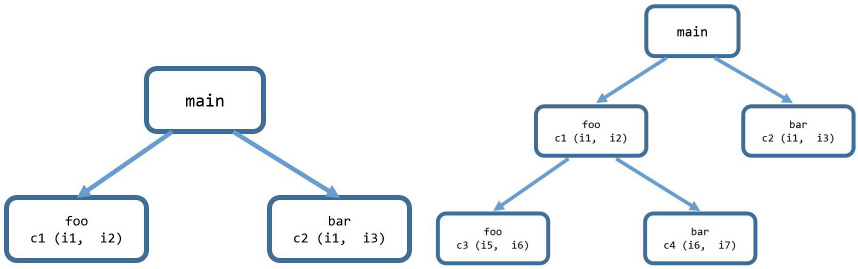


Fig. 10. The shape of the *CallTree* after initialization (left) and after instantiating `foo` (right)

`f` from its parent node during VC generation. SI uses the standard SMT solver API:

- Push: Creates a backtracking point.
- Pop: Undo all asserted formulae until the most recent backtracking point.
- Assert: Add (conjoin) a formula to already asserted formulae.
- Check: Check for satisfiability of asserted formulae.

The procedures Init and Instantiate, shown in Fig. 8, initialize and grow the *CallTree*, respectively. Init takes the name of the starting procedure (`main`) and creates a tree with root labeled `main` and one leaf for each procedure called by `main`. Fig. 10 shows the initial tree for our running example of Fig. 6. Instantiate takes a leaf node and inlines the procedure represented by that node. It does so by generating the VC (line 2), renaming the interface variables and asserting the VC (line 4). The asserted formula says that if c is **true**, then the constraint imposed by `f` must be satisfied. Next, we then create new leaves for all callees of `f` (line 7).

Given a *CallTree*, SI makes two kinds of queries, shown in Fig. 9. QueryUnder tries to see if RMT_b holds: it first blocks all open call-sites (line 3) and then checks if the currently asserted formula is satisfiable (line 5). If so, then we have found a valid program execution (because it only goes through inlined calls) and RMT_b holds. QueryOver tries to see if RMT_b does not hold. First, it blocks all open call-sites whose recursion bound exceeds b (line 4). The sub-routine `RB` takes a leaf node, say $l = (\texttt{f}, \_, \_)$ and simply counts the number of instances of `f` along the path from l to the root. (`RB` simulates the *Count* function of Figure 2.) It is easy to see that this count is the number of times `f` must appear on the call-stack when execution reaches l. For example, `RB` of the leaf node `foo` in Fig. 10 is 1. If the check on line 7 is satisfiable, then we use the model to construct the corresponding program execution. Next, we inline all open calls on this path using Instantiate (line 13). Note: (1) QueryOver will never inline a leaf that has crossed the recursion bound b because such open calls are blocked (line 4), and (2) the blocking of open calls in both QueryUnder and QueryOver is nested inside Push-Pop operations, hence this blocking does not persist beyond a single query.

We can now write down the full SI algorithm. It simply calls the two queries in alternation, until one of them returns "RMT_b holds" or "RMT_b does not hold".

```
1: Init(main)
2: while true do
3:    QueryUnder();
4:    QueryOver();
5: end while
```

The SI algorithm is guaranteed to terminate because each iteration of the while loop, if it is not the last iteration, must grow the *CallTree*. This is because when QueryUnder is not able to find a path within the inlined part, it must be that the trace τ on line 11 of QueryOver passes through some open call. Moreover, the size of *CallTree* is bounded because of the recursion bound. Hence, SI will terminate in bounded time. SI makes at most exponential number of queries on formulas that are atmost exponential in the size of the program. Asymptotically, SI has the same complexity as static inliining.

Related Work. Stratified Inlining draws inspiration from multiple sources. Previous work on *structural abstraction* [4] and *inertial refinement* [33] has similarities with SI. However, work on structural abstraction does not use an underapproximation-based query (by blocking open call sites). Inertial refinement does use both over and under approximations to iteratively build a view of the program that is then analyzed. A distinguishing factor is our use of recursion bounding as well as using lazy inlining to construct a single VC for the entire program view.

It has been illustrated in the SMT community that dealing with eager instantiation of either theory lemmas or quantifiers (e.g. as done in UCLID tool [24]) does not provide the most scalable way to reason about SMT. Instead lazy instantiation tends to scale much better. Similarly, we believe that lazy approaches like SI have much better chance of being successful than full static inlining.

6 The Corral Solver

The Corral tool is a practical realization of a solver for bounded RMT. It is designed for the Boogie programming language. It takes a Boogie program (with assertions) as input, the name of the starting procedure (`main`) and a recursion bound. The assertions in the input program are removed using a source-to-source transformation to obtain a usual bounded RMT problem as follows.

- Introduce a Boolean variable `error` and initialize it to **false** at the entry to `main`.
- Replace **assert** e with `error := `e`; if(error) return`.
- After each procedure call, insert **if** `(error) return`.
- At the exit of `main`, assume that `error` is **true**.

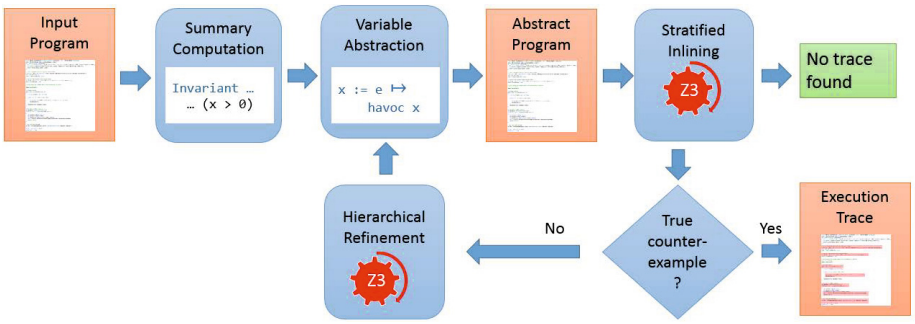


Fig. 11. Corral's architecture

After such a transformation, there is an execution that exits `main` if and only if the original program had a failing assertion.

As output, Corral can either return a feasible path that ends in an assertion violation (i.e., RMT holds), or a message saying that no such path exists (i.e., RMT does not hold), or a message saying that no such path was found within the recursion bound (i.e., bounded RMT does not hold, but no conclusion can be made for unbounded RMT). Corral was the subject of a previous publication [25]. This paper only briefly covers the ideas and algorithms behind Corral.

The design of Corral is shown in Fig. 11. The main component of Corral is the stratified inlining (SI) algorithm that was described in Section 5. Instead of directly giving the input RMT problem to SI, Corral uses two optimizations to reduce the computational burden on SI.

The first optimization is to compute program invariants. In principle, any technique for invariant generation may be used. Corral uses the Houdini algorithm [17] to compute invariants in the form of procedure summaries. The user provides, as additional input to Corral, *candidate* expressions for procedure summaries. Houdini uses theorem prover queries, each on the VC of at most one procedure, to compute the strongest inductive summaries within the given candidates. The number of single-procedure queries is quadratic in the worst case but linear in the common case. The invariants, once computed, are injected back into the program as assume statements. These invariants can help SI prune search because they (soundly) constrain the over-approximate query used by SI, which can rule out many abstract counterexamples.

The second optimization is an abstraction-refinement loop. Corral uses a very simple abstraction in this loop, called *variable abstraction*. Let G be the set of global variables of the input program. Note that G is always a finite set. Variable abstraction is parameterized by a set T of *tracked* variables, where $T \subseteq G$. Variable abstraction works by abstracting away all variables in $G - T$, using a simple source rewriting. For instance, the assignment `x := e`, where `x` $\in T$ and the expression e has some variable in $G - T$, is re-written to **havoc** `x`. The abstracted program is fed to SI. This abstraction can lead to SI returning spurious counterexamples. These counterexamples are used to refine the abstraction by

increasing T. The variable-abstraction based refinement loop substantially differs from one based on predicate abstraction that is used by most software model checking tools: (1) variable abstraction is easy to compute, unlike predicate abstraction that may need an exponential number of SMT queries, and (2) the refinement loop of variable abstraction is bounded because T is bounded above by G, unlike predicate abstraction where the number of predicates is unbounded.

The abstraction-refinement loop of Corral is useful when only a few variables are relevant in solving the RMT problem. Abstracting away variables considerably reduces the size of the program and allows the theorem prover to focus only on the relevant part of the program's data. Moreover, VC generation is quadratic in the number of variables, and a reduction in the number of variables also significantly decreases the VC size.

6.1 Experience Using Corral

Corral is ideally suited for applications where one is more interested in finding bugs than in finding proofs, or when finding proofs is simply too difficult. We now describe our experience with such applications.

The first application is the Static Driver Verifier (SDV), a product supported by the Driver Quality team of Microsoft Windows. SDV supports any driver written using one of four diferent driver models: WDM, KMDF, NDIS, STORPORT. It also comes with a list of rules (or properties) that drivers must satisfy. A driver and rule pair forms a *verification instance* that is fed to the verification engine. SDV has traditionally used Slam as the verification engine. After a comprehensive evaluation, a dual-engine system of Corral and Yogi [20, 29] will replace Slam inside SDV in the next release of the Microsoft Windows operating system. Going into the details of the evaluation is outside the scope of this paper. We briefly present our experience with Corral in comparison to Slam.

Corral is executed with a modest recursion bound of 3 to 6, depending on the driver model. In an initial study [25], this bound was sufficient to find all but 9 defects in a test suite containing around 400 defects in total. The missed defects were due to loops with a constant upper bound, for example, the loop **for**`(int i = 0; i < 27; i++)` requires a bound of at least 27 before code after the loop is reachable. We designed custom techniques to deal with such loops, after which Corral was able to find almost all defects reported by Slam. Furthermore, Corral has 2.5X reduction in timeouts and 40% improvement in running time.

While Corral was able to overtake Slam in the number of defects found, it was also important to compare the number of instances that were proved correct. This will indicate a measure of confidence that Corral will not miss defects in yet unseen drivers. Along with each verification instance, we also supplied summary candidates for Houdini. The candidates are derived heuristically looking solely at the property (not the driver) being verified. On an initial test suite, Corral was able to prove correctness in 91% of the cases (regardless of the recursion bound), with summaries inferred by Houdini playing an important role

in establishing proofs. This means that for most part our heuristically-generated invariant candidates were sufficient.

We obvserved similar speedups against the YOGI tool as well. We now list some other lessons learned from these comparisons.

- While SLAM and YOGI were designed and trained on drivers for SDV, CORRAL was initially designed for finding concurrency bugs. The fact that CORRAL performed well inside SDV demonstrates a degree of robustness of CORRAL in handling programs from multiple sources.
- SLAM avoided using array theory to model the heap, and instead relied on a *logical* memory model [5]. This decision was justified because implementations of array-theory might have been inefficient ten years ago. YOGI inherited a similar memory model as SLAM. CORRAL, on the other hand, makes heavy use of array theory because the entire heap is modeled using maps. Thus, as theorem-prover technology changes, it is reasonable to expect the design of software verifiers to change as well.
- We found CORRAL to be much more dependent on the performance of Z3 than YOGI. For instance, upgrading Z3 almost always resulted in a significant speedup of CORRAL, whereas, we did not observe such speedups in YOGI. In retrospect, this is not surprising because YOGI was designed to avoid invoking the theorem prover to the greatest extent possible.

The second class of applications for CORRAL are what are now called *sequentializations* of concurrent programs. The original sequentialization was a program transformation that converted a safety property on a multi-threaded concurrent program to a safety property on a sequential program, given a bound on the number of context switches between threads [31, 26]. Subsequently, more sequentializations were proposed: for asynchronous task-buffer programs [16], for liveness properties of concurrent programs [15, 2], and for programs on weak-memory models [3]. In each of these cases, CORRAL was used as the solver for finding defects in the generated sequential program. These applications have two common aspects. First, they require bounding the set of program behaviors (e.g., context switches). Thus, verifying correctness cannot be a goal. Second, the generated sequential programs are complicated and so are their invariants. Consequently, software verifiers like SLAM do not perform well on such programs. On the other hand, CORRAL, which builds off the robustness of SMT solvers, is able to work well uniformly across such programs.

CORRAL has been used to automatically detect security vulnerabilities in models of web applications. In one study, Cashier-as-a-Service web payment systems were modeled in the C language; CORRAL was used to find vulnerabilities that would allow an attacker to shop for free [34]. In another study, authentication and authorization SDKs were modeled in the C# language; CORRAL was used to find improper use of SDK APIs leading to insecure access [35].

Besides these applications, Corral is also used inside a debugging tool for .NET, called GetMeHere, where it is able to successfully operate on Boogie programs compiled from C#. We also evaluated Corral on the Software Verification Competition (SV-COMP) benchmarks and obtained favorable results compared to all the other tools participating in the competition. More details are available in the original Corral paper [25].

References

[1] Alur, R.: Formal analysis of hierarchical state machines. In: Dershowitz, N. (ed.) Verification (Manna Festschrift). LNCS, vol. 2772, pp. 42–66. Springer, Heidelberg (2004)

[2] Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012)

[3] Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)

[4] Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)

[5] Ball, T., Bounimova, E., Levin, V., de Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research (2010)

[6] Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7), 68–76 (2011)

[7] Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Language Design and Implementation (2001)

[8] Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN, pp. 113–130 (2000)

[9] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)

[10] Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis for Software Tools and Engineering (2005)

[11] Barnett, M., Qadeer, S.: BCT: A translator from MSIL to Boogie. In: Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (2012)

[12] Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT (2012)

[13] Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)

[14] Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (2009)

[15] Emmi, M., Lal, A.: Finding non-terminating executions in distributed asynchronous programs. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 439–455. Springer, Heidelberg (2012)

[16] Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: Foundations of Software Engineering (2012)
[17] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
[18] Godefroid, P., Yannakakis, M.: Analysis of boolean programs. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 214–229. Springer, Heidelberg (2013)
[19] Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
[20] Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Foundations of Software Engineering (2006)
[21] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages (2002)
[22] Lahiri, S., Qadeer, S.: Back to the future: Revisiting precise program verification using SMT solvers. In: Principles of Programming Languages (2008)
[23] Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 493–508. Springer, Heidelberg (2009)
[24] Lahiri, S.K., Seshia, S.A.: The UCLID decision procedure. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 475–478. Springer, Heidelberg (2004)
[25] Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
[26] Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1) (2009)
[27] Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Computer and System Sciences 21, 317–353 (1980)
[28] Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
[29] Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in YOGI. In: International Conference on Software Engineering, pp. 355–364 (2010)
[30] Piskac, R., de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reasoning 44(4), 401–424 (2010)
[31] Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Programming Language Design and Implementation, pp. 14–24 (2004)
[32] Rakamaric, Z., Emmi, M.: SMACK: Static Modular Assertion ChecKer, `http://smackers.github.io/smack`
[33] Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
[34] Wang, R., Chen, S., Wang, X., Qadeer, S.: How to shop for free online — security analysis of Cashier-as-a-Service based web stores. In: IEEE Symposium on Security and Privacy, pp. 465–480 (2011)
[35] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In: USENIX Security Symposium (2013)

The Dynamic Complexity of the Reachability Problem on Graphs*

Thomas Schwentick

Technische Universität Dortmund

Abstract. Many current data processing scenarios deal with about large collections of permanently changing data. In this context, it is often impossible to compute the answer for a query from scratch. Rather some auxiliary data needs to be stored that helps answering queries quickly, but also requires to be maintained incrementally. This incremental maintenance scenario can be studied in various ways, e.g., from the perspective of dynamic algorithms with the goal to reduce (re-) computation time. Other options are to study the scenario from the perspective of low-level parallel computational complexity [3] or parallelizable database queries [1]. As the "lowest" complexity class AC^0 (with a suitable uniformity condition) and the core of the standard database query language SQL both coincide with first-order predicate logic, one naturally arrives at the question which queries can be answered/maintained dynamically with first-order predicate logic (DynFO).

The most intensily studied query in this dynamic setting is the reachability query on graphs, arguably the "simplest recursive" query. It has been shown that it can be maintained in DynFO on undirected [3] or acyclic directed graphs [1]. However, whether it can be maintained on general directed graphs is considered the main open question of the field.

The talk will give an introduction into dynamic complexity, survey known results on the dynamic complexity of Reachability and report about more recent work on fragments of DynFO and their inability to express Reachability [2,4].

References

1. Dong, G., Su, J.: Incremental and decremental evaluation of transitive closure by first-order queries. Inf. Comput. 120(1), 101–106 (1995)
2. Gelade, W., Marquardt, M., Schwentick, T.: The dynamic complexity of formal languages. ACM Trans. Comput. Log. 13(3), 19 (2012)
3. Patnaik, S., Immerman, N.: Dyn-fo: A parallel, dynamic complexity class. J. Comput. Syst. Sci. 55(2), 199–209 (1997)
4. Zeume, T., Schwentick, T.: On the quantifier-free dynamic complexity of reachability. CoRR, abs/1306.3056 (2013)

* This work was supported by the DFG Grant SCHW678/6-1.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, p. 45, 2013.

Reachability Problems for Hierarchical Piecewise Constant Derivative Systems

Paul C. Bell and Shang Chen

Department of Computer Science, Loughborough University
{P.Bell,S.Chen3}@lboro.ac.uk

Abstract. In this paper, we investigate the computability and complexity of reachability problems for two-dimensional hierarchical piecewise constant derivative (HPCD) systems. The main interest in HPCDs stems from the fact that their reachability problem is on the border between decidability and undecidability, since it is equivalent to that of reachability for one-dimensional piecewise affine maps (PAMs) which is a long standing open problem. Understanding the most expressive hybrid system models that retain decidability for reachability has generated a great deal of interest over the past few years. In this paper, we show a restriction of HPCDs (called RHPCDs) which leads to the reachability problem becoming decidable. We then study which additional powers we must add to the RHPCD model to render it 1D PAM-equivalent. Finally, we show NP-hardness of reachability for nondeterministic RHPCDs.

1 Introduction

Hybrid automata are an important class of mathematical model allowing one to capture both discrete and continuous dynamics in the same framework. There is currently much interest in *hybrid systems* since they can be used to model many practical real world systems in which we have a discrete controller acting in a continuous environment and their analysis has a huge range of potential applications, such as aircraft traffic management systems, aircraft autopilots, automotive engine control [6], chemical plants [7] and automated traffic systems for example.

Hybrid systems are described by a state-space model given by the Cartesian product of a discrete and continuous set. The system evolves over time according to a set of defined rules until some condition or event is satisfied, at which point a discrete, non-continuous event occurs. Such an event can cause an update to certain variables and change the continuous dynamics of the continuous variables.

A fundamental question concerning hybrid systems is that of *reachability*: does there exist a trajectory starting from some initial state (or set of states) which evolves to reach a given final state (or set of states) in finite time? Related questions, such as *convergence* (does there exist a state (or periodic set of states) towards which the system converges for any initial state) or *control problems* (given an input, can the system be controlled to avoid some 'bad' set of

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 46–58, 2013.

states?), are also important, see [9]. In this paper we focus on *reachability*. Unfortunately, many reachability problems are *undecidable*, even for very restricted hybrid systems [2,5,8,10]. The objective of studying the decidability boundary is twofold; to obtain the most expressive system for which reachability is decidable and to study the simplest system for which it is undecidable.

An important and intuitive model of hybrid system is that of a *Piecewise Constant Derivative* (PCD) system. In this model, we partition the continuous state space into a finite number of nonempty regions, each of which is assigned a constant derivative defining the dynamics of a point within that region (see Section 2 for full details). It was proven in [12] that reachability for PCD systems in two-dimensions (2-PCD) is decidable, but for three-dimensions (a 3-PCD), the problem becomes undecidable [2]. One of the important properties of a PCD, which leads to its reachability problem being decidable in dimension 2, is that trajectories can never 'cross' each other since each region has a constant derivative assigned. It can be proven that the trajectories are either periodic, or else form an expanding or contracting spiral which can be proven using geometric arguments on the *edge-to-edge successor* function of a 2-PCD.

In [4], an intermediate model, called a *Hierarchical Piecewise Constant Derivative* (HPCD) system was introduced. Intuitively, this model of linear hybrid system can be thought of as a two-dimensional hybrid automaton where the dynamics in each discrete location is given by a 2-PCD (precise details are given in Section 2). Certain edges in locations of the HPCD are denoted as guards (which can be comparative) and lead to discrete location changes. When changing location, an affine reset rule may also be applied to the continuous variables. If all regions of the underlying PCDs are bounded, then the HPCD is called bounded. Clearly then, the model of HPCD seems more powerful than that of a 2-PCD. Indeed, the reachability problem for a one-dimensional Piecewise Affine Map (1-PAM) was shown to be equivalent to that of reachability for a bounded HPCD with either: i) comparative guards, identity resets and elementary flows in Proposition 3.20 of [3] or else ii) affine resets, non-comparative guards and elementary flows in Lemma 3.4 of [3] (See Section 2 for definitions).

Our reference model in this paper is called a *Restricted HPCD* (an RHPCD). An RHPCD is an HPCD with elementary flows, identity resets and non-comparative guards and is thus a simpler form of HPCD. We prove that reachability for an RHPCD is decidable. We also prove that a 1-PAM can also be simulated by an RHPCD with arbitrary constant flows or with *linear* resets, and is thus equivalent to an RHPCD with affine resets, see Table 1 for an overview of results.

In [13], the reachability problem for planar linear hybrid automata without resets is shown to be decidable, however they focus on the setting in which the flows are *monotonic*, meaning there exists some vector ρ such that the derivatives of all variables in all states have a positive projection along ρ. In dimension 4, the reachability problem becomes undecidable [13].

Table 1. RHPCD (starred results are contributions of this paper)

RHPCD	Infinite number of PCD regions	Linear resets	Affine resets	Comparative guards	Arbitrary constant flows	Number of locations	
Decidable	×	×	×	×	×	$N < \infty$	*
	×	×	×	✓	✓	1	[12]
1-PAM equivalent	×	×	×	×	✓	$\lceil \log_2 n \rceil + 3$	*
	×	×	×	✓	×	$4n$	[3]
	×	×	✓	×	×	1	[3]
	×	✓	×	×	×	$\lceil \log_2 n \rceil + 3$	*
Undecidable	✓	×	×	×	×	1	[3]

2 Preliminaries

Intervals of the form $(s,t), [s,t), (s,t], [s,t]$ are called open, half-open or closed bounded rational intervals (respectively), where $s,t \in \mathbb{Q}$. We write $\langle I, c\rangle$ to denote $\{(x,c)|x \in I\} \subseteq \mathbb{Q}^2$, where $I \subseteq \mathbb{Q}$ is an (open, half-open or closed) bounded rational interval and $c \in \mathbb{Q}$ is a constant. We similarly define $\langle c, I\rangle = \{(c,y)|y \in I\}$. By abuse of notation, for an interval $I = (s,t)$ where $s,t \in \mathbb{Q}$ and $s \leq t$, a function $f(x) : \mathbb{Q} \to \mathbb{Q}$ and a constant $m \in \mathbb{Q}$, we define $f(I) + m = (f(s) + m, f(t) + m)$. Similar definitions exist for half-open and closed intervals. We use similar definitions as [3] for the following.

Definition 1. (HA) *An n-dimensional Hybrid Automaton (HA) [1] is a tuple* $\mathcal{H} = (\mathcal{X}, Q, f, l_0, Inv, \delta)$ *consisting of the following components:*

(1) A continuous state space $\mathcal{X} \subseteq \mathbb{R}^n$. *Each* $\boldsymbol{x} \in \mathcal{X}$ *can be written* $\boldsymbol{x} = (x_1, \ldots, x_n)$, *and we use variables* $x_1, \ldots, x_n$ *to denote components of the state vector.*
(2) A finite set of discrete locations Q.
(3) A function $f : Q \to (\mathcal{X} \to \mathbb{R}^n)$, *which assigns a continuous vector field on* $\mathcal{X}$ *to each location. In location* $l \in Q$, *the evolution of the continuous variables is governed by the differential equation* $\dot{\boldsymbol{x}} = f_l(\boldsymbol{x})$. *The differential equation is called the dynamics of location* l.
(4) An initial condition $I_0 : Q \to 2^{\mathcal{X}}$ *assigning initial values to variables in each location.*
(5) An invariant Inv: $Q \to 2^{\mathcal{X}}$. *For each* $l \in Q$, *the continuous variables must satisfy the condition Inv(l) in order to remain in location* l, *otherwise it must make a discrete transition.*
(6) A set of transitions δ. *Every* $tr \in \delta$ *is of the form* $tr = (l, g, \gamma, l')$, *where* $l, l' \in Q$, $g \subset \mathcal{X}$ *is called the guard, defining when the discrete transition can occur,* $\gamma \subset \mathcal{X} \times \mathcal{X}$ *is called the reset relation applied after the transition from* l *to* l'.

An HA is *deterministic* if it has exactly one solution for its differential equation in each location and the guards for the outgoing edges of locations are mutually exclusive. A *trajectory* of a hybrid automaton $\mathcal{H}$ starting from $(l_0, \mathbf{x_0})$ where $l_0 \in Q, \mathbf{x_0} \in \mathcal{X}$ is a pair of functions $\pi_{l_0,\mathbf{x_0}} = (\lambda_{l_0,\mathbf{x_0}}(t), \xi_{l_0,\mathbf{x_0}}(t))$ such that

(1) $\lambda_{l_0,\mathbf{x_0}}(t) : [0, +\infty) \to Q$ is a piecewise function constant on every interval $[t_i, t_{i+1})$.
(2) $\xi_{l_0,\mathbf{x_0}}(t) : [0, +\infty) \to \mathbb{R}^n$ is a piecewise differentiable function and in each piece $\xi_{l_0,\mathbf{x_0}}$ is càdlàg (right continuous with left limits everywhere).
(3) On any interval $[t_i, t_{i+1})$ where $\lambda_{l_0,\mathbf{x_0}}$ is constant and $\xi_{l_0,\mathbf{x_0}}$ is continuous,

$$\xi_{l_0,\mathbf{x_0}}(t) = \xi_{l_0,\mathbf{x_0}}(t_i) + \int_{t_i}^{t} f_{\lambda_{l_0,\mathbf{x_0}}(t_i)}(\xi_{l_0,\mathbf{x_0}}(\tau))d\tau$$

for all $\tau \in [t_i, t_{i+1})$.
(4) For any t_i, there exists a transition $(l, g, \gamma, l') \in \delta$ such that
  (i) $\lambda_{l_0,\mathbf{x_0}}(t_i) = l$ and $\lambda_{l_0,\mathbf{x_0}}(t_{i+1}) = l'$;
  (ii) $\xi^-_{l_0,\mathbf{x_0}}(t_{i+1}) \in g(l, l')$ where $\xi^-_{l_0,\mathbf{x_0}}(t)$ means the left limit of $\xi_{l_0,\mathbf{x_0}}$ at t;
  (iii) $(\xi^-_{l_0,\mathbf{x_0}}(t_{i+1}), \xi_{l_0,\mathbf{x_0}}(t_{i+1})) \in \gamma$.

Definition 2. (n-PCD) *An n-dimensional Piecewise Constant Derivative (n-PCD) system [2] is a pair $\mathcal{H} = (\mathbb{P}, \mathbb{F})$ such that:*

(1) $\mathbb{P} = \{P_s\}_{1 \le s \le k}$ is a finite family in $\mathbb{R}^n$, where $P_s \subseteq \mathbb{R}^n$ are non-overlapping convex polygonal sets.
(2) $\mathbb{F} = \{\boldsymbol{c}_s\}_{1 \le s \le k}$ is a family of vectors in $\mathbb{R}$.
(3) The dynamics are given by $\dot{\boldsymbol{x}} = \boldsymbol{c}_s$ for $\boldsymbol{x} \in P_s$.

An n-PCD is called bounded if for its regions $\mathbb{P} = \{P_s\}_{1 \le s \le k}$, there exists $r \in \mathbb{Q}^+$, such that for all P_s, we have that $P_s \subseteq B_{\mathbf{0}}(r)$, where $B_{\mathbf{0}}(r)$ is an origin-centered open ball of radius r and appropriate dimension.

We define the *support set* of a PCD $\mathcal{H}$ as $\mathrm{Supp}_{\mathrm{PCD}}(\mathcal{H}) = \bigcup_{1 \le s \le k} P_s$. Given an edge e, we represent a point on e by a one-dimensional *local coordinate*, allowing us to define an edge-to-edge successor function as an affine function between edges.

Definition 3. (HPCD) *A Hierarchical Piecewise Constant Derivative (HPCD) system [3] is a hybrid automaton $\mathcal{H} = (\mathcal{X}, Q, f, l_0, Inv, \delta)$ such that Q and l_0 are defined as in Definition 1, with the dynamics at each $l \in Q$ given by a 2-PCD and each transition $tr = (l, g, \gamma, l')$ is such that: (1) Its guard g is a line segment in $\mathbb{R}^2$; and (2) The reset relation γ is an affine function of the form: $\boldsymbol{x}' = \gamma(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$. We denote the* internal guards *of an HPCD location to be the guards of the underlying PCD regions and the* transition guards *to be the guards used in transitions between locations. The Invariant (Inv) for a location l is defined to be $Supp_{PCD}(\mathcal{H}) \setminus \mathcal{G}_l$, where $Supp_{PCD}(\mathcal{H})$ is the support set of the underlying PCDs of the HPCD and $\mathcal{G}_l$ is the set of transition guards in location l. If all the PCDs are bounded, then $\mathcal{H}_{PCD}$ is said to be bounded.*

It was shown in [3] that reachability for an HPCD is equivalent to reachability for a 1-PAM. An HPCD system has *elementary flows* if the derivatives of all variables in each location are in $\{0, \pm 1\}$, otherwise it has *arbitrary constant flows*. Guards are defined as line segments, described by boolean combinations of linear

inequalities. If each atomic formula contains only one variable (x or y), then the guard is called non-comparative. An HPCD has *non-comparative guards* if all guards are non-comparative, e.g., $\frac{3}{2} \leq x \leq 7 \wedge y = -1$ is non-comparative, but $0 \leq x \leq 1 \wedge 0 \leq y \leq \frac{1}{2} \wedge x = 2y$ is a comparative guard. We define an *affine reset* of variables $\mathbf{z} = (x, y)$ as $\gamma(\mathbf{z}) = A\mathbf{z} + \mathbf{b}$ where $A \in \mathbb{Q}^{2\times 2}$ and $\mathbf{b} \in \mathbb{Q}^2$, a *linear reset* is of the form $\gamma(\mathbf{z}) = A\mathbf{z}$ and an *identity reset* is $\gamma(\mathbf{z}) = \mathbf{z}$. Our reference model is called an RHPCD, a *restricted* HPCD. We later show that adding any one of the additional powers - arbitrary constant flows, comparative guards or linear resets - allows simulation of a 1-PAM. Note that since reachability for HPCDs is equivalent to reachability for 1-PAMs and an RHPCD with these powers is a restricted form of HPCD, then showing reachability for a 1-PAM can be simulated by such an RHPCD shows their equivalence. Equivalence of reachability between 1-PAMs and planar pseudo-billiard systems was shown in [11], whereas more complex 1-dim. functions allow universal computation.

Definition 4. (RHPCD) *A* Restricted Hierarchical Constant Derivative System (RHPCD) *is a bounded HPCD with identity resets, non-comparative guards, elementary flows and a finite number of PCD regions. See Fig. 3b and Fig. 4 for an example of an RHPCD with arbitrary constant flows.*

Our final model is the class of one-dimensional Piecewise Affine Maps (1-PAM). The reachability problem for 1-PAM is currently a long-standing open problem, even for two intervals. Our approach follows a similar style to [3] where we show various classes where reachability is equivalent to that of a 1-PAM.

Definition 5. (1-PAM) - *A one-dimensional Piecewise Affine Map (1-PAM) is a function $f : \mathbb{R} \to \mathbb{R}$ (See Fig. 3a for an example) such that:*

(1) Domain of f : dom(f)=$\bigcup I_i$, where I_i are disjoint rational intervals.
(2) $\exists a_i, b_i \in \mathbb{Q}$ such that $\forall x \in I_i$, $f(x) = a_i x + b_i$.
(3) f is closed, i.e., range(f) $\subseteq$ dom(f).

Simulation: We use the definition of simulation of PCDs described in [2]. If model $\mathcal{A}$ can be simulated by model $\mathcal{B}$, then reachability for $\mathcal{A}$ can be reduced to reachability for $\mathcal{B}$ ($\mathcal{A}$ has a decidable reachability problem as long as $\mathcal{B}$ does).

Reachability Problems: By an instance of a reachability problem we mean a finite description of a model, an initial and final configuration (or set of configurations). Reachability: starting from the initial configuration(s), does the trajectory eventually reach the final configuration(s) in finite time after a finite number of (discrete) transitions? Models of hybrid automata with an *infinite number of transitions in finite time* are said to have the *Zeno property* which we do not consider here. Problems such as convergence, stability and control are not considered in this paper, see [9] for more information about these problems.

Open Problem 6 - 1-PAM Reachability - *Given a 1-dim. Piecewise Affine Map f, and points $x, y \in \mathbb{Q}$. Does there exist $t \in \mathbb{N}$, such that $f^t(x) = y$?* [1]

[1] $f^t(x)$ denotes $\underbrace{f(f(\ldots f(x)\ldots))}_{t}$

3 Reachability for RHPCDs and Extensions

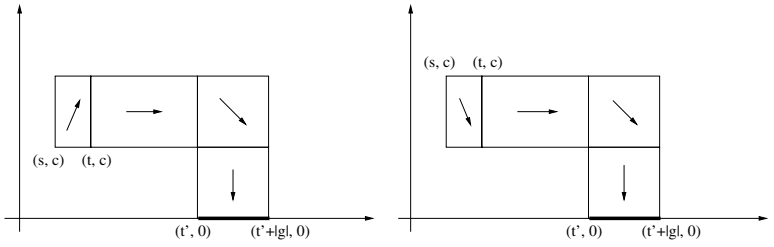
We now explore reachability for our models. We start with a technical lemma.

Lemma 7. *The interval $\langle I, 0\rangle$ can be mapped to $\langle f(I) + m, 0\rangle$ by an RHPCD system with arbitrary constant flows, where $f(x) = ax + b$ is an affine function, $I = (s, t)$ is a 1-dimensional interval and $a, b, m, s, t \in \mathbb{Q}$ are constants.*

Proof. We prove this lemma by 3 steps.
Step 1 - Interval $\langle I, 0\rangle$ can be mapped to interval $\langle I, c\rangle$, where $c \in \mathbb{Q}^+$, by a bounded PCD with non-comparative guards using flow $(0, 1)$. By this, we mean that for any $0 \leq \alpha \leq 1$, point $(s + (t - s)\alpha, 0)$ can be mapped to point $(s + (t - s)\alpha, c)$ by this bounded PCD. We also use similar terminology throughout.

Step 2 - Suppose we have an affine function $f(x) = ax+b$, and the 1-dimensional rational interval $I = (s, t)$. For any constant t' where $t' \geq t > s$, define $g = f(t) - f(s)$ and $s' = t' + |g|$. Assume that $c > |g| + |b| > 0$. Then we show the interval $\langle I, c\rangle$ can be mapped to $\langle I' = (t', s'), 0\rangle$ by a bounded PCD system with non-comparative guards; thus meaning that for any $0 \leq \alpha \leq 1$, point $(s + (t - s)\alpha, c)$ can be mapped to point $(t' + (s' - t')\alpha, 0)$, see Fig. 1. We need to consider 2 cases, $a > 0$ and $a < 0$.

1. $a > 0$. See Fig. 1(a)
   (i) Use flow $(1, a)$ to map $\langle (s, t), c\rangle$ to $\langle t, (c, c + |g|)\rangle$;
   (ii) Use flow $(1, 0)$ to map $\langle t, (c, c + |g|)\rangle$ to $\langle t', (c, c + |g|)\rangle$;
   (iii) Use flow $(1, -1)$ to map $\langle t', (c, c + |g|)\rangle$ to $\langle (t', t' + |g|), c\rangle$;
   (iv) Use flow $(0, -1)$ to map $\langle (t', t' + |g|), c\rangle$ to $\langle (t', t' + |g|), 0\rangle$.
2. $a < 0$. See Fig. 1(b).
   (i) Use flow $(1, a)$ to map $\langle (s, t), c\rangle$ to $\langle t, (c - |g|, c)\rangle$;
   (ii) Use flow $(1, 0)$ to map $\langle t, (c - |g|, c)\rangle$ to $\langle t', (c - |g|, c)\rangle$;
   As we assume $c > |g| + |b| > 0$, so $c - |g| > |b| > 0$, which means the rectangle $\{(x, y) | t < x < t', c - |g| < y < |g|\}$ does not intersect with the x-axis, hence the following steps make sense.
   (iii) Use flow $(1, -1)$ to map $\langle t', (c - |g|, c)\rangle$ to $\langle (t', t' + |g|), c - |g|\rangle$;
   (iv) Use flow $(0, -1)$ to map $\langle (t', t' + |g|), c - |g|\rangle$ to $\langle (t', t' + |g|), 0\rangle$.



(a) $a > 0$ (b) $a < 0$

Fig. 1. Lemma 7 Step 2: map $\langle (s, t), c\rangle$ to $\langle (t', s'), 0\rangle$

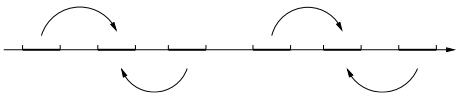


Fig. 2. Idea of Theorem 8: map every two adjacent intervals into one interval

Note that though some of these steps contains 'triangles', we can define the flows in a rectangular area containing that triangle, thus comparative guards are not required. Note the 'orientation' of the interval is reversed after the mapping.

Step 3 - Using a similar idea we can show the interval $\langle I' = (t', s'), 0\rangle$ can be mapped to $\langle f(I) + m, 0\rangle$, where $f(I) = (f(s), f(t))$ if $a > 0$ and $f(I) = (f(t), f(s))$ if $a < 0$, by a bounded PCD system with non-comparative guards. We can use only the upper or lower half plane of the PCD. Here we only prove the case when $a > 0$ and $f(t) + m < t'$ by using the lower half plane, other cases can be proven similarly.

(i) Use flow $(-1, -1)$ to map $\langle (t', s'), 0\rangle$ to $\langle \frac{1}{2}(t' + f(t) + m), (-\frac{1}{2}|t' - f(t) - m| - |g|, -\frac{1}{2}|t' - f(t) - m|)\rangle$;
(ii) Use flow $(-1, 1)$ to map $\langle \frac{1}{2}(t' + f(t) + m), (-\frac{1}{2}|t' - f(t) - m| - |g|, -\frac{1}{2}|t' - f(t) - m|)\rangle$ to $\langle (f(s) + m, f(t) + m), 0\rangle$.

Combining Step 1,2 and 3 we get the result of the lemma. We need a 2-location RHPCD system with arbitrary constant flows and hence each location is a bounded PCD system with non-comparative guards. In location 1 we realize Step 1 and jump to location 2, i.e., the guards are $\langle s_i \leq x < t_i, y = c\rangle$. In location 2 we realize Step 2 and Step 3 together because Step 2 only uses the upper plane of a PCD and Step 3 only requires the lower plane of a PCD. □

Theorem 8. *A 1-PAM with n intervals can be simulated by an RHPCD with $\lceil \log_2 n \rceil + 3$ locations such that one of the variables has arbitrary constant flows.*

Proof. Suppose PAM $\mathcal{A}$ is defined by $f(x) = a_i x + b_i$ if $x \in I_i$, with $1 \leq i \leq n$ and I_i are rational intervals. Let the left and right endpoints of I_i be s_i and t_i respectively. First we show that this PAM can be simulated straightforwardly by an $n + 1$-location RHPCD with arbitrary constant flows. We need a single location p as the global state and n locations q_i for each interval I_i, $1 \leq i \leq n$.

1. In location p, we define the corresponding points of the PAM $\mathcal{A}$ on interval $\langle (s_1, t_n), 0\rangle$. We then map each $\langle I_i, 0\rangle$ to the interval $\langle I_i, c\rangle$, where $c = |\max\{|a_i|\}(t_n - s_1)| + \max\{|b_i|\}$. (See Lemma 7, Step 1). The transition guards of p are: $\langle s_i \leq x < t_i, y = c\rangle$, in which we jump to q_i.
2. In location q_i, map $\langle I_i, c\rangle$ to $\langle f(I_i), 0\rangle$ (see Lemma 7, Step 2&3). The transition guard of q_i is : $\langle s_1 \leq x < t_n, y = 0\rangle$, with a jump to location p.

The above method requires $n+1$ locations for a PAM with n intervals. We now give an improved method using an RHPCD with only $\lceil \log_2 n \rceil + 3$ locations.

Suppose the PAM $\mathcal{A}$ contains n intervals. For every $n \neq 2^d$, $d \in \mathbb{N}$, there exists a minimum integer $k \in \mathbb{N}$ such that $\log_2(n+k) = \lceil \log_2 n \rceil$. The PAM $\mathcal{A}$ can be expanded to $\mathcal{A}'$ such that $f(x) = a_i x + b_i$ if $x \in I_i$, where $i \in \{1, \ldots, n\}$. For every $i' \in \{1, \ldots, k\}$, the length of each new added interval is given by $|I_{\varepsilon_{i'}}| = \epsilon$, and the corresponding affine function is $f(x) = x$. This expansion does not change the dynamics of the PAM $\mathcal{A}$, thus we assume $n = 2^d$, $d \in \mathbb{Z}$.

Again, let the left endpoint and the right endpoint of I_i be s_i and t_i respectively. Define c to be $c = |\max\{|a_i|\}(t_n - s_1)| + \max\{|b_i|\}$ and l to be $l = |t_n - s_1|$.

Step 1 Define the PAM on interval $\langle (s_1, t_n), 0 \rangle$. For every $i \in \{1, 2, ..., n\}$, map $\langle I_i, 0 \rangle$ to interval $\langle I_i, 2(n-i+1)c \rangle$. (See Lemma 7, Step 1). In this step each interval is mapped to a different height $y = 2(n-i+1)c$. There is a $2c$-length 'gap' between every two intervals I_i and I_{i+1} and I_i is higher than I_{i+1}. In Lemma 7 Step 2 this clearly prevents intersections in the following step.

Step 2 Map each interval $\langle I_i, 2(n-i+1)c \rangle$ to $\langle (f(I_i) + 2(n-i+1)l, 0 \rangle$. (See Lemma 7, Step 2). Then between every two intervals there is a 'gap' whose length is l.

Step 3 For i from 1 to $\frac{n}{2}$, let $j = 2i - 1$, we can find an undefined interval between $\langle f(I_j) + 2(n-j+1)l, 0 \rangle$ and $\langle f(I_{j+1}) + 2(n-j+2)l, 0 \rangle$ of length l. By the proof of Lemma 7 (Step 3), we can map $\langle f(I_j) + 2(n-j+1)l, 0 \rangle$ using the upper plane and $\langle f(I_{j+1}) + 2(n-j+2)l, 0 \rangle$ using the lower plane to this interval.

Step 4 Repeat Step 2 for $\log_2(n)$ times until only 1 interval, I_f, remains.

Step 5 If the orientation of I_f is 'reversed' with respect to the initial interval of the PAM $\mathcal{A}$, then map I_f to this initial interval; otherwise, we reverse it before mapping it to the initial interval.

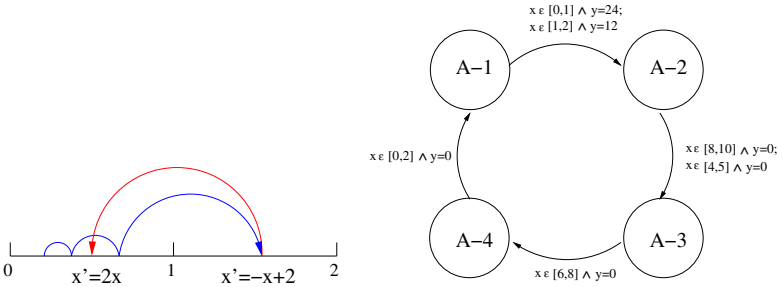
Step 1, 2 and 5 each require 1 location. Step 3 and Step 4 require $\log_2 n$ locations, thus $(\log_2 n) + 3$ locations are required. □

The difficulty of simulating a 1-PAM by a 2-PCD is that regions cannot overlap in a PCD, i.e., one region has only one deterministic constant flow. Thus it is impossible to map several different intervals into a single interval under a 2-PCD, leading us to believe that $\Omega(\log_2 n)$ is a lower bound of the number of locations required to simulate an n-interval 1-PAM by an RHPCD with arbitrary constant flows.

Example 9. *We give an example of a 1-PAM below and show how to simulate it by a RHPCD with arbitrary constant flows in Figs. 3, 4.*

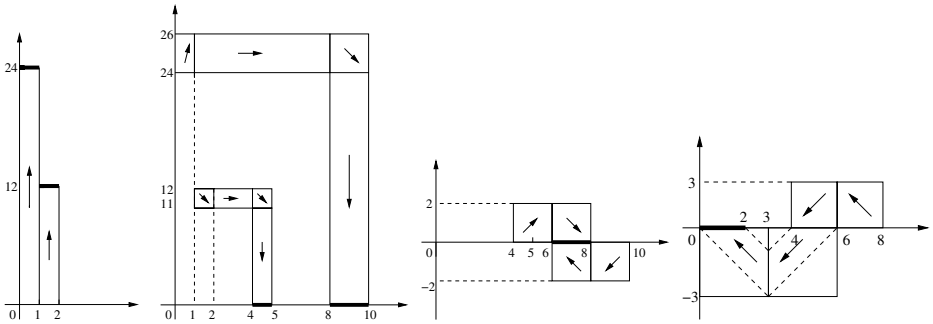
$$f(x) = \begin{cases} 2x, & if \quad x \in [0,1) \\ -x+2, & if \quad x \in [1,2] \end{cases}$$

Let the initial point be x_0. The initial location of the HPCD is A-1, with variables $(x, y) = (x_0, 0)$. PCD A-1 corresponds to Theorem 8, Step 1. PCD A-2 separates



(a) A 1-dim. Piecewise Affine Map (b) RHPCD with arbitrary constant flows

Fig. 3. The 1-PAM with its equivalent HPCD



(a) A-1 (b) A-2 (c) A-3 (d) A-4

Fig. 4. The 2-PCDs of the HPCD in Fig 3b (transition guards in bold)

each interval onto the x axis (Theorem 8, Step 2). PCD A-3 combines together these two intervals (Theorem 8, Step 3). Finally, in A-4, as the final interval $[6, 8]$ *has the same orientation as the initial interval* $[0, 2]$*, we reverse it before mapping it back to the initial interval (Theorem 8, Step 5).*

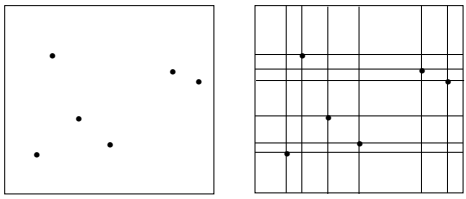
We now show that an RHPCD with linear resets can simulate a 1-PAM.

Lemma 10. *The interval* $\langle I, 0\rangle$ *can be mapped to* $\langle f(I) + m, 0\rangle$ *by an RHPCD system with linear resets, where* $f(x) = ax + b$ *is an affine function,* $I = (s, t)$ *is a 1-dimensional interval and* $a, b, m, s, t \in \mathbb{Q}$ *are constants.*

Proof. The proof is similar to the proof of Lemma 7. We still prove by 3 steps.

Step 1 First map the interval $\langle I, 0\rangle$ to the interval $\langle I, c\rangle$ by flow $(0, 1)$. Define the transition guard to be $\langle I, c\rangle$, which jumps to location 2 with linear reset: $x' = |a|x, y' = y$.

Step 2 Using the similar idea in Lemma 7 Step 2, we can map the interval $\langle |a|I, c\rangle$ to the interval $\langle (t', t' + |g|), 0\rangle$ by the flows $(1, 1)$ if $(a > 0)$ or



(a) Set of Points $V \subseteq \mathbb{R}^2$ (b) Its Rectilinear Tessellation

Fig. 5. Rectilinear Tessellation of the plane

$(1,-1)$ if $a < 0$, $(1,0)$, $(1,-1)$ and $(0,-1)$, where t' and g are the same defined as in Lemma 7.

Step 3 Exactly the same as Lemma 7 Step 3. □

Theorem 11. *A 1-PAM with n intervals can be simulated by an RHPCD containing $\lceil \log_2 n \rceil + 3$ locations with linear resets.*

Proof. Apply Lemma 10 instead of Lemma 7 in the proof of Theorem 8. □

Corollary 12. *An RHPCD with linear resets is equivalent to an RHPCD with affine resets.*

Proof. Immediately from the results of [3] and Theorem 11. □

Definition 13. (Rectilinear Tessellation) - *Let $V \subseteq \mathbb{R}^2$ be a finite set of 2-dimensional points. We define a* rectilinear tessellation *of the plane as a tessellation by rectangles by identifying with each $v \in V$ a splitting of the plane into four quadrants, parallel to the x and y axes. See Fig. 5(a) and Fig. 5(b).*

Lemma 14. *Let $\mathcal{H}$ be an RHPCD. There exists an RHPCD, $\mathcal{H}_R$, which is topologically equivalent to $\mathcal{H}$, such that $\mathcal{H}_R$ has an injective edge-to-edge successor function which preserves local coordinates.*

Proof. Given the n-location RHPCD $\mathcal{H}$, we define the *rectilinear tessellation* of $\mathcal{H}$ in the following way. Let $V_i \subseteq \mathbb{Q}^2$ be the set of points defining the PCD regions of location l_i of $\mathcal{H}$ and let $V = \cup V_i$ for $1 \leq i \leq n$. We assume without loss of generality that $V \subseteq \mathbb{Z}^2$ by forming a (topologically) equivalent HPCD with all points multiplied by the least common multiple of the denominators of coordinates of points in V. We form a rectilinear tessellation of the plane by set of points V. Note that each location of $\mathcal{H}$ can thus be defined on the same set of regions, but allowing a region in different locations to have a different derivative.

We define derivatives of the form $\{(\pm 1, 0), (0, \pm 1)\}$ as *straight flows* and derivatives of the form $\{(\pm 1, \pm 1), (\pm 1, \mp 1)\}$ as *diagonal flows*. Our next step is to further decompose the rectilinear tessellation of the plane such that any non-square region containing a diagonal flow in any location is split into a finite number of square regions, each of which contains the same diagonal flow.

To perform this step, let R by any (non-square) rectangle such that there exists a location l_i where the derivative of R in l_i is a diagonal flow. Let $(r_x, r_y) \in \mathbb{Z}^2$ be the bottom left point of R and $(r'_x, r'_y) \in \mathbb{Z}^2$ be its upper right point. Then we add all points $\{(i,j) | r_x \leq i \leq r'_x, r_y \leq j \leq r'_y\} \subseteq \mathbb{Z}^2$ to set V and recompute the rectilinear tessellation of $\mathcal{H}$. There are a finite number of bounded regions in $\mathcal{H}$ and thus this procedure eventually halts giving a final set of points V' defining the regions shared by all locations of the new RHPCD $\mathcal{H}_R$.

In each location of $\mathcal{H}_R$, only square regions have diagonal flows, thus each edge is mapped to exactly one other under $\mathcal{H}_R$. Clearly then, local coordinates of each edge are preserved by the edge-to-edge successor function by rectilinearity of the plane partition and since each region has elementary flows. □

Theorem 15. *The reachability problem is decidable for an RHPCD.*

Proof. Given $\mathcal{H}$, we can apply the rectilinear tessellation technique of Lemma 14 to form an RHPCD $\mathcal{H}_R$ satisfying the conditions of the lemma. Let (l_0, α) be the initial state of the system and (l_f, β) be the final point of the system, where $\alpha, \beta \in \mathbb{Q}^2$. In the same way as in Lemma 14, these points can be transformed so that $\alpha, \beta \in \mathbb{Z}^2$ are the initial and final points under $\mathcal{H}_R$.

Since the edge-to-edge successor function of $\mathcal{H}_R$ is injective, we can form a finite graph with vertices labelled (e_i, k) where (e_i, k) is an edge in location l_i of $\mathcal{H}_R$. Each vertex is connected to exactly one other, according to the injective edge-to-edge successor function. Since local coordinates of points on edges are preserved by this function by Lemma 14, reachability becomes trivial since the local coordinate of y must be the same as x and we can simply traverse the graph until we reach the correct edge in some location or else detect a cycle. □

Definition 16. (1-POM) *Let f be a 1-PAM. We call f a* one-dimensional piecewise offset map *(1-POM) if $f(x) = x + b_i$ for all $x \in I_i$.*

Corollary 17. *A 1-POM can be simulated by an RHPCD, and an RHPCD an be simulated by a 1-POM.*

The following theorem shows a relationship between the additional computational powers of affine resets and arbitrary constant flows.

Theorem 18. *A k-location RHPCD with arbitrary constant flows can be simulated by a k-location RHPCD with affine resets for any $k \geq 1$.*

Finally, we introduce nondeterminism to the RHPCD model.

Theorem 19. *The reachability problem for a nondeterministic RHPCD system is NP-hard.*

Proof. We use a reduction of the Subset Sum Problem (SSP): given a finite set of positive integers A and a positive integer $N > 0$, is there a subset $A' \subseteq A$ such that the sum of the elements in A' is exactly N? i.e., if $A = \{k_1, k_2, \ldots, k_n\} \subseteq \mathbb{Z}^+$, is there a subset $A' = \{k_{t_1}, k_{t_2}, \ldots, k_{t_m}\} \subseteq A$ such that $\sum_{j=1}^{m} k_{t_j} = N$?

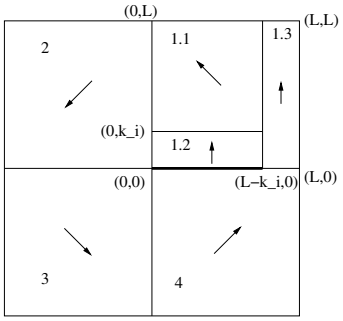


Fig. 6. SSP simulated by a nondeterministic RHPCD with arbitrary constant flows. The bold line denotes the transition guard.

To simulate an n element instance of SSP, we define an $(n+2)$-location nondeterministic RHPCD. Divide each location i $(1 \leq i \leq n)$ into four squares labelled 1 to 4; see Fig. 6. Each square has length L, where $L > N + \max\{|k_i|\} + \varepsilon$, $\varepsilon \in (0,1)$. Define the left lower corner point of square 1 to be the original point $(0,0)$ and the right upper corner point of square 1 to be the point (L,L) and the other regions are located as in Fig. 6. For regions 2, 3, 4 we need elementary flows $(-1,-1),(1,-1)$ and $(1,1)$, respectively. For each location i $(1 \leq i \leq n)$, divide square 1 into three subregions 1.1, 1.2 and 1.3 by internal guards $\langle (0, L-k_i), k_i \rangle$ and $\langle L-k_i, (0,L) \rangle$. Then the region 1.1 is defined as a square of length $L-k_i$ and regions 1.2 and 1.3 are two rectangles located inside square 1 as in Fig. 6. The flows of region 1.1, 1.2 and 1.3 are defined as $(-1,1)$, $(0,1)$ and $(0,1)$, respectively. Because $L > N + \max\{|k_i|\} + \varepsilon$, the point $(0, N+\varepsilon)$ is possible to reach from the x-axis in each location i. Define the transition guard of each location i to be $\langle 0 < x < L, y = 0 \rangle$, which jumps to any one of the locations from location $i+1$ to $n+1$. At last let location 0 be the starting location from which we can jump to location 1 to n, and location $n+1$ be the final location where all the trajectories move with flow $(1,0)$. Clearly, the SSP instance has a solution if the point $(0, N+\varepsilon)$ in any location i $(1 \leq i \leq n)$ can be reached from $(\varepsilon, 0)$ in location 0. Note that the construction of the corresponding RHPCD can be done in time polynomial in the SSP instance size. □

4 Conclusion

We showed decidability of reachability for Restricted Hierarchical Piecewise Constant Derivative (RHPCD) systems. The complexity of this problem is interesting but is currently unresolved. We then showed that adding: comparative guards, arbitrary constant flows or linear resets to an RHPCD makes the problem equivalent to reachability for 1-PAMs, which is a long standing open problem. We also showed that adding nondeterminism to RHPCDs leads to NP-hardness of reachability. In this paper we focused on reachability problems, but stability, convergence and control problems for low-dimensional linear hybrid systems are also important topics under active research.

Acknowledgements. We would like to thank the anonymous referees for their very useful suggestions and comments.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Asarin, E., Maler, O., Pneuli, A.: Reachability analysis of dynamical systems having piecewise constant derivatives. Theoretical Computer Science 138, 35–65 (1995)
3. Asarin, E., Mysore, V., Pneuli, A., Schneider, G.: Low dimensional hybrid systems - decidable, undecidable, don't know. Information and Computation 211, 138–159 (2012)
4. Asarin, E., Schneider, G.: Widening the boundary between decidable and undecidable hybrid systems. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 193–208. Springer, Heidelberg (2002)
5. Asarin, E., Schneider, G., Yovine, S.: Algorithmic analysis of polygonal hybrid systems, part I: Reachability. Theoretical Computer Science 379, 231–265 (2007)
6. Balluchi, A., Benvenuti, L., di Benedetto, M., Pinello, C., Sangiovanni-Vincentelli, A.: Automotive engine control and hybrid systems: challenges and opportunities. Proceedings of the IEEE 88(7), 888–912 (2000)
7. Bauer, N., Kowalewski, S., Sand, G.: A case study: Multi product batch plant for the demonstration of control and scheduling problems. In: ADPM, Dortmund, Germany, pp. 969–974 (2000)
8. Blondel, V., Tsitsiklis, J.: When is a pair of matrices mortal? Information Processing Letters 63, 283–286 (1997)
9. Blondel, V., Tsitsiklis, J.N.: A survey of computational complexity results in systems and control. Automatica 36, 1249–1274 (2000)
10. Henzinger, T.A., Raskin, J.-F.: Robust undecidability of timed and hybrid systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 145–159. Springer, Heidelberg (2000)
11. Kurganskyy, O., Potapov, I., Sancho-Caparrini, F.: Reachability problems in low-dimensional iterative maps. International Journal of Foundations of Computer Science 19(4), 935–951 (2008)
12. Maler, O., Pnueli, A.: Reachability analysis of planar multi-linear systems. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 194–209. Springer, Heidelberg (1993)
13. Prabhakar, P., Vladimerou, V., Viswanathan, M., Dullerud, G.E.: A decidable class of planar linear hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 401–414. Springer, Heidelberg (2008)

Parametric Interrupt Timed Automata*

Beatrice Bérard[1], Serge Haddad[2], Aleksandra Jovanović[3], and Didier Lime[3]

[1] Université Pierre & Marie Curie, LIP6/MoVe, CNRS, Paris, France
[2] ENS Cachan, LSV, CNRS, INRIA, Cachan, France
[3] LUNAM Université, École Centrale de Nantes, IRCCyN, CNRS, Nantes, France

Abstract. Parametric reasoning is particularly relevant for timed models, but very often leads to undecidability of reachability problems. We propose a parametrised version of Interrupt Timed Automata (an expressive model incomparable to Timed Automata), where polynomials of parameters can occur in guards and updates. We prove that different reachability problems, including robust reachability, are decidable for this model, and we give complexity upper bounds for a fixed or variable number of clocks and parameters.

1 Introduction

Parametric Verification. Getting a complete knowledge of a system is often impossible, especially when integrating quantitative constraints. Moreover, even if these constraints are known, when the execution of the system slightly deviates from the expected behaviour, due to implementation choices, previously established properties may not hold anymore. Additionally, considering a wide range of values for constants allows for a more flexible and robust design.

Introducing parameters instead of concrete values is an elegant way of addressing these three issues. Parametrisation however makes verification more difficult. Besides, it raises new problems like parameter synthesis, *i.e.* finding the set (or a subset) of values for which some property holds.

Parameters for Timed Models. Among quantitative features, parametric reasoning is particularly relevant for timing requirements, like network delays, time-outs, response times or clock drifts.

Pioneering work on parametric real time reasoning was presented in [1] for the now classical model of timed automata [2] with parameter expressions replacing the constants to be compared with clock values. Since then, many studies have been devoted to the parametric verification of timed models [3,4,5], mostly establishing undecidability results for questions like parametric reachability, even for a small number of clocks or parameters. Relaxing completeness requirement or guaranteed termination, several methods and tools have been developed for parameter synthesis in timed automata [6,7,8], as well as in hybrid automata [9,10]. Another research direction consists in defining subclasses of parametric timed

* This work has been supported by project ImpRo ANR-2010-BLAN-0317.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 59–69, 2013.

models for which some problems become decidable [11,12,13]. Unfortunately, these subclasses are severely restricted. It is then a challenging issue to define expressive parametric timed models where reachability problems are decidable.

Contributions. The model of interrupt timed automata (ITA) [14,15] was proposed as a subclass of hybrid automata, incomparable with the class of timed automata, where task interruptions are taken into account. Hence ITA are particularly suited for the modelling of scheduling with preemption.

We propose to enrich ITA with parameters in the spirit above. A PITA is a parametric version of ITA where polynomial parameter expressions can be combined with clock values both as additive and multiplicative coefficients. The multiplicative setting is much more expressive and useful in practice, for instance to model clock drifts. We prove that reachability in parametric ITA is decidable as well as its robust variant, an important property for implementation issues. To the best of our knowledge, this is the first time such a result has been obtained for a model including a multiplicative parametrisation. Furthermore, we establish upper bounds for the algorithms complexity: 2EXSPACE and PSPACE when the number of clocks is fixed, which become respectively 2EXPTIME and PTIME for additive parametrisation, when the number of clocks and parameters is fixed. Our technique combines the construction of symbolic class automata from the ITA case and the first order theory of real numbers. Finally, considering only additive parametrisation, we reduce reachability to the same problem in basic ITA.

Outline. The parametric ITA model is introduced in Section 2 and decision procedures are presented in Section 3 with complexity analysis. We conclude and give some perpectives for this work in Section 4. All proofs are given in the appendix.

2 Parametric Interrupt Timed Automata

2.1 Notations

The sets of natural, rational and real numbers are denoted respectively by $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$. Given two sets F, G, we denote by $\mathcal{P}ol(F, G)$, the set of polynomials with variables in F and coefficients in G. We also denote by $\mathcal{L}in(F, G)$ the subset of polynomials with degree at most one and by $\mathcal{F}rac(F, G)$, the set of rational functions with variables in F and coefficients in G (i.e. quotients of polynomials).

Clock and parameter constraints. Let X be a finite set of clocks and let P be a finite set of parameters. An expression over clocks is an element $\sum_{x\in X} a_x \cdot x + b$ of $\mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$. In the sequel we also consider two other sets of expressions: $\mathcal{L}in(X, \mathbb{Q})$ and $\mathcal{L}in(X \cup P, \mathbb{Q})$. The former is the subset of expressions without parameters while the latter can be seen as a subset of expressions where $a_x \in \mathbb{Q}$ for all $x \in X$ and $b \in \mathcal{L}in(P, \mathbb{Q})$. We denote by $\mathcal{C}(X, P)$ the set of constraints

obtained by conjunctions of atomic propositions of the form $C \bowtie 0$, where C is an expression in $\mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$ and $\bowtie \in \{>, \geq, =, \leq, <\}$.

Updates and valuations. An *update* is a conjunction of assignments of the form $\wedge_{x \in X} x := C_x$, where $C_x \in \mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$, with possibly $C_x = 0$ or $C_x = x$. The set of updates is written $\mathcal{U}(X, P)$. For an expression C and an update u, the expression $C[u]$ is obtained by "applying" u to C, *i.e.*, simultaneously substituting each x by C_x in C, if $x := C_x$ is the update for x in u. For instance, for clocks $X = \{x_1, x_2\}$, parameters $P = \{p_1, p_2, p_3\}$, expression $C = p_2x_2 - 2x_1 + 3p_1$ and the update u defined by $x_1 := 1 \wedge x_2 := p_3x_1 + p_2$, applying u to C yields the expression $C[u] = p_2p_3x_1 + p_2^2 + 3p_1 - 2$. Note that the use of multiplicative parameters for clocks may result in polynomial coefficients when updates are applied.

A *clock valuation* is a mapping $v : X \mapsto \mathcal{P}ol(P, \mathbb{R})$, with $\mathbf{0}$ the valuation where all clocks have value 0. For a valuation v and an expression $C \in \mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$, $v(C) \in \mathcal{P}ol(P, \mathbb{R})$ is obtained by evaluating C w.r.t. v. Given an update u and a valuation v, the valuation $v[u]$ is defined by $v[u](x) = v(C_x)$ for x in X if $x := C_x$ is the update for x in u. For instance, let $X = \{x_1, x_2, x_3\}$ be a set of three clocks. For valuation $v = (2p_2, 1.5, 3p_1^2)$ and update u defined by $x_1 := 1 \wedge x_2 := x_2 \wedge x_3 := p_1x_3 - x_1$, applying u to v yields the valuation $v[u] = (1, 1.5, 3p_1^3 - 2p_2)$.

A *parameter valuation* is a mapping $\pi : P \mapsto \mathbb{R}$. For a parameter valuation π and an expression $C \in \mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$, $\pi(C) \in \mathcal{L}in(X, \mathbb{R})$ is obtained by evaluating C w.r.t. π. If $C \in \mathcal{P}ol(P, \mathbb{Q})$, then $\pi(C) \in \mathbb{R}$. Given a parameter valuation π, a clock valuation v and an expression $C \in \mathcal{L}in(X, \mathcal{P}ol(P, \mathbb{Q}))$ we write $\pi, v \models C \bowtie 0$ when $\pi(v(C)) \bowtie 0$.

2.2 Parametric Interrupt Timed Automata

Definitions. The behaviour of an ITA can be viewed as the one of an operating system with interrupt levels. At a given level, exactly one clock is active (rate 1), while the clocks at lower levels are suspended (rate 0), and the clocks at higher levels are not yet activated and thus contain value 0. The enabling conditions of transitions, called *guards*, are constraints in $\mathcal{L}in(X, \mathbb{Q})$ over clocks of levels lower than or equal to the current level. Transitions can *update* the clock values. If the transition decreases (resp. increases) the level, then each clock which is relevant after (resp. before) the transition can either be left unchanged or take a linear expression of clocks of **strictly** lower level.

Parametric ITA include parameters in guards and updates.

Definition 1. *A* parametric interrupt timed automaton *(PITA) is a tuple* $\mathcal{A} = \langle \Sigma, P, Q, q_0, X, \lambda, \Delta \rangle$*, where:*

- Σ *is a finite alphabet,* P *is a finite set of parameters,*
- Q *is a finite set of states,* q_0 *is the initial state,*
- $X = \{x_1, \ldots, x_n\}$ *consists of* n *interrupt clocks,*
- *the mapping* $\lambda : Q \to \{1, \ldots, n\}$ *associates with each state its level; we assume* $\lambda(q_0) = 1$*,* $X_{\lambda(q)} = \{x_i \mid i \leq \lambda(q)\}$ *is the set of* relevant *clocks at this level and* $x_{\lambda(q)}$ *is called the* active *clock in state* q*;*

– $\Delta \subseteq Q \times \mathcal{C}(X,P) \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{U}(X,P) \times Q$ *is a finite set of transitions. Let* $q \xrightarrow{\varphi,a,u} q'$ *be a transition in* Δ *with* $k = \lambda(q)$ *and* $k' = \lambda(q')$*. The guard* φ *is a constraint in* $\mathcal{C}(X_k, P)$ *(using only clocks from levels less than or equal to* k*). The update* u *is of the form* $\wedge_{i=1}^{n} x_i := C_i$ *with:*
  - *if* $k > k'$*, i.e. the transition decreases the level, then for* $1 \leq i \leq k'$*,* C_i *is either of the form* $\sum_{j=1}^{i-1} a_j x_j + b$ *or* $C_i = x_i$ *(unchanged clock value) and for* $i > k'$*,* $C_i = 0$*;*
  - *if* $k \leq k'$ *then for* $1 \leq i \leq k$*,* C_i *is of the form* $\sum_{j=1}^{i-1} a_j x_j + b$ *or* $C_i = x_i$*, and for* $i > k$*,* $C_i = 0$*.*

An ITA is a PITA with $P = \emptyset$. When all expressions occurring in guards and updates are in $\mathcal{L}in(X \cup P, \mathbb{Q})$, the PITA is said to be *additively parametrised*, in contrast to the general case, which is called *multiplicatively parametrised*.

We give a transition system describing the semantics of a PITA w.r.t. a parameter valuation π. A configuration (q, v) consists of a state q of the PITA and a clock valuation v.

Definition 2. *The semantics of a PITA* $\mathcal{A}$ *w.r.t. a parameter valuation* π *is defined by the (timed) transition system* $\mathcal{T}_{\mathcal{A},\pi} = (S, s_0, \to)$*. The set of configurations is* $S = \{(q, v) \mid q \in Q,\ v \in \mathbb{R}^X\}$*, with initial configuration* $s_0 = (q_0, \mathbf{0})$*. The relation* $\to$ *on* S *consists of two types of steps:*

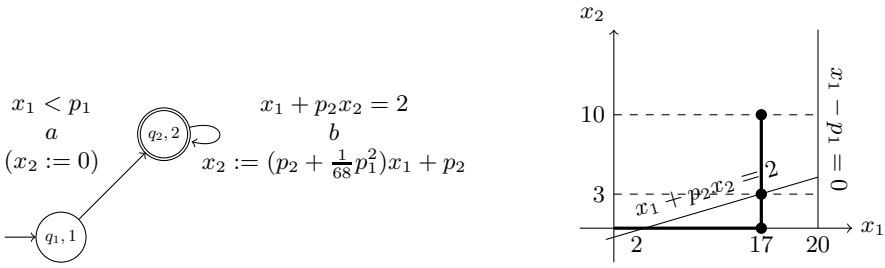
Time steps: *Only the active clock in a state can evolve, all other clocks are suspended. For a state* q *with active clock* $x_{\lambda(q)}$*, a time step of duration* d *is defined by* $(q, v) \xrightarrow{d} (q, v')$ *with* $v'(x_{\lambda(q)}) = v(x_{\lambda(q)}) + d$ *and* $v'(x) = v(x)$ *for any other clock* x*. We write* $v' = v +_q d$*.*

Discrete steps: *A discrete step* $(q, v) \xrightarrow{e} (q', v')$ *can occur for some transition* $e = q \xrightarrow{\varphi,a,u} q'$ *in* Δ *such that* $\pi, v \models \varphi$ *and* $v'(x) = \pi(v[u](x))$*.*

A *run* of $\mathcal{A}$ for some parameter valuation π is a finite path in the transition system $\mathcal{T}_{\mathcal{A},\pi}$, which can be written as an alternating sequence of (possibly null) time and discrete steps. A state $q \in Q$ is *reachable* from q_0 for π if there is a path from $(q_0, \mathbf{0})$ to (q, v) in $\mathcal{T}_{\mathcal{A},\pi}$, for some valuation v.

Example 1. A PITA $\mathcal{A}$ is depicted in Fig. 1(a), with two interrupt levels. Fixing the parameter valuation π: $p_1 = 20$ and $p_2 = -5$, the run $(q_1, 0, 0) \xrightarrow{17} (q_1, 17, 0) \xrightarrow{a} (q_2, 17, 0) \xrightarrow{3} (q_1, 17, 3) \xrightarrow{b} (q_2, 17, 10)$ is obtained as follows. After staying in q_1 for 17 time units, a can be fired and the value of x_1 is then frozen in state q_2, while x_2 increases. Transition b can be taken if $x_1 + p_2 x_2 = 2$, hence for $x_2 = 3$, after which x_2 is updated to $x_2 = 18p_2 + \frac{17}{68}p_1^2 = 10$. A geometric view of this run w.r.t. π is given (in bold) in Fig. 1(b).

Problems. We consider here reachability problems for PITA. Let $\mathcal{A}$ be a PITA with initial state q_0 and q be a state of $\mathcal{A}$. The *Existential (resp. Universal) Reachability Problem* asks whether q is reachable from q_0 for some (resp. all) parameter valuation(s). *Scoped* variants of these problems are obtained by adding



(a) A PITA $\mathcal{A}_1$ with two interrupt levels (b) A possible run in $\mathcal{A}$ for π

Fig. 1. An example of PITA and a possible execution

as input a set of parameter valuations given by a first order formula over the reals or a polyhedral constraint. The *Robust Reachability Problem* asks whether there exists a parameter valuation π and a real $\varepsilon > 0$ such that for all π' with $\|\pi - \pi'\|_\infty < \varepsilon$, q is reachable from q_0 for π' (where $\|\pi\|_\infty = max_{p \in P} |\pi(p)|$). When satisfied, this property ensures that small parameter perturbations do not modify the reachability result. It is also related to parameter synthesis where a valuation has to be enlarged to an open region with the same reachability goal.

3 Reachability Analysis

In this section, we give the main construction for the decidability result (Point 1 below), the remaining part of the proof is given in appendix.

Theorem 1. *1. The (scoped) existential, universal and robust reachability problems for PITA are decidable and belong to 2EXPSPACE. The complexity reduces to PSPACE when the number of clocks is fixed.*

2. The (polyhedral scoped) existential reachability problem is decidable for additively parametrised PITA, and belongs to 2EXPTIME. It belongs to PTIME when the number of clocks and parameters is fixed.

We briefly present the main ideas underlying the proof. Given a PITA $\mathcal{A}$, the first step is to build a finite partition of the set $\mathbb{R}^P$ of parameter valuations. An element Π of this partition is specified by a satisfiable first-order formula over $(\mathbb{R}, +, \times)$, with the parameters as variables. Intuitively, inside Π the qualitative behaviour of $\mathcal{A}$ does not depend on the precise parameter valuation. In a second step, we build a finite automaton $\mathcal{R}(\Pi)$ for each non empty Π. In $\mathcal{R}(\Pi)$, a state R, called a *class*, defines a set $[\![R]\!]_\pi$ of reachable configurations of $\mathcal{T}_{\mathcal{A},\pi}$ for a valuation $\pi \in \Pi$. The transition relation of $\mathcal{R}(\Pi)$ contains discrete steps $R \xrightarrow{e} R'$ (for a transition e of $\mathcal{A}$) and abstract time steps $R \to Post(R)$ with the following properties:

Discrete Step (DS): If there is a transition $R \xrightarrow{e} R'$ in $\mathcal{R}(\Pi)$ then for each $\pi \in \Pi$ and each $(q, v) \in [\![R]\!]_\pi$ there exists $(q', v') \in [\![R']\!]_\pi$ such that $(q, v) \xrightarrow{e} (q', v')$.

Conversely, let $\pi \in \Pi$ and $(q, v) \in [\![R]\!]_\pi$. If there exists a transition $(q, v) \xrightarrow{e} (q', v')$ in $\mathcal{T}_{\mathcal{A},\pi}$ then for some R', there is a transition $R \xrightarrow{e} R'$ in $\mathcal{R}(\Pi)$ and (q', v') belongs to $[\![R']\!]_\pi$.

Time Step (TS): Let $\pi \in \Pi$ and $(q, v) \in [\![R]\!]_\pi$. There exists $d > 0$ such that $(q, v +_q d) \in [\![Post(R)]\!]_\pi$ and for each d' with $0 \leq d' \leq d$, $(q, v +_q d') \in [\![R]\!]_\pi \cup [\![Post(R)]\!]_\pi$.

Hence, we obtain a finite family of abstract time bisimulations of the transition systems $\mathcal{T}_{\mathcal{A},\pi}$, for all parameter valuations, which gives the decidability result.

The key idea for the construction of $\mathcal{R}(\Pi)$ is based on the fact that, at some level k, the active clock x_k evolves in a one dimensional space and must be compared to a set E_k of expressions, the values of which are based on parameter values and the (fixed) clock values of levels below. For instance, in the automaton of Fig. 1(a), if $p_2 = 0$, the guard reduces to a comparison of $x_1 - 2$ with 0. If $p_2 \neq 0$, clock x_2 must be compared to $-\frac{x_1-2}{p_2}$ (in a sense depending on the sign of p_2 if the constraint was an inequality). After transition b is fired, updates must also be taken into account which leads to enlarge the set of expressions. Due to the syntactic restrictions of PITA this procedure terminates. Hence, we first need to define a set $PolPar$ of polynomials (appearing in the denominators like p_2) and a family $\{E_k\}_{k \leq n}$ of expressions in $\mathcal{L}in(X_k, \mathcal{F}rac(P, \mathbb{Q}))$.

3.1 Construction of $PolPar$ and Expressions $\{E_k\}_{k \leq n}$

We define operations on expressions, relatively to a level k, to help building the elements in E_k to which the clock x_k will be compared.

Definition 3. *Let $k \leq n$ be some level and let $C = \sum_{i \leq n} a_i x_i + b$ be an expression in $\mathcal{L}in(X, \mathcal{F}rac(P, \mathbb{Q}))$, with $a_k = \frac{r_k}{s_k}$, for some r_k and s_k in $\mathcal{P}ol(P, \mathbb{Q})$. We associate with C the following expressions:*

- $\texttt{lead}(C, k) = r_k$;
- *if* $\texttt{lead}(C, k) \notin \mathbb{Q} \setminus \{0\}$, $\texttt{comp}(C, k) = \sum_{i<k} a_i x_i + b$;
- *if* $\texttt{lead}(C, k) \neq 0$ *then* $\texttt{compnorm}(C, k) = -\sum_{i<k} \frac{a_i}{a_k} x_i - \frac{b}{a_k}$.

In the previous example, `comp` corresponds to $x_1 - 2$ while `compnorm` corresponds to $-\frac{x_1-2}{p_2}$. More examples are given after the construction of $PolPar$ and $\{E_k\}_{k \leq n}$. This construction proceeds top down from level n to level 1 after initialising $PolPar$ to $\emptyset$ and E_k to $\{x_k, 0\}$ for all k. When handling level k, we add new terms to E_i for $1 \leq i \leq k$.

1. At level k the first step consists in adding new expressions to E_k and new polynomials to $PolPar$. More precisely, let C be any expression occurring in a guard of an edge leaving a state of level k. We add $\texttt{lead}(C, k)$ to $PolPar$ when it does not belong to $\mathbb{Q}$ and we add $\texttt{comp}(C, k)$ and $\texttt{compnorm}(C, k)$ to E_k when they are defined.
2. The second step consists in iterating the following procedure until no new term is added to any E_i for $1 \leq i \leq k$.

(a) Let $q \xrightarrow{\varphi,a,u} q'$ with $\lambda(q) \geq k$ and $\lambda(q') \geq k$, and let $C \in E_k$. Then we add $C[u]$ to E_k (recall that $C[u]$ is the expression obtained by applying update u to C).

(b) Let $q \xrightarrow{\varphi,a,u} q'$ with $\lambda(q) < k$ and $\lambda(q') \geq k$. Let $\{C, C'\}$ be a set of two expressions in E_k. We compute $C'' = C[u] - C'[u]$, choosing an arbitrary order between C and C'. This step ends by handling C'' w.r.t. $\lambda(q)$ as done for C w.r.t. k in step 1 above.

Example 2. For the automaton of Fig. 1(a), initially, we have $PolPar = \emptyset$, $E_1 = \{x_1, 0\}$ and $E_2 = \{x_2, 0\}$. Starting with level $k = 2$, we consider in step 1 the expression $C_2 = p_2x_2 + x_1 - 2$ appearing in the guard of the single edge leaving q_2. We compute $\texttt{lead}(C_2, 2) = p_2$, $\texttt{comp}(C_2, 2) = x_1 - 2$, and $\texttt{compnorm}(C_2, 2) = -\frac{x_1-2}{p_2}$. We obtain $PolPar = \{p_2\}$ and $E_2 = \{x_2, 0, x_1 - 2, -\frac{x_1-2}{p_2}\}$. For step 2(a) and the same edge, we apply its update to the expressions of E_2 that contain x_2, add them to E_2, and thus obtain $E_2 = \{x_2, 0, x_1 - 2, -\frac{x_1-2}{p_2}, (p_2 + \frac{1}{68}p_1^2)x_1 + p_2\}$.

In step 2(b), considering the single edge from q_1 to q_2, we compute the differences between any two expressions from E_2 (after applying update) and the resulting expressions `lead`, `comp` and `compnorm`, which yields: $PolPar = \{p_2, p_2 + 1, 1 - p_2 - \frac{1}{68}p_1^2, -p_2^2 - \frac{1}{68}p_1^2p_2 - 1\}$ and $E_1 = \{x_1, 0, 2, -\frac{2(p_2+1)}{p_2}, -2 - p_2, \frac{2+p_2}{1-p_2-\frac{1}{68}p_1^2}, \frac{2-p_2^2}{p_2}, \frac{2-p_2^2}{1+p_2^2+\frac{1}{68}p_1^2p_2}\}$.

We proceed with level 1 and add $\texttt{compnorm}(C_1, 1) = p_1$ to E_1, hence:
$E_1 = \{x_1, 0, 2, -\frac{2(p_2+1)}{p_2}, -2 - p_2, \frac{2+p_2}{1-p_2-\frac{1}{68}p_1^2}, \frac{2-p_2^2}{p_2}, \frac{2-p_2^2}{1+p_2^2+\frac{1}{68}p_1^2p_2}, p_1\}$,
$E_2 = \{x_2, 0, x_1 - 2, -\frac{x_1-2}{p_2}, (p_2 + \frac{1}{68}p_1^2)x_1 + p_2\}$ and,
$PolPar = \{p_2, p_2 + 1, 1 - p_2 - \frac{1}{68}p_1^2, -p_2^2 - \frac{1}{68}p_1^2p_2 - 1\}$.

Lemma 1 below is used for the class automata construction. Its proof is obtained by a straightforward examination of the above procedure. The other two lemmata are related to the termination and complexity of this procedure and used in the computation of the upper bound of the reachability algorithm. This algorithm manipulates rationals numbers (resp. rational functions) as pairs of integers (resp. polynomials).

Lemma 1. *Let C belong to E_k for some k and $c = \frac{r}{s}$ be a coefficient of C with $s \notin \mathbb{Q}$. Then there exists polynomials $P_1, \ldots, P_\ell \in PolPar$ and some constant $K \in \mathbb{Q} \setminus \{0\}$ such that $s = K.\prod_{1 \leq i \leq \ell} P_i$.*

Lemma 2. *The construction procedure of $\{E_k\}_{k \leq n}$ terminates and the size of every E_k is bounded by $(2E + 2)^{2^{n(n-k+1)}+1}$ where E is the number of atomic propositions in edges of the PITA.*

Lemma 3. *Let $\mathcal{A}$ be a PITA, and let b_0 be the maximal number of bits for integers and d_0 the maximal degree of polynomials, occurring in $\mathcal{A}$. If b is the number of bits of an integer constant and d is the degree of a polynomial, occurring in an expression of $PolPar$ or some E_k, then $b \leq (n+2)!2^n b_0$ and $d \leq (n+2)!d_0$.*

We now explain the partition construction. Starting from the finite set $PolPar$, we split the set of parameter valuations in parameter regions specified by the result of comparisons to 0 of the values of the polynomials in $PolPar$. For instance, for the set $PolPar$ computed above, the inequalities $p_2 < 0$, $p_2 + 1 < 0$, $1 - p_2 - \frac{1}{68}p_1^2 > 0$ and $-1 - p_2^2 - \frac{1}{68}p_1^2 p_2 > 0$ define a set $preg$ of parameter valuations containing $p_1 = 20$ and $p_2 = -5$. The set of non empty such regions can be computed by solving an existential formula of the first-order theory of reals.

Then, given a non empty parameter region $preg$, we consider the following subset of E_k for $1 \leq k \leq n$: $E_{k,preg} = \{C \in E_k \mid \text{the denominators of coefficients of } C \text{ are non null in } preg\}$. Due to Lemma 1, these subsets are obtained by examining the specification of $preg$.

Observe that expressions in $E_{1,preg} \setminus \{x_1\}$ belong to $\mathcal{F}rac(P, \mathbb{Q})$ and that, depending on the parameter valuation, two different expressions can produce the same value. We refine $preg$ according to a linear pre-order $\preceq_1$ on $E_{1,preg} \setminus \{x_1\}$ which is satisfiable within $preg$. We denote this refined region by $\Pi = (preg, \preceq_1)$ and we now build a finite automaton $\mathcal{R}(\Pi)$.

3.2 Construction of the Class Automata

In this paragraph, we fix a non empty parameter region $\Pi = (preg, \preceq_1)$.

Class Definition. A state of $\mathcal{R}(\Pi)$, called a class, is defined as a pair $R = (q, \{\preceq_k\}_{1 \leq k \leq \lambda(q)})$ where q is a state of $\mathcal{A}$ and $\preceq_k$ is a total preorder over $E_{k,preg}$, for $1 \leq k \leq \lambda(q)$. For a parameter valuation $\pi \in \Pi$, the class R describes the following subset of configurations in $\mathcal{T}_{\mathcal{A},\pi}$:

$$[\![R]\!]_\pi = \{(q, v) \mid \forall k \leq \lambda(q)\ \forall g, h \in E_{k,preg},\ \pi(v(g)) \leq \pi(v(h)) \text{ iff } g \preceq_k h\}$$

The initial state of $\mathcal{R}(\Pi)$ is the class R_0, such that $(q_0, \mathbf{0}) \in [\![R_0]\!]_\pi$, which can be straightforwardly determined by extending $\preceq_1$ to $E_{1,preg}$ with $x_1 \preceq_1 0$ and $0 \preceq_1 x_1$ and closing $\preceq_1$ by transitivity.

As usual, transitions in $\mathcal{R}(\Pi)$ consist of discrete and time steps:

Discrete Step. Let $R = (q, \{\preceq_i\}_{1 \leq i \leq \lambda(q)})$ and $R' = (q', \{\preceq'_i\}_{1 \leq i \leq \lambda(q')})$ be two classes. There is a transition $R \xrightarrow{e} R'$ for a transition $e : q \xrightarrow{\varphi, a, u} q'$ if for some $\pi \in \Pi$, there is some $(q, v) \in [\![R]\!]_\pi$ and $(q', v') \in [\![R']\!]_\pi$ such that $(q, v) \xrightarrow{e} (q', v')$. In this case, for all $(q, v) \in [\![R]\!]_\pi$ there is a $(q', v') \in [\![R']\!]_\pi$ such that $(q, v) \xrightarrow{e} (q', v')$. We prove in the sequel that the existence of transition $R \xrightarrow{e} R'$ is independent of $\pi \in \Pi$ and of $(q, v) \in [\![R]\!]_\pi$. It can be decided as follows.

Firability condition. Write $\varphi = \bigwedge_{j \in J} C_j \bowtie_j 0$. For a given j, let us write $C_j = \sum_{i \leq \lambda(q)} a_i x_i + b$. We consider three cases.

• **Case** $a_{\lambda(q)} = 0$**.** Then $C_j = \texttt{comp}(C_j, \lambda(q)) \in E_{\lambda(q),preg}$ and using the positions of 0 and C_j w.r.t. $\preceq_{\lambda(q)}$, we can decide whether $C_j \bowtie_j 0$.

• **Case** $a_{\lambda(q)} \in \mathbb{Q} \setminus \{0\}$**.** Then $\texttt{compnorm}(C_j, \lambda(q)) \in E_{\lambda(q),preg}$, hence using the sign of $a_{\lambda(q)}$ and the positions of $x_{\lambda(q)}$ and $\texttt{compnorm}(C_j, \lambda(q))$ w.r.t. $\preceq_{\lambda(q)}$, we can decide whether $C_j \bowtie_j 0$.

• **Case** $a_{\lambda(q)} \notin \mathbb{Q}$**.** According to the specification of *preg*, we know the sign of $a_{\lambda(q)}$ as it belongs to $PolPar$. In case $a_{\lambda(q)} = 0$, we decide as in the first case. Otherwise, we decide as in the second case.

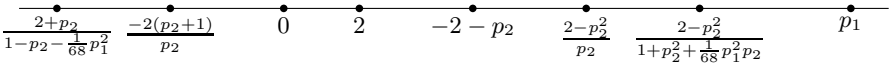
Successor R' definition. Let $k \leq \lambda(q')$ and $g', h' \in E_{k,preg}$.

1. Either $k \leq \lambda(q)$, by step 2(a) of the construction, $g'[u], h'[u] \in E_{k,preg}$. Then $g' \preceq'_k h'$ iff $g'[u] \preceq_k h'[u]$.
2. Or $k > \lambda(q)$, let $D = g'[u] - h'[u] = \sum_{i \leq \lambda(q)} a_i x_i + b$.
   • **Case** $a_{\lambda(q)} = 0$**.** Then $D = \texttt{comp}(D, \lambda(q)) \in E_{\lambda(q),preg}$, so we can decide whether $D \preceq_{\lambda(q)} 0$ and $g' \preceq'_k h'$ iff $D \preceq_{\lambda(q)} 0$.
   • **Case** $a_{\lambda(q)} \in \mathbb{Q} \setminus \{0\}$**.** Then $\texttt{compnorm}(D, \lambda(q)) \in E_{\lambda(q),preg}$. There are four subcases to consider. For instance if $a_{\lambda(q)} > 0$ and $x_{\lambda(q)} \preceq_{\lambda(q)} \texttt{compnorm}(D, \lambda(q))$ then $g' \preceq'_k h'$. The other subcases are similar.
   • **Case** $a_{\lambda(q)} \notin \mathbb{Q}$**.** Let us write $a_{\lambda(q)} = \frac{r_{\lambda(q)}}{s_{\lambda(q)}}$. According to the specification of *preg*, we know the sign of $a_{\lambda(q)}$ as $r_{\lambda(q)}$ belongs to $PolPar$ and $s_{\lambda(q)}$ is a product of items in $PolPar$. In case $a_{\lambda(q)} = 0$, we decide $g' \preceq'_k h'$ as in the first case. Otherwise, we decide in a similar way as in the second case. For instance if $a_{\lambda(q)} > 0$ and $x_{\lambda(q)} \preceq_{\lambda(q)} \texttt{compnorm}(D, \lambda(q))$ then $g' \preceq'_k h'$.

Time Step. For $R = (q, \{\preceq_k\}_{1 \leq k \leq \lambda(q)})$, there is a transition $R \xrightarrow{succ} Post(R)$, where $Post(R) = (q, \{\preceq'_k\}_{1 \leq k \leq \lambda(q)})$ is the time successor of R, defined as follows. Intuitively, all preorders below $\lambda(q)$ are fixed, so $\preceq'_i = \preceq_i$ for each $i < \lambda(q)$. On level $\lambda(q)$, the clock value simply progresses along the one dimensional time line, where the expressions are ordered. More precisely, let $\sim$ be the equivalence relation $\preceq_{\lambda(q)} \cap \preceq^{-1}_{\lambda(q)}$ induced by the preorder. A $\sim$-equivalence class groups expressions yielding the same value, and on these classes, the (total) preorder becomes a (total) order. Let V be the $\sim$-equivalence class containing $x_{\lambda(q)}$.

1. Either $V = \{x_{\lambda(q)}\}$. If V is the greatest $\sim$-equivalence class, then $\preceq'_{\lambda(q)} = \preceq_{\lambda(q)}$ (and $Post(R) = R$). Otherwise, let V' be the next $\sim$-equivalence class. Then $\preceq'_{\lambda(q)}$ is obtained by merging V and V', and preserving $\preceq_{\lambda(q)}$ elsewhere.
2. Or V is not a singleton. Then we split V into $V \setminus \{x_{\lambda(q)}\}$ and $\{x_{\lambda(q)}\}$ and "extend" $\preceq_{\lambda(q)}$ by $V \setminus \{x_{\lambda(q)}\} \preceq'_{\lambda(q)} \{x_{\lambda(q)}\}$.

Example 3. This construction is illustrated on automaton $\mathcal{A}_1$ of Fig. 1(a), for the region $\Pi = (preg, \preceq_1)$, where *preg* was defined above by: $p_2 < 0$, $p_2 + 1 < 0$, $1 - p_2 - \frac{1}{68}p_1^2 > 0$ and $-1 - p_2^2 - \frac{1}{68}p_1^2 p_2 > 0$ and $\preceq_1$ is the ordering of the expressions in $E_{1,preg} = E_1$ specified by the line below.



A part of the resulting class automaton $\mathcal{R}(\Pi)$, including the run corresponding to the one in Fig. 1(b), is depicted in Fig. 2, where dashed lines indicate (abstract) time steps. The initial class is $R_0 = (q_0, Z_0)$ where Z_0 is $\preceq_1$ extended

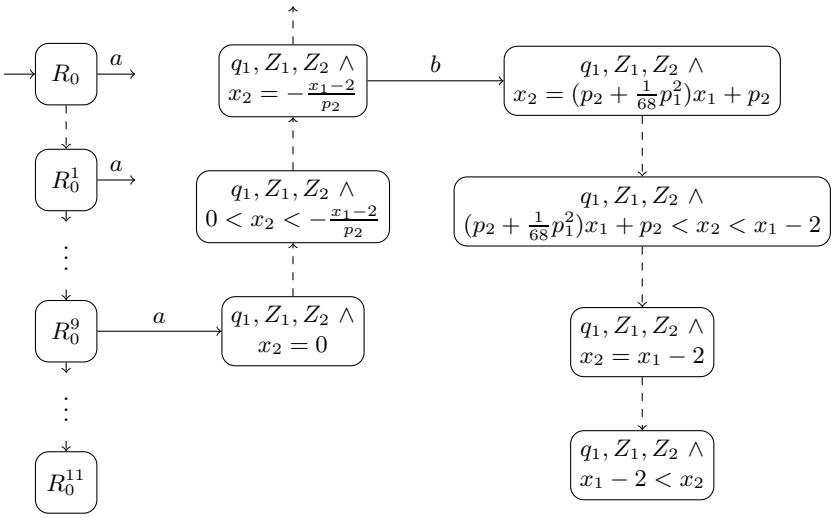
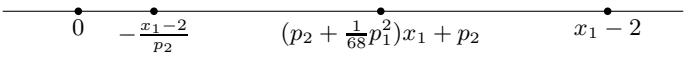


Fig. 2. A part of $\mathcal{R}(\Pi)$ for $\mathcal{A}_1$

with $x_1 = 0$. Time successors of the initial state are obtained by moving x_1 to the right along the line: $R_0^1 = (q_0, \preceq_1 \wedge\ 0 < x_1 < 2)$, $R_0^2 = (q_0, \preceq_1 \wedge\ x_1 = 2)$, ..., up to $R_0^{11} = (q_0, \preceq_1 \wedge\ p_1 < x_1)$.

Transition a can be fired from all these classes except from R_0^{10} and R_0^{11}. In Fig. 2, we represent only the one from R_0^9, and we denote by Z_1 the ordering $\preceq_1$ extended with $\frac{2-p_2^2}{1+p_2^2+\frac{1}{68}p_1^2p_2} < x_1 < p_1$. Region Π and Z_1 determine the ordering $Z_2 = \preceq_2$ on $E_{2,preg} \setminus \{x_2\} = E_2 \setminus \{x_2\}$, specified by the line below. This firing produces $R_1 = (q_1, Z_1, Z_2 \wedge x_2 = 0))$. Transition b is fired from the time (second) successor of R_1 for which $x_2 = -\frac{x_1-2}{p_2}$.



To conclude, observe that the automaton $\mathcal{R}(\Pi)$ defined above has the properties **(DS)** and **(TS)** mentionned previously, and is hence a finite time abstract bisimulation of $\mathcal{T}_{\mathcal{A},\pi}$, for all parameter valuations $\pi \in \Pi$.

4 Conclusion

While seminal results on parametrised timed models leave little hope for decidability in the general case, we provide here an expressive formalism for the analysis of parametric reachability problems. Our setting includes a restricted form of stopwatches and polynomials in the parameters occurring as both additive and multiplicative coefficients of the clocks in guards and updates. We plan to investigate which kind of timed temporal logic would be decidable on PITA.

References

1. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: ACM Symp. on Theory of Computing, pp. 592–601. ACM (1993)
2. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
3. Bérard, B., Fribourg, L.: Automated verification of a parametric real-time program: The ABR conformance protocol. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 96–107. Springer, Heidelberg (1999)
4. Miller, J.S.: Decidability and complexity results for timed automata and semi-linear hybrid automata. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 296–309. Springer, Heidelberg (2000)
5. Doyen, L.: Robust Parametric Reachability for Timed Automata. Information Processing Letters 102(5), 208–213 (2007)
6. André, É., Chatain, T., Encrenaz, E., Fribourg, L.: An inverse method for parametric timed automata. Int. J. of Foundations of Comp. Sci. 20(5), 819–836 (2009)
7. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012)
8. Jovanović, A., Lime, D., Roux, O.H.: Integer parameter synthesis for timed automata. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 401–415. Springer, Heidelberg (2013)
9. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic Symbolic Verification of Embedded Systems. IEEE Transactions on Software Engineering 22, 181–201 (1996)
10. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A Model-Checker for Hybrid Systems. Software Tools for Technology Transfer 1, 110–122 (1997)
11. Bozzelli, L., Torre, S.L.: Decision problems for lower/upper bound parametric timed automata. Formal Methods in System Design 35(2), 121–151 (2009)
12. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.: Linear parametric model checking of timed automata. J. of Logic and Alg. Prog. 52-53, 183–220 (2002)
13. Jovanović, A., Faucou, S., Lime, D., Roux, O.H.: Real-time control with parametric timed reachability games. In: WODES 2012, IFAC, pp. 323–330 (2012)
14. Bérard, B., Haddad, S.: Interrupt Timed Automata. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 197–211. Springer, Heidelberg (2009)
15. Bérard, B., Haddad, S., Sassolas, M.: Interrupt timed automata: Verification and expressiveness. Formal Methods in System Design 40(1), 41–87 (2012)

Deciding Continuous-Time Metric Temporal Logic with Counting Modalities*

Marcello M. Bersani[1], Matteo Rossi[1], and Pierluigi San Pietro[1,2]

[1] Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano
[2] CNR IEIIT-MI, Milano, Italy
{marcellomaria.bersani,matteo.rossi,pierluigi.sanpietro}@polimi.it

Abstract. We present a satisfiability-preserving translation of QTL formulae with counting modalities interpreted over finitely variable signals into formulae of the CLTL-over-clocks logic. The satisfiability of CLTL-over-clocks can be solved through a suitable encoding into the input logics of SMT solvers, so our translation constitutes an effective decision procedure for QTL with counting modalities. It is known that counting modalities increase the expressiveness of QTL (hence also of the expressively equivalent MITL logic); to the best of our knowledge, our decision procedure for such modalities is the first actually implemented.

1 Introduction

When developing computer systems that monitor and control time-continuous physical quantities, such as position, speed, temperature or pressure, the typical discrete-time models of computer science, e.g., finite state machines, may no longer be adequate. To address these shortcomings, many notations, and related tools, supporting continuous-time have been developed [10], most often based on operational mechanisms, e.g., Timed Automata [3] supported by tools such as Uppaal [20]. Descriptive notations, such as temporal logics, have been used mainly as a concise and convenient way to express the required properties of a system, in the path of traditional verification of finite-state models through model checking [5]. Temporal logics, however, also allow designers to pursue a descriptive approach to the specification and modeling of reactive systems (see, e.g., [16,10]), where the system is defined by means of its general properties, rather than by a machine behaving in the desired way. In this case, verification typically consists of satisfiability checking of the conjunction of the model and of the (negation of) its desired properties.

Quantitative Temporal Logic (QTL [11]) is one of the more interesting continuous-time temporal logics: its satisfiability is PSPACE-complete; it has a very simple syntax, with only one metric operator, for expressing statements of the form "Event θ will happen within one time unit"; nevertheless, it is expressively

* Work supported by the Programme IDEAS-ERC, Project 227977-SMScom, and by PRIN Project 2010LYA9RH-006.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 70–82, 2013.

equivalent with other syntactically richer logics, in particular with the Quantitative Monadic Logic of Order (QMLO), and with the Metric Interval Temporal Logic (MITL). In fact, a translation has been defined that, from a QTL formula, produces an equivalent QMLO (resp. MITL) formula, and vice-versa.

However, the expressive power of QTL, and of other typical metric temporal logics on continuous time, is often not enough. Pnueli conjectured (as later proved in [12]) that logics such as QTL and MITL are unable to express many naturally-occurring constraints, such as "Events θ_1 and θ_2 will both happen within one time unit". This has led [12] to define an extension of QTL, by introducing a new modality, called $\mathbf{P}_2$, where $\mathbf{P}_2(\theta_1, \theta_2)$ means that θ_1 and θ_2 will both happen within one time unit, with θ_1 occurring before θ_2. This "Pnueli modality" was also generalized, by introducing, for every natural number $n > 0$, the modality $\mathbf{P}_n(\theta_1, \ldots, \theta_n)$, with the meaning that it holds in t if there exists an increasing sequence of time instants $t < t_1 < t_2 < \cdots < t_n < t + 1$ such that every θ_i holds at t_i, for all $1 \leqslant i \leqslant n$. For every $\mathbf{P}_n$ there is also corresponding modality $\overleftarrow{\mathbf{P}}_n$ with the same meaning, but in the past.

The rationale for introducing these "generalized" modalities is that QTL with the $\mathbf{P}_n$ modality is strictly more powerful than QTL with only the $\mathbf{P}_{n-1}$ modality. For instance, a language, say, with only $\mathbf{P}_2$ cannot express constraints about the occurrence of three events in the next time unit. [12] also introduced a simpler "counting" modality, defined as $\mathbf{C}_n(\theta) = \mathbf{P}_n(\theta, \ldots, \theta)$, i.e., with the meaning that θ holds in at least n time instants in the open unit interval ahead. $\overleftarrow{\mathbf{C}}_n$ is the past version of $\mathbf{C}_n$. According to [13], each $\mathbf{C}_n$ modality is strictly more powerful than the $\mathbf{C}_{n-1}$ modality and, somewhat surprisingly, the logic *QTLc*, defined as QTL extended with every $\mathbf{C}_n$ modality, is as expressive as *QTLp*, i.e., QTL extended with every $\mathbf{P}_n$ modality. Satisfiability of QTLc is PSPACE-complete when each index n in a modality $\mathbf{C}_n$ is encoded in unary, although it is EXPSPACE-complete if n is encoded in binary [17].

In general, the verification of continuous-time temporal logics is not as well-supported by tools as for discrete-time models, especially when the logic is endowed with metric operators. Decision procedures for determining satisfiability mostly rely on timed automata-based techniques [4,14,18] but they appear to be very difficult to realize in practice. To the best of our knowledge, no implementation exists for them. [19] also defines a decision procedure for a temporal logic that is equi-expressive with MITL, one that is essentially akin to building a suitable automaton. Finally, the techniques used in [12,17,13] to prove decidability and complexity of QTLc appear to be remote from any actual implementation.

In this paper we introduce a satisfiability-preserving translation from QTLc formulae into formulae of CLTL-over-clocks, a decidable logic whose satisfiability is PSPACE-complete. This translation is polynomial in the size of the QTLc formula, hence preserving the complexity, at least when counting modalities are encoded in unary. The advantage of CLTL-over-clocks is that it allows the definition of a decision procedure based on Satisfiability Modulo Theories (SMT) techniques that are implemented in a variety of tools. A prototype tool for CLTL-over-clocks already exists [8], hence an actual implementation of a satisfiability

checker for QTLc can be obtained through the (almost straightforward) implementation of the translation described in this paper.

For brevity, this paper only shows the translation for the case of $\mathbf{C}_n$ modalities, the case of the past $\overleftarrow{\mathbf{C}}_n$ modalities being very similar. Also, we do not show the encoding of the $\mathbf{P}_n$ modalities, which is more involved than for the $\mathbf{C}_n$ modalities, though it is known that QTLp and QTLc are expressively equivalent.

Although QTLc is decidable over unrestricted models, we will focus on models that are *finitely variable*, i.e. such that in every bounded time interval there can only be a finite number of changes. This is a very common requirement for continuous-time models, which only rules out pathological behaviors (e.g., Zeno [10]) which do not have much practical interest.

The paper is organized as follows: Sect. 2 defines QTLc and CLTL-over-clocks, and Sect. 3 defines a reduction from the former to the latter; Sect. 4 shows that the translation is satisfiability-preserving, and discusses its complexity. Sect. 5 concludes.

All proofs and parts of the translation are in the full version of the paper.

2 Languages

We define the syntax and semantics of QTLc, i.e., the general case where counting modalities are allowed.

Let AP be a finite set of atomic propositions and $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots\}$ be the infinite sets of (names of) counting modalities, respectively. The syntax of (well-formed) QTLc formulae over AP and $\mathcal{C}$ is defined by the grammar:

$$\phi := p \mid \phi \wedge \phi \mid \neg\phi \mid \phi \mathbf{U}_{(0,\infty)} \phi \mid \phi \mathbf{S}_{(0,\infty)} \phi \mid \mathbf{C}_i(\phi) \mid \overleftarrow{\mathbf{C}}_i(\phi)$$

where p is in AP, $i > 0$ stands for any integer constant (note that $\mathbf{C}_1$ is the usual "eventually" operator $\mathbf{F}_{(0,1)}$ of QTL).

The semantics of QTLc may be defined with respect to a generic linear order, but in what follows we will focus on the nonnegative real line, i.e., the linear order $(\mathbb{R}_{\geqslant 0}, <)$. A *structure* M for QTLc over alphabet AP is a pair $M = \langle \mathbb{R}_{\geqslant 0}, \mathcal{B}^M \rangle$, where $\mathcal{B}^M$ is a valuation mapping every propositional variable $p \in AP$ to a set $\mathcal{B}^M(p) \subseteq \mathbb{R}_{\geqslant 0}$. Hence, a structure may be considered as providing continuous-time Boolean *signals* over the set AP. *Satisfaction* of a QTLc formula over M at a point $t \in \mathbb{R}_{\geqslant 0}$ is a relation $\models$ defined inductively as in Table 1. Given a QTLp formula ϕ, $sub(\phi)$ is the set of its subformulae.

Constraint LTL (CLTL [9,6]) formulae are defined with respect to a finite set V of variables and a structure $\mathcal{D} = (D, \mathcal{R})$ where D is a specific domain of interpretation for variables and constants and $\mathcal{R}$ is a family of relations on D, with the set AP of atomic proposition being the set $\mathcal{R}_0$ of 0-ary relations. An *atomic constraint* is a term of the form $R(x_1, \ldots, x_n)$, where R is an n-ary relation of $\mathcal{R}$ on D and $x_1, \ldots, x_n \in V$. A *valuation* is a mapping $v : V \to D$. A constraint is *satisfied* by v, written $v \models_{\mathcal{D}} R(x_1, \ldots, x_n)$, if $(v(x_1), \ldots, v(x_n)) \in R$.

Table 1. Semantics of QTLc (past operators are omitted for brevity)

$$
\begin{aligned}
M,t \models p &\Leftrightarrow t \in \mathcal{B}^M(p)\\
M,t \models \neg\phi &\Leftrightarrow M,t \not\models \phi\\
M,t \models \phi \wedge \psi &\Leftrightarrow M,t \models \phi \text{ and } M,t \models \psi\\
M,t \models \phi \mathbf{U}_{(0,\infty)} \psi &\Leftrightarrow \exists t' > t : M,t' \models \psi \text{ and } \forall t'', 0 < t'' < t', M,t'' \models \phi\\
M,t \models \mathbf{C}_i(\phi) &\Leftrightarrow \exists t_1, \ldots t_i, t < t_1 < \ldots t_i < t+1, \text{ and } M,t_k \models \phi \text{ for all } k \in \{1 \ldots i\}
\end{aligned}
$$

Table 2. Semantics of CLTL (propositional connectives are omitted for brevity)

$$
\begin{aligned}
(\pi,\sigma), i \models p &\Leftrightarrow p \in \pi(i) \text{ for } p \in AP\\
(\pi,\sigma), i \models R(\alpha_1, \ldots, \alpha_n) &\Leftrightarrow (\sigma(i + |\alpha_1|, x_{\alpha_1}), \ldots, \sigma(i + |\alpha_n|, x_{\alpha_n})) \in R\\
(\pi,\sigma), i \models \mathbf{X}(\phi) &\Leftrightarrow (\pi,\sigma), i+1 \models \phi\\
(\pi,\sigma), i \models \mathbf{Y}(\phi) &\Leftrightarrow (\pi,\sigma), i-1 \models \phi \wedge i > 0\\
(\pi,\sigma), i \models \phi \mathbf{U}_\psi &\Leftrightarrow \exists\, j \geqslant i : (\pi,\sigma), j \models \psi \ \wedge (\pi,\sigma), n \models \phi \ \forall\ i \leqslant n < j\\
(\pi,\sigma), i \models \phi \mathbf{S}_\psi &\Leftrightarrow \exists\, 0 \leqslant j \leqslant i : (\pi,\sigma), j \models \psi \ \wedge (\pi,\sigma), n \models \phi \ \forall\ j < n \leqslant i
\end{aligned}
$$

Temporal terms α are defined by the syntax $\alpha := c \mid x \mid \mathrm{X}\alpha$, where c is a constant in D and $x \in V$. CLTL formulae are defined as follows:

$$\phi := R(\alpha_1, \ldots, \alpha_n) \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}(\phi) \mid \mathbf{Y}(\phi) \mid \phi \mathbf{U} \phi \mid \phi \mathbf{S} \phi$$

where α_i's are temporal terms, $R \in \mathcal{R}$, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since" operators of LTL, with the same meaning. Operator X is similar to $\mathbf{X}$, but it only applies to temporal terms, with the meaning that $\mathrm{X}\alpha$ is the *value* of temporal term α in the next time instant. Operators "globally" $\mathbf{G}$ and "release" $\mathbf{R}$ are introduced as customary as abbreviations: $\phi_1 \mathbf{R} \phi_2 = \neg(\neg\phi_1 \mathbf{U} \neg\phi_2)$, $\mathbf{G}(\phi) = \bot \mathbf{R} \neg\phi$.

The *depth* $|\alpha|$ of a temporal term is the total amount of temporal shift needed in evaluating α: $|x| = 0$ when x is a variable, and $|\mathrm{X}\alpha| = |\alpha| + 1$. The semantics of CLTL formulae is defined with respect to a strict linear order representing time $(\mathbb{N}, <)$. Truth values of propositions in AP and values of variables belonging to V are defined by a pair (π, σ), where $\sigma : \mathbb{N} \times V \to D$ and $\pi : \mathbb{N} \to \wp(AP)$, which define the value of variables and a subset of AP for each element of $\mathbb{N}$. The value of terms is defined with respect to σ by $\sigma(i, \alpha) = \sigma(i + |\alpha|, x_\alpha)$, assuming that x_α is the variable in V occurring in term α. The semantics of a CLTL formula ϕ at instant $i \geqslant 0$ over a pair (π, σ) is recursively defined as in Table 2, where x_{α_i} is the variable that appears in temporal term α_i, and $R \in \mathcal{R} \backslash \mathcal{R}_0$ (recall that $\mathcal{R}_0 = AP$). A formula CLTL ϕ is *satisfiable* if there exists a pair (π, σ) such that $(\pi, \sigma), 0 \models \phi$; in this case, we say that (π, σ) is a *model* of ϕ.

In this paper, we restrict the set of pairs (π, σ) which can be model for formulae to the ones where variables in V are evaluated as *clocks*. A clock "measures" the

time elapsed since the last time it was "reset" (i.e., the variable was equal to 0). Each position $i \in \mathbb{N}$ is associated with a "time delay" $\delta(i)$, where $\delta(i) > 0$ for all i, which corresponds to the "time elapsed" between the current position i and the next one $i + 1$. More precisely, given a clock x,

$$\sigma(i+1, x) = \begin{cases} \sigma(i, x) + \delta(i), & \text{time elapsing} \\ 0 & \text{reset } x. \end{cases}$$

The set $\mathcal{R}$ is restricted to $\{<, =\}$ because CLTL formulae need only to measure the time elapsing among events, as later explained. Under these two restrictions, the resulting logic is called CLTL-over-clocks and it is decidable (as shown in [8]); an effective decision procedure can be devised by encoding it into the decidable theory of Quantifier-free Linear Real Arithmetic, which can be solved by SMT solvers such as, for example, Z3 [15]. A prototype solver for CLTL-over-clocks formulae is available as part of the Zot tool [2].

3 Reduction of QTLc to CLTL-Over-Clocks

Reducing QTLc to CLTL-over-clocks requires a way to represent models of QTL formulae, i.e., continuous-time signals over a finite set of atomic propositions, by means of CLTL models where time is discrete. In CLTL, variables behaving as clocks represent time progress, while discrete positions in models represent, for each subformula occurring in ϕ, whether a change of truth value (an "event") occurs or not for the subformula at that point. Time progress between two discrete points is measured by clocks; between events, the truth value of formulae is stable (i.e., there is no change). CLTL models embed, in every (discrete) position, the information defining both the truth value of all the subformulae occurring in QTLc formula ϕ and the time progress between two consecutive changing points. Therefore, our reduction defines, by means of CLTL-over-clocks formulae, the semantics of every subformula occurring in ϕ.

Consider a QTLc formula ϕ. For each subformula θ of ϕ we introduce two predicates, $\restriction_\theta$ and $\overleftarrow{\theta}$, which represent the value of θ in, respectively, the first instant and the rest of the interval between two events (hence, $\restriction_\theta$ represents the value of θ exactly when the event occurs). We also introduce n_θ clocks $z_\theta^0, \ldots z_\theta^{n_\theta - 1}$, where, if n' is the greatest value such that $\mathbf{C}_{n'}(\theta)$ appears in ϕ, then $n_\theta = \max(n, n') + 1$ if $\theta = \mathbf{C}_n(\gamma)$, and $n_\theta = \max(1, n') + 1$ otherwise. Clocks z_θ^i (with $i \in \{0, \ldots n_\theta - 1\}$) measure the time elapsed since the last n_θ "events" of θ. Let $\theta \in sub(\phi)$. Define the event "become true" e_θ^u to occur at time instant $t \geqslant 0$ of a signal M when: $\exists \varepsilon > 0$ s.t. $\forall t' \in (t, t + \varepsilon)$ it is $M, t \models \theta$ and either $t = 0$ or $\exists \varepsilon > 0$ s.t. $\forall t' \in (t - \varepsilon, t)$ it is $M, t \models \neg\theta$.
The opposite event "become false" e_θ^d may be defined by replacing θ with $\neg\theta$ in the above definition. QTLc events e_θ^u and e_θ^d are represented in the CLTL formula through combinations of the basic predicates $\restriction_\theta$ and $\overleftarrow{\theta}$ that are abbreviated by $\lrcorner_\theta$ and $\llcorner_\theta$, respectively, whose definitions are shown in Table 3.

Table 3. CLTL predicates and abbreviations used in the encoding. Note that $\mathbf{Y}(\overleftarrow{\xi})$ and $\mathbf{Y}(\neg \overleftarrow{\xi})$ are false in the origin, no matter ξ, and elsewhere $\neg\mathbf{Y}(\neg \overleftarrow{\xi}) \equiv \mathbf{Y}(\overleftarrow{\xi})$; hence, $\lrcorner_\xi$ holds in 0 iff $\overleftarrow{\xi}$ holds there, $\lrcorner\!\!\lrcorner_\xi$ does not hold in 0, and so on.

$\uparrow\!\xi$ $= \xi$ holds in the first instant of the current interval	
$\overleftarrow{\xi}$ $= \xi$ holds in the current interval (save possibly for its first instant)	
$\lrcorner_\xi = \neg\mathbf{Y}(\overleftarrow{\xi}) \wedge \overleftarrow{\xi}$	$\lrcorner\!\!\lrcorner_\xi = \mathbf{Y}(\neg \overleftarrow{\xi}) \wedge \uparrow\!\xi \wedge \neg \overleftarrow{\xi}$
$\urcorner_\xi = \neg\mathbf{Y}(\neg \overleftarrow{\xi}) \wedge \neg \overleftarrow{\xi}$	$\urcorner\!\!\urcorner_\xi = \mathbf{Y}(\overleftarrow{\xi}) \wedge \neg \uparrow\!\xi \wedge \overleftarrow{\xi}$
$\overset{\xi}{\downarrow} = \urcorner_\xi \vee \urcorner\!\!\urcorner_\xi \vee (\mathrm{orig} \wedge \neg \uparrow\!\xi)$	$\overset{\xi}{\hookrightarrow} = \urcorner_\xi \vee \lrcorner\!\!\lrcorner_\xi$
$\overset{\xi}{\uparrow} = \lrcorner_\xi \vee \lrcorner\!\!\lrcorner_\xi \vee (\mathrm{orig} \wedge \uparrow\!\xi)$	$\overset{\xi}{\Rightarrow} = \lrcorner_\xi \vee \urcorner\!\!\urcorner_\xi$
$\mathrm{orig} = \neg\mathbf{Y}(\top)$	

We do not impose any restrictions on signals other than that they be finitely variable. In particular, subformulae θ can have *singularities*, i.e., instants in which the value of θ is different than in their neighborhood. More precisely, we say that a formula θ has an "up-singularity" s^u_θ in instant t if the following holds: $t > 0$, $M, t \models \theta$ and $\exists \varepsilon > 0$ s.t. $\forall t' \neq t \in (t - \varepsilon, t + \varepsilon)$ it is $M, t' \models \neg\theta$.

Conversely, a "down-singularity" s^d_θ occurs if the formula above holds with $\neg\theta$ instead of θ. By definition singularities do not occur in 0. In CLTL formulae, we represent up- and down-singularities with propositions $\lrcorner\!\!\lrcorner_\theta$ and $\urcorner\!\!\urcorner_\theta$, respectively.

Tab. 3 summarizes the CLTL predicates used here. In a nutshell, $\overset{\xi}{\downarrow}$ (resp. $\overset{\xi}{\uparrow}$) indicates that formula ξ held (resp. did not hold) in an interval before the current one, and now it switches; the switch can be singular (in which case ξ immediately takes the same value it held before), or not, in which case ξ stays false (resp. true) for some time after the switch. Formula $\overset{\xi}{\Rightarrow}$ (resp. $\overset{\xi}{\hookrightarrow}$), instead, holds if ξ becomes true (resp. false) in the current instant, and it holds in an interval after now. Formula $\neg\mathbf{Y}(\top)$, that holds only in 0, is denoted by orig.

The translation from QTLc to CLTL-over-clocks has three parts: (i) a set $\texttt{genconstr}_\theta$ of general formulae, which are written for any $\theta \in sub(\phi)$, defining constraints that guarantee that clock resets occur at suitable points; (ii) the translation of QTL basic connectives and operators; (iii) the translation of the counting modalities. Every $\theta \in sub(\phi)$ is translated into a CLTL-over-clocks formula $m(\theta)$ that captures its semantics by describing how θ becomes true and false depending on the value of its own subformula(e).

General constraints $\texttt{genconstr}_\theta$ and the translation $m(\theta)$ in the case of basic QTL connectives and operators (i.e., cases (i) and (ii) above) have already been presented in [7]. The next section shows the translation $m(\theta)$ for $\mathbf{C}_n$, with $n \geqslant 1$.

A QTLc formula ϕ is initially satisfiable iff in 0 it has an edge such that it holds in the origin, i.e., $\mathtt{init}_\phi = \lrcorner_\phi \vee \llcorner_\phi$. Then, the corresponding CLTL-over-clocks formula ϕ_{CLTL} is:

$$\phi_{\mathrm{CLTL}} = \mathtt{init}_\phi \wedge \bigwedge_{\theta \in sub(\phi)} (\mathtt{genconstr}_\theta \wedge \mathbf{G}(m(\theta))) . \tag{1}$$

3.1 Semantics of the Counting Modalities

Some general properties of $\mathbf{C}_n$ are introduced, before defining $m(\mathbf{C}_n(\gamma))$.

Lemma 1. *Let $\theta = \mathbf{C}_n(\gamma)$. If $M, t \models \theta$ then there is $\varepsilon \in \mathbb{R}_{>0}$ such that, for all $t' \in [t, t+\varepsilon]$ it is $M, t' \models \theta$; moreover, when $t > 0$, there exists $\varepsilon \in \mathbb{R}_{>0}$ such that $\varepsilon < t$ and for all $t' \in [t-\varepsilon, t]$ it is $M, t' \models \theta$.*

Because of Lemma 1, an up-singularity $\lrcorner\!\!\llcorner_\theta$ can never occur for a formula of the form $\mathbf{C}_n(\gamma)$. In addition, if θ holds at the beginning of an interval (i.e., $\restriction_\theta$ holds), then it must hold also in the rest of the interval and, if $t > 0$, it must also hold in the interval before. Then, the following constraint holds in every instant:

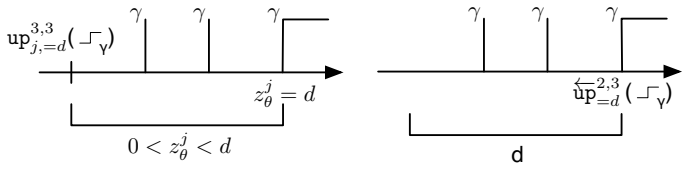
$$\restriction_\theta \Rightarrow \overleftarrow{\theta} \wedge (\mathbf{Y}(\overleftarrow{\theta}) \vee \mathrm{orig}) \tag{2}$$

For the sake of readability, some shorthands are useful. Let $z^j_{\theta,\gamma} > 0$ stand for $(z^j_\theta > 0 \wedge \bigwedge_{i \in \{0,\ldots,n_\gamma - 1\}} z^i_\gamma > 0)$, where n_γ is the number of clocks introduced for γ. We also write $z^{\hat{p}}_\gamma \sim d$ (where $\sim \in \{<, \leqslant, =, \geqslant, >\}$) to state that there are exactly p clocks of γ satisfying $\sim d$. Formula (9) of App. A defines $z^{\hat{p}}_\gamma \sim d$.

Fig. 1 recursively defines $\mathtt{up}^{n,p}_{j,=d}(B_\gamma)$, whose actual meaning is that it holds in every instant such that: 1) in the next time instant t such that clock z^j_θ has value d, also the Boolean combination B_γ of propositional letters associated with γ ($\lrcorner_\gamma$, $\overset{\gamma}{\perp}\!\!\uparrow$, etc.) holds; 2) γ has a $n-1$ true singularities (i.e., instants where $\lrcorner\!\!\llcorner_\gamma$ holds) before t; 3) in instant t there are only p clocks associated with γ whose value is $\leqslant d$ (i.e., γ has changed value p times between the instants in which z^j_θ was 0 and d). Fig. 1(a) depicts a situation in which $\mathtt{up}^{3,3}_{j,=d}(\lrcorner_\gamma)$ holds.

Formula $\mathtt{up}^{n}_{\mathrm{orig},\sim d}(B_\gamma)$ (with $\sim \in \{<, =\}$) is similar to $\mathtt{up}^{n,p}_{j,=d}(B_\gamma)$, except that the reference clock used for formula θ is fixed to z^0_θ (hence j is no more a parameter), and when B_γ holds it is $z^0_\gamma \sim d$; in addition, the number of change points of γ before B_γ holds is certainly n, so parameter p is unnecessary. For brevity, the definition of $\mathtt{up}^{n}_{\mathrm{orig},\sim d}(B_\gamma)$ is shown in App. A, Formulae (7)-(8).

Fig. 1 also introduces shortands similar to $\mathtt{up}^{n,p}_{j,=d}(B_\gamma)$, but which refer to the interval *before* B_γ holds. Formula $\overleftarrow{\mathtt{nspikes}}_n(\gamma)$ holds if the last n times when γ changed value before the current instant are of the form $\lrcorner\!\!\llcorner_\gamma$. Formula $\overleftarrow{\mathtt{up}}^{n,p}_{=d}(B_\gamma)$, then holds if B_γ holds, the last n times when γ changed value were up-singularities, and the number of clocks associated with γ that are less than d is p, hence, if $p = n+1$, all n "spikes" occurred within the last d time units. Fig. 1(b) shows an example of $\overleftarrow{\mathtt{up}}^{2,3}_{=d}(\lrcorner_\gamma)$ holding.



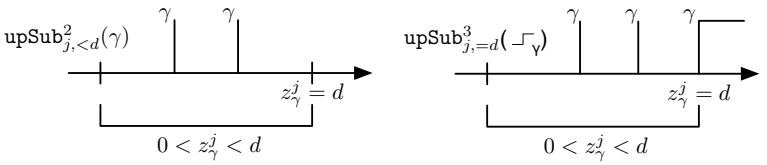
(a) $\mathtt{up}^{3,3}_{j,=d}(\lrcorner\gamma)$ **(b)** $\overleftarrow{\mathtt{up}}^{2,3}_{=d}(\lrcorner\gamma)$

Fig. 1. A first batch of abbreviations:
$\mathtt{up}^{1,p}_{j,=d}(B_\gamma) = \mathbf{X}(z^j_{\theta,\gamma} > 0\ \mathbf{U}(B_\gamma \wedge z^j_\theta = d \wedge z^{\hat{p}}_\gamma \leqslant d))$
$\mathtt{up}^{n,p}_{j,=d}(B_\gamma) = \mathbf{X}(z^j_{\theta,\gamma} > 0\ \mathbf{U}(\lrcorner\!\llcorner_\gamma \wedge 0 < z^j_\theta < d \wedge \mathtt{up}^{n-1,p}_{j,=d}(B_\gamma)))$
$\overleftarrow{\mathtt{nspikes}}_1(\gamma) = \mathbf{Y}(\overset{\gamma}{\not\downarrow}\mathbf{S}(\lrcorner\!\llcorner_\gamma \vee (\uparrow_\gamma \wedge \neg \overset{\leftarrow}{\gamma} \wedge \mathrm{orig})))$
$\overleftarrow{\mathtt{nspikes}}_n(\gamma) = \mathbf{Y}(\overset{\gamma}{\not\downarrow}\mathbf{S}(\lrcorner\!\llcorner_\gamma \wedge \overleftarrow{\mathtt{nspikes}}_{n-1}(\gamma)))$
$\overleftarrow{\mathtt{up}}^{0,p}_{=d}(B_\gamma) = B_\gamma \wedge z^{\hat{p}}_\gamma < d, \quad \overleftarrow{\mathtt{up}}^{n,p}_{=d}(B_\gamma) = B_\gamma \wedge \overleftarrow{\mathtt{nspikes}}_n(\gamma) \wedge z^{\hat{p}}_\gamma < d$

Using the abbreviations of Fig. 1, we capture through CLTL-over-clocks formulae the conditions that make $\theta = \mathbf{C}_n(\gamma)$ have a raising edge (i.e., that corresponds to $\lrcorner_\theta$). Formula (3) describes that, when θ becomes true with a raising edge $\lrcorner_\theta$ in an instant $t > 0$, then it does so in a left-open manner (i.e., θ does not hold in t), a clock z^j_θ is reset, and (i) either γ has $n-1$ up-singularities before z^j_θ hits 1 and γ becomes true again also with an up-singularity when $z^j_\theta = 1$, or (ii) γ has a raising edge when $z^j_\theta = 1$ (hence it is true infinitely many times in a right neighborhood of that instant), and it also has *up to* $n-1$ (possibly 0) up-singularities before $z^j_\theta = 1$. If instead θ becomes true in $t = 0$ in a left-closed manner (i.e., θ holds in t; the left-open case is similar to the one above), before clock $z^0_\theta = 1$ either γ has a raising edge (so it is true infinitely many times before $z^0_\theta = 1$) preceded by up to $n-1$ (possibly 0) up-singularities, or there are n up-singularities followed, before $z^0_\theta = 1$, by an instant in which γ becomes true.

$$\lrcorner_\theta \Leftrightarrow \left(\begin{array}{l} \mathrm{orig} \wedge \left(\begin{array}{l} \uparrow_\theta \wedge \left(\lrcorner_\gamma \ \vee \mathtt{up}^n_{\mathrm{orig},<1}(\lrcorner\!\llcorner_\gamma) \vee \bigvee\limits_{k\in\{1,\ldots,n\}} \mathtt{up}^k_{\mathrm{orig},<1}(\lrcorner_\gamma) \right) \vee \\ \quad \neg \uparrow_\theta \wedge \left(\mathtt{up}^n_{\mathrm{orig},=1}(\lrcorner\!\llcorner_\gamma) \vee \bigvee\limits_{k\in\{1,\ldots,n\}} \mathtt{up}^k_{\mathrm{orig},=1}(\lrcorner_\gamma) \right) \end{array} \right) \vee \\ \neg\mathrm{orig} \wedge \bigvee\limits_{j=0}^{n_\theta-1} \left(\neg \uparrow_\theta \wedge z^j_\theta = 0 \wedge \left((\mathtt{up}^{n,n}_{j,=1}(\lrcorner\!\llcorner_\gamma) \wedge \not\lrcorner_\gamma) \vee \bigvee\limits_{k\in\{1,\ldots,n\}} \mathtt{up}^{k,k}_{j,=1}(\lrcorner_\gamma) \right) \right) \end{array} \right) \tag{3}$$

Formula (4) states that if t is an instant in which either (i) in the preceding interval of length 1 γ has $n-1$ up-singularities and γ also becomes true in t with an up-singularity (i.e., $\overleftarrow{\mathtt{up}}^{n-1,n}_{<1}(\lrcorner\!\llcorner_\gamma)$ holds), or (ii) γ has a raising edge and in the preceding interval of length 1 γ has *at most* $n-1$ up-singularities (i.e., $\overleftarrow{\mathtt{up}}^{k-1,k}_{=1}(\lrcorner_\gamma)$ holds for some $k \leqslant n-1$), then in t one of the clocks associated with θ must be 1 (in fact, $\mathbf{C}_n(\gamma)$ started to hold exactly 1 time unit before t, see also Fig. 1(b)), and all others are greater than 1.



(a) $\mathtt{upSub}^2_{j,<d}(\gamma)$ **(b)** $\mathtt{upSub}^3_{j,=d}(\lrcorner\!\lceil_\gamma)$

Fig. 2. A second batch of abbreviations:
$\mathtt{upSub}^0_{j,\lesssim d}(\gamma) = \neg\mathbf{X}(\not\uparrow^{\gamma}\mathbf{U}(\uparrow^{\gamma} \wedge 0 < z^j_\gamma \leqslant d))$
$\mathtt{upSub}^n_{j,\lesssim d}(\gamma) = \mathbf{X}(\not\uparrow^{\gamma}\mathbf{U}(\lrcorner\!\llcorner_\gamma \wedge 0 < z^j_\gamma \lesssim d \wedge \mathtt{upSub}^{n-1}_{j,\lesssim d}(\gamma)))$
$\mathtt{upSub}^1_{j,=d}(B_\gamma) = \mathbf{X}(\not\uparrow^{\gamma}\mathbf{U}(B_\gamma \wedge z^j_\gamma = d))$
$\mathtt{upSub}^n_{j,=d}(B_\gamma) = \mathbf{X}(\not\uparrow^{\gamma}\mathbf{U}(\lrcorner\!\llcorner_\gamma \wedge 0 < z^j_\gamma < d \wedge \mathtt{upSub}^{n-1}_{j,d}(B_\gamma)))$

$$\left(\overleftarrow{\mathtt{up}}^{n-1,n}_{=1}(\lrcorner\!\llcorner_\gamma) \vee \bigvee_{k\in\{1,\ldots,n\}} \overleftarrow{\mathtt{up}}^{k-1,k}_{=1}(\lrcorner\!\lceil_\gamma)\right) \Rightarrow \bigvee_{i\in\{0,\ldots,n_\theta-1\}} z^i_\theta = 1 \tag{4}$$

To describe the conditions under which θ becomes false, either with a falling edge (i.e., $\urcorner\!\llcorner_\theta$ holds), or with a singularity (i.e., $\urcorner\!\lceil_\theta$ holds) we introduce a pair of further shorthands, shown in Fig. 2. Formula $\mathtt{upSub}^0_{j,\lesssim d}(\gamma)$ (where $\lesssim \in \{<, \leqslant\}$) holds if, from the current instant (excluded) until the instant when clock z^j_γ hits value d (included), γ never becomes true. Then, $\mathtt{upSub}^n_{j,\lesssim d}(\gamma)$ holds if, in the interval that starts in the current instant and ends when clock $z^j_\gamma = d$ (both endpoints excluded if $\lesssim = <$), γ has exactly n up-singularities. Fig. 2(a) exemplifies when $\mathtt{upSub}^2_{j,<d}(\gamma)$ holds. Note that if there are at least $n+1$ clocks associated with γ, if $\mathtt{upSub}^n_{j,<d}(\gamma)$ holds, then z^j_γ is not reset before it becomes d. Similarly, $\mathtt{upSub}^n_{j,=d}(B_\gamma)$ holds if, in the interval that starts in the current instant and ends when $z^j_\gamma = d$ (endpoints excluded), γ has $n-1$ up-singularities, and B_γ holds when $z^j_\gamma = d$. Fig. 2(b) depicts a case where $\mathtt{upSub}^3_{j,=d}(\gamma)$ holds.

When $\theta = \mathbf{C}_n(\gamma)$ becomes false with either a falling edge ($\urcorner\!\llcorner_\gamma$) or in a singular manner ($\urcorner\!\lceil_\gamma$), γ becomes false, and a clock z^i_γ is reset. Let us first consider the former case (Formula (5)). There are two cases: γ becomes false with a falling edge $\urcorner\!\llcorner_\gamma$, or it has an up-singularity $\lrcorner\!\llcorner_\gamma$. In the former case, γ can have *up to* $n-1$ up-singularities before $z^i_\gamma = 1$ (it can have less than $n-1$, since γ holds infinitely many times before it has a falling edge). In the latter case, γ must have *exactly* $n-1$ up-singularities before $z^i_\gamma = 1$, or θ does not have a falling edge.

$$\urcorner\!\llcorner_\theta \Leftrightarrow \left(\begin{array}{c} \urcorner\!\llcorner_\gamma \wedge \displaystyle\bigwedge_{i\in\{0,\ldots,n_\gamma-1\}} \left(z^i_\gamma = 0 \Rightarrow \bigvee_{k\in\{0,\ldots,n-1\}} \mathtt{upSub}^k_{i,\leqslant 1}(\gamma)\right) \vee \\ \lrcorner\!\llcorner_\gamma \wedge \displaystyle\bigwedge_{i\in\{0,\ldots,n_\gamma-1\}} \left(z^i_\gamma = 0 \Rightarrow \mathtt{upSub}^{n-1}_{i,<1}(\gamma)\right) \end{array}\right) \tag{5}$$

Finally, as captured by Formula (6), for θ to have a down-singularity $\urcorner\llcorner_\theta$, not only γ must become false with $\urcorner\llcorner_\theta$, but it must also become true again exactly when the clock z^i_γ, which is reset with $\urcorner\llcorner_\theta$, takes value 1.

$$\urcorner\llcorner_\theta \Leftrightarrow \left(\begin{array}{c} \neg\text{orig} \wedge \llcorner_\gamma \wedge \bigwedge_{i=0}^{n_\gamma - 1} \left(z^i_\gamma = 0 \Rightarrow \texttt{upSub}^n_{i,=1}(\lrcorner\uparrow_\gamma) \vee \bigvee_{k=1}^{n-1} \texttt{upSub}^k_{i,=1}(\lrcorner\urcorner_\gamma) \right) \vee \\ \lrcorner\llcorner_\gamma \wedge \bigwedge_{i=0}^{n_\gamma - 1} \left(z^i_\gamma = 0 \Rightarrow \texttt{upSub}^n_{i,=1}(\lrcorner\uparrow_\gamma) \right) \end{array} \right) \quad (6)$$

Finally, for $\theta = \mathbf{C}_n(\gamma)$, $m(\theta)$ is (2) $\wedge$ (3) $\wedge$ (4) $\wedge$ (5) $\wedge$ (6).

4 Correctness and Complexity of the Reduction

To complete the results, we need to show that a QTLc formula ϕ is satisfiable iff there exists a pair (π, σ) that satisfies ϕ_{CLTL} defined by Formula (1). The proof is an extension of the one presented in [7], so in this paper we summarize its salient points and we outline the QTLc-specific parts.

First of all, we define a correspondence between QTLc signals and CLTL-over-clocks interpretations. Let us consider a finitely variable signal M that is an interpretation for a QTLc formula θ; we call $r_\theta(M)$ the set of CLTL-over-clocks interpretations (π, σ) built according to the rules presented below.

Since M is finitely variable, the set of "events" in M for formula θ is denumerable. Also, it can be shown that, in any closed interval of length 1, a formula of the form $\mathbf{C}_n(\phi)$ cannot have more than $n + 1$ events.

Let $T = \{t_k\}_{k\in\mathbb{N}} \subset \mathbb{R}_+$ be a denumerable set of time instants such that $t_k < t_j \Leftrightarrow k < j$, for all $t' \in \mathbb{R}_+$ there is $t_k \in T$ such that $t_k > t'$, and if t is an instant when at least one event for θ occurs in M, then $t \in T$. In the following we say that a clock v is reset at position k when $\sigma(v, k) = 0$.

If an event among $e^u_\theta, e^d_\theta, s^u_\theta$ or s^d_θ occurs at $t_k \in T$, the event marker captured by the corresponding formula $\lrcorner\urcorner_\theta, \llcorner_\theta, \lrcorner\llcorner_\theta, \urcorner\llcorner_\theta$ holds in $\pi(k)$, i.e., if $M, t_k \models e^u_\theta$, then $\lrcorner\urcorner_\theta$ holds in $\pi(k)$ (hence $\overleftarrow{\theta} \notin \pi(k-1)$, $\overleftarrow{\theta} \in \pi(k)$), etc. In addition, if $M, t_k \models e^u_\theta$ and $M, t_k \models \theta$ (resp. $M, t_k \not\models \theta$), then $\uparrow_\theta \in \pi(k)$ (resp. $\uparrow_\theta \notin \pi(k)$); similarly for the falling edge. By the definition of events of Sect. 3, θ has an event in $t = 0$, so $t_0 = 0$. If no events for θ occur in $t_k \in T$, then none of $\{\lrcorner\urcorner_\theta, \llcorner_\theta, \lrcorner\llcorner_\theta, \urcorner\llcorner_\theta\}$ holds in $\pi(k)$ (so $\overleftarrow{\theta} \in \pi(k-1)$ iff $\uparrow_\theta, \overleftarrow{\theta} \in \pi(k)$). For each $t_k \in T$ where an event for θ occurs, a z^i_θ is reset at k. z^0_θ is reset in 0; after 0, clocks are reset in a circular manner, modulo n_θ (i.e., z^i_θ is reset after $z^{(i-1) \bmod n_\theta}_\theta$, but before $z^{(i+1) \bmod n_\theta}_\theta$, and so on). For each z^i_θ it is $\sigma(z^i_\theta, k+1) = \sigma(z^i_\theta, k) + t_{k+1} - t_k$ unless z^i_θ is reset.

For a given signal M there is more than one possible compatible sequence $T = \{t_k\}_{k\in\mathbb{N}}$, each one corresponding to a different CLTL interpretation. In addition, one can show that if two signals $M_1 \neq M_2$ differ for θ in at least one instant $t \in \mathbb{R}_+$, then $r_\theta(M_1) \cap r_\theta(M_2) = \varnothing$. Let us now consider a set of formulae $\mathcal{F}$; we indicate by $r_\mathcal{F}(M)$ the set of CLTL interpretations built as before, but

by considering all the events related to the formulae in $\mathcal{F}$. In particular, we will be interested, given a formula ϕ, in $r_{sub(\phi)}(M)$. Not all CLTL interpretations (π, σ) represent QTLc signals, so there are pairs (π, σ) such that $r_\theta^{-1}((\pi, \sigma))$ is undefined. However, we have the following result (from [7], *mutatis mutandis*).

Lemma 2. *Let θ be a QTLc formula and M be a signal. For all interpretations (π, σ) s.t. $(\pi, \sigma) \in r_\theta(M)$ it is $(\pi, \sigma), 0 \models \mathtt{genconstr}_\theta$. Dually, let (π, σ) be a CLTL-over-clocks interpretation where time diverges (i.e., where $\sum_{i \in \mathbb{N}} \delta(i) = \infty$); if $(\pi, \sigma), 0 \models \mathtt{genconstr}_\theta$, there is exactly one signal M s.t. $(\pi, \sigma) \in r_\theta(M)$.*

From Lemma 2 we can prove the following result.

Lemma 3. *Let M be a signal, and ϕ a QTLc formula. For any $(\pi, \sigma) \in r_{sub(\phi)}(M)$ it is $(\pi, \sigma), 0 \models \bigwedge_{\theta \in sub(\phi)} \mathtt{genconstr}_\theta$ and for all $k \in \mathbb{N}, \theta \in sub(\phi)$ it is $(\pi, \sigma), k \models m(\theta)$. Conversely, if $(\pi, \sigma), 0 \models \bigwedge_{\theta \in sub(\phi)} \mathtt{genconstr}_\theta \wedge \mathbf{G}(m(\theta))$ and $M = r_{sub(\phi)}^{-1}((\pi, \sigma))$, then $\lrcorner_\phi \in \pi(k)$ iff $M, t_k \models e_\phi^u$ (similarly for the other events).*

Finally, the following theorem derives from Lemma 3, by observing that signal M is model for ϕ iff $M, 0 \models \phi$, which means that in 0 either ϕ has a left-closed raising edge or it has a left-open falling edge.

Theorem 1. *A QTLc formula ϕ is satisfiable if, and only if, the CLTL-over-clocks formula ϕ_{CLTL} (defined by Formula (1)) is satisfiable.*

Consider a QTLc formula ϕ. The translation of Sect. 3 introduces, for each $\theta \in sub(\phi)$, 2 atomic propositions $\restriction_\theta, \overleftarrow{\theta}$ and n_θ variables $z_\theta^0, z_\theta^{n_\theta - 1}$ (where n_θ is bounded by the maximum n' such that $\mathbf{C}_{n'}$ appears in ϕ). All CLTL-over-clocks formulae $m(\theta)$ have fixed size, except $m(\mathbf{C}_n(\gamma))$, whose size is $O(n^3)$ (it can be made quadratic with optimizations that are not shown here for simplicity). Hence, the size of Formula (1) is polynomial in the size of ϕ, if one considers a unary encoding of the indexes of the counting modalities. [8] shows that satisfiability for a CLTL-over-clocks formula ϕ_{CLTL} is PSPACE in the number of subformulae of ϕ_{CLTL} and the maximum constant occurring in it (i.e., 1 in the case of QTLc). Then our translation preserves the PSPACE complexity (considering a unary encoding of the indexes) of the satisfiability of QTLc [17].

5 Conclusions and Preliminary Results

This paper presents a satisfiability-preserving translation from QTLc to the logic CLTL-over-clocks. The advantage of this approach is that the resulting translation can be encoded into the input language of SMT solvers, thus providing an effective decision procedure for QTLc (the first one available in the literature, to the best of our knowledge). An open-source prototype [1] handling QTL formulae and implementing the encoding of [7] and the one presented in this work has been used to carry out some preliminary experiments on verifying the satisfiability of representative, albeit not large, formulae. Although the tool has not been optimized, it was able to solve some interesting

examples in a matter of minutes. For example, it was able to show in 24 seconds[1] that specification $S = \mathbf{G}(\mathbf{F}(q) \wedge (q \rightarrow \mathbf{C}_2(q)))$ is satisfiable, in 50 seconds that property $\mathbf{G}\big(q \rightarrow \mathbf{F}_{(0,0.5)}(q)\big)$ does not hold for S, and in 57 minutes that $\mathbf{G}\big(\mathbf{F}\big(q \wedge \mathbf{F}_{(0,0.5)}(q)\big)\big)$ does instead hold.

References

1. qtlsolver, `http://qtlsolver.googlecode.com`
2. Zot: a bounded satisfiability checker, `http://zot.googlecode.com`
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. Journal of the ACM 43(1), 116–146 (1996)
5. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
6. Bersani, M.M., Frigeri, A., Rossi, M., San Pietro, P.: Completeness of the bounded satisfiability problem for constraint LTL. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 58–71. Springer, Heidelberg (2011)
7. Bersani, M.M., Rossi, M., San Pietro, P.: On the satisfiability of metric temporal logics over the reals. In: Proceedings of AVOCS (2013)
8. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. In: Proceedings of TIME (2013)
9. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Inf. Comput. 205(3), 380–415 (2007)
10. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling Time in Computing. EATCS Monographs in Theoretical Computer Science. Springer (2012)
11. Hirshfeld, Y., Rabinovich, A.: Timer formulas and decidable metric temporal logic. Information and Computation 198(2), 148–178 (2005)
12. Hirshfeld, Y., Rabinovich, A.: Expressiveness of metric modalities for continuous time. Logical Methods in Computer Science 3(1:3), 1–11 (2007)
13. Hirshfeld, Y., Rabinovich, A.: Continuous time temporal logic with counting. Information and Computation 214, 1–9 (2012)
14. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
15. Microsoft Research. Z3: An efficient SMT solver
16. Morzenti, A., San Pietro, P.: Object-oriented logical specification of time-critical systems. ACM TOSEM 3(1), 56–98 (1994)
17. Rabinovich, A.: Complexity of metric temporal logics with counting and the Pnueli modalities. Theoretical Computer Science 411, 2331–2342 (2010)
18. Raskin, J.-F., Schobbens, P.-Y.: State clock logic: a decidable real-time logic. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 33–47. Springer, Heidelberg (1997)
19. Raskin, J.-F., Schobbens, P.-Y., Henzinger, T.A.: Axioms for real-time logics. Theoretical Computer Science 274(1-2), 151–182 (2002)
20. The UPPAAL model checker, `http://www.uppaal.org`

[1] All tests have been carried out on a desktop computer with a 2.8GHz AMD PhenomTMII processor and 8MB RAM; the solver was Microsoft Z3 3.2. The experiments were carried out using a bound $k = 25$.

A Completing the Translation of Counting Modalities

Shorthands $\mathtt{up}^1_{\mathrm{orig},\sim d}(B_\gamma)$ and $\mathtt{up}^n_{\mathrm{orig},\sim d}(B_\gamma)$, where $\sim \in \{<, =\}$.

$$\mathtt{up}^1_{\mathrm{orig},\sim d}(B_\gamma) = \mathbf{X}\big(z^0_{\theta,\gamma} > 0 \ \mathbf{U}\ \big(B_\gamma \wedge 0 < z^0_\theta \sim d\big)\big) \tag{7}$$

$$\mathtt{up}^n_{\mathrm{orig},\sim d}(B_\gamma) = \mathbf{X}\left(z^0_{\theta,\gamma} > 0 \ \mathbf{U}\Big(\text{⅃L}_\gamma \wedge 0 < z^0_\theta < d \wedge \mathtt{up}^{n-1}_{\mathrm{orig},\sim d}(B_\gamma)\Big)\right) \tag{8}$$

Shorthand for $z^{\hat{p}}_\gamma \sim d$ (where $\sim \in \{<, \leqslant, =, \geqslant, >\}$, n_γ is the number of clocks associated with γ, $p < n_\gamma$, and $+_{n_\gamma}$ is the addition modulo n_γ):

$$z^{\hat{p}}_\gamma \sim d = \bigvee_{i \in \{0,\ldots,n_\gamma - 1\}} \left(\bigwedge_{j \in \{i +_{n_\gamma} 1, \ldots, i +_{n_\gamma} p\}} (z^i_\gamma \sim d) \wedge \bigwedge_{j \in \{i +_{n_\gamma} (p+1), \ldots, i\}} (z^j_\gamma \nsim d) \right) \tag{9}$$

MaRDiGraS: Simplified Building of Reachability Graphs on Large Clusters

Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga

Università degli Studi di Milano, Milan, Italy
{bellettini,camilli,capra,monga}@di.unimi.it

Abstract. Dealing with complex systems often needs the building of huge reachability graphs, thus revealing all the challenges associated with big data access and management. It also requires high performance data processing tools that would allow scientists to extract the knowledge from the unprecedented amount of data coming from these analyzed systems. In this paper we present MaRDiGraS, a generic framework aimed at simplifying the construction of very large state transition systems on large clusters and cloud computing platforms. Through a simple programming interface, it can be easily customized to different formalisms, for example Petri Nets, by either adapting legacy tools or implementing brand new distributed reachability graph builders. The outcome of several tests performed on benchmark specifications are presented.

Keywords: Reachability Graph, Big Data, Formal Methods, Distributed Computing, Cloud Computing, MapReduce.

1 Introduction

Formal verification of dynamic, concurrent and real-time systems has been the focus of several decades of software engineering research. One of the most challenging task in this context is the development of tools able to cope with the complexity of the models needed in the analysis of real word examples. "Big Data" is a recent buzzword used to represent the collection of tools, methods, and techniques that can be employed in the manipulation of very large datasets: the analysis of big models certainly falls in this area, although formal verification has not yet been considered by big data scientists. Indeed the challenges to be tackled in formal verification include those associated with big data access and management. In fact formal verification requires several different skills: On the one hand, one needs an adequate background on formal methods in order to understand specific formalisms and proper abstraction techniques for modeling and interpreting the analysis results; On the other hand, one should strive to deploy this techniques into software tools able to analyze large amount of data very reliably and efficiently similarly to "big data" projects. Recent approaches have shown the convenience of employing distributed memory and computation to manage large amount of reachable states, but unfortunately exploiting this

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 83–95, 2013.

requires further skills in developing complex applications with knotty communication and synchronization issues. In particular, adapting an application for exploiting the scalability provided by cloud computing facilities as the Amazon Cloud Computing platform [2] might be a daunting task without the proper knowledge of the subtleties of data-intensive and distributed analyses.

In this paper, we try to reduce the gap between these different areas of expertise by providing the MaRDiGraS[1] generic library. Our framework is built on top of Hadoop MapReduce [14,23] and can be easily specialized to deal with the construction of very big state spaces of different kinds of formalisms (e.g., different kind of Petri Nets), thus it is suitable for simplifying the task of dealing with a large amount of reachable states by exploiting large clusters of machines. The MapReduce programming model, which has become the de facto standard for large scale data-intensive applications, has provided researchers with a powerful tool for tackling big-data problems in the areas of machine learning, text processing, bioinformatics, and large-scale graph processing [22]. The building of the huge graphs emerging from the analysis of discrete-event systems is a problem that could benefit from a MapReduce based approach, but this has currently to be done by *ad-hoc* solutions [3], since the topic is still poorly explored as far as we know. We firmly believe that the delivery of MaRDiGraS framework would be helpful for both communities: First, we supply a flexible tool for constructing high-performance distributed applications without the need for the developers (supposed to be more familiar with formalisms than with MapReduce or distributed systems) to care about all non-functional aspects; Second, we aim at introducing the state explosion problem [24], as an instance of a big-data problem. Exposing this issue to scientists with different backgrounds could stimulate the development new interesting and more efficient solutions.

The rest of the paper is organized as follows. Section 2 gives a brief background and we try to extract the intrinsic characteristics associated to state spaces coming from different formalisms; Section 3 describes our generic approach and gives some technical details about the MapReduce model; Section 4 describes some use cases; Section 5 describes some experimental results; Section 6 gives an overview on related works; finally Section 7 reports our conclusion.

2 Background

The behaviour of a discrete-event dynamic system is formally given in terms of a labeled state transition system $(S, \Lambda, \rightarrow)$, where S is the set of system's states, Λ is a set of labels, and $\rightarrow \subseteq S \times \Lambda \times S$: $(s, \lambda, s') \in \rightarrow$ if and only if s' is reachable from s through the occurrence of λ (s' is said to be a *successor* of s and it is written as $s \xrightarrow{\lambda} s'$).

A way to face state explosion (or the fact that in general S may be infinite, or even uncountable, like in some time PN extensions) consists of building a (hopefully finite) abstraction of the original (concrete) state transition system.

[1] MapReduce-based DIstributed building of GRAphS.

Different techniques are employed for that, depending on the formalisms. A non exhaustive survey regarding high-level PNs may be found in [18]. In general, $(A, L, \Rightarrow)$ is an abstraction of $(S, \Lambda, \rightarrow)$ if each $a \in A$ represents a set of concrete states, A is a coverage of S, i.e., $\bigcup_{a \in A} a \supseteq S$, and, letting f be a morphism $\Lambda \rightarrow L$, relation $\Rightarrow \subseteq A \times L \times A$ satisfies condition EE (exists-exists)[6,3]:

EE-(1) if $a \stackrel{l}{\Rightarrow} a'$, then $\exists s \in a, s' \in a', \lambda \in f^{-1}(l) : s \stackrel{\lambda}{\rightarrow} s'$

EE-(2) if $s \stackrel{\lambda}{\rightarrow} s'$, then $\forall a \in A$ s.t. $s \in a$, $\exists a' \in A$ s.t. $s' \in a' : a \stackrel{f(\lambda)}{\Rightarrow} a'$

The first part of condition EE avoids two abstract states from being connected if no corresponding concrete states are. The second part ensures that each concrete path corresponds to some abstract path.

Depending on the particular abstraction technique, and the properties one is interested to check [3], it is possible/necessary to further refine condition EE, either locally or globally, as informally shown in Fig. 1. For example, condition EA (exists-for all) imposes that any abstract edge between states a, a' must correspond to a set of concrete ones, between some $s \in a$ and *each* $s' \in a'$. Any (abstract) state-transition system can thus be described by annotating edges with additional information, indicating which kind of connectivity among EE, EA, AE (for all-exists), and AA (for all-for all) is locally met. According to this convention, a concrete state space can be represented using only edges labeled AA. Edges, as well as nodes, usually carry other annotations that are specific to the particular formalism one is using.

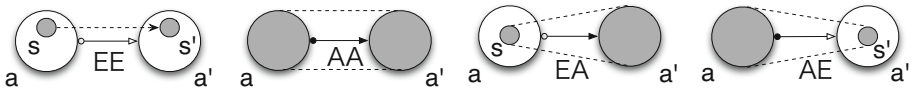


Fig. 1. Edges types of an abstract state space

Independently of the formalism used in the modelling phase (PNs, in their several extensions, process-algebras, etc.), we can reformulate most of the algorithms for building (abstract) state-transition systems in terms of an elementary iterative schema, whose essential points are outlined below:

(a) For each unexplored state a, we calculate the set of successors $succ(a)$, identifying which connectivity conditions are met. Then we mark a as explored.
(b) For each $a' \in succ(a)$, we try to identify equivalence/inclusion relationships between a' and any state a''. If a' has been shown equivalent to/included in a'', it is discarded and all existing edges towards a' are redirected to a'' (in the inclusion case the edges of kind *A are relabelled as *E).

Typically, such schema cycles until there are no unexplored states, using states coming from the previous iteration as input to the next one. The operations

which depend on the adopted formalisms are the calculation of the successors of a state, the evaluation of relationships between states (often the more computationally expensive operation), and the identification of connectivity conditions. The basic idea of the framework is gathering them into a very simple programming interface, clearly separated from the core of the distributed algorithm, based on a generic MapReduce. The complexity of evaluating equivalence/inclusion relationships between abstract states can be alleviated by identifying any *syntactical* feature which defines a necessary condition for states' overlapping, e.g., the merely numerical distribution of tokens in a Coloured Petri net marking. Examples of algorithms that could be rephrased according to the schema above are presented in [4] for TB nets, in [5] for time Petri nets, and in [9] for Well-Formed Coloured Petri nets. The construction of the reachability graph for low-level Petri nets trivially falls in this category.

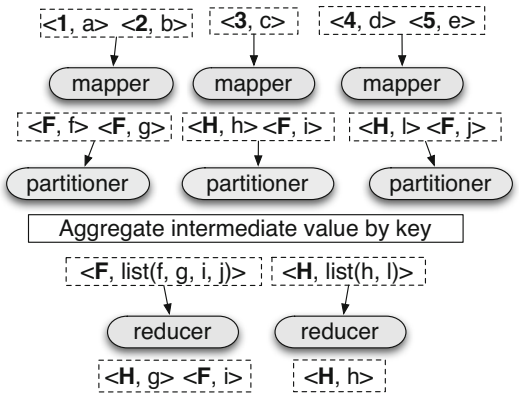


Fig. 2. The MapReduce model: The keys are in **bold**

3 Programming Model

The MaRDiGraS framework is built on top of Hadoop MapReduce. In the following sections we briefly recall the MapReduce model and we present the key points of the framework.

3.1 MapReduce

MapReduce relies on the observation that many information processing activities have the same basic design: a same operation is applied over a large number of records (e.g., database records, vertices of a graph) to generate partial results, which are then aggregated to compute the final output. In the MapReduce model

users create their own application through a "map" function (specifying per-record computations) and a "reduce" function (specifying the aggregation of map computations): both operate in parallel on key-value pairs which represent the input of the problem. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs. The reducer is then applied to all values associated with the same intermediate key to generate an arbitrary number of final key-value pairs as output. This two-step processing structure is presented in Figure 2.

The execution framework transparently handles all non-functional aspects of execution on big clusters. It is responsible, among other things, for scheduling (moving code to data), handling faults, and the large distributed sorting and shuffling needed between the map and reduce phases since intermediate key-value pairs must be grouped by key. The "partitioner" is responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. The default partitioner computes a hash function on the value of the key modulo the number of reducers. In order to rewrite a state-space (possibly abstract) builder tool according to the MapReduce model, one has to assign to mappers the task (a) described in Section 2, and to the reducers the task (b).

3.2 MaRDiGraS

MaRDiGraS follows the *Hybrid Iterative MapReduce* model sketched in Figure 3. Computation starts by considering the initial state of the system under analysis and goes on with a sequential state-space building phase until the set N of states not yet explored becomes *large enough*: in other words there is a configurable *threshold* (in terms of number of states) below which a (all-in-RAM) sequential approach is considered more efficient than the distributed one. Once we go above the threshold, an *iterative* MapReduce algorithm runs over a cluster of machines. We carried out several experiments to determine a good setting of the threshold. Experimental evidences suggest that this parameter is strictly related to the number of new nodes created at each iteration: this makes us confident in a possible run-time setting of the threshold.

The *map* step (computation of new states) and the *reduce* step (identification of equivalence/inclusion relationships), iterate until $|N|$ remains above the threshold. Between them, the default partitioner splits the intermediate key set, ensuring that all possibly related states belong to the same partition. This is done by using as intermediate keys a function g such that if two states s_1, s_2 are related then $g(s_1) = g(s_2)$. This way the partitioner delivers all possibly related states to the same reducer. Whenever $|N|$ goes back below the threshold the output of all reducers is merged in order to proceed with the sequential algorithm. This operation might cause a memory overflow in some cases, due to the huge size of the state-space computed until that point (potentially many GB or TB). This is why the user can choose not to switch to the sequential algorithm anymore, after the first exceeding of the threshold. Once the set of unexplored states becomes empty, the entire-state space is supplied as output either in a single file, or distributed over different files.

The code base of MaRDiGraS is made up by two main packages which split logically the framework into two different parts: the `data` package and the `core` package. The `data` package contains all entities concerning the data of our framework: the state-space and the model. These entities must be extended in order to be customized to a specific formalism: for example, in the case of time PN extensions, one may want to attach specific meta-data to nodes and edges holding timing properties. The `core` package contains all the algorithms of the framework. They implement, together with the user defined building blocks, the *Hybrid Iterative MapReduce* model.

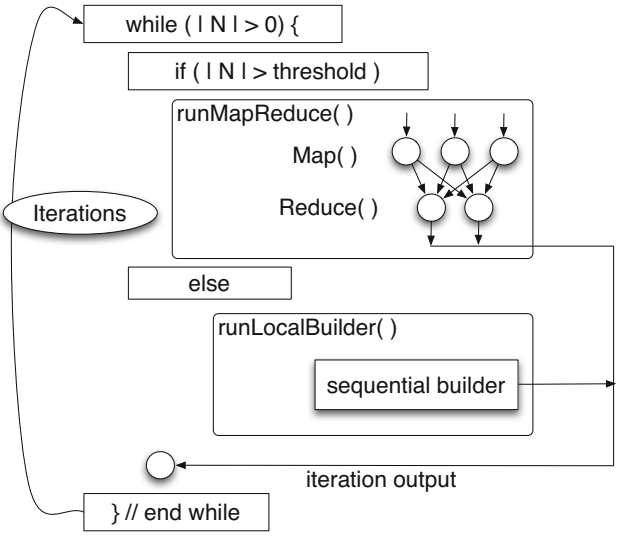


Fig. 3. Hybrid Iterative MapReduce model

The `data` package contains the *Model*, the *State* and the *Edge* entities.

The *Model* is an interface which should be implemented by the class representing the model under analysis. It contains two methods which must be implemented in order to correctly interpret the user input model and to build the root state of the reachability graph.

The *State* is an abstract class which should be extended to instatiate the state concept in a particular formalism. The user can also add properties to this entity, other than the standard ones supplied by the framework: an identifier and a list of incoming edges. The framework largely uses and manipulates these objects during the computation through a few user-implemented methods. The `createSuccessors` method must return a list of new `State` objects representing the states directly reachable from the subject of the call. MaRDiGraS calls this

method during the "map" phase in order to compute all new reachable states from the unexplored ones. The `identifyRelationship` method must evaluate the actual relationship between (abstract) states sharing some specific features. The possible returned values are: `NONE, EQUALS, INCLUDED, INCLUDES`. It is invoked during the "reduce" phase. Depending on the returned value, the framework discards from the state space those states evaluated included or equal to other ones, modifying all incoming edges of the remaining state, as explained in Section 2. The `getFeatures` method must provide some state features so that the equality of these features must be a necessary condition for equivalence/inclusion relationships between states. MaRDiGraS uses this method to compute the key of each intermediate key-value pair. This way the default partitioner assigns all possibly related states to the same reducer. Whenever the equality of computed state features is also a sufficient condition for state equivalence, one should more conveniently use an optimized version of the *reducer* called `SimpleReducer`.

The *Edge* is an abstract class which should be extended to represent the edge concept. The extending class should implement all the properties that the user want to attach to edges, MaRDiGraS invokes the `addLabel` user defined method to initialize an edge between two states. During this stage, we can change the default edge type (`EE` type), and we can attach additional information to the edge, in order to supply the label concept.

The main components of the `core` package work together with the user supplied building blocks to implement the *Hybrid Iterative MapReduce* schema described above. The *reduce* phase can be performed in two different ways: the standard reducer works by evaluating the user-supplied `identifyRelationship` method for each pair of states potentially related. This is a very expensive task and it must be done whenever the actual relationship between two states is unknown, because we supplied a necessary condition, but not sufficient for evaluating the relationship between states. But, if the implementation of the `getFeatures` method gives also a sufficient condition for evaluating state equivalence, the framework already knows that all states sharing a key are equal. In that case `SimpleReducer` should be used, which performs the reduce phase much more efficiently: it simply returns one among the input states, redirecting all incoming edges of the others into that state.

It is worth noting that no particular knowledge on MapReduce and the Hadoop framework is required in order to use MaRDiGraS. The user only cares about the functional aspects of the application, leaving to the framework the management of all other aspects of execution on big clusters. A tool based on MaRDiGraS will produce a set of binary files containing the representation of the state transition system computed from the user's model (given as input using any reasonable format chosen by the user). The output could be used in turn to extract the knowledge from the analyzed systems: for example to model-check it or to verify particular structural properties on the graph.

The MaRDiGraS framework can be found at `http://goo.gl/do9aw` together with the API description, installation instructions and a working application.

4 Use Cases

Time Basic PNs. Time-Basic (TB) nets [17] belong to the category of PNs in which time dependencies are expressed as numerical intervals associated to each transition, denoting the possible firing instants since enabling. Tokens atomically produced by a firing are thereby associated to time-stamps in a given domain (e.g., $\mathbb{R}+$). Transition interval bounds are functions of time-stamps in transition presets and each transitions may be assigned either a weak or a strong semantics. In order to exploit MaRDiGraS to compute the associated abstract state transition system (called TRG) [4], we have extended `State`, `Edge`, and `Model` classes. In particular, TRG states are defined as pairs $\langle M, C \rangle$, where M is an association between *places* and a multi-set of symbols denoting time-stamps, C is a predicate formed by linear inequalities involving such symbols. Labels on edges include the firing transition and the minimum-maximum firing times. Once created all data structure, the application logic has been supplied to the framework by implementing the abstract methods described above. The `createSuccessors` individuates all transition instances enabled in the current TRG state, and for each of those computes a new reachable state; `identifyRelationship` figures out the actual relationship between given states, according to the following sufficient condition for $a \subseteq a'$: $M = M'$ and $C \wedge \neg C' \equiv false$. Depending on the actual computed relationship, the framework modifies the incoming edges' type (either EE, AA, AE or EA). The `getFeatures` method just returns the topological part (M) of a TRG state.

P/T Nets. In order to prove the effectiveness of using MaRDiGraSto improve legacy tools, we adapted an existing P/T nets tool: PIPE [15], an open source Java tool (∼82400 lines of code). It supports the design and analysis of P/T nets with priorities, and their stochastic extension (GSPN). In particular a module is in charge of computing the reachability graph (without any particular smart technique such as decision diagrams, use of structural information, partial order techniques, etc.). For this reason, in such a situation, the memory consumption and the execution time become heavy even during the analysis of relatively small models. In order to exploit the MaRDiGraS framework to overcome these troubles, we first simply identified all those parts of PIPE representing our needed building blocks described in section 3.2. Then we encapsulated these blocks with proper *adapter* classes. To adapt the sequential algorithm of PIPE into a distributed one, we just needed 290 lines of code: a very small number also if compared with the dimension of the effectively used PIPE modules (∼6500 lines of code). In this particular implementation, `State`s correspond to reachable *marking*s, `Edge`s are of the type AA and they carry on information about firing transitions. The `createSuccessors` method simply identifies all reachable states from a given one, by making all enabled transitions fire. The `getFeatures` method just returns a compact representation of the actual *marking* of the `State`, and because the equality of the *marking* is a necessary and sufficient condition for equality between states, the application can exploits the *SimpleReducer* version.

Well-formed Nets. Well-formed Nets (WN) [9] are a power-retaining version of Colored Petri nets characterized by a structured syntax that permits the construction of a quotient graph, called Symbolic Reachability Graph (SRG). The SRG relies on the notion of Symbolic Marking (SM). SMs provide a syntactical equivalence relation on the set of concrete markings. They are formally expressed using dynamic subclasses instead of colors, representing parametric partitions of static subclasses in which WN color classes are in turn partitioned. The SRG is directly built from a given SM, through a symbolic firing rule. By using the *canonical representation* of a SM, the equivalence between SMs boils down to the syntactical identity. In order to exploit the MapReduce based framework for the SRG construction we first need a `SymbolicMarking` extension of `State`, in which `createSuccessors` (according to the symbolic firing rule) simply returns the list of successor SMs of the current SM, in a non-canonical form. Each SRG edge is by construction of kind AA. The `getFeatures` method should return the canonical representation of the current SM. In such a case the reduce phase is similar to the P/T nets case, thus we can exploit the *SimpleReducer* to fold the incoming lists of equivalent SMs. A possible adaptation of modules of GreatSPN package [11] (written in C), that natively supports the analysis of WN models, is currently under investigation.

Table 1. Experiments report

model	# machines	machine-type	# states	# reducers	threshold	time (m)
gas-burner	4	m2.2xlarge	14563	16	200	95
gas-burner	8	m2.2xlarge	14563	32	200	39
shared-memory	2	m2.2xlarge	1.831×10^6	2	1000	325
shared-memory	4	m2.2xlarge	1.831×10^6	4	1000	163
shared-memory	8	m2.2xlarge	1.831×10^6	4	1000	100
shared-memory	16	m2.2xlarge	1.831×10^6	4	1000	74
simple-lbs	20	m2.2xlarge	4.060×10^8	40	1000	530

5 Experiments

The experiments described in this section are executed using the Amazon Elastic MapReduce [2] on the Amazon Web Service cloud infrastructure and are partially supported by "AWS in Education Grant award" [1].

Gas Burner. The Gas Burner [4] is a benchmark real-time system model specified with a TB PN. Our experiments show that MaRDiGraS can be conveniently used to increase performances with respect to a sequential builder. The MaRDiGraS based tool, executed on the input model, generates a graph with 14563 nodes (23635 states are generated during computation) and it takes only 39 minutes, over 8 *m2.2xlarge* machines. Despite the generated graph is quite small, the execution time is 80% faster than the sequential approach running on

the same environment (2 hours and 55 minutes). It is worth noting that with this formalism we choose to implement a *getFeature* function that returns a necessary but not sufficient condition for the inclusion relationships between states, thus since we cannot exploit the *SimpleReducer*, the reduce phase becomes very expensive. For this reason, we gain substantial benefits by increasing the number of reducers, as shown in table 1.

Shared Memory. This model is taken from the GreatSPN benchmarks [10,19]. This P/T net models a system composed of 10 processors competing for the access to a shared memory using a unique shared bus. The PIPE tool fails after more than 20 hours of computation on a *m2.2xlarge* due to an out of memory error (Garbage Collector overhead limit exceeded). In such a situation the benefits deriving from using the adapted tool, as shown in table 1, are clear. As we can see, the construction is scalable even for this relatively small state space.

Simple Load Balancing. This P/T net represents a simple load balancing system composed of 10 clients, 2 servers, and between these, a load balancer process. In order to analyze this model, we implemented the building blocks of MaRDiGraS from scratch, to overcome some inefficiencies introduced by PIPE.

As shown in table 1, the reachability graph generated is very large: 4.060×10^8 states and 3.051×10^9 arcs for a total size of 104 GB of data. Thus this computation goes clearly beyond the capacity of a single machine. Fig. 4 shows the state space dimension over different MaRDiGraS iterations. As we can see, it explodes very quickly, but the computation slows at the end because the number of new states foreach iteration becomes very small. This condition could be tackled for example by considering different optimizations coming from the big data community: In particular we are evaluating the possibility of splitting old and new states into different files, and applying the *schimmy pattern* [22]. This would allow to highly decrease the time required by the last iterations.

6 Related Work

Several works, in the literature, describe tools and techniques for generating the state space associated to discrete-event systems in a parallel/distributed fashion. Among others, we may cite [8,12,13,20,7]. However, most of these works are related to a specific formalism, and they do not consider new emerging distributed solutions. Moreover, we considered another important aspect: we wanted to completely remove the costs of deploying our framework into an end-to-end solution, for this reason we developed our software on top of the consolidated Hadoop MapReduce framework. Works presented in [21,16] describe large-scale graph processing application reformulated in terms of MapReduce programming model, but unfortunately, this large class of graph algorithms doesn't fit well with the state explosion problem in large-scale graph building, which remains rather unexplored. As a common point, both iterate a number of times, using graph states

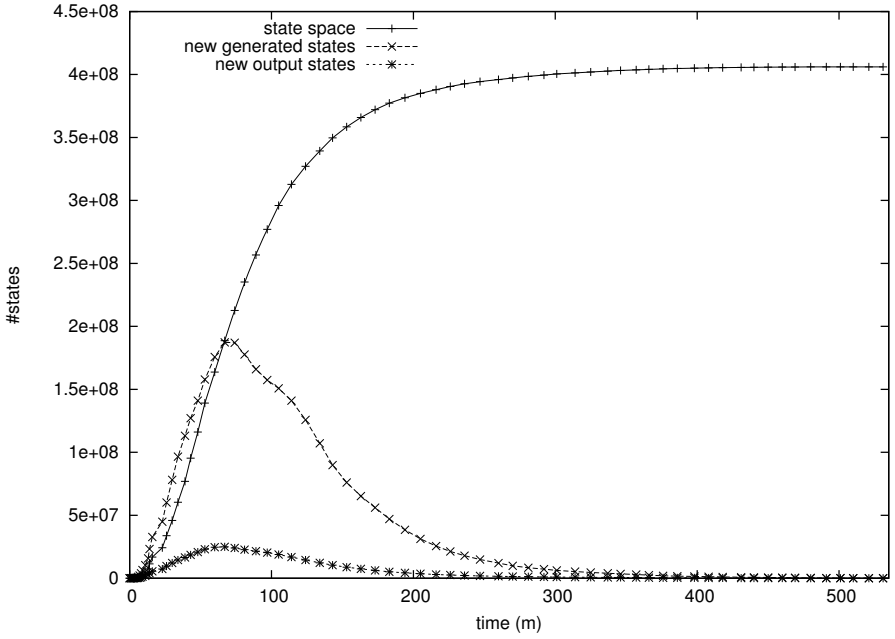


Fig. 4. Reachability graph computation of the simple-lbs model

from the previous iteration as input to the next one, until some stopping criterion is met. Thus both use an Iterative MapReduce approach [16,3]. But there are also significant differences: first of all, we have to deal with large graph building, not with large graph processing. Second, the input of each iteration is different: in graph processing, it is the internal status of all nodes of the graph; in graph building, it is a portion of the final graph. As a direct consequence, in the latter case, the input dimension greatly varies at each iteration making a standard iterative MapReduce approach ineffective. The input, in graph building, is also partitioned into two different classes of states: "explored states" and "unexplored states" and Mappers must act differently depending on the membership class. Moreover, some key points of graph building algorithms depend on the specific adopted formalisms, thus they must become user defined parameters.

7 Conclusion and Future Work

In this paper we presented MarDiGraS: a generic framework which can easily adapted for tackling the state explosion problem within the computation of the reachability graph associated to different formalisms. This framework exploits techniques typically used by the big data community and so far poorly explored for this kind of issues. Thanks to its very simple programming interface, it provides a powerful tool for constructing high-performance distributed applications

without the need to deal with the complex communication and synchronization issues required for exploiting a computation distributed on large clusters. Our experiments report that MaRDiGraS can be used effectively to compute state spaces sized with different orders of magnitude. We believe that this work could be a first step towards a synergy between two very different, but related communities: the "formal methods" community and the "big data" community. Exposing this issue to scientists with different backgrounds could stimulate the development of new interesting and more efficient solutions. Concerning future works, we are now working to develop it as the basic component of a generic library for distributed model checking. In particular we are currently developing a software tool which exploits the MaRDiGraS computed graphs by applying iterative map-reduce algorithms based on fixpoint characterizations of the basic temporal operators of CTL (Computational Tree Logic) and LTL (Linear Temporal Logic). Moreover, several questions remains open and require further investigation: for example, could a dynamic programming approach help in choosing partitions and/or thresholds? How the proposed computational model can be optimized when the number of new states gets very small? Are there classes of formalisms for which this approach cannot be used? And how can we adapt it to these classes?

References

1. Amazon Web Services. AWS in Education, `http://aws.amazon.com/grants/`
2. Amazon Web Services. Elastic MapReduce, `http://aws.amazon.com/-documentation/elasticmapreduce/`
3. Bellettini, C., Camilli, M., Capra, L., Monga, M.: Symbolic state space exploration of RT systems in the cloud. In: Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, pp. 295–302. IEEE CS Press, Los Alamitos (2012)
4. Bellettini, C., Capra, L.: Reachability analysis of time basic Petri nets: a time coverage approach. In: Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2011, pp. 110–117. IEEE CS Press, Los Alamitos (2011)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. Softw. Eng. 17, 259–273 (1991)
6. Boucheneb, H., Hadjidj, R.: CTL* model checking for time Petri nets. Theor. Comput. Sci. 353(1), 208–227 (2006)
7. Boukala, M.C., Petrucci, L.: Distributed model-checking and counterexample search for CTL logic. Int. J. Crit. Comput.-Based Syst. 3(1/2), 44–59 (2012)
8. Caselli, S., Conte, G., Marenzoni, P.: Parallel state space exploration for GSPN models. In: DeMichelis, G., Díaz, M. (eds.) ICATPN 1995. LNCS, vol. 935, pp. 181–200. Springer, Heidelberg (1995)
9. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: A symbolic reachability graph for coloured Petri nets. Theor. Comput. Sci. 176(1-2), 39–65 (1997)
10. Chiola, G., Franceschinis, G.: Colored GSPN models and automatic symmetry detection. In: Petri Nets and Performance Models, PNPM 1989, pp. 50–60 (1989)
11. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: Greatspn 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. Performance Evaluation 24, 47–68 (1995)

12. Ciardo, G.: Automated parallelization of discrete state-space generation. J. Parallel Distrib. Comput. 47(2), 153–167 (1997)
13. Ciardo, G., Gluckman, J., Nicol, D.: Distributed state space generation of discrete-state stochastic models. INFORMS J. on Comp. 10(1), 82–93 (1998)
14. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (2008)
15. Dingle, N.J., Knottenbelt, W.J., Suto, T.: Pipe2: a tool for the performance evaluation of generalised stochastic Petri nets. SIGMETRICS Perform. Eval. Rev. 36(4), 34–39 (2009)
16. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: Proc. of Symp. on High Performance Distributed Computing, pp. 810–818 (2010)
17. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified high-level petri net formalism for time-critical systems. IEEE Trans. Softw. Eng. 17(2), 160–172 (1991)
18. Jensen, K., Rozenberg, G.: High-level Petri nets: theory and application. Springer (1991)
19. Kordon, F., Linard, A., Buchs, D., Colange, M., Evangelista, S., Fronc, L., Hillah, L.-M., Lohmann, N., Paviot-Adet, E., Pommereau, F., Rohr, C., Thierry-Mieg, Y., Wimmel, H., Wolf, K.: Raw report on the model checking contest at Petri nets 2012. CoRR, abs/1209.2382 (2012)
20. Kristensen, L., Petrucci, L.: An Approach to Distributed State Space Exploration for Coloured Petri Nets. In: 25th International Conference on Application and Theory of Petri Nets, Bologna, Italy (2004)
21. Lin, J.: Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In: Research and Development in Information Retrieval, SIGIR 2009, pp. 155–162. ACM (2009)
22. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in mapreduce. In: Mining and Learning with Graphs, pp. 78–85. ACM Press, New York (2010)
23. The Apache Software Foundation. Hadoop MapReduce, `http://hadoop.apache.org/mapreduce/`
24. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)

Modular Synthesis with Open Components*

Ilaria De Crescenzo and Salvatore La Torre

Dipartimento di Informatica
Università degli Studi di Salerno

Abstract. We introduce and solve a new component-based synthesis problem that combines the synthesis from libraries of recursive components introduced by Lustig and Vardi with the modular synthesis introduced by Alur et al. for recursive game graphs. We model the components of our libraries as game modules of a recursive game graph with unmapped boxes, and consider as correctness specification a target set of vertices. To solve this problem, we give an exponential-time fixed-point algorithm that computes annotations for the vertices of the library components by exploring them backwards. We also show a matching lower-bound via a direct reduction from linear-space alternating Turing machines, thus proving EXPTIME-completeness. Finally, we give a second algorithm that solves this problem by annotating in a table the result of many local reachability game queries on each game component. This algorithm is exponential only in the number of the exits of the game components, and thus shows that the problem is fixed-parameter tractable.

1 Introduction

Synthesis is the construction of a system that satisfies a given correctness specification. This problem has been studied in different settings, and in particular the controller synthesis problem has a natural formulation as a two-player game (see [13,14]). Given a description of the system, where some of the choices depend upon the input and some represent uncontrollable nondeterminism (which may depend on the interaction with the external environment), the controller synthesis problem asks to determine a controller that supplies inputs to the system such that this satisfies a given correctness specification. Synthesizing a controller corresponds to computing winning strategies in a two-player game.

For pushdown systems modeled as recursive game graphs, where the system is composed of modules that can call each other in a potentially recursive manner (the game counterpart of recursive state machines [1]), it naturally arises the notion of *modular strategy* [3]. Asking for a modular strategy in a recursive game graph equals to require that the synthesized controller is formed of a set of finite state controllers (thus adhering to the modular design of controllers), one for each of the system modules. In executing such a controller, whenever a

* This work was partially funded by the MIUR grants FARB 2010-2011-2012, Università degli Studi di Salerno (Italy).

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 96–108, 2013.

module is called, the finite state controller for that module re-starts, i.e., it is oblivious of the previous history in the computation.

Component-based design plays a key role in configurable and scalable development of efficient hardware as well as software systems. For example, it is current practice to design specialized hardware using some base components that are more complex than universal gates at bit-level, and programming by using library features and frameworks.

In the modular synthesis for recursive game graphs, the call-return structure is given and cannot be modified. Therefore, the synthesis process concerns only the internal structure of each module and the modules cannot be freely composed. The work presented in this paper goes in the direction of providing new results for the automatic synthesis from components. In particular, we formulate and solve a new modular component-based synthesis problem for recursive game modules that, besides requesting the modularity of the solutions, allows also to re-configuring the call-return structure and using multiple instances of a same game module, each instance being controlled in a possibly different manner.

The game modules for our component-based synthesis are taken from a finite set (*library*) of *game components*. Game components differ from game modules in that the boxes are not mapped to any module (as an empty position in a code where we could insert a function call). Namely, a game component is a two-player finite game graphs with two kinds of vertices: standard *nodes* and *boxes*. Each box has *call* and *return* points, and each component has distinguished *entry* and *exit* nodes. The edges are *from* a node or a return *to* a node or a call within the same component. Moreover, the nodes and the returns are split among the two players (pl_0 and pl_1).

The correctness specification is given as a set of target exits $\mathcal{T}$ of a game component C_{main}. The *modular synthesis problem* asks to construct (1) a recursive game graph G by using as game modules copies of the library components and (2) a modular strategy f for pl_0 in G such that: all the maximal plays σ, starting from the entry of C_{main} and that conform to f, visit a vertex in $\mathcal{T}$. We solve this problem and address its computational complexity.

Our first contribution is a fixed-point algorithm $\mathcal{A}_1$ that decides in exponential time the above modular synthesis problem. This algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b \in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of another component C_b or undefined. Each such tuple summarizes for vertex u a reachable *local target* E (via a modular strategy of pl_0) and a set of *assumptions* $\{\mu_b\}_{b \in B}$ that are used to get across the boxes in order to reach the local target. We start from the tuples of the target exits $\mathcal{T}$ and then propagate the search backwards in the game components. Internally to each component, the search proceeds as in the standard attractor set construction [12] and it is propagated through calls to other components from the returns to the exits and then back from the entries to the calls. In this, tuples that have incompatible assumptions or refer to a different local target are

treated as belonging to different searches and thus are not used together in the update rules.

Our second contribution is to show a matching lower bound by a reduction from linear-space alternating Turing machines. In the reduction, we use only four game components and $O(n)$ exits, where n is the number of cells used in the configurations of the Turing machine.

Finally, we delve deeper in the computational complexity of this problem, and give a second decision algorithm $\mathcal{A}_2$ for it. The main idea here is to solve many reachability game queries "locally" to each game component and maintain a table with the obtained results to avoid recomputing. Each table entry corresponds to a game component and a set of its exits (used as targets in the query), and for the successful queries, contains a link to each table entry that has been used to reach the target (we look up into the table to propagate the search across the boxes). We observe that $\mathcal{A}_2$ takes time exponential only in the number of exits, while $\mathcal{A}_1$ takes time exponential also in the number of boxes. This is due mainly to the fact that $\mathcal{A}_1$ may compute and store exponentially many different ways of assigning the boxes to modules, in contrast, $\mathcal{A}_2$ computes and stores just one of them. Therefore, since alternating reachability in finite game graphs is already PTIME-hard, by algorithm $\mathcal{A}_2$ we get that the considered problem is PTIME-complete when the number of exits is fixed.

Related Work. The synthesis problem addressed in this paper combines the synthesis from libraries of recursive components [11] with the synthesis of modular strategies for recursive game graphs [3]. In fact, if the game components of the considered library do not contain pl_1 vertices, the problem reduces to a synthesis problem from recursive component libraries. If we instead constrain the solution to use at most one copy for each game component, we can encode our synthesis problem as a synthesis of modular strategies for recursive game graphs.

We recall that in [11] the components are modeled with transducers with call-return structures, and the correctness specification is given as a temporal logic formula over nested words. The same synthesis problem with LTL specifications and components modeled as standard finite-state transducers is addressed in [10]. In [4], this problem is formulated for synthesizing hierarchical systems bottom-up with respect to a different μ-calculus specification for each component in the hierarchy. All these synthesis problem turn out to be 2EXPTIME-complete. The synthesis from libraries of components with simple specifications has been also implemented in tools (see [8] and references therein).

Modular synthesis of recursive game graphs with several classes of ω-regular specifications is solved in [2] and is shown to be EXPTIME-complete already with finite automata specifications. The computational complexity of this problem turns out to be NP-complete for reachability specifications [3]. In [6], the modular synthesis of recursive game graphs is shown to be 2EXPTIME-complete with respect to visibly pushdown specifications. A solution to CaRet games that computes winning strategies that are modular for the recursive game graph extended with set of subformulas of the specification formula is given in [5]. The notion of modular strategy is also of independent interest and has recently found

application in the automatic transformation of programs for ensuring security policies in privilege-aware operating systems [7].

2 A Modular Synthesis Problem

In this section, we define our modular synthesis problem. For this, we introduce first some preliminary notions and recall known ones.

For $n \in \mathbb{N}$, with $[n]$ we denote the set of naturals i s.t. $1 \leq i \leq n$.

Library of (game) Components. For $h, k \in \mathbb{N}$, a (h,k)-*component* is a finite graph with two kinds of vertices, the standard *nodes* and the *boxes*, and with h *entry* nodes and k *exit* nodes. Each box has h *call* points and k *return* points, and the edges take from a node/return to a node/call in the component.

Formally, for a box b, we denote with (i,b) the i-th call of b for $i \in [h]$, and with (b,i) the i-th return of b for $i \in [k]$. A (h,k)-*component* is a tuple (N, B, En, Ex, δ) where N is a finite set of nodes, B is a finite set of boxes, $En \subseteq N$ is the set of entries, $Ex \subseteq N$ is the set of exits, and $\delta : N \cup Retns \to 2^{N \cup Calls}$ where $Retns = \{(b,i) \mid b \in B, i \in [k]\}$ and $Calls = \{(i,b) \mid b \in B, i \in [h]\}$. The calls, returns and nodes of a component form its set of vertices. In the following, when we do not need to specify h and k, we simply write component.

A *game component* is a component whose nodes and returns are split into two sets P^0 and P^1, where P^0 is the set of player 0 (pl_0) positions and P^1 is the set of player 1 (pl_1) positions. We denote it as a tuple $(N, B, En, Ex, \delta, P^0, P^1)$.

For $h, k > 0$, a *library* of (game) components is a finite set $\mathcal{L}ib = \{C_i\}_{i \in [n]}$ where each C_i is a (game) (h,k)-component.

To ease the presentation we make the following standard assumptions:

- there is only one entry node for every (game) component and thus just one call for each box, i.e., we refer to (game) $(1,k)$-components;
- in each (game) component there are no transitions taking to its entry and no transitions leaving from its exits, i.e., the entries are sources and the exits are sinks in the graph representation of the component;
- there is no transition from a return to a call, i.e., two boxes are not directly connected by a single transition.

Instances from a Library. Intuitively, an instance of a (game) component C is a copy A of C where each box is mapped to an instance of a (game) component (possibly A itself). Depending on whether we consider a library of components or of game components, the instances define a *recursive state machine* [1] or a *recursive game graph* [3].

Fix a library $\mathcal{L}ib = \{C_1, \ldots, C_n\}$ of game components.

A *recursive game graph* from $\mathcal{L}ib$ is $G = (M, m_{in}, \{S_m\}_{m \in M})$ where M is a finite set of module names, $m_{in} \in M$ is the name of the initial module and for each $m \in M$, S_m is a game module. A *game module* S_m is defined as $(N_m, B_m, Y_m, \{e_m\}, Ex_m, \delta_m, P^0_m, P^1_m)$ where:

- $Y_m : B_m \to (M \setminus \{m_{in}\})$ is a labeling function that maps every box to a game module;
- $(N_m, B_m, \{e_m\}, Ex_m, \delta_m, P_m^0, P_m^1)$ is equal to a component C of $\mathcal{L}ib$ up to a renaming of nodes and boxes such that calls and returns of a box b are 1-to-1 mapped to the entries and the exits of $M_{Y_m(b)}$, that is, denoting $Ex_{Y_m(b)} = \{x_1, \ldots, x_k\}$: the call of b is renamed to $(e_{Y_m(b)}, b)$ and each return (b, i) is renamed to (b, x_i).

The calls, returns and vertices of S_m are denoted respectively $Calls_m$, $Retns_m$ and V_m. We also assume the following notation: $V = \bigcup_m V_m$ (set of all vertices); $B = \bigcup_m B_m$ (set of all boxes); $Calls = \bigcup_m Calls_m$ (set of all calls); $Retns = \bigcup_m Retns_m$ (set of all returns); and $P^i = \bigcup_m P_m^i$ for $i = 0, 1$ (set of all positions of pl_i).

The definition of a *recursive state machine* from $\mathcal{L}ib$ can be obtained from that of recursive game graph by ignoring the splitting among pl_0 and pl_1 nodes.

A (global) *state* of G is composed of a call stack and a vertex. Formally, the states are of the form $(\gamma, u) \in B^* \times V$ where $\gamma = b_1 \ldots b_h$, $b_1 \in B_{m_{in}}$, $b_{i+1} \in B_{Y(b_i)}$ for $i \in [h-1]$ and $u \in V_{Y(b_h)}$. In the following, for a state $s = (\gamma, u)$, we denote with $V(s)$ its vertex, that is $V(s) = u$.

A *play* of G is a (possibly finite) sequence of states $s_0 s_1 s_2 \ldots$ such that $s_0 = (\epsilon, e_{m_{in}})$ and for $i \in \mathbb{N}$, denoting $s_i = (\alpha_i, u_i)$, one of the following holds:

- **Internal move:** $u_i \in (N_m \cup Retns_m) \setminus Ex_m$, $u_{i+1} \in \delta_m(u_i)$ and $\alpha_i = \alpha_{i+1}$;
- **Call to a module:** $u_i \in Calls_m$, $u_i = (b, e_{m'})$, $u_{i+1} = e_{m'}$ and $\alpha_{i+1} = \alpha_i.b$;
- **Return from a call:** $u_i \in Ex_m$, $\alpha_i = \alpha_{i+1}.b$, and $u_{i+1} = (b, u_i)$.

Modular Strategies. A *strategy* of a player pl is a function f that associates a legal move to every play ending in a node controlled by pl. A *modular strategy* [3] for G consists of a set of local strategies, that are used together as a global strategy for a player. A local strategy for a game module S can only refer to the local memory of S, i.e. the sequence of S vertices that are visited in the play in the current invocation of S.

Formally, fix $j \in \{0, 1\}$. A *modular strategy* f of pl_j is a set of functions $\{f_m\}_{m \in M}$, one for each game module, where for every m, $f_m : V_m^*.P_m^j \to V_m$ such that $f_m(\sigma.u) \in \delta_m(u)$ for every $\sigma \in V_m^*, u \in P_m^j$.

Fix a play $\sigma = s_0 s_1 ... s_n$ where $s_i = (\gamma_i, u_i)$ for any i. Denote with $\sigma_i = s_0 s_1 ... s_i$, i.e., the prefix of σ up to index i. With $ctr(\sigma_i)$ we denote $m \in M$ such that $u_i \in V_m$, that is the name of the game module where the control is after σ_i. The *local history* at σ_i, denoted $\lambda(\sigma_i)$, is the maximal sequence of S_m vertices u_j, $j \leq i$, starting with the most recent occurrence of entry e_m where $m = ctr(\sigma_i)$.

A play σ *conforms to* a modular strategy $f = \{f_m\}_{m \in M}$ of pl_j if for every $i <\mid \sigma \mid$, denoting $ctr(\sigma_i) = m$, $u_i \in P_m^j$ implies that $u_{i+1} = f_m(\lambda(\sigma_i))$.

Modular Synthesis from Libraries of Game Components. A *modular game over a library* is $(\mathcal{L}ib, C_{main}, \mathcal{T})$ where $\mathcal{L}ib$ is a library of game components, $C_{main} \in \mathcal{L}ib$ and $\mathcal{T}$ is a set of exits of C_{main}.

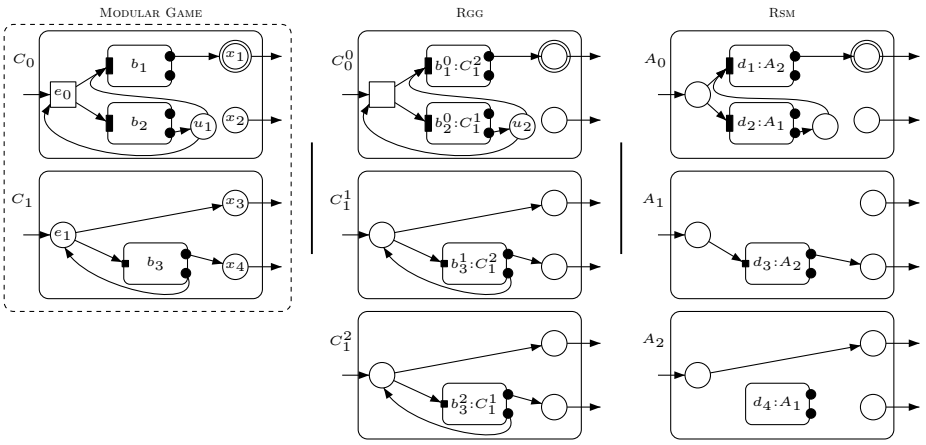


Fig. 1. An example of modular synthesis

Given an instance $(\mathcal{L}ib, C_{main}, \mathcal{T})$ of a modular game over a library, the *modular synthesis problem* is the problem of determining whether: for some recursive game graph G from $\mathcal{L}ib$ whose initial module is an instance of C_{main}, there exists a modular strategy f for pl_0 in G such that all the maximal plays that conform to f reach an exit of the initial module of G that corresponds to an exit in $\mathcal{T}$.

Such a strategy f for pl_0 is called a *winning modular strategy.*

Example. We illustrate the definitions with an example. In the first column of Fig. 1, we give $(\mathcal{L}ib, C_0, \{x_1\})$, an instance of a modular game over a library of game components. Each game component has two exits, and $\mathcal{L}ib$ is composed of two game components C_0 and C_1. In the figure, we denote the nodes of pl_0 with circles and the nodes of pl_1 with squares. Rounded squares are used to denote the boxes. The target is marked with a double circle. C_0 has one entry e_0, two exits x_1 and x_2, and two boxes b_1 and b_2. C_1 has one entry e_1, two exits x_3 and x_4, and one box b_3.

In the second column of the figure, we show one of the possible recursive game graphs that can be obtained from $\mathcal{L}ib$ and whose initial module C_0^0 is an instance of C_0. Note that we have marked as target the vertex of C_0^0 that corresponds to (i.e., is a copy of) x_1. The other modules C_1^1 and C_1^2 are instances of C_1. Note that each box now is mapped to a game module, for example b_1^0 is mapped to C_1^2. Also, the box b_3^1 of C_1^1 is mapped to C_1^2 and the box b_3^2 of C_1^2 is mapped to C_1^1 thus forming a cycle in the chain of recursive calls.

Consider a modular strategy for pl_0, where the local strategy of C_0^0 selects the call from u_2, the local strategy of C_1^1 selects the call from its entry and the local strategy for C_1^2 selects the upper exit from its entry. This strategy is winning and modular. In the third column of the figure, we show a recursive state machine, obtained from the considered recursive game graph by resolving the moves of pl_0 according to this modular strategy. To simplify the figure, we

have deleted all the unreachable transitions. Clearly, each run of this machine reaches the target. Also, note that in the considered game it is not possible to win if we do not instantiate at least two instances of C_1.

3 Solving Our Modular Synthesis Problem

In this section, we describe an exponential-time fixed-point algorithm to solve the modular synthesis problem.

We fix a library of game components $\mathcal{L}ib = \{C_{main}, C_1, \ldots, C_n\}$ and a target set $\mathcal{T}$ of C_{main} exits.

Intuitively, our algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b \in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of a game component or undefined (we use $\perp$ to denote this). The intended meaning of such tuples is that: there is a local strategy f of pl_0 in C such that starting from u, each maximal play conforming to f reaches an exit within E, under the assumption that: for each box $b \in B$, if μ_b is defined, then from the call of b the play continues from one of the returns of b corresponding to a $x \in \mu_b$ (if μ_b is undefined means that no play conforming to f visits b starting from u). Thus, each tuple $(u, E, \{\mu_b\}_{b \in B})$ summarizes for vertex u a reachable *local target* E and a set of *assumptions* $\{\mu_b\}_{b \in B}$ that are used to get across the boxes.

For computing Φ, we use the concept of compatibility of the assumptions. Namely, we say that two assumptions μ and μ' are *compatible* if either $\mu = \mu'$, or $\mu' = \perp$, or $\mu = \perp$ (i.e., there is at most one assumption that has been done). Moreover, we say that the assumptions $\mu_1, \ldots, \mu_m$ are *passed to* μ if $\mu = \bigcup_{i \in [m]} \mu_i$ (we assume that $\perp \cup X = X \cup \perp = X$ holds for each set X).

The set Φ is initialized with all the tuples of the form $(u, \mathcal{T}, \{\perp\}_{b \in B_{main}})$ where $u \in \mathcal{T}$ and B_{main} is the set of boxes of C_{main}. Then, Φ is updated by exploring the components backwards according to the game semantics, and in particular: within the components, tuples are propagated backwards as in an attractor set construction, by preserving the local target and passing to a node the assumptions of its successors (provided that multiple assumptions on the same box are are passed they are pairwise compatible); the exploration of a component is started from the exits with no assumptions on the boxes, whenever the corresponding returns of a box b have been discovered with no assumptions on b; the visit of a component is resumed at the call of a box b, whenever (1) there is an entry of a component that has been discovered with local target X and (2) there is a set of b returns corresponding to the exits X with no assumptions on b (thus, that can be responsible for discovering the exits in X as in the previous case) and with compatible assumptions on the remaining boxes; if this is the case, then the call is discovered with the assumption X on box b and passing the local target and the assumptions on the other boxes as for the above returns.

Below, we denote with b_x the return of a box b corresponding to an exit x (recall that all game components of a library have the same number of exits, and so do the boxes). The update rules are formally stated as follows:

UPDATE 1: For a pl_0 vertex v, we add $(v, E, \{\mu_b\}_{b\in B})$ provided that there is a transition from v to u and $(u, E, \{\mu_b\}_{b\in B}) \in \Phi$ (the local target and the assumptions of a v successor are passed on to a pl_0 vertex v).

UPDATE 2. For a pl_1 vertex v, denote $u_1, \ldots, u_m$ all the vertices s.t. there is a transition from v to u_i, $i \in [m]$, then we add $(v, E, \{\mu_b\}_{b\in B})$ to Φ provided that for each $i, j \in [m]$ and $b \in B$: (1) there is a $(u_i, E_i, \{\mu_b^i\}_{b\in B}) \in \Phi$, (2) $E_i = E_j$, (3) μ_b^i and μ_b^j are compatible, and (4) $\mu_b = \bigcup_{i\in[m]} \mu_b^i$ (all the v successors must be discovered under the same target and with compatible assumptions; target and assumptions are passed on to a pl_1 vertex v).

UPDATE 3. For an exit u, we add a tuple $(u, E, \{\bot\}_{b\in B'})$ to Φ provided that $u \in E$ and for a box b' it holds that there are tuples $(b'_x, E_x, \{\mu_b^x\}_{b\in B}) \in \Phi$, one for each $x \in E$, such that for all $x, y \in E$ and $b \in B$, (1) $\mu_{b'}^x = \bot$, (2) $E_x = E_y$, and (3) μ_b^x and μ_b^y are compatible (the discovery of the exits follows the discovery of the corresponding returns under compatible assumptions and the same local target).

UPDATE 4. For a call u of a box b', we add a tuple $(u, E_u, \{\mu_b^u\}_{b\in B})$ to Φ provided that (i) there is an entry e s.t. $(e, E_e, \{\mu_b^e\}_{b\in B'}) \in \Phi$, (ii) for each return b'_x, $x \in E_e$, there is a tuple $(b_x, E, \{\mu_b^x\}_{b\in B}) \in \Phi$ s.t. all these tuples satisfy (1), (2) and (3) of UPDATE 3, and moreover, (iii) $E_u = E$, $\mu_b^u = \bigcup_{x\in E_e} \mu_b^x$ for $b \neq b'$, and $\mu_{b'}^u = E_e$ (the discovery of a call u of box b' follows the discovery of an entry e from exits E_e that in turn have been discovered by matching returns b'_x, $x \in E$; thus on u we propagate the local target and the assumptions on the boxes $b \neq b'$ of the returns b'_x and make an assumption E_e on box b').

We compute Φ as the fixed-point of the recursive definition given by the above rules and outputs "YES" iff $(e, \mathcal{T}, \{\mu_b\}_{b\in B_{main}}) \in \Phi$ for the entry e of C_{main}.

Observe that, the total number of tuples of the form $(u, E, \{\mu_b\}_{b\in B})$ is bounded by $|\mathcal{L}ib|\, 2^{O(k\beta)}$ where k is the number of exits of each game component in $\mathcal{L}ib$ and β is the maximum over the number of boxes of each game component. Therefore, the algorithm always terminates and takes at most time exponential in k and β, and linear in the size of $\mathcal{L}ib$.

Soundness of the algorithm is a consequence of the fact that each visit of a game component is done according to the standard attractor set construction, and repeated explorations of each component are kept separate by allowing to progress backwards in the graph only with the same local target and compatible assumptions on the boxes. By not allowing to change the box assumptions (when defined), we ensure that we cannot cheat by using different assumptions in repeated visits of a box within the same exploration. The computed strategy is clearly modular since we compute it locally to each graph component. Note that we can end up computing more than a local strategy for each graph component, but this does not break the modularity of the solution since this happens when in the computed solution we use different instances of the component. Also, observe that for each game component we construct at most a local strategy for each possible subset of its exits, thus we bound the search of a solution to modular strategies of this kind.

To prove completeness, we first observe that using standard arguments one can show that:

Lemma 1. *If there is a modular winning strategy for an instance of the modular synthesis problem over a library $\mathcal{L}ib$, then there is a winning modular strategy f for a recursive game graph G from $\mathcal{L}ib$ such that: for each two instances S and S' of a same game component in $\mathcal{L}ib$, the sets of exits visited along any play conforming to f in S and S' differ.*

Observe that by the above lemma, we can restrict the search for a solution within the modular strategies of the instances of a $\mathcal{L}ib$ that have at most 2^k copies of each game component, where k is the number of exits for the components. Therefore, combining this with the results from [3] we get a simple argument to show membership to NEXPTIME of the considered problem.

The next step in the completeness argument is to show that if there is a winning modular strategy f as for Lemma 1, then our algorithm outputs YES. Denoting with G the recursive game graph from $\mathcal{L}ib$ for which f is winning, this can be shown by proving by induction on the structure of G that: if on a game module S of G that is an instance of $C \in \mathcal{L}ib$, f forces to visit a set of exits corresponding to the exits X of C, then the algorithm adds to Φ the tuples $(x, X, \{\bot\}_{b\in B})$ for each $x \in X$ and eventually discovers the entry of C with local target X. We omit the proof of this here.

Therefore, we get that the algorithm is a solution of the modular synthesis problem from game component libraries, and the following theorem holds.

Theorem 1. *The modular synthesis problem from libraries of game components with k exits and at most β boxes can be solved in time linear in the size of $\mathcal{L}ib$ and exponential in k and β.*

4 Computational Complexity Analysis

Lower-bound. We reduce the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components, thus showing EXPTIME-hardness for this problem.

Consider a linear-space alternating Turing machine A and an input word $w = a_1 \dots a_n$. Without loss of generality, we assume that the transition function δ of A is the union of two functions δ_1 and δ_2 where $\delta_i : Q \times \Sigma \to \{L, R\} \times Q$ for $i \in [2]$, and Q is the set of control locations, Σ is the tape alphabet, and L/R cause to move the tape head to left/right. A configuration of A is represented as $b_1 \dots (q, b_i) \dots b_n$ where b_j is the symbol at cell j of the input tape for $j \in [n]$, q is the control state and the tape head is on cell i. The control states are partitioned into states where the $\exists$-player can move, and states where the $\forall$-player can move. A computation of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation ρ that reaches a configuration with a final state on all the plays conforming to ρ.

Denoting $h = n\,|\Sigma|\,(|Q|+1)$, fix two sets $X = \{x_1, \dots, x_h\}$ and $Y = \{y_1, \dots, y_h\}$ such that each x_i and y_i correspond exactly to a symbol and a position in a configuration of A (i.e., for each symbol in $\Sigma \cup Q \times \Sigma$ we have exactly n variables from X

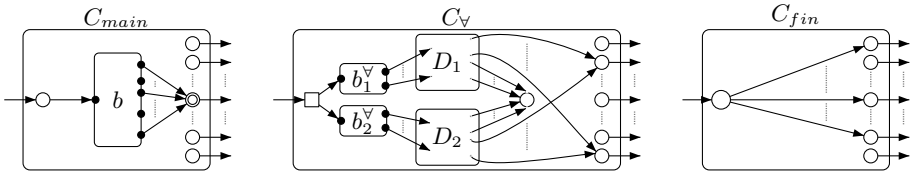


Fig. 2. Graphical representation of the game components C_{main}, $C_\forall$ and C_{fin}

and n from Y, one for each position on the tape). We can encode each configuration $\sigma_1 \dots \sigma_n$ of A by setting to true a variable x_j (resp. y_j) iff it corresponds to a σ_i for $i \in [n]$ (that is, to a configuration symbol and its position in the configuration). It is well-known that for each δ_i, we can construct a Boolean circuit (using only the logical gates AND and OR) with inputs $\bar{x} = x_1 \dots x_h$ and outputs $\bar{y} = y_1, \dots, y_h$, such that if $\bar{x}$ is an encoding of a configuration, then $\bar{y}$ is the next configuration after the application of the only possible transition of δ_i.

From each such circuit we can construct a game graph by replacing each AND gate with a node of pl_1 and each OR gate with a node of pl_0. We denote with D_1 and D_2 the game graphs corresponding to the above circuits for δ_1 and δ_2, respectively. The encoding of the bits is done by reachability, that is, true at an input x_i corresponds to connecting it to a vertex that can lead to the target, and false otherwise. Since the circuits compute a next configuration, from each output wire y_i that evaluates to true we will be able to get to the target by a strategy that resolves the choices on the pl_0 nodes (and thus the OR gates), and this will not be possible for those y_i that evaluates to false.

We construct a library $\mathcal{L}ib$ containing exactly the game components C_{main}, $C_\forall$, $C_\exists$, and C_{fin} (see Fig. 2). Each component has exactly h exits, each one corresponding to a variable x_i for $i \in [h]$. In C_{main}, we arbitrarily select an exit as the only vertex in the target $\mathcal{T}$, and link to it all the returns of the box that encode the initial configuration (we can assume that A has only one initial state). In $C_\forall$, all the exits are wired as inputs to both D_1 and D_2 except for those that correspond to states of the $\exists$-player. We add a pl_0 node that has no out-going edges and is wired as input to D_1 and D_2 for the remaining inputs. The outputs of D_1 and D_2 are wired respectively to the boxes $b_1^\forall$ and $b_2^\forall$, and the calls of these boxes are connected to the entry, that is a pl_1 node. $C_\exists$ is as $C_\forall$ except that the entry is a pl_0 node and the exits that are not connected correspond to $\forall$-player states. The component C_{fin} has just the entry and the exits. The entry is a pl_0 node and is connected to all the exits that correspond to a final state.

It is simple to verify that if, starting from an instance of C_{main}, we map the boxes such that to reproduce an accepting computation of A, then we get a recursive game graph that admits a modular winning strategy of pl_0. Vice-versa, suppose that there is a modular winning strategy of pl_0 in the synthesis problem $(\mathcal{L}ib, C_{main}, \mathcal{T})$. First, observe that since the returns from which we reach the target encode a legal initial configuration, each game module to which

we map the box b will have the corresponding exits with the same property. Moreover, in order to reach backwards the entries of all the used instances of C_{main}, $C_\forall$, and $C_\exists$, at some point we need to use a copy of C_{fin}. Now, if the initial state is a $\forall$-player state and we map b to an instance of $C_\exists$, since the exit encoding the head position and the state will not be wired to D_1 and D_2, in all the modules below in the hierarchy of calls, none of such exits will be connected to the target. Thus, also the entry of each copy of C_{fin} in this hierarchy would not be connected to the target, and so all the entries up to the entry of the copy of C_{main}, thus contradicting the hypothesis. A contradiction can be shown also in the dual case. Thus, at any point we must have mapped each box to an instance of either $C_\exists$ or $C_\forall$ depending on whether the next move is of the $\exists$-player or the $\forall$-player. Since, the graphs D_1 and D_2 ensure the correct propagations of the reachability according to the computed configurations, we can correctly reconstruct a computation ρ of A from the modular strategy. Moreover, since a winning modular strategy ensures that each maximal sequence of module calls ends with a call to an instance of C_{fin}, then each play of ρ ends in a final configuration and thus ρ is accepting, that concludes the proof.

Lemma 2. *There is a polynomial-time reduction from the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components. Moreover, the resulting library has four game components each one with at most two boxes and a number of exits which is linear in the size of the input word.*

Complexity and fixed-parameter tractability. The algorithm from the previous section, say $\mathcal{A}_1$, shows membership to EXPTIME for the modular synthesis problem. Therefore, by Lemma 2, we get:

Theorem 2. *The modular reachability problem is* EXPTIME*-complete.*

Note that $\mathcal{A}_1$ takes time exponential in both the number of boxes β and the number of exits k. We sketch a different algorithm that shows that this problem is indeed in PTIME when the number of exits for each game component is fixed.

The main idea is to solve many reachability game queries on standard finite game graphs, where each query asks to determine for a game component C and a subset of its exits E: if there exists a modular strategy f of pl_0 such that all the maximal plays, which conform to f and start from the entry of C, reach one of the exits from E. To avoid recomputing, the results of such queries are stored in a table T, and the algorithm halts when no more queries can be answered positively.

To solve the query for a component C and a set of its exits E, we extend the standard attractor set construction. Namely, we accumulate the winning set for pl_0 as usual for nodes and returns. To add the call of a box b, we look in the table for a positively answered query whose target set correspond to returns of b that are already in the winning set. If the entry of C is added to the winning set, then we update the T entry for E and C to YES, and store the links to the table entries that have been used to add the calls (observe that we just need to store

exactly a link for each box that is traversed to win in the game query in order to synthesize the recursive game graph and the winning modular strategy).

With similar arguments as those used in Section 3, we can show that pl_0 has a winning modular strategy in the input modular synthesis problem if and only if the T entry for the target set $\mathcal{T}$ is set to YES. Since the size of the table is exponential in k and linear in β, and that solving the "local" reachability games is linear in the size of the game component and in the size of the table, we get that the whole algorithm takes time exponential in k and linear in β (and the size of the library). Since already alternating reachability is PTIME-hard, we get:

Theorem 3. *The modular reachability problem for a fixed number of exits is* PTIME*-complete.*

We observe that $\mathcal{A}_1$ computes all the solutions of the kind as from Lemma 1, by trying all the possible ways of assigning each box with all the game components. This causes the exponential in the number of boxes, but also gives a quite simple and direct way to show completeness. Moreover, the fixed-point updates of $\mathcal{A}_1$ can be implemented quite efficiently and only the sets of exits from which we can reach the target (in a series of calls) are used in the computation.

Algorithm $\mathcal{A}_2$ arbitrarily computes, for each game component and each set of exits, only one assignment of each box with a game module. Moreover, it computes (several times) all the game queries, even those with exits that cannot reach the global target $\mathcal{T}$.

Both algorithms can be used to synthesize the winning modular strategy as a recursive state machine. Also, we can modify them to compute optimal winning modular strategies with respect to some criteria, such as minimizing the number of modules, the depth of the call stack or the number of used exits.

5 Conclusion

In this paper, we have introduced a formulation of the synthesis problem that generalizes both the modular synthesis of recursive game graphs and the synthesis from component libraries. We have solved this problem for reachability specifications, and in particular, we have shown that it is EXPTIME-complete and is fixed-parameter tractable when the number of exits is fixed.

Besides the optimization problems mentioned at the end of previous section, we see several other future directions that could be investigated.

In our formulation, the number of instances of each component that are allowed in a solution is unbounded. It is realistic to consider some limitations, in particular, we plan to investigate variations of the considered synthesis problem where in each solution there is at most one game module that instantiates each component, or where for all the game modules that instantiate a same component we require the same local function.

We have considered only reachability specifications. It is natural to investigate more complex specifications such as regular or pushdown specifications expressed

as temporal logic formulas or automata models. Moreover, to synthesize more succinct solutions, it could be interesting to investigate the effect of a hierarchical labeling such as in [9].

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. 27(4), 786–818 (2005)
2. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for infinite games on recursive graphs. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 67–79. Springer, Heidelberg (2003)
3. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for recursive game graphs. Theor. Comput. Sci. 354(2), 230–249 (2006)
4. Aminof, B., Mogavero, F., Murano, A.: Synthesis of hierarchical systems. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 42–60. Springer, Heidelberg (2012)
5. De Crescenzo, I., La Torre, S.: Winning caret games with modular strategies. In: Fioravanti, F. (ed.) CILC. CEUR Workshop Proceedings, vol. 810, pp. 327–331. CEUR-WS.org (2011)
6. De Crescenzo, I., La Torre, S.: Visibly pushdown modular games. Technical Report, University of Salerno, pp. 1–19 (2013)
7. Harris, W.R., Jha, S., Reps, T.: Secure programming via visibly pushdown safety games. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 581–598. Springer, Heidelberg (2012)
8. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 215–224 (May 2010)
9. La Torre, S., Napoli, M., Parente, M., Parlato, G.: Verification of scope-dependent hierarchical state machines. Inf. Comput. 206(9-10), 1161–1177 (2008)
10. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 395–409. Springer, Heidelberg (2009)
11. Lustig, Y., Vardi, M.Y.: Synthesis from recursive-components libraries. In: D'Agostino, G., La Torre, S. (eds.) GandALF. EPTCS, vol. 54, pp. 1–16 (2011)
12. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic 65(2), 149–184 (1993)
13. Thomas, W.: Infinite games and verification (Extended abstract of a tutorial). In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)
14. Thomas, W.: Facets of synthesis: Revisiting church's problem. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 1–14. Springer, Heidelberg (2009)

Parameterized Verification of Broadcast Networks of Register Automata

Giorgio Delzanno[1], Arnaud Sangnier[2], and Riccardo Traverso[1]

[1] DIBRIS, University of Genova, Italy
[2] LIAFA, Univ Paris Diderot, Paris Cité Sorbonne, CNRS, France

Abstract. We study parameterized verification problems for networks of interacting register automata. We consider safety properties expressed in terms of reachability, from arbitrarily large initial configurations, of a configuration exposing some given control states and patterns.

1 Introduction

We introduce a formal model of data-sensitive distributed protocols, called Broadcast Networks of Register Automata (BNRA), aimed at modelling both the local knowledge of distributed nodes as well as their interaction via broadcast communication. A network is modelled via a finite graph where each node runs an instance of a common protocol. A protocol is specified via a register automaton, an automaton equipped with a finite set of registers [20]. Each register assumes values taken from the set of natural numbers. Node interaction is specified via broadcast communication, well-suited to model scenarios in which individual nodes have partial information about the network topology. Messages are allowed to carry data, that can be assigned to or tested against the local registers of receivers. Dynamic updates of the current configuration are modelled via non-deterministic reconfigurations of the underlying connectivity graph. A node may disconnect from its neighbours and connect to other ones at any time of the execution. This behaviour models in a natural way unexpected power-off and dynamic movement of devices. The resulting model can be used to reason about core parts of client-server protocols as well as of routing protocols, e.g. route maintainance as in Link Reversal Routing.

In the paper we focus our attention on the decidability and complexity of parameterized verificatiom, i.e., the problem of finding a sufficient number of nodes and an initial topology that may lead to a configuration exposing a bad pattern (e.g. a loop in the information contained in the routing tables). The considered class of verification problems is parametric in four dimensions, namely, the number of nodes, the topology of the initial configuration to be discovered, and the amount of data contained in local registers and exchanged messages.

Related Works. Our formal model of topology-sensitive broadcast communication with data naturally extends those obtained in [11,12,10]. Formal models of broadcast networks date back to CBS [22], extended in several ways (time,

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 109–121, 2013.

asynchrony, etc) in successive works. Automated verification methods have been tested on protocols for Ad Hoc Networks with a fixed number of processes in [16,25,15]. Verification of broadcast protocols in fully connected networks in which nodes and messages range over a finite set of states has been considered, e.g., in [13,18,5]. Via an adequate counting abstraction, the problem can be reformulated in terms of Petri nets with transfer arcs [14,7]. The non-elementary complexity of coverability in this class of nets is proved in [24]. Symbolic backward exploration procedures for network protocols specified in graph rewriting have been presented in [19] (termination guaranteed for ring topologies) and [23] (approximations without termination guarantees). Decidability issues for broadcast communication in fully connected networks have been studied in [14]. Verification of unreliable communicating FIFO systems has been studied in [3]. Coverability problems for broadcast communication in fully connected networks with data is investigated in [2,21,8].

2 Broadcast Networks of Register Automata

2.1 Syntax and Semantics

We model a distributed network using a graph in which the behaviour of each node is described via an automaton with operations over a finite set of registers. A node can transmit part of its current data to adjacent nodes using broadcast messages. A message carries both a type and a finite tuple of data. Receivers can test/store/ignore the data contained inside a message. We assume that broadcasts and receptions are executed without delays (i.e. we simultaneously update the state of sender and receiver nodes).

Actions. Let us first describe the set of actions. We use $r \geq 0$ to denote the number of registers in each node. We use $f \geq 0$ to denote the number of data fields available in each message and we consider a finite alphabet Σ of message types. We often use $[i..j]$ to denote the set $\{k \in \mathbb{N} \mid i \leq k \leq j\}$. We also assume that if $r = 0$ then $f = 0$ (no registers, no information to transmit). The set of broadcast actions parameterized by r, f and Σ is defined follows:

$$Send_{\Sigma}^{r,f} = \{\mathbf{b}(m, p_1, \ldots, p_f) \mid m \in \Sigma \text{ and } p_i \in [1..r] \text{ for } i \in [1..f]\}$$

The action $\mathbf{b}(a, p_1, \ldots, p_f)$ corresponds to a broadcast message of type a whose i-th field contains the value of the register p_i of the sending node. For instance, for $r = 2$ and $f = 4$, $\mathbf{b}(req, 1, 1, 2, 1)$ corresponds to a message of type req in which the current value of the register 1 of the sender is copied in the first two fields and in the last field, and the current value of register 2 of the sender is copied into the third field.

A receiver node can then either compare the value of a message field against the current value of a register, store the value of a message field in a register, or simply ignore a message field. Reception actions parameterized by r, f and Σ

are defined as follows:

$$Rec_{\Sigma}^{r,f} = \left\{ \mathbf{r}(m, \alpha_1, \ldots, \alpha_f) \;\middle|\; \begin{array}{l} m \in \Sigma,\ \alpha_i \in Act^r \text{ for } i \in [1..f] \\ \text{and if } \alpha_i = \alpha_k = \downarrow j \text{ then } i = k \end{array} \right\}$$

where the set of field actions Act^r is: $\{?k, ?\overline{k}, \downarrow k, * \mid k \in [1..r]\}$. When used in a given position of a reception action, $?k$ [resp. $?\overline{k}$] tests whether the content of the k-th register is equal [resp. different] to the corresponding value of the message, $\downarrow k$ is used to store the corresponding value of the message into the k-th register, and $*$ is used to denote that the corresponding value is ignored.

As an example, for $r = 2$ and $f = 4$, $\mathbf{r}(req, ?\overline{2}, ?1, *, \downarrow 1)$ specifies the reception of a message of type req in which the first field is tested for inequality against the current value of the second register, the second field is tested for equality against the first register, the third field is ignored, and the fourth field is assigned to the first register. We now provide the definition of a protocol that models the behaviour of an individual node.

Definition 1. *A (r, f)-*protocol *over Σ is a tuple $\mathcal{P} = \langle Q, R, q_0 \rangle$ where: Q is a finite set of control states, $q_0 \in Q$ is an initial control state, and $R \subseteq Q \times (Send_{\Sigma}^{r,f} \cup Rec_{\Sigma}^{r,f}) \times Q$ is a set of broadcasting and reception rules.*

In the rest of the paper we call a (r, f)-protocol over Σ simply a (r, f)-protocol when the alphabet is clear from the context.

A configuration is a graph in which nodes represent the current state of the corresponding protocol instance running on it (control state and current value of registers) and edges denote communication links. In this paper we assume that the value of registers are naturals. Therefore, a valuation of registers is defined as a map from register positions to naturals. More formally, a configuration γ of a (r, f)-protocol $\mathcal{P} = \langle Q, R, q_0 \rangle$ is an undirected graph $\langle V, E, L \rangle$ such that V is a finite set of nodes, $E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$ is a set of edges, and $L : V \to Q \times \mathbb{N}^r$ is a labelling function (current valuation of registers).

Before we give the semantics of our model, we introduce some auxiliary notations. Let $\gamma = \langle V, E, L \rangle$ be a configuration. For a node $v \in V$, we denote by $L_Q(v)$ and $L_M(v)$ the first and second projection of $L(v)$. For $u, v \in V$, we write $u \sim_\gamma v$ – or simply $u \sim v$ when γ is clear from the context – the fact that $(u, v) \in E$, i.e. the two nodes are neighbours. Finally, the configuration γ is said to be initial if $L_Q(v) = q_0$ for all $v \in V$ and, for all $u, v \in V$ and all $i, j \in [1..r]$, if $u \neq v$ or $i \neq j$ then $L_M(v)[i] \neq L_M(v)[j]$. In an initial configuration, all the registers of the nodes contain different values. We write Γ [resp. Γ_0] for the set of all [resp. initial] configurations, and Γ^{fc} [resp. Γ_0^{fc}] for the set of configurations [resp. initial configurations] $\langle V, E, L \rangle$ that are fully connected, i.e. such that $E = V \times V \setminus \{(v, v) \mid v \in V\}$. Note that for a given (r, f)-protocol the sets Γ, Γ_0, Γ^{fc}, and Γ_0^{fc} are infinite since we do not impose any restriction on the number of processes present in the graph.

Furthermore, from two nodes u and v of a configuration $\gamma = \langle V, E, L \rangle$ and a broadcast action of the form $\mathbf{b}(m, p_1, \ldots, p_f)$, let $\mathcal{R}(v, u, \mathbf{b}(m, p_1, \ldots, p_f)) \subseteq Q \times \mathbb{N}^r$ be the set of the possible labels that can take u on reception of the corresponding message sent by v, i.e. we have $(q'_r, M) \in \mathcal{R}(v, u, \mathbf{b}(m, p_1, \ldots, p_f))$ if and only

if there exists a receive action of the form $\langle L_Q(u), \mathbf{r}(m, \alpha_1, \ldots, \alpha_f), q'_r \rangle \in R$ verifying the two following conditions:

(1) For all $i \in [1..f]$, if there exists $j \in [1..r]$ s.t. $\alpha_i = ?j$ [resp. $\alpha_i = ?\overline{j}$], then $L_M(u)[j] = L_M(v)[p_i]$ [resp. $L_M(u)[j] \neq L_M(v)[p_i]$];
(2) For all $j \in [1..r]$, if there exists $i \in [1..f]$ such that $\alpha_i = \downarrow j$ then $M[j] = L_M(v)[p_i]$ otherwise $M[j] = L_M(u)[j]$.

Given a (r, f)-protocol $\mathcal{P} = \langle Q, R, q_0 \rangle$, we define a Broadcast Network of Register Automata (BNRA) as the transition system $BNRA(\mathcal{P}) = \langle \Gamma, \Rightarrow, \Gamma_0 \rangle$ where Γ [resp. Γ_0] is the set of all [resp. initial] configurations and $\Rightarrow \subseteq \Gamma \times \Gamma$ is the transition relation. Specifically, for $\gamma = \langle V, E, L \rangle$ and $\gamma' = \langle V', E', L' \rangle \in \Gamma$, we have $\gamma \Rightarrow \gamma'$ if and only if $V = V'$ and one of the following conditions holds:

(Broadcast) $E = E'$ and there exist $v \in V$ and $\langle q, \mathbf{b}(m, p_1, \ldots, p_f), q' \rangle \in R$ such that $L_Q(v) = q$, $L'_Q(v) = q'$ and for all $u \in V \setminus \{v\}$:
- if $u \sim v$ then $L'(u) \in \mathcal{R}(v, u, \mathbf{b}(m, p_1, \ldots, p_f))$, or, $\mathcal{R}(v, u, \mathbf{b}(m, p_1, \ldots, p_f)) = \emptyset$ and $L(u) = L'(u)$;
- if $u \nsim v$, then $L(u) = L'(u)$.

(Reconfiguration) $L = L'$ (no constraint on new edges E').

Reconfiguration steps model dynamic changes of the connection topology, e.g., loss of links and messages or node movement. An internal transition τ can be defined using a broadcast of a special message such that there are no reception rules associated to it. A register $j \in [1..r]$ is said to be read-only if and only if there is no $\langle q, \mathbf{r}(m, \alpha_1, \ldots, \alpha_f), q' \rangle \in R$ and $i \in [1..f]$ such that $\alpha_i = \downarrow j$. Read-only registers can be used as identifiers of the associated nodes.

Given $BNRA(\mathcal{P}) = \langle \Gamma, \Rightarrow, \Gamma_0 \rangle$, we use $\Rightarrow_b$ to denote the restriction of $\Rightarrow$ to broadcast steps only, and $\Rightarrow^*$ [resp. $\Rightarrow_b^*$] to denote the reflexive and transitive closure of $\Rightarrow$ [resp. $\Rightarrow_b$]. Now we define the set of reachable configurations as: $Reach(\mathcal{P}) = \{\gamma' \in \Gamma \mid \exists \gamma \in \Gamma_0 \text{ s.t. } \gamma \Rightarrow^* \gamma'\}$, $Reach^b(\mathcal{P}) = \{\gamma' \in \Gamma \mid \exists \gamma \in \Gamma_0 \text{ s.t. } \gamma \Rightarrow_b^* \gamma'\}$, and $Reach^{fc}(\mathcal{P}) = Reach^b(\mathcal{P}) \cap \Gamma^{fc}$.

2.2 Coverability Problem

Our goal is to decide whether there exists an initial configuration (of any size and topology) from which it is possible to reach a configuration exposing (covered by w.r.t. graph inclusion) a bad pattern. We express bad patterns using reachability queries defined as follows. Let $\mathcal{P} = \langle Q, R, q_0 \rangle$ be a (r, f)-protocol and Z a denumerable set of variables. A reachability query φ for $\mathcal{P}$ is a formula generated by the following grammar:

$$\varphi ::= q(\mathbf{z}) \mid M_i(\mathbf{z}) = M_j(\mathbf{z}') \mid M_i(\mathbf{z}) \neq M_j(\mathbf{z}') \mid \varphi \wedge \varphi$$

where $\mathbf{z}, \mathbf{z}' \in Z$, $q \in Q$ and $i, j \in [1..r]$. We now define the satisfiability relation for such queries. Given a configuration $\gamma = \langle V, E, L \rangle \in \Gamma$, a valuation is a function $f : Z \mapsto V$. The satisfaction relation $\models$ is parameterized by a valuation and is defined inductively as follows:

- $\gamma \models_f q(\mathbf{z})$ if and only if $L_Q(f(\mathbf{z})) = q$,
- $\gamma \models_f M_i(\mathbf{z}) = M_j(\mathbf{z}')$ if and only if $L_M(f(\mathbf{z}))[i] = L_M(f(\mathbf{z}'))[j]$,
- $\gamma \models_f M_i(\mathbf{z}) \neq M_j(\mathbf{z}')$ if and only if $L_M(f(\mathbf{z}))[i] \neq L_M(f(\mathbf{z}'))[j]$,
- $\gamma \models_f \varphi \wedge \varphi'$ if and only if $\gamma \models_f \varphi$ and $\gamma \models_f \varphi'$.

We say that a configuration γ satisfies a reachability query φ, denoted by $\gamma \models \varphi$ if and only if there exists a valuation f such that $\gamma \models_f \varphi$. Furthermore we assume that our queries do not contain contradictions w.r.t. $=$ and $\neq$. We now define the parameterized verification problem, i.e., finding an initial configuration that leads to a configuration containing a sub-configuration that matches the query.

Definition 2. *The problem $Cov(r, f)$ is defined as follows: given a (r, f)-protocol $\mathcal{P}$ and a reachability query φ, does there exist $\gamma \in Reach(\mathcal{P})$ such that $\gamma \models \varphi$?*

The problem $Cov^b(r, f)$ [resp. $Cov^{fc}(r, f)$] is obtained by replacing the reachability set with $Reach^b(\mathcal{P})$ [resp. $Reach^{fc}(\mathcal{P})$]. Finally, $Cov(*, f)$ denotes the disjunction of the problems $Cov(r, f)$ varying on $r \geq 0$ (i.e. for any (finite) number of registers).

3 An Example: Route Discovery Protocol

Consider the problem of building a route from nodes of type *sender* to nodes of type *dest*. We assume that nodes have two registers, called *id* and *next*, used to store a pointer to the next node in the route to *dest*. The protocol that collects such information is defined in Figure 1. Initially nodes have type *sender*, *idle*, and *dest*. Request messages like *rreq* are used to query adjacent nodes in search

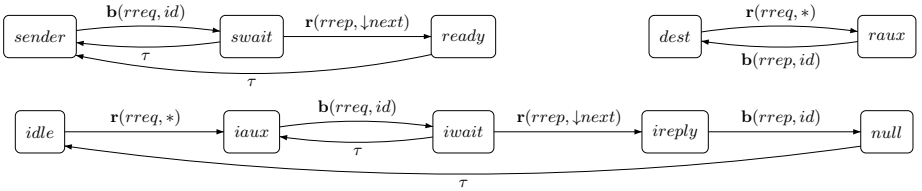


Fig. 1. Route discovery example

for a valid neighbour. Back edges are used to restart the protocol in case of loss of intermediate messages or no reply at all.

In this example an undesired state is, e.g., any configuration in which two adjacent nodes n and n' point to each other. Bad patterns like this one can be specified using a query like $ready(z_1) \wedge ready(z_2) \wedge M_{id}(z_1) = M_{next}(z_2) \wedge M_{next}(z_1) = M_{id}(z_2)$.

4 Reconfiguration in Arbitrary Graphs

4.1 Undecidability of $Cov(2, 2)$

Our first result is the undecidability of coverability for nodes with two registers (one read-only) and messages with two data fields. The proof is based on a

reduction from reachability in two counter machines. The reduction builds upon an election protocol that can be applied to select a linked list (of arbitrary length) of nodes in the network. The existence of such a list-builder protocol is at the core of the proof. The simulation of a two counter machine becomes easy once a list has been constructed. We assume that protocols have at least one read-only register $id \in [1..r]$. We formalize next the notion of list and list-builder that we use in the undecidability proofs presented across the paper. We first say that a node v points to a node v' via x if the register x of v contains the same value as register id of v'. For $q_a, q_b, q_c \in Q$, a list (linked via x) is a set of nodes $\{v_1, \cdots, v_k\}$ such that v_1 has label q_a, v_k has label q_c, v_i has label q_b for $i \in [2..k-1]$, and v_j is the unique node in V that points to v_{j+1} via x and has label in $\{q_a, q_b\}$ for $j \in [0..k-1]$. In other words q_a and q_c are sentinels for a list made of q_b elements. A backward list is defined as before but with reversed pointers, i.e., v_{j+1} points to v_j.

Definition 3. *A protocol $\mathcal{P} = \langle Q, R, q_0 \rangle$ with $\{q_a, q_b, q_c\} \subseteq Q$ is a list-builder for q_a, q_b, and q_c on $x \in [1..r]$ if, for any γ such that $\gamma \in Reach(\mathcal{P})$, if a node v in γ has label q_a, then v is the first node of a list linked via x.*

A backward list-builder is defined in a similar way for backward lists.

Lemma 1. *For $r \geq 2$ and $f \geq 1$, $Cov(r, f)$ is undecidable if there exists a list-builder (r, f)-protocol on $x \in [1..r]$ that can generate lists of any finite length.*

The proof exploits the list (of arbitrary length) generated by a list-builder protocol to build a simulation of a two counter machine. Indeed, notice that if node v is the only one pointing to node v' then the pair of actions $\mathbf{b}(m, x)$ and $\mathbf{r}(m, ?id)$ can be used to send a message from v to v' (v' is the only node that can receive m from v). Furthermore, the pair of actions $\mathbf{b}(m, id)$ and $\mathbf{r}(m, ?x)$ can be used to send a message from v' to v (v is the only node that can receive m from v'). This property can be exploited to simulate counters by using intermediate nodes as single units (the value of the counter is the sum of unit nodes in the list). One of the sentinels is used as program location, and the links in the list are used to send messages (in two directions) to adjacent nodes to increment or decrement (update of labels) the counters. Test for zero is encoded by a double traversal of the list in order to check that each intermediate node represents zero units. The details of the protocol that extends a list-builder are given in [9]. A similar result can be stated for backward list-builders.

The previous lemma tells us that to prove undecidability of coverability we just have to exhibit a list-builder protocol. In the case of $Cov(2, 2)$, we apply Lemma 1, by showing that protocol $\mathcal{P}_{lb}$ of Figure 2 is a backward list-builder for q_h, q_z, and q_t on $x \in [1..r]$. The rationale is as follows. Lists $\{v_1, \cdots, v_k\}$ are built one node at a time, starting from the tail v_k, in state q_t. The links point from each node to the previous one, up to the head v_1, in state q_h. Any node in the initial state q_0 (e.g., v_1) may decide to become a tail by starting to build its own list. Every such construction activity, however, is guaranteed not to interfere in any way with the others, thanks to point to point communication between nodes simulated on top of network reconfigurations and broadcast by exploiting

the two payload fields. This is achieved via a three-way handshake where the first and second fields respectively identify the sender and the recipient. When the sub-protocol is done, v_1 moves to state q_t, v_2 moves to the intermediate state q_i, and one points to the other. Node v_2 decides whether to stop building the list by becoming the head q_h, or to continue by executing another handshake to elect node v_3. The process continues until some v_k finally ends the construction by moving to state q_h. The following theorem then holds (the proof can be found in [9]).

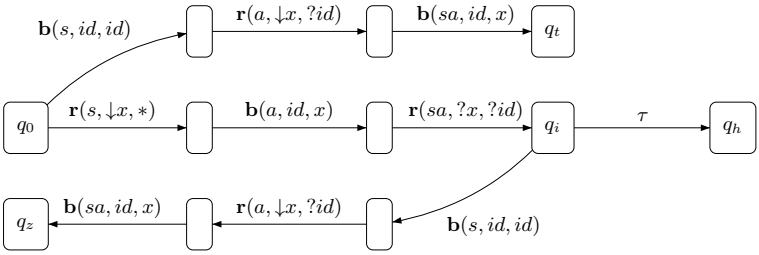


Fig. 2. $\mathcal{P}_{lb}$: backward list-builder for q_h, q_z, q_t, and Γ_0 on x

Theorem 1. *$Cov(2, 2)$ is undecidable even when restricting one register to be read-only.*

4.2 Decidability of $Cov(*, 1)$

In this section, we will prove that $Cov(*, 1)$, i.e. the restriction of our coverability problem to processes with only one field in the message, is PSPACE-complete.

We obtain PSPACE-hardness through a reduction from the reachability problem for 1-safe Petri nets, which is PSPACE-complete [6]. The detail of this construction is provided in [9].

Proposition 1. *$Cov(*, 1)$ is* PSPACE*-hard.*

We now provide a PSPACE algorithm for solving $Cov(*, 1)$. The algorithm is based on a saturation procedure that computes a symbolic representation of reachable configurations. The representation is built using graphs that keep track of control states that may appear during a protocol execution and of relations between values in their registers. The set of symbolic configurations we consider is finite and each symbolic configuration can be encoded in polynomial space.

Assume a $(r, 1)$-protocol $\mathcal{P} = \langle Q, R, q_0 \rangle$ over Σ. A symbolic configuration θ for $\mathcal{P}$ is a labelled graph $\langle W, \delta, \lambda \rangle$ where W is a set of nodes, $\delta \subseteq W \times [1..r] \times [1..r] \times W$ is the set of labelled edges and $\lambda : W \mapsto Q \times \{0, 1\}^r$ is a labelling function (as for configurations, we will denote λ_Q [resp. λ_M] the projection of λ to its first [resp. second] component) such that the following rules are respected:

- For $w, w' \in W$, $w \neq w'$ implies $\lambda_Q(w) \neq \lambda_Q(w')$, i.e. there cannot be two nodes with the same control state;

- If $(w, a, b, w') \in \delta$ then $\lambda_M(w)[a] = 1$ or $\lambda_M(w')[b] = 1$ (or both);
- For $w \in W$ and $j \in [1..r]$, if $\lambda_M(w)[j] = 1$ then $(w, j, j, w) \in \delta$.

The labels $\{0,1\}^r$ are redundant (they can be derived from edges) but simplify some of the constructions needed in the algorithm. We denote by Θ the set of symbolic configurations for $\mathcal{P}$. Let $\theta = \langle W, \delta, \lambda \rangle$ be a symbolic configuration for $\mathcal{P}$. Then, $\langle V, E, L \rangle \in [\![\theta]\!]$ iff the following conditions are satisfied:

1. For each $v \in V$, there is a node $w \in W$ such that $L_Q(v) = \lambda_Q(w)$, i.e. v and w have the same control state;
2. For each $v \neq v' \in V$, if there exist registers $j, j' \in [1..r]$ s.t. $L_M(v)[j] = L_M(v')[j']$, i.e., two distinct nodes with the same value in a pair of registers, then there exists an edge $(w, j, j', w') \in \delta$ with $\lambda_Q(w) = L_Q(v)$ and $\lambda_Q(w') = L_Q(v')$, i.e. we store possible relations on data in registers using edges in θ;
3. For each $v \in V$, if there exist $j \neq j' \in [1..r]$ s.t. $\lambda_M(v)[j] = \lambda_M(v)[j']$, i.e. a node with the same value in two distinct registers, then there exists a self loop $(w, j, j', w) \in \delta$.

We remark that we do not include any information on the communication links of γ, indeed reconfiguration steps can change the topology in an arbitrary way. We define the initial symbolic configuration $\theta_0 = \langle \{w_0\}, \emptyset, \lambda_0 \rangle$ with $\lambda_0(w_0) = (q_0, \mathbf{0})$. Clearly, we have $[\![\theta_0]\!] = \Gamma_0$, i.e. the set of concrete configurations represented by θ_0 is the set of initial configurations of the protocol $\mathcal{P}$. In order to perform a symbolic reachability on symbolic configurations, we use a operator $\texttt{POST}_{\mathcal{P}}$ that, by working on a graph θ simulates the effect of the application of a broadcast rule on its instances $[\![\theta]\!]$. The formal definition of the $\texttt{POST}_{\mathcal{P}}$ operation is given in [9]. We illustrate the key points underlying its definition with the help of an example. Consider the symbolic configurations θ_1 and θ_2 in Figure 3, where we represent edges $(w, a, b, w') \in \delta$ with arrows from w to w' labelled by a, b. Please note that, even though we use directed edges for the graphical representation, the relation between nodes in W symmetrical as $(w, a, b, w') \in \delta$ is equivalent to (w', b, a, w). θ_1 denotes configurations with any number of nodes with label q_0 or q_1. Nodes in state q_0 must have registers containing distinct data (label $0, 0$). Nodes in state q_1 may have the same value in their second register (label $0, 1$ is equivalent to edge $\langle q_1, 2, 2, q_1 \rangle$), that in turn may be equal to the value

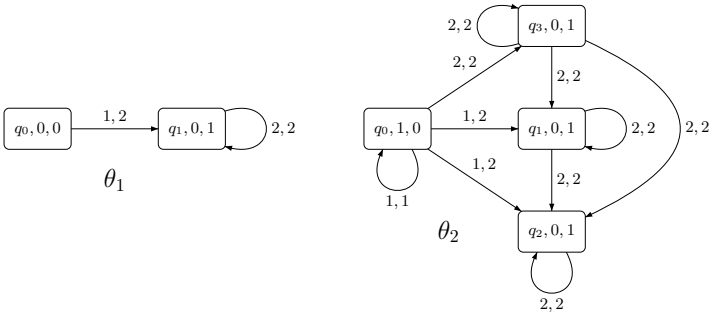


Fig. 3. Example of computations of symbolic post

of the first register in a node labelled q_0 (edge $\langle q_0, 1, 2, q_1\rangle$). θ_1 can be obtained from the initial symbolic configuration by applying rules like $\langle q_0, \mathbf{b}(\alpha, 1), q_0\rangle$ and $\langle q_0, \mathbf{r}(\alpha, \downarrow 2), q_1\rangle$. Indeed, in q_0 we can send the value of the first register to other nodes in q_0 that can then move to q_1 and store the data in the second register (i.e. we create a potential data relation between the first and second register).

We now give examples of rules that can generate the symbolic configuration θ_2 starting from θ_1. The pair $\langle q_0, \mathbf{b}(\beta, 1), q_0\rangle$ and $\langle q_0, \mathbf{r}(\beta, \downarrow 1), q_0\rangle$ generates a new data relation between nodes in state q_0 modelled by changing from 0 to 1 the value of $\lambda_M(q_0)[1]$. We remark that a label 1 only says that registers in distinct nodes may be (but not necessarily) equal.

Consider now the reception rule $\langle q_1, \mathbf{r}(\beta, ?2), q_2\rangle$ for the same message β. The data relation between nodes in state q_0 and q_1 in θ_1 tells us that the rule is fireable. To model its effect we need to create a new node with label q_2 with data relations between registers expressed by the edges between labels q_0, q_1 and q_2 in the figure. Due to possible reconfigurations, not all nodes in q_1 necessarily react, i.e. θ_2 contains the denotations of θ_1.

A rule like $\langle q_1, \mathbf{r}(\beta, ?\overline{2}), q_3\rangle$ can also be fireable from instances of θ_1. Indeed, the message β can be sent by a node in state q_0 that does not satisfy the data relation specified by the edge $(1, 2)$ in θ_1, i.e., the sending node is not the one having the same value in its first register as the node q_1 reacting to the message, hence the guard $?\overline{2}$ could also be satisfied. This leads to a new node with state q_3 which inherits from q_1 the constraints on the first register, but whose second register can have the same value as the second register of nodes in any state.

We now define how to evaluate a reachability query over a symbolic configuration . Let $\theta = \langle W, \delta, \lambda\rangle$ be a symbolic configuration and φ be a reachability query. We denote by $\mathit{Vars}(\varphi)$ the subset of variables used in the query φ and we assume that $\varphi = \bigwedge_{k\in[1..m]} \varphi_k$ where for each $k \in [1..m]$, φ_k is of the form $q(\mathbf{z})$ or $M_i(\mathbf{z}) = M_j(\mathbf{z}')$ or $M_i(\mathbf{z}) \neq M_j(\mathbf{z}')$. We will then say that $\theta \models \varphi$ if there exists a function $g : \mathit{Vars}(\varphi) \mapsto W$ such that for all $k \in [1..m]$ we have the following properties: if $\varphi_k = q(\mathbf{z})$, then $\lambda_Q(g(\mathbf{z})) = q$; if $\varphi_k = (M_i(\mathbf{z}) = M_j(\mathbf{z}'))$ with $\mathbf{z} \neq \mathbf{z}'$ or $i \neq j$, then $(g(\mathbf{z}), i, j, g(\mathbf{z}')) \in \delta$. We have then the following lemma.

Lemma 2. *Given a symbolic configuration θ and a reachability query φ, we have $\theta \models \varphi$ if and only there exists $\gamma \in [\![\theta]\!]$ such that $\gamma \models \varphi$.*

Before giving the properties of the $\mathtt{POST}_{\mathcal{P}}$ operator, we introduce some notations. First we introduce an order on symbolic configurations. Given two symbolic configurations $\theta = \langle W, \delta, \lambda\rangle$ and $\theta' = \langle W', \delta', \lambda'\rangle$, we say that $\theta \sqsubseteq \theta'$ if and only if there exists an injective function $h : W \mapsto W'$ such that for all $w, w' \in W$:

- $\lambda_Q(w) = \lambda'_Q(h(w))$;
- for all $j \in [1..r]$, if $\lambda_M(w)[j] = 1$ then $\lambda'_M(h(w))[j] = 1$;
- if $(w, a, b, w') \in \delta$ then $(h(w), a, b, h(w')) \in \delta'$.

In other words, we have $\theta \sqsubseteq \theta'$ if there are more nodes in θ' than in θ and all the labels of θ appears in θ' as well, and for what concerns the symbolic register valuation, the one of θ' should "cover" the one of θ. One can easily prove the following result.

Lemma 3. *(1) If $\theta \sqsubseteq \theta'$ then $[\![\theta]\!] \subseteq [\![\theta']\!]$. (2) If there exists an infinite increasing sequence $\theta_0 \sqsubseteq \theta_1 \sqsubseteq \theta_2 \ldots$ then there exists $i \in \mathbb{N}$ s.t. for all $j \geq i$, $\theta_j = \theta_i$.*

Furthermore, given a set of configurations $S \subseteq \Gamma$ of the $(r,1)$-protocol $\mathcal{P} = \langle Q, R, q_0 \rangle$ (with $BNRA(\mathcal{P}) = \langle \Gamma, \Rightarrow, \Gamma_0 \rangle$), we define $\mathtt{post}_{\mathcal{P}}(S) = \{\gamma' \in \Gamma \mid \exists \gamma \in \Gamma \text{ s.t. } \gamma \Rightarrow \gamma'\}$ and $\mathtt{post}^*_{\mathcal{P}}$ is the reflexive and transitive closure of $\mathtt{post}_{\mathcal{P}}$ (and $\mathtt{POST}^*_{\mathcal{P}}$ the reflexive and transitive closure of $\mathtt{POST}_{\mathcal{P}}$). Note that since symbolic configuration generate a single node for each label, repeated application of $\mathtt{POST}_{\mathcal{P}}$ are ensured to terminate. We can now give the properties of the $\mathtt{POST}_{\mathcal{P}}$ operator.

Lemma 4. *Let θ be a symbolic configuration of the protocol $\mathcal{P}$. Then we have $\theta \sqsubseteq \mathtt{POST}_{\mathcal{P}}(\theta)$ and for all reachability query φ, there exists $\gamma \in \mathtt{post}^*_{\mathcal{P}}([\![\theta]\!])$ such that $\gamma \models \varphi$ iff $\mathtt{POST}^*_{\mathcal{P}}(\theta) \models \varphi$.*

We have consequently an algorithm to solve whether there exists $\gamma \in \mathit{Reach}(\mathcal{P}) = \mathtt{post}_{\mathcal{P}}(\Gamma_0)$. In fact it is enough to compute $\mathtt{POST}^*_{\mathcal{P}}(\theta_0)$ and to check whether $\mathtt{POST}^*_{\mathcal{P}}(\theta) \models \varphi$. This computation is feasible thanks to Lemma 3 and thanks to the first point of the previous lemma. Note that each each symbolic configuration of the $(r,1)$-protocol $\mathcal{P}$ is a graph with at most $|Q|$ nodes and at most $|Q|^2 * |r|^2$ edges and hence we need only polynomial space in the size of the protocol $\mathcal{P}$ to compute $\mathtt{POST}^*_{\mathcal{P}}(\theta_0)$. Finally we can check in non-deterministic linear time whether $\mathtt{POST}^*_{\mathcal{P}}(\theta_0) \models \varphi$ (it is enough to guess the function g from $\mathit{Vars}(\varphi)$ to the nodes of $\mathtt{POST}^*_{\mathcal{P}}(\theta_0)$). Using Lemma 2, this gives us a polynomial space procedure to check whether there exists $\gamma \in \mathit{Reach}\mathcal{P}$ such that $\gamma \models \varphi$. Furthermore, thanks to the lower bound given by Proposition 1, we can deduce the exact complexity of coverability for protocols using a single field in their messages.

Theorem 2. *$\mathit{Cov}(*,1)$ is* PSPACE*-complete.*

5 Fully Connected Topologies and No Reconfiguration

5.1 Undecidability of $\mathit{Cov}^{fc}(2,1)$

We now move to coverability in fully connected topologies. In contrast with the results obtained without identifiers in [11] it turns out that, without reconfiguration, coverability is undecidable already in the case of nodes with two registers and one payload field. Following the same line as in Lemma 1, to prove the result it is enough to define a (forward) list-builder protocol. We refer to Lemma 1* as the variation of Lemma 1 obtained considering the relation $\Rightarrow_b$ (see [9]). The protocol builds lists backwards from the tail q_t. At each step, a node v among the ones which are not part of the list broadcasts its identifier to the others (which store the value, thus pointing to v), and moves to q_z (or q_t, if it is the first step) electing itself as the next node in the list. The construction ends when such a node will instead move to q_h and force everyone else to stop. By applying Lemma 1*, the following theorem then holds (a complete proof is in [9]).

Theorem 3. *$\mathit{Cov}^{fc}(2,1)$ is undecidable even when one register is read-only.*

5.2 Decidability of $Cov^{fc}(1,1)$

We now consider the problem $Cov^{fc}(1,1)$, where configurations are fully connected and do not change dynamically, processes have a single register, and each message has a single data field. To show decidability, we employ the theory of well-structured transition systems [1,17] to define an algorithm for backward reachability based on a symbolic representation of infinite set of configurations, namely multisets of multisets of states in Q. In the following we use $[a_1, \ldots, a_k]$ to denote a multiset containing (possibly repeated) occurrences $a_1, \ldots, a_k$ of elements from some fixed domain. For a multiset m, we use $m(q)$ to denote the number of occurrences of q in m.

Let $\mathcal{P} = \langle Q, R, q_0 \rangle$ be a $(1,1)$-protocol. The set Ξ of symbolic configurations contains, for every $k \in \mathbb{N}$, all multisets of the form $\xi = [m_1, \ldots, m_k]$, where m_i for $i \in [1..k]$ is in turn a multiset over Q. Given $\xi = [m_1, \ldots, m_k] \in \Xi$, $\langle V, E, L \rangle \in [\![\xi]\!]$ iff there is a function $f : V \to [1..k]$ such that (1) for every $v, v' \in V$, if $L_M(v) = L_M(v')$ then $f(v) = f(v')$ and (2) for all $i \in [1..k]$ and $q \in Q$, $m_i(q)$ is equal to the number of nodes $v \in V$ s.t. $f(v) = i$ and $L_Q(v) = q$ Intuitively, each m_i is associated to one of the k distinct values of the register (the actual values do not matter), and $m_i(q)$ counts how many nodes in state q have the corresponding value. We now define an ordering over Ξ.

Definition 4. *Given $\xi = [m_1, \ldots, m_k] \in \Xi$ and $\xi' = [m'_1, \ldots, m'_p] \in \Xi$, $\xi \prec \xi'$ iff $k \leq p$ and there exists an injection $h : [1..k] \to [1..p]$ such that for all $i \in [1..k]$ and all $q \in Q$, $m_i(q) \leq m_{h(i)}(q)$, i.e. m_i is included in $m_{h(i)}$.*

The following properties then hold.

Proposition 2. *The ordering $(\xi, \prec)$ over symbolic configurations is a well-quasi ordering (wqo), i.e. for any infinite sequence $\xi_1 \xi_2 \ldots$ there exist $i < j$ s.t. $\xi_i \prec \xi_j$.*

Proposition 3. *Let $pre_{\mathcal{P}}(S) = \{\gamma \mid \gamma \Rightarrow_b \gamma', \gamma' \in S\}$. There exists an algorithm $\texttt{PRE}_{\mathcal{P}}$ taking in input $I \subseteq \Xi$ and returning a set $I' \subseteq \Xi$ s.t. $[\![I']\!] = pre_{\mathcal{P}}([\![I]\!])$.*

The formal definition of the predecessor operator is given in [9], together with an example. Following [4], the algorithm for $\texttt{PRE}_{\mathcal{P}}$ can be used to effectively compute a finite representation of the set of predecessors $pre^*_{\mathcal{P}}([\![Bad]\!])$ for a set of symbolic configurations Bad. The computation iteratively applies PRE until a fixpoint is reached. The termination test is defined using $\prec$. The wqo $\prec$ ensures termination of the computation [1]. The following theorem then holds.

Theorem 4. *$Cov^{fc}(1,1)$ is decidable.*

An alternative proof can be given by resorting to an encoding into coverability in data nets [21]. We present such an encoding in [9].

We consider now the complexity. We observe that, without registers and fields our model boils down to the AHNs of [11]. For fully connected topologies, AHN can simulate reset nets as shown in [12]. Following from the complexity of coverability in reset nets [24], we have the the following theoretical lower bound.

Corollary 1. *$Cov^{fc}(0,0)$ and $Cov^{fc}(1,1)$ are non elementary.*

Transitions/Topology	r	f	Status	Complexity
B+R,G	0	0	DEC	PTIME
	$k \geq 1$	1	DEC	PSPACE
	2	2	UNDEC	–
B,FC	0	0	DEC	NON EL
	1	1	DEC	NON EL
	2	1	UNDEC	–
B,G	0	0	UNDEC	–

Fig. 4. Decidability and complexity boundaries: B=broadcast transitions, R=reconfiguration, FC=fully connected topologies, and G=arbitrary graphs.

6 Conclusions

We have investigated decidability and complexity for coverability in a data-sensitive model of broadcast communication (Figure 4). From a technical point of view, our results can be viewed as a fine grained refinement of those obtained for the case without data. For instance, undecidability follows from constructions similar to those adopted in [11]. They are based on special use of data for building synchronization patterns that can be applied even in fully connected networks. Concerning possible applications, the symbolic algorithm for messages with a single data field can be applied to abstract models of routing protocols like the protocol of Section 3. Finally, as future extensions it would be interesting to study ordered data fields and time-sensitive communication.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE Computer Society (1996)
2. Abdulla, P.A., Delzanno, G., Van Begin, L.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209(3), 248–279 (2011)
3. Abdulla, P.A., Jonsson, B.: Undecidable verification problems for programs with unreliable channels. Inf. Comput. 130(1), 71–90 (1996)
4. Abdulla, P.A., Jonsson, B.: Ensuring completeness of symbolic verification methods for infinite-state systems. Theor. Comput. Sci. 256(1-2), 145–167 (2001)
5. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
6. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. TCS 147(1&2), 117–136 (1995)
7. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. FMSD 23(3), 257–301 (2003)
8. Delzanno, G., Rosa-Velardo, F.: On the coverability and reachability languages of monotonic extensions of petri nets. Theor. Comput. Sci. 467, 12–29 (2013)

9. Delzanno, G., Sangnier, A., Traverso, R.: Parameterized verification of broadcat networks of register automata (technical report) (2013), http://verify.disi.unige.it/publications/
10. Delzanno, G., Sangnier, A., Traverso, R., Zavattaro, G.: On the complexity of parameterized reachability in reconfigurable broadcast networks. In: FSTTCS 2012. LIPIcs, vol. 18, pp. 289–300. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
11. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
12. Delzanno, G., Sangnier, A., Zavattaro, G.: On the power of cliques in the parameterized verification of ad hoc networks. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 441–455. Springer, Heidelberg (2011)
13. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS 1998, pp. 70–80. IEEE Computer Society (1998)
14. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359. IEEE Computer Society (1999)
15. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 173–187. Springer, Heidelberg (2012)
16. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and verification of the LMAC protocol for wireless sensor networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
17. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
18. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
19. Joshi, S., König, B.: Applying the graph minor theorem to the verification of graph transformation systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
20. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. 134(2), 329–363 (1994)
21. Lazic, R., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Inform. 88(3), 251–274 (2008)
22. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
23. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
24. Schnoebelen, P.: Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)
25. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: Query-based model checking of ad hoc network protocols. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 603–619. Springer, Heidelberg (2009)

Monomial Strategies for Concurrent Reachability Games and Other Stochastic Games*

Søren Kristoffer Stiil Frederiksen and Peter Bro Miltersen

Aarhus University

Abstract. We consider two-player zero-sum finite (but infinite-horizon) stochastic games with limiting average payoffs. We define a family of stationary strategies for Player I parameterized by $\varepsilon > 0$ to be *monomial*, if for each state k and each action j of Player I in state k except possibly one action, we have that the probability of playing j in k is given by an expression of the form $c\varepsilon^d$ for some non-negative real number c and some non-negative integer d. We show that for all games, there is a monomial family of stationary strategies that are ε-optimal among stationary strategies. A corollary is that all concurrent reachability games have a monomial family of ε-optimal strategies. This generalizes a classical result of de Alfaro, Henzinger and Kupferman who showed that this is the case for concurrent reachability games where all states have value 0 or 1.

1 Introduction

We consider two-player zero-sum finite (but infinite-horizon) stochastic games G with state set $\{1, 2, \ldots, N\}$ and set of actions $\{1, 2, \ldots, m\}$ available to each of the two players in each state. The reward to Player I when Player I plays i and Player II plays j in state k is denoted a^k_{ij}. Transition probabilites are denoted p^{kl}_{ij}. We assume stopping probabilitites are 0, i.e., for all k, i, j we have $\sum_l p^{kl}_{ij} = 1$. We are interested in games with limiting average (undiscounted) payoffs [8,12], i.e, payoff $\liminf_{T\to\infty}(\sum_{i=0}^{t-1} r_t)/T$ to Player I, where r_t is the reward collected by Player I at stage t. A *stationary strategy* x for a player in a stochastic game is a fixed (time independent) assignment of probabilities to his actions, for each of the states of the game. We let x^k_j denote the probability of playing action j in state k according to stationary strategy x. We denote the set of stationary strategies for Player I (II) by S_{I} (S_{II}). For a state k, the *lower value in stationary strategies* of k, denoted $\underline{v}_k$, is defined as $\sup_{x\in S_{\text{I}}} \inf_{y\in S_{\text{II}}} u_k(x, y)$, where $u_k(x, y)$ is the expected limiting average payoff when stationary strategy x of Player I

* The authors acknowledge support from The Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation and from the Center for re- search in the Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 122–134, 2013.

is played against stationary strategy y of Player II and play starts in state k. Given $\varepsilon > 0$, a stationary strategy x^* for Player I is called *ε-optimal among stationary strategies* if for all states k, we have $\inf_{y \in S_{\mathrm{II}}} u_k(x^*, y) \geq \underline{v}_k - \varepsilon$. Notice that when Player I has fixed his stationary strategy, Player II is just playing a Markov decision process, so he has an optimal positional reponse.

The main purpose of the present paper is to prove that all stochastic games have a family of ε-optimal strategies among stationary strategies of a particular regular kind. We introduce the following definition.

Definition 1. A family of stationary strategies $(x_\varepsilon)_{0<\varepsilon\leq\varepsilon_0}$ for Player I in a stochastic game is called *monomial* if for all states k, and all actions j available to Player I in state k *except possibly one action*, we have that $x^k_{\varepsilon,j}$ is given by a monomial in ε, i.e., an expression of the form $c^k_j \varepsilon^{d^k_j}$, where d^k_j is a non-negative integer and c^k_j is a non-negative real number.

The exception made in the definition for some single action in each state is natural and necessary: The sum of probabilities assigned to the actions in each state must be 1, so without this exception, it is easy to see that a monomial family would have $d^k_j = 0$ for all j, k, i.e., it would be a single strategy rather than a family. Also note that when we specify a monomial family of strategies, we do not have to specify the probability assigned to the "special" action in each state, as it is simply the result of subtracting the sum of the probabilities assigned to the remaining actions from one. We can now state our main theorem:

Theorem 1. *For any game G, there is an $\varepsilon_0 > 0$ and a monomial family of stationary strategies $(x_\varepsilon)_{0<\varepsilon\leq\varepsilon_0}$ for Player I, so that for each $\varepsilon \in (0, \varepsilon_0]$, we have that x_ε is ε-optimal among stationary strategies.*

Discussion of the Main Theorem. A monomial family of strategies can be naturally interpreted as a parameterized strategy where probabilities have well-defined "orders of magnitude", given by the degrees d^k_j. Our main theorem informally states that such "clean" strategies are sufficient for playing stochastic games well, at least if one is restricted to the use of stationary strategies. Our main motivation for the theorem is computational: A monomial family of strategies is a *finite* object, and our theorem makes it possible to *ask* the question of whether a family of ε-optimal strategies parameterized by ε can be efficiently computed for a given game, as the result makes this question well-defined. The existence proof of the present paper is essentially non-constructive and provides no efficient algorithm (although it is possible to derive an inefficient algorithm using standard techniques), so we do not answer the question in this paper. It should also be noted that it is easy to give examples of games with rational rewards and transition probabilties where the coefficients c^k_j cannot be rational numbers, so one has to worry about how to represent those. Fortunately, a straightforward application of the Tarski transfer principle yields that algebraic coefficients suffice, and such a number has a finite representation in the form of a univariate polynomial with rational coefficients and an isolating interval within which the number is the only root of the polynomial.

Our main theorem is particularly natural for classes of stochastic games that are guaranteed to have a *value* in stationary strategies, that is, games for which the lower value $\sup_{x \in S_{\mathrm{I}}} \inf_{y \in S_{\mathrm{II}}} u_k(x, y)$ and the *upper* value $\inf_{y \in S_{\mathrm{II}}} \sup_{x \in S_{\mathrm{I}}} u_k(x, y)$ coincide. A natural subclass of stochastic games with this property is Everett's *recursive* games [6]. In a recursive game, all non-zero rewards occur at absorbing states: states k with only one action "1" available to each player and $p_{1,1}^{kk} = 1$ ("terminal states"). Everett presents several examples of families of ε-optimal strategies for natural recursive games and upon inspection, we note that they are monomial. An interesting subclass of recursive games widely studied in the computer science literature [5,3,11,9] is the class of *concurrent reachability games*. In a concurrent reachability game, Player I is trying to reach a distinguished "goal" state and Player II is trying to prevent him from reaching this state. To view such a game as a recursive game, we simply interpret the goal state as an absorbing state g with reward $r_{1,1}^{g} = 1$. Then, the (lower) value $\underline{v}_k$ of a state k is naturally interpreted as the optimal probability of reaching the goal state from k. De Alfaro, Henzinger and Kupferman [5] presented a polynomial time algorithm for deciding which states in a concurrent reachability game have value 1. Inspecting their proof of correctness, we see that it yields an explicit construction of a monomial family of ε-optimal strategies for Player I if the concurrent reachability games satisfy the (very restrictive) property that each state has value either 0 or 1. Note that even this case requires non-trivial strategies for near optimal play [11]. Also, their polynomial time algorithm can easily be adapted to output this strategy. It is interesting to note that in the computed strategy, all coefficients c_j^k are either 0 or 1.

Discussion of the Proof. Our proof relies heavily on semi-algebraic geometry. In this respect, the proof technique is much in line with classical works on stochastic games, in particular the work of Bewley and Kohlberg [1], and semi-algebraic geometry has seen several uses in stochastic games, see for example [13,4,15,10]. Our proof can be outlined as follows. First, we show that it is possible in first order logic over the reals to uniquely define a particular distinguished ε-optimal strategy among stationary strategies, with ε being a free variable in this definition. Then, standard theorems of semi-algebraic geometry imply that there is a family of ε-optimal strategies the probabilities of which can be described as Pusieux series in the parameter $\varepsilon > 0$. We then "round" these series to their most significant terms and finally massage them into monomials. To argue that ε-optimality is not lost in the process, we appeal to theorems upper bounding the sensitivity of the limiting average values of Markov chains to perturbations of their transition probabilities. These sensitivity theorems are due to Solan [14], building on work on Freidlin and Wentzell [7]. As our main theorem is very simply stated, one might speculate that it has an elementary proof, avoiding the use of semi-algebraic geometry. However, we are not aware of any such proof, even for the case of concurrent reachability games. It should be noted that the proof by De Alfaro, Henzinger and Kupferman is combinatorial in nature, and does not rely on semi-algebraic geometry, so at least for the simpler case considered by them, elementary arguments do exist.

Organization of Paper. In section 2 we will introduce the definitions, lemmas and previous results necessary for the proof. In section 3 we prove a version of the main theorem with monomials replacing Puiseux series. In section 4 we prove the actual main theorem.

2 Preliminaries

For $n \in \mathbb{N}$, let $[n]$ denote $\{1, \ldots, n\}$. A *Puiseux series* p over some indeterminate T and field $\mathbb{F}$ is an expression of the form $p = \sum_{i=K}^{\infty} a_i T^{\frac{i}{M}}$ where $K \in \mathbb{Z}, M \in \mathbb{N}$, and for all i, $a_i \in \mathbb{F}$, with the expression satisfying that if $p \neq 0$ then there $\exists i \in \mathbb{Z} : a_i \neq 0 \wedge \gcd(i, M) = 1$. Similarly, a function $p : \mathbb{R} \to \mathbb{R}$ is a *Puiseux function* on an interval I, if there exists $K \in \mathbb{Z}, M \in \mathbb{N}, a_i \in \mathbb{R}$ such that $p(\epsilon) = \sum_{i=K}^{\infty} a_i \epsilon^{\frac{i}{M}}$ for all $\epsilon \in I$. In the context of this paper we will only look at Puiseux functions, and we will often call the function $p(\epsilon)$ a Puiseux series. The *order* of a Puiseux series $p = \sum_{i=K}^{\infty} a_i T^{\frac{i}{M}}$ is the smallest integer i such that $a_i \neq 0$, and we will write $\mathrm{ord}(p) = i$. If $p = 0$ then the order is defined to be ∞. The proofs of the following elementary lemmas on Puiseux series are easy and we omit them.

Lemma 1. *if $q(\epsilon) = \sum_{i=K}^{\infty} c_i \epsilon^{\frac{i}{M}}$ is a Puiseux series that is convergent and bounded on some $(0, \epsilon_0)$, then $c_i = 0$ for all $i < 0$. In other words, the order of q is greater than or equal to 0.*

Lemma 2. *For any Puiseux series $q(\epsilon) = \sum_{i=K}^{\infty} c_i \epsilon^{\frac{i}{M}}$ with $ord(q) = K \geq 0$ there exists an ϵ_0 such that $sign(q(\epsilon)) = sign(c_K)$ for all $\epsilon \in (0, \epsilon_0)$.*

A *semi-algebraic set* is a subset of real Euclidean space defined by a finite set of polynomial equalities and inequalities. The well-known *Tarsi-Seidenberg theorem* states that any set that can be defined in the language of first order arithmetic is semi-algebraic. We will use this theorem throughout this paper to establish that sets are semi-algebraic. A *semi-algebraic function* is a real-valued function whose graph is a semi-algebraic set. We shall use the following lemma, establishing a close relationship between semi-algebraic functions and Puiseux functions.

Lemma 3. *[13, lemma 6.2] Let $a > 0$, if $f : (0, a) \to \mathbb{R}$ is a semi-algebraic function, then there exists an $0 < \epsilon' < a$ such that f is a Puiseux function on $(0, \epsilon')$.*

For stochastic games, we use the notation introduced in the introduction. We shall use the following theorem, due to Solan, as an important lemma. The theorem applies to *1-player* stochastic games (a.k.a., Markov decision processes). In a 1-player stochastic game, Player 2 has only a single action in each state. We therefore write p_i^{kl} rather than p_{ij}^{kl} for the transition probabilities.

Theorem 2. *[14, theorem 6] Let G and $\tilde{G}$ be 1-player stochastic games with identical state set $\{1, 2, \ldots, N\}$, transition probabilities p_i^{kl}, $\tilde{p}_i^{kl}$ and identical*

rewards. Let c be an upper bound on the absolute value of all rewards. Let $\underline{v}$, $\underline{\tilde{v}}$ be the lower value in stationary strategies in each of the games. Let $\delta \in (0, \frac{1}{2N})$ satisfy $\max_{i,k,l}(\frac{p_i^{kl}}{\tilde{p}_i^{kl}}, \frac{\tilde{p}_i^{kl}}{p_i^{kl}}) - 1 \leq \delta$, where $\frac{x}{0} := \infty, \frac{0}{0} := 1$. Then, $|\underline{v} - \underline{\tilde{v}}| \leq 4cN\delta$.

3 Puiseux Family of Strategies

Lemma 4. *For any game G there exists an ϵ_0 and a family of stationary strategies $(x_\epsilon)_{0<\epsilon\leq\epsilon_0}$ that are ϵ-optimal among stationary strategies, where for all states k and all actions j, $x^k_{\epsilon,j}$ is given by a Puiseux series in ϵ, that is, there is an expression $q^k_j(\epsilon) = \sum_{i=K^k_j}^{\infty} c^k_{i,j}\epsilon^{\frac{i}{M^k_j}}$ such that $x^k_{\epsilon,j} = q^k_j(\epsilon)$ for $\epsilon \in (0, \epsilon_0]$.*

Proof. We want to create a first-order formula $\Phi^k_j(x, \epsilon)$ for every state k and every action j, which is true if and only if x is the probability that Player I should play action j in state k in a specific strategy that is ϵ-optimal among stationary strategies. Then, since we have described the function by a first-order formula, it is semi-algebraic, and by Lemma 3 we get that there exists a Puiseux series that is equal to the function, thus completing the proof. We are going to use several smaller first-order formulas to describe the formulas $\Phi^k_j(x, \epsilon)$.

To ease notation, during the proof k, l will only be refering to states in the game, so they will be numbers $k, l \in [N]$. i, j will be refering to actions in a given state, so they will be numbers $i, j \in [m]$. We will also use the following vectors

$$\boldsymbol{x} := (x_i^k)_{i\in[m]}^{k\in[N]} \quad , \quad \boldsymbol{y} := (y_i^k)_{i\in[m]}^{k\in[N]} \quad , \quad \boldsymbol{v} := (v^k)^{k\in[N]} \quad , \quad \boldsymbol{\nu} := (\nu^k)^{k\in[N]}$$

$\boldsymbol{x}$ and $\boldsymbol{y}$ will represent the strategies of Player I and Player II respectively, while $\boldsymbol{v}$ and $\boldsymbol{\nu}$ will be used to represent different values of stationary strategies of the game starting in each position.

The first two formulas $\Delta_\alpha(\boldsymbol{x})$, $\Delta_\beta(\boldsymbol{y})$ describe that $\boldsymbol{x}$ is a stationary strategy and $\boldsymbol{y}$ is a stationary strategy respectively.

$$\Delta_\alpha(\boldsymbol{x}) := \bigwedge_{k\in[N],i\in[m]} \left[x_i^k \geq 0\right] \wedge \bigwedge_{k\in[m]} \left[\sum_{i\in[m]} x_i^k = 1\right]$$

$$\Delta_\beta(\boldsymbol{y}) := \bigwedge_{k\in[N],i\in[m]} \left[y_i^k \geq 0\right] \wedge \bigwedge_{k\in[N]} \left[\sum_{i\in[m]} y_i^k = 1\right]$$

Next we want to create a first-order formula $\Psi(\boldsymbol{v})$ which expresses that v^k is the lower value in stationary strategies when the game starts in state k, that is, the quantity:

$$\sup_{\boldsymbol{x}\in S_I} \inf_{\boldsymbol{y}\in S_{II}} \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \liminf_{T\to\infty} \sum_{t=0}^{T-1} \frac{r_t}{T}$$

We can rewrite this quantity by using the following equations proved in [2, Theorem 5.2]

$$\inf_{\boldsymbol{y}\in S_{II}} \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \liminf_{T\to\infty} \sum_{t=0}^{T-1} \frac{r_t}{T} = \inf_{\boldsymbol{y}\in S_{II}} \liminf_{\lambda\to 0} \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \frac{\lambda}{1+\lambda} \sum_{t=0}^{\infty} \frac{1}{(1+\lambda)^t} r_t \quad , \forall \boldsymbol{x} \in S_I$$

So the suprema over the two sets are the same, and we can express the value by creating a formula which express that

$$v^k = \sup_{\boldsymbol{x}\in S_I} \inf_{\boldsymbol{y}\in S_{II}} \liminf_{\lambda\to 0} \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \frac{\lambda}{1+\lambda} \sum_{t=0}^{\infty} \frac{1}{(1+\lambda)^t} r_t \quad \forall k \in [N]$$

A common way of rewriting these value equations is by expanding the expectations for one state and substituting v^l into the equations

$$v^k = \sup_{\boldsymbol{x}\in S_I} \inf_{\boldsymbol{y}\in S_{II}} \liminf_{\lambda\to 0} \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \frac{\lambda}{1+\lambda} \sum_{t=0}^{\infty} \frac{1}{(1+\lambda)^t} r_t \quad \forall k \in [N]$$

$$\Leftrightarrow v^k = \sup_{\boldsymbol{x}\in S_I} \inf_{\boldsymbol{y}\in S_{II}} \liminf_{\lambda\to 0} \frac{\lambda}{1+\lambda} \sum_{i,j\in[m]} x_i^k y_j^k \left(a_{ij}^k + \sum_{l\in[N]} p_{ij}^{kl} \frac{1}{\lambda} v^l \right) \quad \forall k \in [N]$$

First notice that for any semi-algebraic sets A and B, and any function $f : A \to B$ where there is a formula $\Pi(a, b)$ that is true if and only is $f(a) = b$, we can express the supremum $\sup_{a\in A} f(a)$ in the following way

$$\begin{aligned} \Pi_{sup}(s) := &[\forall a \in A \exists b \in B : \Pi(a, b) \wedge s \geq b] \\ &\wedge [\forall \epsilon > 0 \exists a \in A \exists b \in B : \Pi(a, b) \wedge s < b + \epsilon] \end{aligned}$$

And similar formulas can be created for the infimum and the limit, and since $\liminf_{\lambda\to 0} f(\lambda)$ is $\lim_{\lambda'\to 0} \inf_{0<\lambda<\lambda'} f(\lambda)$, we only need to create a formula for the inner part:

$$\frac{\lambda}{1+\lambda} \sum_{i,j\in[m]} x_i^k y_j^k \left(a_{ij}^k + \sum_{l\in[N]} p_{ij}^{kl} \frac{1}{\lambda} v^l \right)$$

We then create the formula

$$\Pi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\nu}, \lambda) := \bigwedge_{k\in[N]} \left[\nu^k = \frac{\lambda}{1+\lambda} \sum_{i,j\in[m]} x_i^k y_j^k \left(a_{ij}^k + \sum_{l\in[N]} p_{ij}^{kl} \frac{1}{\lambda} \nu^l \right) \right]$$

Since $S_I = \{\boldsymbol{x} \in \mathbb{R}^{Nm} | \Delta_\alpha(\boldsymbol{x})\}$, we have that S_I, S_{II} are semi-algebraic. Then from the previous argument we can create a formula $\Pi_{sup}(\boldsymbol{v})$ for the lower value in stationary strategies. Also, by not removing the last supremum, we can create a formula $\Xi(\boldsymbol{x}, \boldsymbol{v})$ that is true if the value of Player I playing strategy $\boldsymbol{x}$ is $\boldsymbol{v}$.

It is now straightforward to create a formula $\Upsilon(\boldsymbol{x}, \epsilon)$ that is true if and only if $\boldsymbol{x}$ is a stationary strategy that is ϵ-optimal among stationary strategies.

$$\Upsilon(\boldsymbol{x},\epsilon) := \exists \boldsymbol{v} \in \mathbb{R}^N \exists \boldsymbol{\nu} \in \mathbb{R}^N : \Lambda_\alpha(\boldsymbol{x}) \wedge (0 < \epsilon < 1) \\ \wedge \Pi_{sup}(\boldsymbol{v}) \wedge \Xi(\boldsymbol{x},\boldsymbol{\nu}) \bigwedge_{k\in[N]} \left[\nu^k \geq v^k - \epsilon\right]$$

Now to create $\Phi_j^k(x,\epsilon)$, we need to select a unique strategy from the set of stationary strategies that are ϵ-optimal among stationary strategies. Let $\varphi : [N]\times[m] \to [Nm]$ be some bijection, which we will use to get an ordering on the pairs consisting of an action i and a state k. Using this we can write a strategy as $(x_\iota)_{\iota\in[Nm]}$. We define formulas $P_\iota(x_1,\ldots,x_\iota,\epsilon)$ for $\iota \in [Nm]$ which are true if there exists a strategy that is ϵ-optimal among stationary strategies and the first ι entries are $(x_1,\ldots,x_\iota)$.

$$P_\iota(x_1,\ldots,x_\iota,\epsilon) := \exists x_{\iota+1},\ldots,x_{Nm} \in \mathbb{R} : \Upsilon(x_1,\ldots,x_\iota,x_{\iota+1},\ldots,x_{Nm},\epsilon)$$

Notice that for each $\iota \in [Nm]$, if we assume that we have chosen $x_1,\ldots,x_{\iota-1}$ such that $P_{\iota-1}(x_1,\ldots,x_{\iota-1},\epsilon)$ is true, then the set $\{x \in \mathbb{R} | P_\iota(x_1,\ldots,x_{\iota-1},x,\epsilon)\}$ is non-empty. From the Tarski-Seidenberg theorem the set is semi-algebraic, so it is defined by a finite set of polynomial equalities and inequalities. This implies that the set must consist of a finite set of intervals[1], so we can choose a unique strategy by the middle of the interval which lower endpoint is closest to 0. Using this observation, we can now create a new series of formulas $\Psi_\iota(x_1,\ldots,x_{\iota-1},x,\epsilon)$ for $\iota \in [Nm]$ which given that $P_{\iota-1}(x_1,\ldots,x_{\iota-1},\epsilon)$ is true, x is the middlepoint of the interval with the lower endpoint closest to 0 among the intervals in the set $\{x \in \mathbb{R} | P_\iota(x_1,\ldots,x_{\iota-1},x,\epsilon)\}$.

$$\begin{aligned}\Psi_\iota(x_1,\ldots,x_{\iota-1},x,\epsilon) := \exists x_{\iota+1},\ldots,x_{Nm},a,b \in \mathbb{R} : a \leq b \wedge x = \frac{a+b}{2} : \\ \Upsilon(x_1,\ldots,x_{\iota-1},x,x_{\iota+1},\ldots,x_{Nm},\epsilon) \\ \wedge [P_\iota(x_1,\ldots,x_{\iota-1},a,\epsilon) \vee (a < b \wedge \forall y \in (a,b) : P_\iota(x_1,\ldots,x_{\iota-1},y,\epsilon))] \\ \wedge [\forall y < a : \neg P_\iota(x_1,\ldots,x_{\iota-1},y,\epsilon)] \\ \wedge [\exists \epsilon > 0 \forall y \in (b,b+\epsilon) : \neg P_\iota(x_1,\ldots,x_{\iota-1},y,\epsilon)]\end{aligned}$$

Now to select our unique strategy we will do the following: For each ϵ, pick x_1 to be the middlepoint of the interval with the lower endpoint closest to 0 among the intervals in the set $\{x \in \mathbb{R} | P_\iota(x,\epsilon)\}$, next we pick x_2 to be the middlepoint of the interval with the lower endpoint closest to 0 among the intervals in the set $\{x \in \mathbb{R} | P_\iota(x_1,x,\epsilon)\}$, and so on. We can then recursively define new formulas $\Omega_\iota(x_1,\ldots,x_\iota,\epsilon)$ for $\iota \in [Nm]$ that are true if and only if the unique choice of the first ι indices described by the above procedure is exactly $x_1,\ldots,x_\iota$.

$$\Omega_1(x,\epsilon) := \Psi_1(x,\epsilon) \ , \ \Omega_\iota(x_1,\ldots,x_\iota,\epsilon) := \Omega_{\iota-1}(x_1,\ldots,x_{\iota-1},\epsilon) \wedge \Psi_\iota(x_1,\ldots,x_\iota,\epsilon)$$

Using this we can now immediately create the formulas $\Phi_\iota(x,\epsilon)$ for $\iota \in [Nm]$ in the following way:

$$\Phi_\iota(x,\epsilon) := \exists x_1,\ldots,x_{Nm} \in \mathbb{R} : \Omega_{Nm}(x_1,\ldots,x_{Nm},\epsilon) \wedge x = x_\iota$$

[1] In this terminology we allow for the interval $[a,a]$ and identify it with the point $\{a\}$.

Now we have obtained that each formula $\Phi_\iota(x,\epsilon)$ implicitly defines a semi-algebraic function $x_\iota(\epsilon)$ and due to Lemma 3 we have that there exists Puiseux series $q_\iota(\epsilon)$ and numbers ϵ_ι such that $x_\iota(\epsilon) = q_\iota(\epsilon)$ for $\epsilon \in (0,\epsilon_\iota)$. Now take $\epsilon_0 = \min_{\iota\in[Nm]} \epsilon_\iota$ and we have the lemma.

4 Proof of Main Theorem

The proof will be carried out in two steps. First we will use the family of strategies obtained from Lemma 4 to create a family of strategies only consisting of the first term of the Puiseux series of the original family. Then by using Theorem 2, we prove their value can not be much worse. Then finally we transform this family into a monomial family of strategies that are $\epsilon-$optimal among stationary strategies.

Proof (of Theorem 1). From Lemma 4 we know that there exists an ϵ_1 and a family of stationary strategies $(x_\epsilon)_{0<\epsilon\leq\epsilon_1}$ that are ϵ-optimal among stationary strategies such that $x^k_{\epsilon,j} = q^k_j(\epsilon) = \sum_{i=K^k_j}^{\infty} c^k_{i,j}\epsilon^{\frac{i}{M^k_j}}$ for $\epsilon \in (0,\epsilon_1]$ and for all states k and actions j. Assume without loss of generality that $K^k_j = \mathrm{ord}(q^k_j)$, and observe that K^k_j can be ∞ if the Puiseux series is identically 0. Also observe that since each $x^k_{\epsilon,j}$ is a probability, it is positive and bounded, so by Lemma 1 we know that all $K^k_j \geq 0$.

Now for each k, look at the set of Puiseux series $\{q^k_j(\epsilon)\}_{j\in[m]}$ and let j_k be an index so $q^k_{j_k}(\epsilon)$ is one of the Puiseux series in the set which has minimal order. Observe that $q^k_{j_k}(\epsilon)$ has order 0. To see this, assume for contradiction that $\mathrm{ord}(q^k_j) > 0$ for all actions j, then all of them behave as power series around 0, thus $q^k_j(\epsilon) \to 0$ for $\epsilon \to 0$ so the sum $\sum_{j\in[N]} q^k_j(\epsilon) \to 0$ for $\epsilon \to 0$, which contradicts that $\sum_{j\in[N]} q^k_j(\epsilon) = 1$ for all $\epsilon \in (0,\epsilon_1]$.

Now look at any k again. We want to approximate the family of strategies defined by $q^k_j(\epsilon)$ by a new family of strategies defined by finite Puiseux series $\rho^k_j(\epsilon)$ for $\epsilon \in (0,\epsilon_2]$, where ϵ_2 will be defined later. We define $\rho^k_j(\epsilon)$ as a conditional function on the following sets

$$
\begin{aligned}
S_1 &= \{(k,j) \in [N]\times[m] \mid \mathrm{ord}(q^k_j) = \infty\}\\
S_2 &= \{(k,j) \in [N]\times[m] \mid j \neq j_k \wedge \mathrm{ord}(q^k_j) \neq \infty\}\\
S_3 &= \{(k,j) \in [N]\times[m] \mid j = j_k\}
\end{aligned}
$$

Then $\rho^k_j(\epsilon)$ is defined as follows

$$
\rho^k_j(\epsilon) = \begin{cases} 0 & \text{if } (k,j) \in S_1 \\ c^k_{K^k_j,j}\epsilon^{\frac{K^k_j}{M^k_j}} & \text{if } (k,j) \in S_2 \\ 1 - \sum_{j\in S_2} c^k_{K^k_j,j}\epsilon^{\frac{K^k_j}{M^k_j}} & \text{if } (k,j) \in S_3 \end{cases}
$$

So $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ is the derived family of strategies from $q_j^k(\epsilon)$, defined by $\rho_j^k(\epsilon) \equiv 0$ when $q_j^k(\epsilon) \equiv 0$, and otherwise equal to the first term in $q_j^k(\epsilon)$ except for one action, $q_{j_k}^k(\epsilon)$ which is 1 minus the sum of the other probabilities, to ensure $\rho_j^k(\epsilon)$ is a probability distribution. Since $q_{j_k}^k(\epsilon)$ is a probability, then it is positive, so from Lemma 2 we have that for $(k,j) \in S_2$ the constant is positive. But then we can choose ϵ_2 to be small enough so that for all $(k,j) \in S_2$, $\rho_j^k(\epsilon) \leq 1$. So for each $k \in [N]$, $(\rho_j^k(\epsilon))_{j\in[m]}$ becomes a probability distribution.

We will use Theorem 2 to prove that the value of the game where Player I fixes his strategy to $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$, is not much different than the value of the game where Player I fixes his strategy to $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$. To do this, we must show that for all states k and all actions j, $\rho_j^k(\epsilon)$ is multiplicatively close to $q_j^k(\epsilon)$ in the sense of Theorem 2. We look at the three cases where a pair (k,j) is either in S_1,S_2 and S_3.

For the case $(k,j) \in S_1$, $q_{j_k}^k(\epsilon) = 0 = \rho_j^k(\epsilon)$, so they are trivially close.

Now we look at an arbritrary $(k,j) \in S_2$. To simplify notation we omit the k,j in the notation, and hence $\rho_j^k(\epsilon)$ becomes $\rho(\epsilon) = c_K\epsilon^{\frac{K}{M}}$ and $q_j^k(\epsilon)$ becomes $q(\epsilon) = \sum_{i=K}^{\infty} c_K\epsilon^{\frac{i}{M}}$. We want to show that there exists an ϵ_j^k for this $(k,j) \in S_2$ such that for all $\epsilon \in (0,\epsilon_j^k)$ we have

$$q(\epsilon)\left(1-\epsilon^{\frac{1}{M}}\frac{1+|c_{K+1}|}{c_K}\right) \leq \rho(\epsilon) \leq q(\epsilon)\left(1+\epsilon^{\frac{1}{M}}\frac{1+|c_{K+1}|}{c_K}\right)$$

To see this holds, we look at the difference between the two numbers

$$\begin{aligned} q(\epsilon)\left(1+\epsilon^{\frac{1}{M}}\frac{1+|c_{K+1}|}{c_K}\right) - \rho(\epsilon) &= \sum_{i=K+1}^{\infty} c_i\epsilon^{\frac{i}{M}} + \epsilon^{\frac{1}{M}}\frac{1+|c_{K+1}|}{c_K}\sum_{i=K}^{\infty} c_i\epsilon^{\frac{i}{M}} \\ &= \epsilon^{\frac{K+1}{M}}(c_{K+1} + c_K\frac{1+|c_{K+1}|}{c_K}) + \ldots \end{aligned}$$

So the first term is positive, and Lemma 2 gives us that the series is positive on some area $(0,\epsilon')$. Similarly we can show that $q(\epsilon)\left(1-\epsilon^{\frac{1}{M}}\frac{1+|c_{K+1}|}{c_K}\right) - \rho(\epsilon)$ is negative on some area $(0,\epsilon'')$, so by letting $\epsilon_j^k = \min(\epsilon',\epsilon'')$ we get the desired inequalities. Since this works for an arbritary state k and action j where $(k,j) \in S_2$, we can create similar inequalities that work for all the states and actions in S_2 by defining

$$C := \max_{(k,j)\in S_2} \frac{1+|c^k_{K_j^k+1,j}|}{c^k_{K_j^k,j}} \quad , \quad Q := \min_{(k,j)\in S_2} \frac{1}{M_j^k} \quad , \quad \epsilon_3 := \min_{(k,j)\in S_2} \epsilon_j^k$$

This immediately implies that for all $(k,j) \in S_2$ we get the following multiplicative relation between $q_j^k(\epsilon)$ and $\rho_j^k(\epsilon)$

$$q_j^k(\epsilon)\left(1-\epsilon^Q C\right) \leq \rho_j^k(\epsilon) \leq q_j^k(\epsilon)\left(1+\epsilon^Q C\right) \quad \forall \epsilon \in (0,\epsilon_3)$$

Now we look at $(k,j) \in S_3$. From the observations on S_2 we have that for all $(l,i) \in S_2$, that $\rho_i^l(\epsilon) \geq q_i^l(\epsilon)\left(1-\epsilon^Q C\right)$ for $\epsilon \in (0,\epsilon_3)$. Furthermore since we know that $\sum_{i\in[m]} q_i^k(\epsilon) = 1$, it holds that $q_j^k(\epsilon) = 1 - \sum_{i\in S_2} q_i^k(\epsilon)$. We use these observations to compute the following

$$\begin{aligned}
\rho_j^k(\epsilon) &= 1 - \sum_{i\in S_2} \rho_i^k(\epsilon) \leq 1 - (1-\epsilon^Q C)\sum_{i\in S_2} q_i^k(\epsilon) \\
&= \epsilon^Q C + (1-\epsilon^Q C) - (1-\epsilon^Q C)\sum_{i\in S_2} q_i^k(\epsilon) \\
&= \epsilon^Q C + (1-\epsilon^Q C)(1-\sum_{i\in S_2} q_i^k(\epsilon)) = \epsilon^Q C + (1-\epsilon^Q C) q_j^k(\epsilon) \\
&= q_j^k(\epsilon)\left(\frac{\epsilon^Q C}{q_j^k(\epsilon)} + 1 - \epsilon^Q C\right) \leq q_j^k(\epsilon)\left(\frac{2\epsilon^Q C}{c_{0,j}^k} + 1 - \epsilon^Q C\right)
\end{aligned}$$

The last inequality is conditioned on ϵ being small enough. To see how small ϵ must be, consider the Puiseux series $q_j^k(\epsilon)$. First recall that for $(i,l) \in S_3$, $q_i^l(\epsilon)$ has order 0, so the initial term is just a constant $c_{0,j}^k$, and from Lemma 2 we know that the constant is positive. Now look at the the tail $\sum_{i=1}^{\infty} c_{i,j}^k \epsilon^{\frac{i}{M_j^k}}$ without the first term. The tail is just a fractional power series, so it tends to 0 for $\epsilon \to 0$. This means that for any constant κ, then there exists an ϵ' such that for all $\epsilon < \epsilon'$ the tail is smaller than κ. By using the constant $\frac{c_{0,j}^k}{2}$, we get that $\rho_j^k(\epsilon)$ must be larger than $\frac{c_{0,j}^k}{2}$ when $\epsilon \in (0,\epsilon')$, giving us the inequality for $\epsilon \in (0,\epsilon')$. If we then chose $\epsilon'' = \min(\epsilon', \epsilon_3)$, then all the inequalities of the above computation hold. In the same way, we get that there exists an ϵ''' such that

$$\rho_j^k(\epsilon) \geq q_j^k(\epsilon)\left(\frac{-2\epsilon^Q C}{c_{0,j}^k} + 1 + \epsilon^Q C\right) \quad \forall \epsilon \in (0,\epsilon''')$$

Now let $\epsilon_j^k = \min(\epsilon'', \epsilon''')$, and let $\epsilon_4 = \min_{(j,k)\in S_3} \epsilon_j^k$. We now get that both inequalities hold for all $(k,j) \in S_3$

$$q_j^k(\epsilon)\left(\frac{-2\epsilon^Q C}{c_{0,j}^k} + 1 + \epsilon^Q C\right) \leq \rho_j^k(\epsilon) \leq q_j^k(\epsilon)\left(\frac{2\epsilon^Q C}{c_{0,j}^k} + 1 - \epsilon^Q C\right)$$

Next by defining $c = \min_{(j,k)\in S_3} c_{0,j}^k$, and inverting the signs of $\epsilon^Q C$ in the above inequalities, the bound also covers $(k,j) \in S_2$ as well. But then we have that for all $\epsilon \in (0,\epsilon_4)$ and all $(k,j) \in S_1 \cup S_2 \cup S_3$ that

$$q_j^k(\epsilon)\left(\frac{-2\epsilon^Q C}{c} + 1 - \epsilon^Q C\right) \leq \rho_j^k(\epsilon) \leq q_j^k(\epsilon)\left(\frac{2\epsilon^Q C}{c} + 1 + \epsilon^Q C\right)$$

Notice that $\frac{2\epsilon^Q C}{c} + 1 + \epsilon^Q C = 1 + \epsilon^Q \frac{2C+cC}{c}$. To ease the notation of the upcoming calculations we define

$$lw(\epsilon) := 1 - \epsilon^Q \frac{2C+cC}{c} \quad , \quad up(\epsilon) := 1 + \epsilon^Q \frac{2C+cC}{c}$$

Now we are ready to use Theorem 2 to bound the difference in the value of the two Markov Decision processes that appear when we fix the strategy of Player I to be $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ and $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$.

Since the strategy $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ is $\epsilon-$optimal among stationary strategies, then when Player I fixes its strategy to $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$, Player II can not gain more than ϵ more than $\underline{v}_k + \epsilon$. Similarly we can look at the game where Player I fixes his strategy to $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$. If we can prove that Player II can not gain more than $\underline{v}_k+\gamma$ in this game, then we get that the strategy is $\gamma-$optimal among stationary strategies.

Let $(p_j^{kl}(\epsilon))_{j\in[m]}^{k,l\in[N]}$ be the transition probabilities of the Markov Decision process where we fix the strategy of Player I to be $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$. Similarly, let $(\tilde{p}_j^{kl}(\epsilon))_{j\in[m]}^{k,l\in[N]}$ be the transition probabilities when we fix Player I's strategy to be $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$. Then, we get:

$$\frac{\tilde{p}_j^{kl}(\epsilon)}{p_j^{kl}(\epsilon)} = \frac{\sum_{j\in\{1,\ldots,m\}} \rho_i^k(\epsilon) p_{ij}^{kl}}{\sum_{j\in\{1,\ldots,m\}} q_i^k(\epsilon) p_{ij}^{kl}} \Rightarrow lw(\epsilon) \leq \frac{\tilde{p}_j^{kl}(\epsilon)}{p_j^{kl}(\epsilon)} \leq up(\epsilon)$$

So we have an upper bound on the fraction $\frac{\tilde{p}_j^{kl}(\epsilon)}{p_j^{kl}(\epsilon)}$. To upper bound the fraction $\frac{p_j^{kl}(\epsilon)}{\tilde{p}_j^{kl}(\epsilon)}$, observe that when ϵ is smaller than some ϵ', then $lw(\epsilon), up(\epsilon) > 0$ and we get the following upper bound

$$lw(\epsilon) \leq \frac{\tilde{p}_j^{kl}(\epsilon)}{p_j^{kl}(\epsilon)} \Rightarrow \frac{p_j^{kl}(\epsilon)}{\tilde{p}_j^{kl}(\epsilon)} \leq \frac{1}{lw(\epsilon)}$$

Also, since $lw(\epsilon) \cdot up(\epsilon) \leq 1$, then $\frac{1}{lw(\epsilon)} \geq up(\epsilon)$, so the fraction $\frac{\tilde{p}_j^{kl}(\epsilon)}{p_j^{kl}(\epsilon)}$ is also upper bounded by $\frac{1}{lw(\epsilon)}$.

We now use Theorem 2 with $\delta := \frac{1}{lw(\epsilon)} - 1$, and a as a an upper bound on the absolute value of the rewards. Now look at any state k, and let $\gamma, \tilde{\gamma} > 0$ be the numbers such that $\underline{v}_k + \gamma$ and $\underline{v}_k + \tilde{\gamma}$ are the values for Player II of the games where Player I has fixed his strategy to $(q_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ and $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ respectively. Then from Theorem 2 we get

$$\begin{aligned} &\underline{v}_k + \gamma - (\underline{v}_k + \tilde{\gamma}) = \gamma - \tilde{\gamma} \geq -4N\delta a \\ \Rightarrow &\tilde{\gamma} \leq 4N \left(\frac{1}{1 - \epsilon^Q \frac{2C+cC}{c}} - 1 \right) a + \gamma \leq 4N \frac{\epsilon^Q \frac{2C+cC}{c}}{1 - \epsilon^Q \frac{2C+cC}{c}} a + \epsilon \end{aligned}$$

Since the denominator $1 - \epsilon^Q \frac{2C+cC}{c}$ tends to 1 for $\epsilon \to 0$, then for ϵ smaller than some ϵ'''', the denominator is always larger than $\frac{1}{2}$. So by letting $\epsilon_0 := \min(\epsilon'''', \epsilon_4)$ we get that $\tilde{\gamma} \leq \frac{8Na(2C+cC)}{c}\epsilon^Q + \epsilon$. This implies that $(\rho_j^k(\epsilon))_{j\in[m]}^{k\in[N]}$ is a $\left(\frac{8Na(2C+cC)}{c}\epsilon^Q + \epsilon\right)$-optimal strategy among stationary strategies. Now consider

the strategy defined by $\varphi_j^k := \rho_j^k\left(\left(\frac{c}{8Na(2C+cC)}\epsilon\right)^{\frac{1}{Q}}\right)$, which is then $(\epsilon^Q+\epsilon)$-optimal among stationary strategies. The strategy $\left(\varphi_j^k(\frac{\epsilon}{2})\right)_{j\in[m]}^{k\in[N]}$ is then an ϵ-optimal strategy, since $\left(\frac{\epsilon}{2}\right)^Q + \frac{\epsilon}{2} \leq \epsilon$.

Finally notice that the strategy $\left(\varphi_j^k(\frac{\epsilon}{2})\right)_{j\in[m]}^{k\in[N]}$ is not a monomial family of strategies, since it could have fractional exponents. To fix this, we define

$$M := \mathrm{lcm}_{j\in\{1,\ldots,m\},k\in\{1,\ldots,N\}} M_j^k,$$

and let $x_{\epsilon,j}^k := \rho_j^k\left(\left(\left(\frac{\epsilon}{2}\right)^Q\right)^M\right)$. Then $(\boldsymbol{x}_\epsilon)_{0<\epsilon\leq\epsilon_0}$ is a monomial family of strategies, which is also $\epsilon-$optimal amough stationary strategies, because $\left(\frac{\epsilon}{2}\right)^{QM} \leq \frac{\epsilon}{2}$, hence adding the exponent QM only improves the approximation of the value.

References

1. Bewley, T., Kohlberg, E.: The asymptotic theory of stochastic games. Mathematics of Operations Research 1(3), 197–208 (1976)
2. Bewley, T., Kohlberg, E.: On stochastic games with stationary optimal strategies. Mathematics of Operations Research 3(2), 104–125 (1978)
3. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: Strategy improvement for concurrent reachability games. In: Third International Conference on the Quantitative Evaluation of Systems, QEST 2006, pp. 291–300. IEEE Computer Society (2006)
4. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Stochastic limit-average games are in EXPTIME. International Journal of Game Theory 37(2), 219–234 (2008)
5. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theor. Comput. Sci. 386(3), 188–217 (2007)
6. Everett, H.: Recursive games. In: Kuhn, H.W., Tucker, A.W. (eds.) Contributions to the Theory of Games III. Annals of Mathematical Studies, vol. 39, Princeton University Press (1957)
7. Freidlin, M., Wentzell, A.: Random Perturbations of Dynamical Systems. Springer (1984)
8. Gillette, D.: Stochastic games with zero stop probabilities. In: Contributions to the Theory of Games III. Ann. Math. Studies, vol. 39, pp. 179–187. Princeton University Press (1957)
9. Hansen, K.A., Ibsen-Jensen, R., Miltersen, P.B.: The complexity of solving reachability games using value and strategy iteration. In: Kulikov, A., Vereshchagin, N. (eds.) CSR 2011. LNCS, vol. 6651, pp. 77–90. Springer, Heidelberg (2011)
10. Hansen, K.A., Koucky, M., Lauritzen, N., Miltersen, P.B., Tsigaridas, E.P.: Exact algorithms for solving stochastic games. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing, pp. 205–214. ACM (2011)
11. Hansen, K.A., Koucky, M., Miltersen, P.B.: Winning concurrent reachability games requires doubly exponential patience. In: 24th Annual IEEE Symposium on Logic in Computer Science (LICS 2009), pp. 332–341. IEEE (2009)

12. Mertens, J.F., Neyman, A.: Stochastic games. International Journal of Game Theory 10, 53–66 (1981)
13. Milman, E.: The semi-algebraic theory of stochastic games. Mathematics of Operations Research 27(2), 401–418 (2002)
14. Solan, E.: Continuity of the value of competitive Markov decision processes. Journal of Theoretical Probability 16, 831–845 (2003)
15. Solan, E., Vieille, N.: Computing uniformly optimal strategies in two-player stochastic games. Economic Theory 42, 237–253 (2010)

Stability Controllers for Sampled Switched Systems

Laurent Fribourg and Romain Soulat

LSV, ENS de Cachan & CNRS, France

Abstract. We consider in this paper switched systems, a class of hybrid systems recently used with success in various domains such as automotive industry and power electonics. We propose a state-dependent control strategy which makes the trajectories of the analyzed system converge to finite cyclic sequences of points. Our method relies on a technique of decomposition of the state space into local regions where the control is uniform. We have implemented the procedure using zonotopes, and applied it successfully to several examples of the literature.

1 Introduction

Switched systems are now widely used in industrial applications in domains such as power electronics or automative industry. A switched system can be viewed as a family of continuous-time subsystems with a rule that orchestrates the switching between them. A suitable switching rule allows to steer the system to interesting operating regions which are not accessible using a single subsystem. However, it becomes impossible to stabilize the system around a unique equilibrium point, as in classical systems. The stabilization problem is relaxed as a problem of "practical stabilization", as follows: given a region R of the state space, find a switching rule that makes the system converge to a region, located inside R, as small as possible. In practice, the controlled trajectories of switched systems often converge to *limit cycles* (see, e.g., [His01]). We present here a forward-oriented method that performs a *decomposition* of the region R, and induces a state-dependent control which, under certain conditions, makes the system converge to a cyclic trajectory.

Related Work

To the best of of our knowledge, applying a process of state space decomposition in order to stabilize system dynamics is original, at least in the context of switched systems. The method presents some similarities with the method of *box invariance* of [ATS09] which exhibits rectangular invariant subregions of affine hybrid systems containing an equilibrium point, and with the method of *bisection* used in [JKDW01] for the purpose of "set inversion".

The classical methods that are used for proving the existence and stability of limit cycles are based on various techniques such as Lyapunov functions (see, e.g.,

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 135–145, 2013.

[BRC05, RRL00]), Poincaré map (see, e.g., [Gon03, His01]), sensibility functions [FRL06], or describing functions [San93]. We give here a couple of conditions, called (A1)-(A2), from which the existence of stable limit cycles follows in an elementary way.

Outline of This Paper

We first present the decomposition method in Section 2. We then show that the decomposition induces a state-dependent control in Section 3. We explain that, under certain conditions, the controlled trajectories converge to limit cycles (Section 4). Experimental results are described in Section 5. We conclude in Section 6.

2 State Space Decomposition

A *switched system* Σ is defined by a finite family of differential equations of the form $\{\dot{x} = f_u(x)\}_{u \in U}$ where U is a finite set of *modes* (see, e.g., [GPT10, Tab09]). In the following, we consider that the dynamics of the subsystems are *affine* (i.e., $f_u(x)$ is of the form $A_u x + b_u$ with $A_u \in \mathbb{R}^{n \times n}$ and b_u a vector of $\mathbb{R}^n$). The control problem for a switched system Σ is to find a piecewise constant law $\mathbf{u} : \mathbb{R}_{\geq 0} \to U$ in order to achieve some pertained goals. The *switching instants* are the times at which $\mathbf{u}$ changes its value. A *sampled switched system* is a switched system for which the switching instants occur at integer multiples of τ (called *sampling parameter*). We will use $\mathbf{x}(t, x, u)$ to denote the point reached by Σ at time t under mode u from the initial condition x. This gives a transition relation $\rightarrow_u^\tau$ defined for x and x' in $\mathbb{R}^n$ by: $x \rightarrow_u^\tau x'$ iff $\mathbf{x}(\tau, x, u) = x'$. Given a set $X \subseteq \mathbb{R}^n$, we define:

$$Post_u(X) = \{x' \mid x \rightarrow_u^\tau x' \text{ for some } x \in X\}.$$

It can be seen that $Post_u(X)$ is the result of an affine transformation of the form $C_u X + d_u$ with $C_u \in \mathbb{R}^{n \times n}$ and d_u a vector of $\mathbb{R}^n$.

We say that a subset X of $\mathbb{R}^n$ is *controlled invariant* if:

$$\forall x \in X \ \exists u \in U \ \exists x' \in X : \ x \rightarrow_u^\tau x'.$$

A *pattern* π is defined as a finite sequence of modes. A k-pattern is a pattern of length at most k. Given a pattern π of the form $(u_1 \cdots u_n)$ and a subset X of $\mathbb{R}^n$, we define:

$$Post_\pi(X) = \{x' \mid x \rightarrow_{u_1}^\tau x_1 \ \wedge x_1 \rightarrow_{u_2}^\tau x_2 \ \wedge \cdots \wedge x_{m-1} \rightarrow_{u_m}^\tau x' \text{ for some } x \in X \text{ and } x_1, \ldots, x_{m-1} \in \mathbb{R}^n\}.$$

Given a pattern π of the form $(u_1 \cdots u_m)$, and a set $X \subseteq \mathbb{R}^n$, the *unfolding of* X *via* π, denoted by $Unf_\pi(X)$, is the set $\bigcup_{i=0}^m X_i$ with:

- $X_0 = X$,
- $X_{i+1} = Post_{u_{i+1}}(X_i)$, for all $0 \leq i \leq m-1$.

Definition 1. *Given a set $R \subseteq \mathbb{R}^n$, a* k-invariant decomposition *of R is a set Δ of the form $\{V_i, \pi_i\}_{i \in I}$, where I is a finite set of indices, V_is are subsets of R, π_is are k-patterns, such that:*

- $\bigcup_{i \in I} V_i = R$, *and*
- *for all $i \in I$: $Post_{\pi_i}(V_i) \subseteq R$.*

Given a set $R \subseteq \mathbb{R}^n$ and a k-invariant decomposition $\Delta = \{(V_i, \pi_i)\}_{i \in I}$ of R, the Δ-*unfolding* of a subset X of R is defined by $\bigcup_{i \in I} Unf_{\pi_i}(V_i \cap X)$. The operator $Post_\Delta$ is defined, for all subset X of R by:

$$Post_\Delta(X) = \bigcup_{i \in I} Post_{\pi_i}(X \cap V_i).$$

We have:

Proposition 1. *Suppose that a set R has a k-invariant decomposition Δ. Then we have: $Post_\Delta(R) \subseteq R$.*

We now give a simple algorithm, called Decomposition algorithm, which, given a set R, outputs, when it succeeds, a k-invariant decomposition Δ of the form $\{V_i, \pi_i\}_{i \in I}$ for R. The input set R is given under the form of a *box* of $\mathbb{R}^n$, that is a cartesian product of n closed intervals. The subsets V_is of R are boxes that are obtained by bisection. Two adjacent boxes thus share a common border.

The Decomposition procedure first calls sub-procedure Find_Pattern in order to get a k-pattern π such that $Post_\pi(R) \subseteq R$. If it succeeds, then it is done. Otherwise, it divides R into 2^n sub-boxes $V_1, \ldots, V_{2^n}$ of equal size. If for each V_i, Find_Pattern gets a k-pattern π_i such that $Post_{\pi_i}(V_i) \subseteq R$, it is done. If, for some V_j, no such pattern exists, the procedure is recursively applied to V_j. It ends with success when a k-invariant decomposition of R is found, or failure when the maximal degree d of decomposition is reached. The algorithmic form of the procedure is given in Algorithms 1 and 2. (For the sake of simplicity, we consider the case of dimension $n = 2$, but the extension to $n > 2$ is straightforward.) The main procedure Decomposition(W,R,D,K) is called with R as input value for W, d for input value for D, and k as input value for K; it returns either $\langle \{(V_i, \pi_i)\}_i, True \rangle$ with $\bigcup_i V_i = W$ and $\bigcup_i Post_{\pi_i}(V_i) \subseteq R$, or $\langle \_, False \rangle$. Procedure Find_Pattern(W,R,K) looks for a K-pattern π for which $Post_\pi(W) \subseteq R$: it selects all the K-patterns by non-decreasing length order until either it finds such a pattern π (output: $\langle \pi, True \rangle$), or none exists (output: $\langle \_, False \rangle$). The correctness of the procedure is stated as follows.

Theorem 1. *If Decomposition(R,R,d,k) returns $\langle \Delta, True \rangle$, then Δ is a k-invariant decomposition of R.*

Example 1. (**Boost DC-DC Converter**). This example is taken from [BPM05] (see also, e.g., [BRC05, GPT10, SEK03]). This is a boost DC-DC converter with one switching cell (see Figure 1). There are two operation modes depending on the position of the switching cell. An example of pattern of length 4 is illustrated

Algorithm 1. Decomposition(W,R,D,K)

Input: A box W, a box R, a degree D of decomposition, a length K of pattern
Output: $\langle \{(V_i, \pi_i)\}_i, True\rangle$ with $\bigcup_i V_i = W$ and $\bigcup_i Post_{\pi_i}(V_i) \subseteq R$, or $\langle \_, False\rangle$

$(\pi, b) := Find\_Pattern(W, R, K)$
if $b = True$ **then**
  return $\langle \{(W, \pi)\}, True\rangle$
else
  if $D = 0$ **then**
    return $\langle \_, False\rangle$
  else
    Divide equally W into (W_1, W_2, W_3, W_4) /* (case $n = 2$) */
    $(\Delta_1, b_1) :=$ Decomposition(W_1,R,$D-1$,K)
    $(\Delta_2, b_2) :=$ Decomposition(W_2,R,$D-1$,K)
    $(\Delta_3, b_3) :=$ Decomposition(W_3,R,$D-1$,K)
    $(\Delta_4, b_4) :=$ Decomposition(W_4,R,$D-1$,K)
    return $(\Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta_4, b_1 \wedge b_2 \wedge b_3 \wedge b_4)$

Algorithm 2. Find_Pattern(W,R,K)

Input: A box W, a box R, a length K of pattern
Output: $\langle \pi, True\rangle$ with $Post_\pi(W) \subseteq R$, or $\langle \_, False\rangle$ when no pattern maps W into R

for $i = 1 \ldots K$ **do**
  $\Pi :=$ set of patterns of length i
  while Π *is non empty* **do**
    Select π in Π
    $\Pi := \Pi \setminus \{\pi\}$
    if $Post_\pi(W) \subseteq R$ **then**
      return $\langle \pi, True\rangle$
return $\langle \_, False\rangle$

in Figure 2: it corresponds to the application of mode 2 on $[0, \tau)$ and mode 1 on $[\tau, 4\tau)$. The state of the system is $x(t) = [i_l(t)\ v_c(t)]^T$ where i_l is the current intensity in inductor, and $v_c(t)$ the voltage of capacitor. The aim of the control is to maintain the system inside a given zone R while the output voltage stabilizes around a desired value. The dynamics associated with mode u is of the form $\dot{x}(t) = A_u x(t) + b_u$ $(u = 1, 2)$ with

$$A_1 = \begin{pmatrix} -\frac{r_l}{x_l} & 0 \\ 0 & -\frac{1}{x_c}\frac{1}{r_0+r_c} \end{pmatrix} \quad b_1 = \begin{pmatrix} \frac{v_s}{x_l} \\ 0 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} -\frac{1}{x_l}(r_l + \frac{r_0 . r_c}{r_0+r_c}) & -\frac{1}{x_l}\frac{r_0}{r_0+r_c} \\ \frac{1}{x_c}\frac{r_0}{r_0+r_c} & -\frac{1}{x_c}\frac{1}{r_0+r_c} \end{pmatrix} \quad b_2 = \begin{pmatrix} \frac{v_s}{x_l} \\ 0 \end{pmatrix}$$

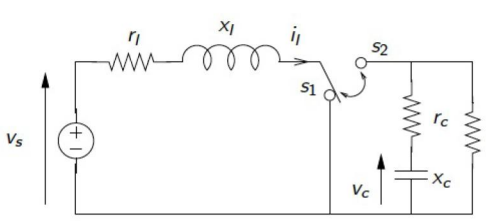


Fig. 1. Scheme of the boost DC-DC converter

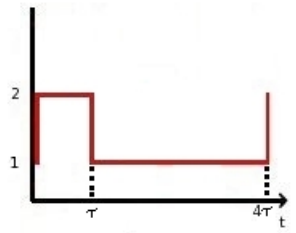


Fig. 2. Cell switching for pattern (2.1.1.1)

We will use the numerical values of [BPM05], expressed in the per unit system: $x_c = 70$, $x_l = 3$, $r_c = 0.005$, $r_l = 0.05$, $r_0 = 1$, $v_s = 1$. The sampling period is $\tau = 0.5$. For $R = [1.55, 2.15] \times [1.0, 1.4]$, the Decomposition algorithm yields a decomposition $\Delta = \{(V_i, \pi_i)\}_{i=1,\ldots,4}$, which is depicted in the left part of Figure 3: the sub-region $V_1 = [1.55, 1.85] \times [1.0, 1.2]$ is associated pattern $\pi_1 = (1{\cdot}1{\cdot}2{\cdot}2{\cdot}2)$, $V_2 = [1.85, 2.15] \times [1.0, 1.2]$ with $\pi_2 = (2)$, $V_3 = [1.85, 2.15] \times [1.2, 1.4]$ with $\pi_3 = (2 \cdot 1 \cdot 2)$, and $V_4 = [1.55, 1.85] \times [1.2, 1.4]$ with $\pi_4 = (1)$. For all $1 \leq i \leq 4$, we have: $Post_\Delta(V_i) = Post_{\pi_i}(V_i) \subseteq R$. This is visualized in the right part of Figure 3.

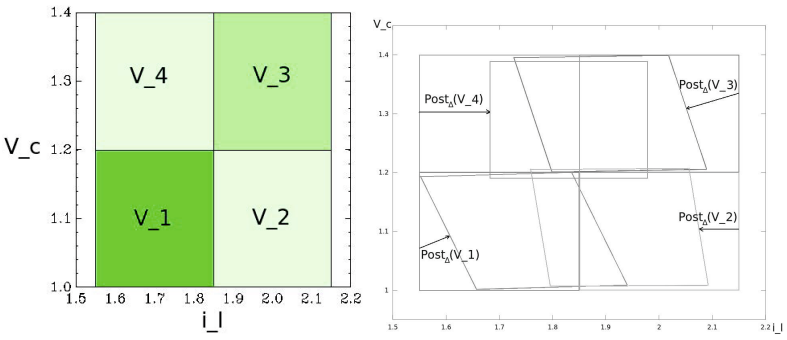


Fig. 3. Decomposition Δ of $R = [1.55, 2.15] \times [1.0, 1.4]$ for the boost DC-DC converter example (left), and visualization of $Post_\Delta(V_i) \subseteq R$, $i = 1, \ldots, 4$ (right)

3 Δ-trajectories

A k-invariant decomposition Δ of R induces a state-dependent control that makes any trajectory starting fom R go back to R within at most k steps: given a starting state x_0 in R, we know that $x_0 \in V_i$ for some $i \in I$ (since $R = \bigcup_{i \in I} V_i$); one thus applies π_i to x_0, which gives a new state x_1 that belongs

to R (since $Post_\pi(V_i) \subseteq R$); the process is repeated on x_1, and so on iteratively. Given a point $x \in R$, we will denote by $succ_\Delta(x)$ the point of R obtained by applying π_i to x when x is in V_i. [1]

Definition 2. *Suppose that Δ is a k-invariant decomposition of a given set R. A* discrete trajectory induced by Δ, *or more simply, a* Δ-trajectory, *is a sequence of points $\{x_i\}_{i\geq 0}$ of R, with $x_{i+1} = succ_\Delta(x_i)$ for all $i \geq 0$.* [2] *A* Δ-cycle *is a Δ-trajectory of R of the form $\{x_0, x_1, \ldots, x_{m-1}\}$ with $x_0 = succ_\Delta(x_{m-1})$.*

Example 2. Consider the boost example 1. Figure 4 depicts a Δ-trajectory starting from the left upper corner of $R = [1.55, 2.15] \times [1.0, 1.4]$, together with its Δ-unfolding.

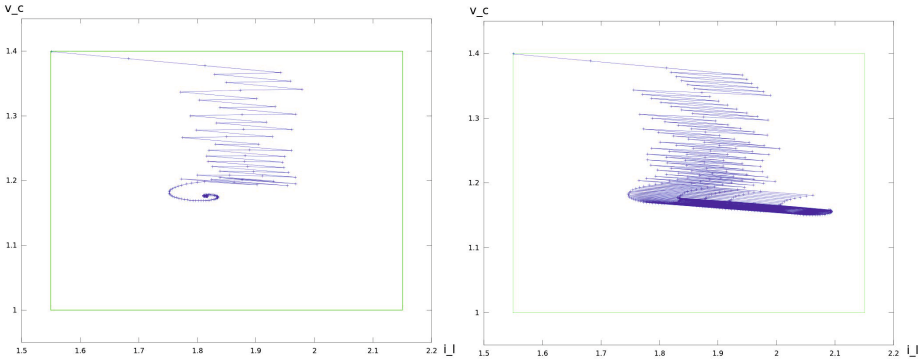


Fig. 4. Δ-trajectory for the boost example (left), and its Δ-unfolding (right)

In Figure 4, we can see that the Δ-trajectory and its Δ-unfolding seem to converge to cycles. We now formally state that, under certain assumptions, this is actually the case.

4 Limit Cycles

We suppose that we are given a region $R \subseteq \mathbb{R}^n$ and a k-invariant decomposition $\Delta = \{(V_i, \pi_i)\}_{i\in I}$ of R (produced, e.g., by the Decomposition algorithm of Section 2).

Proposition 2. *Consider a k-invariant decomposition $\Delta = \{(V_i, \pi_i)\}_{i\in I}$ of R. Let R^j_Δ be defined by $R^0_\Delta = R$, and $R^j_\Delta = Post_\Delta(R^{j-1}_\Delta)$ for $j > 0$. The sequence $\{R^j_\Delta\}_{j\geq 0}$ is a decreasing nested sequence and the set $R^*_\Delta = \bigcap_{j\geq 0} R^j_\Delta$ is well-defined. Furthermore, R^*_Δ is an* attractor set *of R, i.e.:*

[1] A nondeterministic choice has to be done when a point x belongs to more than one subset V_i. When x belongs to a single subset V_i, then $succ_\Delta(x) = Post_\Delta(x)$.

[2] We will sometimes denote such a trajectory under the form: $x_0 \rightarrow_{\pi_{i_1}} x_1 \rightarrow_{\pi_{i_2}} \cdots$ with $i_1, i_2, \cdots \in I$.

– $Post_{\Delta}(R^*_{\Delta}) = R^*_{\Delta}$ *(invariance property)*
– $\forall x \in R,\ d(succ^j_{\Delta}(x), R^*_{\Delta}) \to 0$ *as* j *tends to* ∞ *(attraction property).*[3]

Furthermore we have:

Proposition 3. *For all* $i \geq 0$*, the set* R^i_{Δ} *is a finite union of polyhedra.*

We now make the following assumption:

(A1): There exists $N > 0$ such that R^N_{Δ} is a finite union of polyhedra $P_1, \ldots, P_q$ (with $q \in \mathbb{N}$) such that:

$$\forall j \in \{1, \ldots, q\}\ \exists! i \in I :\ P_j \cap V_i \neq \emptyset.$$

Assumption $(A1)$ states that every polyhedral component of R^N_{Δ} shares common points with a single subset V of R. In particular no polyhedron can cross a common intersection ("border") of two distinct subsets V and V' of R. This implies that operator $Post_{\Delta}$ applied to any polyhedron of R^N_{Δ} is deterministic: $\forall j \in \{1, \ldots, q\}\ \exists! i \in I\ Post_{\Delta}(P_j) = Post_{\pi_i}(P_j)$. Furthermore, we have:

$$\forall j \in \{1, \ldots, q\}\ \exists! j' \in \{1, \ldots, q\} :\ Post_{\Delta}(P_j) \subseteq P_{j'}.$$

Therefore, R^N_{Δ} can be seen as a directed graph of vertices $P_1, \ldots, P_q$, with an edge from P_j to $P_{j'}$ iff $Post_{\Delta}(P_j) \subseteq P_{j'}$. The vertices of this graph have a single outgoing edge. The sets R^i_{Δ} for $i \geq N$ are generated by further application of $Post_{\Delta}$. The polyhedral components of R^i_{Δ} which have no incoming edge will disppear at iteration $i + 1$. After a finite number of iterations, the graph of the polyhedral components of R^i_{Δ} corresponds to the strongly connected components of R^N_{Δ}. Furthermore, these strongly connected components correspond to disjoint cycles, since the vertices of the graphs have only one outgoing edge. This is formally stated as follows.

Theorem 2. *Under assumptions (A1), we have:*

1. R^*_{Δ} *is a finite union of disjoint cycles of polyhedra.*
2. *The* Δ*-unfolding of each cycle of* R^*_{Δ} *is a controlled invariant set.*

Let $\mathcal{C}_1, \ldots, \mathcal{C}_r$ denote the cycles of polyhedra of R^*_{Δ}. Each cycle $\mathcal{C}_i$ $(1 \leq i \leq r)$ is made of a finite set of polyhedra. Each polyhedron P of a cycle $\mathcal{C}$ is associated with a pattern π such that $Post_{\pi}(P) = P$. Let us now consider the additional assumption

(A2): For each pattern π associated with a polyhedron P of a cycle $\mathcal{C}$, π is locally contractive in R, i.e.:

$$\forall x, y \in R\ \ \|Post_{\pi}(x) - Post_{\pi}(y)\| < \|x - y\|$$

for some norm $\|\cdot\|$ of $\mathbb{R}^n$.

[3] $d(y, Z)$ denotes the smallest distance between a point y and any point of Z, and $succ^j_{\Delta}(x)$ the point obtained from x after j applications of $succ_{\Delta}$.

Then we have:

Theorem 3. *Under assumptions (A1) and (A2), we have:*

1. R^*_Δ *is a finite union of disjoint cycles of points of* R.
2. *The* Δ*-unfolding of each cycle of* R^*_Δ *is a controlled invariant set.*
3. *Each* Δ*-trajectory* $\{x_0, x_1, \dots\}$ converges to a cycle *of the form* $\{y_0, y_1, \dots, y_{m-1}\}$ *in the following sense:*

$$\exists M \in \mathbb{N}\ \forall \ell = 0, \dots, m-1 \ \lim_{i \to \infty} x_{M+i \cdot m+\ell} = y_\ell.$$

for all $\ell = 0, \dots, m-1$.

The proof of a variant of Theorem 3 is given in [FS13].

We now illustrate the convergence of R^k_Δ to a cyclic set of points as k tends to infinity, on the boost example.

Example 3. (**Boost DC-DC Converter**). One can check that the modes of the boost converter are locally contractive in $R = [1.55, 1.85] \times [1.0, 1.2]$, hence (A2) is satisfied. Likewise, (A1) is satisfied: for $N = 100$, all the polyhedral components of R^N_Δ belong to a single box (viz., V_1) of the decomposition Δ. This is shown in Figure 5, which depicts the iterated images R^k_Δ for $k = 0, 20, 40, 60, 80, 100$. The limit set R^*_Δ is here composed of a unique limit cycle that is made of a single point $y_0 \in V_1$. We have: $y_0 \rightarrow_{\pi_1} y_1 = y_0$, with $\pi_1 = (1 \cdot 1 \cdot 2 \cdot 2 \cdot 2)$. The Δ-unfolding of this limit cycle is thus made of 5 points (corresponding to the composing modes of π_1) and is depicted in Figure 6.

5 Implementation

The implementation of the method is made of two basic procedures: a procedure Decomposition (described in Section 2), which outputs Δ, and a procedure, called Iteration which constructs R^i_Δ for $i \geq 0$. The Decomposition procedure makes use of zonotopes [KÖ98], and has been written in Octave [oct], except for the multilevel examples which have been implemented using PLECS [ple]. The procedure Iteration does not use the data structure zonotopes because it involves the intersection operator which does not preserve the structure of zonotopes. It has been written in Ocaml [oca], using the PPL library [ppl] of polyhedra. The Iteration procedure receives Δ from module Decomposition and outputs the successive iterations of $Post_\Delta$. The sequence of post sets can also be visualized as an animation (see Figure 5).

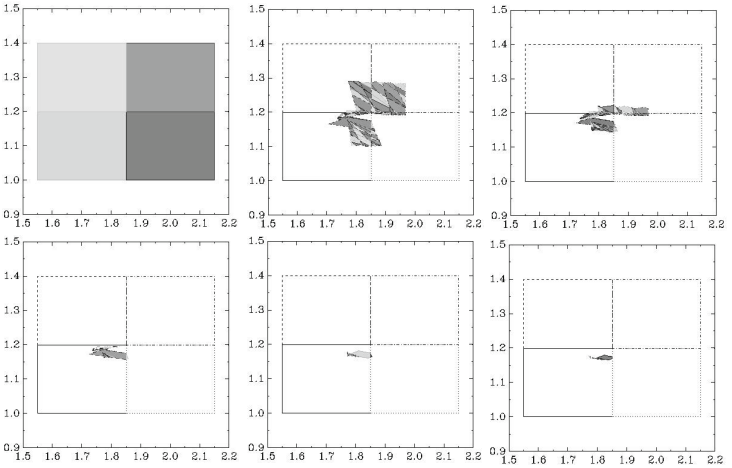


Fig. 5. Visualization of R^k_Δ for $k = 0, 20, 40, 60, 80, 100$ for the boost example

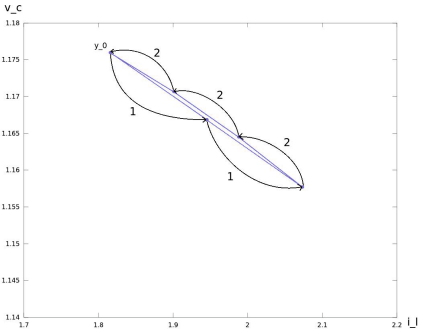


Fig. 6. Δ-unfolding of the limit cycle $\{y_0\}$ for the boost example

The examples have run on a machine equipped with an Intel Core2 CPU X6800 at 2.93GHz and with 2GB of Ram memory. Some figures of the experiments are listed in the following table.

Example	Running time	# patterns	$\|U\|$	k	d	n	(A1)	(A2)	cycle
Boost [BPM05]	150 seconds	12113	2	5	1	2	yes	yes	yes
Two-tank [His01]	4 seconds	1423	4	3	1	2	yes	yes	yes
Heating [Gir12]	1 second	134	2	2	4	2	yes	yes	yes
Helicopter [DLHT11]	$\approx$ 2 hours	$\approx$ 1.5 million	9	6	4	2	yes	no	yes
5-level [FFL$^+$12]	3 minutes	-	16	8	1	3	yes	no	yes
7-level [FFL$^+$12]	35 minutes	-	64	32	1	5	yes	no	yes
9-level [FFL$^+$12]	$\approx$ 5 hours	-	256	128	1	7	yes	no	yes

The first column indicates the name of the example together with its reference. The second column indicates the running time to obtain a decomposition, and the third one the numbers of patterns generated to obtain this decomposition[4]. The subsequent columns labeled by $|U|$, k, d and n indicate the number of modes, the input parameter of maximal pattern length, the input parameter of decomposition depth and the space dimension respectively. Finally, the column '(A1)' (resp. '(A2)') indicates if (A1) (resp. (A2)) is satisfied, and the column 'cycle' if the controlled trajectories converge to a limit cycle.

6 Final Remarks

We have presented an original technique to synthesize stability controllers for switched systems. We have implemented the procedure, and applied it successfully to several examples of the literature. The method can also be used for synthesizing *safety controllers* in order to guarantee safety properties of the controlled system (see [FS13]). A sufficient condition for the existence of a k-invariant decomposition of a given box R is also given in [FS13].

References

[ATS09] Abate, A., Tiwari, A., Sastry, S.: Box invariance in biologically-inspired dynamical systems. Automatica 45(7), 1601–1610 (2009)

[BPM05] Beccuti, A.G., Papafotiou, G., Morari, M.: Optimal control of the boost dc-dc converter. In: Proc. 44th IEEE Conference on Decision and Control European Control Conference (CDC-ECC 2005), pp. 4457–4462 (December 2005)

[BRC05] Buisson, J., Richard, P.-Y., Cormerais, H.: On the stabilisation of switching electrical power converters. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 184–197. Springer, Heidelberg (2005)

[DLHT11] Ding, J., Li, E., Huang, H., Tomlin, C.J.: Reachability-based synthesis of feedback policies for motion planning under bounded disturbances. In: IEEE International Conference on Robotics and Automation (ICRA 2011), pp. 2160–2165 (2011)

[FFL+12] Feld, G., Fribourg, L., Labrousse, D., Revol, B., Soulat, R.: Correct by design control of 5-level and 7-level converters. Research Report LSV-12-25, LSV, ENS Cachan, France (December 2012)

[FRL06] Flieller, D., Riedinger, P., Louis, J.-P.: Computation and stability of limit cycles in hybrid systems. Nonlinear Analysis 64(2), 352–367 (2006)

[FS13] Fribourg, L., Soulat, R.: Finite controlled invariants for sampled switched systems. Research Report LSV-13-09, Laboratoire Spécification et Vérification, ENS Cachan, France, 27 pages (April 2013)

[Gir12] Girard, A.: Low-complexity switching controllers for safety using symbolic models. In: Heemels, M., De Schutter, B., Lazar, M. (eds.) 4th IFAC conference on Analysis and Design of Hybrid Systems, Eindhoven, Pays-Bas (June 2012)

[4] This figure is not available for the multilevel converters because they have been implemented using PLECS, rather than Octave.

[Gon03] Gonçalves, J.M.: Region of stability for limit cycles of piecewise linear systems. In: IEEE Conference on Decision and Control (2003)

[GPT10] Girard, A., Pola, G., Tabuada, P.: Approximately bisimilar symbolic models for incrementally stable switched systems. IEEE Trans. on Automatic Control 55, 116–126 (2010)

[His01] Hiskens, I.A.: Stability of limit cycles in hybrid systems. In: 34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 3-6. IEEE Computer Society (2001)

[JKDW01] Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis: with examples in parameter and state estimation. In: Robust Control and Robotics. Springer (2001)

[K̈98] Kühn, W.: Zonotope dynamics in numerical quality control. In: Mathematical Visualization, pp. 125–134 (1998)

[oca] OCaml Web page, `http://caml.inria.fr/ocaml/index.en.html`

[oct] Octave Web page, `http://www.gnu.org/software/octave/`

[ple] PLECS Web page, `http://www.plexim.com`

[ppl] PPL Web page, `http://bugseng.com/products/ppl/`

[RRL00] Rubensson, M., Rubensson, M., Lennartson, B.: Stability of limit cycles in hybrid systems using discrete-time lyapunov techniques. In: Proceedings of the 39th IEEE Conference on Decision and Control (2000)

[San93] Sanders, S.R.: On limit cycles and the describing function method in periodically switched circuits. IEEE Trans. Circuits and Systems 40(9), 564–572 (1993)

[SEK03] Senesky, M., Eirea, G., Koo, T.J.: Hybrid modelling and control of power electronics. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 450–465. Springer, Heidelberg (2003)

[Tab09] Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach, 1st edn. Springer Publishing Company, Incorporated (2009)

Formal Languages, Word Problems of Groups and Decidability

Sam A.M. Jones and Richard M. Thomas

Department of Computer Science, University of Leicester, Leicester LE1 7RH, U.K.

Abstract. This paper considers questions relating formal languages to word problems of groups with a particular emphasis on the decidability of some problems that arise. We investigate the decidability of certain natural conditions that characterize word problems for various classes of languages and we then turn our attention to the question of a language actually being a word problem. We show that this is decidable for the classes of regular and deterministic context-free languages but undecidable for the one-counter languages.

1 Introduction

The word problem of a finitely generated group G (i.e. the set of words representing the identity element of G) is a fundamental notion in group theory and there have been some intriguing connections between this and formal language theory. In particular, various classifications have been obtained as to which groups have their word problems lying in certain classes of languages (see Section 4). We focus on subfamilies of the context-free languages and, given a result of Herbst (Theorem 7) which says that, under certain closure assumptions, there are essentially only three cases, we concentrate on those particular families, namely the regular languages, the one-counter languages and the context-free languages.

There is also a simple necessary and sufficient criterion (see Theorem 9) for a language to be the word problem of a group. This involves the conjunction of two conditions (universal prefix closure and deletion closure - see Section 2 for definitions of these concepts) and we consider the question of the decidability of these two conditions in Sections 5 and 6 respectively. It is intriguing that we have a connection between word problems of groups and natural formal language conditions such as deletion closure as studied in [15].

Having established the decidability of these conditions for the classes of languages we are considering we turn our attention to the question of deciding their conjunction, i.e. that of deciding whether a given language is the word problem of a group. Whilst this is easily seen to be decidable for the regular languages we build on the work in [17] to show that this is undecidable for one-counter languages (and hence for context-free languages as well); see Theorem 28. However, we know that any context-free language that is the word problem of a group is deterministic context-free and we show that the problem of deciding whether a deterministic context-free language is the word problem of a group is actually decidable (see Theorem 31).

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 146–158, 2013.

2 Background from Formal Language Theory

Throughout this paper we will be discussing regular and context-free languages accepted by finite automata and pushdown automata respectively. We will also be discussing *one-counter languages* which are those languages accepted by a *one-counter automaton*, i.e. a pushdown automaton where we have only a single stack symbol (apart from a symbol marking the bottom of the stack); these automata are nondeterministic and accept by final state. We will use some standard definitions and properties of classes of languages (such as their closure properties under certain operations and decidability results); see [2,11,14] for example.

In particular, it is well known (see [11] for example) that one cannot decide whether or not a context-free language $L \subseteq \Sigma^*$ is equal to Σ^* (the so called *universe problem*). In fact, this problem remains undecidable if one restricts oneself to the certain subsets of the context-free languages such as the one-counter languages [13]. We will need a slight strengthening of this fact where we restrict to the case where the alphabet has size 2:

Theorem 1. *The following decision problem is undecidable:*
Input: *a one-counter automaton M with input alphabet Ω of size 2.*
Output: *"yes" if $L(M) = \Omega^*$; "no" otherwise.*

Proof. We use a standard technique to show that, if we had an algorithm $\mathfrak{A}$ solving this decision problem, then we would have an algorithm solving the universe problem for alphabets of arbitrary size. So suppose that we have such an algorithm $\mathfrak{A}$. Let $\Sigma = \{x_1, x_2, \ldots, x_n\}$ be an arbitrary alphabet and let $\Omega = \{a, b\}$. Let $M = (Q, \Sigma, \Gamma, \tau, s, A)$ be a one-counter automaton and let $L = L(M)$. We want to determine whether or not $L(M) = \Sigma^*$.

Define $\varphi : \Sigma^* \to \Omega^*$ by $x_1 \mapsto ab$, $x_2 \mapsto a^2b$, $\ldots$, $x_n \mapsto a^nb$, and let $K = \Sigma^*\varphi$. Since K is regular, $R = \Omega^* - K$ is regular. Since φ is injective we have that $L = \Sigma^*$ if and only if $L\varphi = \Sigma^*\varphi = K$ which is equivalent to saying that $L\varphi \cup R = K \cup R = \Omega^*$. Since $L\varphi \cup R$ is a one-counter language we may use algorithm $\mathfrak{A}$ to decide this problem, and so we could determine whether or not $L(M) = \Sigma^*$, a contradiction. □

Remark 2. In Theorem 1 all we have used about the family $\mathcal{F}$ of one-counter languages are the facts that the universe problem is undecidable for $\mathcal{F}$ and that $\mathcal{F}$ is closed under homomorphism and union with regular languages; so Theorem 1 applies to any such family of languages. □

In this paper we will also need the idea of the *prefix closure* of a language $L \subseteq \Sigma^*$ which is defined to be:

$$\text{prefix}(L) = \{\alpha \in \Sigma^* : \alpha\beta \in L \text{ for some } \beta \in \Sigma^*\}.$$

In the case where $\text{prefix}(L) = \Sigma^*$, we say that L has the *universal prefix closure property*.

We also say that a language $L \subseteq \Sigma^*$ is *deletion closed* if it satisfies the following condition:

$$\alpha, u, \beta \in \Sigma^*,\ \alpha u\beta \in L,\ u \in L \implies \alpha\beta \in L.$$

3 Background from Group Theory

If G is a group and Σ is a finite set of symbols such that there is a surjective (monoid) homomorphism $\varphi : \Sigma^* \to G$, then we say that Σ is a *generating set* for G. Note that Σ is a *monoid generating set* for G as opposed to a *group generating set*; in the latter case, we have a set of symbols X and then let $\Sigma = X \cup X^{-1}$ where X^{-1} is a set of symbols in a (1-1) correspondence with X (and where we insist that $x^{-1}\varphi = (x\varphi)^{-1}$). In either case the *word problem* of the group G is the set of all words in Σ^* that represent the identity element of G.

A *presentation* for a group G is an expression of the form $\langle A : R \rangle$ where A is a generating set for G and R is a set of relations of the form $\alpha = \beta$. If A is a monoid generating set for G and $R \subseteq A^* \times A^*$, we have a *monoid presentation* for G and, if A is a group generating set for G, $\Sigma = A \cup A^{-1}$ as above, and $R \subseteq \Sigma^* \times \Sigma^*$ we have a *group presentation* for G. In each case the set R must be a set of *defining relations* for G: if $\approx$ is the congruence generated by R (together with all pairs of the form $(x^{-1}x, \epsilon)$ or (xx^{-1}, ϵ) with $x \in X$ in the case of a group generating set, where ϵ denotes the empty word), then G is isomorphic to $\Sigma^*/\approx$, i.e. $\alpha \approx \beta$ if and only if $\alpha\varphi = \beta\varphi$. The free group on a set X has the (group) presentation $\langle X : \emptyset \rangle$.

If P is any property of groups, then we say that a group is *virtually* P if it has a subgroup of finite index with property P. We will need the following fact:

Theorem 3. *Given a finite group presentation $\wp = \langle X : R \rangle$ and the promise that the group G presented by $\wp$ is virtually free, triviality of G is decidable.*

Proof. We start two processes running. The first enumerates the consequences of R, terminating if all the pairs (x, ϵ) with $x \in X$ have been output; this terminates if G is trivial. The second enumerates all the subgroups of finite index (one can do this for any finitely presented group; see [16] for example) and terminates if it finds a proper subgroup (any non-trivial virtually free group must possess such a subgroup). Eventually one of these two processes must terminate. □

4 Characterizing Word Problems

When examining groups based on their word problem as a formal language it is quite common to classify groups based on what class of languages their word problem lies in. However, there is no guarantee that the word problem will lie in the same class $\mathcal{F}$ of languages for different generating sets. The following result (see [8]) shows that, under certain mild assumptions on $\mathcal{F}$, this is not a problem:

Theorem 4. *If a class of languages $\mathcal{F}$ is closed under inverse homomorphism and the word problem of a group G lies in $\mathcal{F}$ with respect to some finite generating set then the word problem of G will lie in $\mathcal{F}$ for all finite generating sets.*

Anisimov [1] classified the groups with a regular word problem:

Theorem 5. *A finitely generated group has a regular word problem if and only if it is a finite group.*

Further work was done by Muller and Schupp [18] which, along with a result of Dunwoody [4], characterised the groups with a context-free word problem:

Theorem 6. *A finitely generated group G has a context-free word problem if and only if it is a virtually free group.*

One might ask what other families of languages $\mathcal{F}$ contained in the context-free languages give rise to interesting classes of groups. Herbst [7] showed that, if $\mathcal{F}$ satisfies certain natural closure conditions, then there are not many possibilities:

Theorem 7. *If $\mathcal{F}$ is a subset of the context-free languages closed under homomorphism, inverse homomorphism and intersection with regular languages then the class of finitely generated groups whose word problem lies in $\mathcal{F}$ is the class of groups with a regular word problem, the class of groups with a one-counter word problem or the class of groups with a context-free word problem.*

In the light of Theorems 5 and 6 it is natural to ask which groups have a one-counter word problem. Herbst characterised these groups in [7] (see also [10]):

Theorem 8. *A finitely generated group G has a one-counter word problem if and only if it is a virtually cyclic group.*

Given Theorem 7 it is natural to ask if one can decide if a language lying in one of these three families of languages is a word problem of a group and we answer this question in Section 7.

The following characterisation of word problems of groups was given in [19]:

Theorem 9. *A language L over an alphabet Σ is the word problem of a group with generating set Σ if and only if L satisfies the following two conditions:*

W1 *for all $\alpha \in \Sigma^*$ there exists $\beta \in \Sigma^*$ such that $\alpha\beta \in L$;*

W2 $\alpha u \beta \in L, u \in L \Rightarrow \alpha\beta \in L$.

Condition W1 says that the prefix closure $\text{prefix}(L)$ of L is Σ^*, i.e. that L has the universal prefix closure property, whereas condition W2 says that the language L is deletion closed. It is natural to ask, for the families of languages in Theorem 7, whether one can decide whether a language in that family satisfies these conditions and we will answer these questions in Sections 5 and 6 respectively.

5 Universal Prefix Closure and Decidability

In this section we investigate the decidability of the question $\text{prefix}(L) = \Sigma^*$. It is clear this is decidable if L is specified by means of a finite automaton:

Proposition 10. *The following problem is decidable:*

Input: *a finite automaton $N = (Q, \Sigma, \tau, s, A)$.*

Output: *"yes" if prefix$(L(N)) = \Sigma^*$; "no" otherwise.*

For example, given a finite state automaton N we construct the minimal deterministic automaton M accepting $L(N)$. As M has no unreachable states, $\text{prefix}(L(N)) = \Sigma^*$ if and only if M has no fail states, which is clearly decidable.

Remark 11. We are only interested in decidability questions in this paper and not with computational complexity. In our one non-trivial result about decidability (see Theorem 31) we do not have an easily computable time complexity (see Remark 32). For more information about the complexity of problems related to that described in Proposition 10 see [20] for example. □

When we consider the corresponding problem for one-counter languages we need the idea of a *counter machine*. There are several ways of describing these machines and we give one possibility here.

A counter machine M (as distinct from a one-counter automaton) is a two-tape machine. The first tape is the input tape; it is read only and the head can only move to the right. The second tape is a stack: whenever we move left, M erases the symbol it moved away from. There is only one stack symbol, a say. Intuitively M can only store a natural number (so that we can think of M as having an input tape and a counter). As we will see, the stack is never empty.

More formally, a *counter machine* is a sextuple $M = (Q, \Sigma, a, \delta, q_0, q_f)$ where Q is a finite set of states containing two distinguished states, q_0, the start state, and q_f, the final state. The input alphabet Σ is a finite set of symbols such that $a \notin \Sigma$. A *configuration* of M is a word of the form qa^n where $q \in Q$ and $n > 0$ (where the current state is q and the current stack contents are a^n).

We take C to be $\{1, 2, 3, 5, 7, \frac{1}{2}, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}\}$; there is no particular significance in our choice of 2, 3, 5 and 7, in that any four pair-wise coprime natural numbers would suffice. The transition relation δ is a function from $(Q - \{q_f\}) \times \Sigma \times C$ to $(Q - \{q_0\}) \times (Q - \{q_0\})$; the fact that δ is a function means that M is deterministic. M starts with just a on its stack (i.e. with the counter set to 1) and must set its counter to 1 again before entering q_f.

A move $(p, b, x, q, r) \in \delta$ is interpreted as follows. If M is in state p reading an input b and if the result of multiplying the current value n of the counter (i.e. we have a^n on the stack) by the value x is an integer, then we set the counter to xn and move to state q; if xn is not an integer then the counter remains set at n and M moves to state r. We write $pa^n \vdash qa^{xn}$ or $pa^n \vdash ra^n$ as appropriate.

Given a Turing Machine, one can effectively construct a counter machine accepting the same language (see [11] for example). We now turn to the computations of a counter machine:

Definition 12. *Let M be a counter machine. A* valid computation *of M is a word $C_0 C_1 \ldots C_n \in Q \cup \{a\}^*$ such that the C_i are configurations of M and*

$$C_0 = q_0 a \vdash C_1 \vdash \ldots \vdash C_{n-1} \vdash C_n = q_f a.$$

An invalid computation *is a word in $(Q \cup \{a\})^*$ which is not a valid computation.*

In any valid computation of M, any configuration qa^n will have $n = 2^b 3^c 5^d 7^e$ for some $b, c, d, e \geqslant 1$. Multiplying by 2, 3, 5 or 7 increases b, c, d or e by 1

and multiplying by $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$ or $\frac{1}{7}$ (if possible) decreases b, c, d or e by 1; so we effectively have four counters each of which can be increased or decreased. The fact that we can only multiply by x if nx is an integer is effectively saying that we can test each counter individually for zero (for example, if $n = 2^b3^c5^d7^e$ and we want to multiply by $\frac{1}{2}$, then we must have that $b > 0$).

Our aim is to show that the problem of deciding whether a language has the universal prefix property is undecidable for one-counter languages. In order to do this we need to relate the set of invalid computations of M to a one-counter language (i.e. a language accepted by a one-counter automaton as defined above). There have been other similar approaches to such problems (see [22] for example).

Proposition 13. *If $M = (Q, \Sigma, a, \delta, q_0, q_f)$ is a counter machine then the following language is a one-counter language:*

$K = \{qa^npa^j$: *the following conditions hold:*
if (q, b, k, p, r) is a quintuple of δ and kn is an integer then $kn \neq j$;
if (q, b, k, p, r) is a quintuple of δ and kn is not an integer then $j \neq n$}

Proof. Since δ is a finite set of quintuples (q, b, k, p, r) and the one-counter languages are closed under union, it is sufficient to show that the language

$$\{qa^npa^j : (kn \in \mathbb{Z} \Rightarrow kn \neq j) \wedge (kn \notin \mathbb{Z} \Rightarrow n \neq j)\}$$

is a one-counter language for any fixed quintuple (q, b, k, p, r). Now the condition

$$(kn \in \mathbb{Z} \Rightarrow kn \neq j) \wedge (kn \notin \mathbb{Z} \Rightarrow n \neq j)$$

is equivalent to

$$(kn \in \mathbb{Z} \wedge kn \neq j) \vee (kn \notin \mathbb{Z} \wedge n \neq j),$$

and (using again the fact that the one-counter languages are closed under union) we only need to show that the languages

$$\{qa^npa^j : kn \in \mathbb{Z} \wedge kn \neq j\}, \quad \{qa^npa^j : kn \notin \mathbb{Z} \wedge n \neq j\}$$

are both one-counter languages.

If $k \in \mathbb{N}$ then the condition $kn \in \mathbb{N}$ is automatically satisfied; if $k \notin \mathbb{N}$, then the condition $kn \in \mathbb{N}$ is equivalent to $n \bmod \frac{1}{k}$ being zero which we may check in the states of the machine. As far as $kn \neq j$ or $n \neq j$ is concerned this can be easily verified for any fixed k using the stack and the result follows. □

We now use Proposition 13 to prove the following result:

Theorem 14. *The following problem is undecidable:*
Input: *a one-counter automaton $N = (Q, \Sigma, \Gamma, \delta, s, A)$.*
Output: *"yes" if prefix$(L(N)) = \Sigma^*$; "no" otherwise.*

Proof. Let $M = (Q, \Sigma, a, \delta, q_0, q_f)$ be a counter machine and $\Gamma = Q \cup \{a\}$. The unique halting configuration of M is q_fa and the following language over Γ

$K = \{qa^npa^j$: the following conditions hold:
if (q, b, k, p, r) is a quintuple of δ and kn is an integer then $kn \neq j$;
if (q, b, k, p, r) is a quintuple of δ and kn is not an integer then $j \neq n\}$

is a one-counter language by Proposition 13. We consider the following languages:

(i) $L_1 = \Gamma^* - q_0 a \Gamma^*$. This is the set of all words in Γ^* which don't start with the unique initial configuration of M.
(ii) $L_2 = \Gamma^* - \Gamma^* q_f a \Gamma^*$. This is the set of all words not containing the unique halting configuration of M.
(iii) $L_3 = L_2 Q Q \Gamma^* \cup a\Gamma^*$. This is the set of all words which are badly formed (as a sequence of configurations of M) before the halting configuration of M appears (if it appears).
(iv) $L_4 = L_2 K \Gamma^*$. This is the set of words which contain two successive configurations where the second does not follow from the first before the halting configuration of M appears (if it appears).

L_1, L_2 and L_3 are regular and L_4 is a one-counter language since the one-counter languages are closed under concatenation. If $L = L_1 \cup L_2 \cup L_3 \cup L_4$ we see that L is a one-counter language as the one-counter languages are closed under union.

L is the set of all invalid computations α of M such that no prefix of α is a valid computation; so L is prefix-closed. We have $L = \Gamma^*$ precisely when M does not accept any input and so deciding if $L = \Gamma^*$ is equivalent to deciding if $L(M) = \emptyset$; as counter machines are Turing complete this is undecidable.

If one could decide, given a one-counter automaton $N = (Q, \Sigma, \Omega, \delta, s, A)$, if $\text{prefix}(L(N)) = \Sigma^*$, then one could decide if $L = \Gamma^*$. However, L is prefix-closed; so its prefix closure is equal to Γ^* precisely when L itself is equal to Γ^*. We have just pointed out that determining whether or not $L = \Gamma^*$ is undecidable. □

Remark 15. Given Theorem 14 it is immediate that the problem of deciding whether $\text{prefix}(L(N)) = \Sigma^*$ is undecidable for pushdown automata. Given our interest in word problems of groups, we have focussed on one-counter and context-free languages in this paper, but the argument used in Theorem 14 would easily apply to other classes of languages as well □

6 Deletion Closed Languages and Decidability

We now investigate the decidability of the question as to whether a language L is deletion closed. As with Proposition 10, we note that this is easily seen to be decidable if L is specified by means of a finite automaton:

Proposition 16. *The following problem is decidable:*
Input: *a finite automaton* $N = (Q, \Sigma, \tau.s, A)$.
Output: *"yes" if* $L(N)$ *is deletion closed; "no" otherwise.*

Proof. Given N we construct the minimal automaton P accepting $L = L(N)$ and then calculate the syntactic monoid M of L as the transition monoid of P.

Let φ be the natural map from Σ^* to M, so that $L = S\varphi^{-1}$ for some $S \subseteq M$. For each element $x \in M$ we can test whether or not $x\varphi^{-1} \in L$ (this is independent of the choice of $x\varphi^{-1}$), and so we may determine S. Since the condition that L is deletion closed, i.e. that $\alpha u \beta \in L, u \in L \Rightarrow \alpha\beta \in L$, is equivalent to

$$\alpha u \beta \in S, u \in S \Rightarrow \alpha\beta \in S,$$

and since the latter is clearly decidable (as S is a subset of a finite monoid M), we can decide whether or not L is deletion closed. □

It is clear that the technique used in the proof of Proposition 16 will apply to many other properties of regular languages.

We will use the following result from [9], known as *Higman's Lemma*, in what follows:

Theorem 17. *The set of finite words over a finite alphabet, as partially ordered by the subsequence relation, is well-quasi-ordered. This, in particular, implies that there does not exist an infinite sequence where the elements of the sequence are all pairwise incomparable or, equivalently, any set containing only pairwise incomparable finite words is finite.*

We will write $\alpha \prec \beta$ if α can be obtained by deleting some symbols in β, i.e. if α is a proper subsequence of β; if α is a (not necessarily proper) subsequence of β, we will write $\alpha \preceq \beta$. The following may be of some independent interest:

Proposition 18. *A language $L \subseteq \Sigma^*$ which is deletion closed and contains Σ is regular.*

Proof. Given that L is deletion closed and contains Σ, deleting any symbols from a word in L always results in another word in L; so, if $\alpha \in L$ and $\beta \prec \alpha$, then $\beta \in L$. If $L = \Sigma^*$ then the result is clearly true; so we will assume that $L \neq \Sigma^*$ in what follows.

First, consider words $\beta \notin L$ such that $\alpha \prec \beta \Rightarrow \alpha \in L$ (such words are guaranteed to exist since $\emptyset \neq L \neq \Sigma^*$). Given two such words γ and β we must have that $\gamma \not\prec \beta$ and that $\beta \not\prec \gamma$; so, by Theorem 17, the set U of all such words must be finite.

Consider the language $V = \{\alpha \in \Sigma^* : \exists\ \beta \in U \text{ such that } \beta \preceq \alpha\}$. This language is regular (as all we are doing is checking that a subsequence lies in a finite set).

If $\alpha \in V$ then there exists $\beta \in U$ such that $\beta \preceq \alpha$ and so $\alpha \notin L$ (as $\beta \notin L$).

Conversely, if $\alpha \notin L$, then choose β minimal such that $\beta \preceq \alpha$ and $\beta \notin L$. If $\gamma \prec \beta$, then $\gamma \in L$ by the minimality of β; so $\beta \in U$ and hence $\alpha \in V$.

Given this we see that $\Sigma^* - L = V$ is regular and hence L is regular. □

Remark 19. The hypothesis that L contains Σ in Proposition 18 is necessary; for example, the language $\{a^n b^n : n \geqslant 0\}$ is deletion closed but not regular. Indeed, since the word problem of any finitely generated group is deletion closed and there are finitely generated groups with unsolvable word problem, there exist deletion closed languages that are not even recursively enumerable. □

Recall that a language $L \subseteq \Sigma^*$ is said to be *bounded* if there exist non-empty words $w_1, w_2, \ldots, w_k$ in Σ^* such that $L \subseteq w_1^* w_2^* \ldots w_k^*$. It is known that the problem of deciding, given a context-free grammar G, if $L(G)$ is bounded is decidable (see Theorem 5.5.2 in [6] for example). Given this, we can now prove:

Theorem 20. *The following problem is undecidable:*
Input: a one-counter automaton M.
Output: "yes" if $L = L(M)$ is deletion closed; "no" otherwise.

Proof. We show that, if we could solve this decision problem, then we could solve the universe problem for alphabets of size 2, contradicting Theorem 1.

Let us assume that we have an algorithm determining whether or not $L(M)$ is deletion closed for a one-counter automaton $M = (Q, \Sigma, \Gamma, \tau, s, A)$ where $\Sigma = \{a, b\}$. First we note that, if the language $L = L(M)$ is not deletion closed, then it cannot be Σ^*. Our next observation is that, if L does not contain Σ (which we can test as membership is decidable for one-counter languages), then it also cannot be Σ^*; so in these cases we simply output "no" and terminate.

Now assume that L is deletion closed and contains Σ. Each non-empty word α in L is uniquely expressible in the form $a^{n(1)}b^{m(1)} \dots a^{n(\ell)}b^{m(\ell)}$ for some $\ell \geqslant 1$ where $n(1) \geqslant 0$, $n(i) > 0$ for $i > 1$, $m(i) > 0$ for $i < \ell$ and $m(\ell) \geqslant 0$; let us call this the *standard decomposition* for α and, given such a decomposition, let $\|\alpha\|$ denote ℓ. One of the following two possibilities must occur:

(i) there is a bound on $\|\alpha\|$ for $\alpha \in L$, i.e. there exists $k > 0$ such that every word of L has a standard decomposition $a^{n(1)}b^{m(1)} \dots a^{n(\ell)}b^{m(\ell)}$ with $\ell \leqslant k$;
(ii) there is no such bound on $\|\alpha\|$ and so, for any k, we have a word of the form $a^{n(1)}b^{m(1)} \dots a^{n(\ell)}b^{m(\ell)}$ in L with $\ell \geqslant k$. Given this, for every $k > 0$ there exists $\beta \in L$ such that $(ab)^k$ is a subsequence of β.

If possibility (i) occurs then L is bounded and $L \neq \Sigma^*$. If possibility (ii) occurs then, as every word in Σ^* is a subsequence of $(ab)^k$ for some k and L is deletion closed, we must have that $L = \Sigma^*$ in this case (and so L is not bounded).

So we test if L is bounded. If L is bounded then it is not Σ^* and we output "no" and, if L is not bounded, then $L = \Sigma^*$ and we output 'yes'. This gives us our contradiction. □

Remark 21. Given Theorem 20, it is immediate that the problem of deciding whether or not $L(M)$ is deletion closed is undecidable for pushdown automata. Given our interest in word problems of groups, we have focussed on one-counter and context-free languages in this paper, but the argument used in Theorem 20 would apply to other classes of languages as well.

If one were only interested in context-free languages then there are other approaches simpler than the one we have presented here. For example the property of a language being deletion closed distinguishes Σ^* from $\Sigma^* - \{w\}$ for any word w and one can build undecidability proofs from this based on the invalid computations of a Turing machine (see [11] for example). □

7 Word Problems and Decidability

We now turn our attention to word problems, i.e. those languages satisfying both the conditions W1 and W2 in Theorem 9. Given Theorem 9, together with Propositions 10 and 16, we immediately have the following result:

Proposition 22. *The following decision problem is decidable:*
Input: a finite automaton N.
Output: "yes" if $L(N)$ is the word problem of a group; "no" otherwise.

When we come to the one-counter languages, however, this problem becomes undecidable. This was shown for context-free languages in [17] and the argument used there extends to one-counter languages as well. This result does not follow immediately from Theorems 14 and 20; it is possible to have two undecidable problems whose conjunction is decidable. In order to prove the undecidability of this problem we need the concept of a *Hotz group* from [12]:

Definition 23. *The* Hotz group $H(G)$ *of a grammar $G = (V, \Sigma, P, S)$ is the group with presentation $\langle V \cup \Sigma : \{\alpha = \beta : (\alpha \to \beta) \in P\}\rangle$.*

Hotz showed that the group $H(G)$ for a reduced context-free grammar G depends only on $L(G)$. We also need the idea of a *collapsing group*:

Definition 24. *The* collapsing group $C(L)$ *of a language $L \subseteq \Sigma^*$ is the group with presentation $\langle \Sigma : \{\alpha = \beta : \alpha, \beta \in L\}\rangle$.*

The following connection between these two concepts will play a central role in what follows:

Definition 25. *A language $L \subseteq \Sigma^*$ is called a* language with Hotz isomorphism *if there exists a reduced grammar $G = (V, \Sigma, P, S)$ with $L = L(G)$ such that the collapsing group of L is isomorphic to $H(G)$.*

It is known [5] that all context-free languages are languages with Hotz isomorphism. In fact it is shown in [3] that:

Theorem 26. *A language $L \subseteq \Sigma^*$ is a language with Hotz isomorphism if and only if the collapsing group $C(L)$ is finitely presentable.*

Remark 27. The collapsing group $C(L)$ of a language $L \subseteq \Sigma^*$, where the empty word ϵ lies in L, will have every word in L representing the identity element of $C(L)$ but it may have other words representing the identity element as well.

Let $\wp$ denote the presentation $\langle \Sigma : \{\alpha = 1 : \alpha \in L\}\rangle$. If L is the word problem of some group K, then $\wp$ is a presentation for K and so K is isomorphic to $C(L)$. If L is not the word problem of a group then the word problem of the group with presentation $\wp$ must contain L as a proper subset.

In particular, if L is a context-free language which is the word problem of a group K, then $C(L)$ is isomorphic to K and we may obtain a finite presentation for K using the facts that K is isomorphic to $H(G)$ (where G is a context-free grammar generating L) and that the definition of $H(G)$ in Definition 23 is via a finite presentation. □

We are now in a position to prove our undecidability result:

Theorem 28. *The following decision problem is undecidable:*
Input: a one-counter automaton $N = (Q, \Sigma, \Gamma, \tau, s, A)$.
Output: "yes" if $L(N)$ is the word problem of a group; "no" otherwise.

Proof. Suppose we had an algorithm $\mathfrak{A}$ which could decide, given a one-counter automaton N, whether or not $L = L(N)$ were the word problem of some group G. We will show that one could then decide whether or not $L = \Sigma^*$ which is a contradiction by Theorem 1.

Since Σ^* is the word problem of the group $\{1\}$, if $\mathfrak{A}$ outputs "no", then we have that $L \neq \Sigma^*$. On the other hand, since L is context-free, if $\mathfrak{A}$ outputs "yes", then we know that the corresponding group G has a context-free word problem, and so G is virtually free by Theorem 6. We can now obtain a finite presentation $\wp$ for G as in Remark 27 and then use the presentation $\wp$ to test G for triviality as in Theorem 3. Since G is trivial if and only if $L = \Sigma^*$ we now have an algorithm for determining whether or not $L = \Sigma^*$, a contradiction. □

Remark 29. Since every one-counter language is context-free, in that a one-counter automaton is a special case of a pushdown automaton, it immediately follows from Theorem 28 that there is no algorithm to decide whether or not $L(M)$ is the word problem of a group for a pushdown automaton M (as proved in [17]). The proof given in Theorem 28 will, in fact, work for any family $\mathcal{F}$ of context-free languages where the universe problem is undecidable (provided that $\mathcal{F}$ is specified in such a way that a finite presentation for the Hotz group of any language L in $\mathcal{F}$ can be effectively determined). □

8 Deterministic Context-Free Languages

We saw in Theorem 6 that a group has a context-free word problem if and only if it is virtually free. It is not hard to show that the word problem of a virtually free group is deterministic context-free. So we have the following immediate consequence of Theorem 6:

Theorem 30. *If a group G has a context-free word problem, then it has a deterministic context-free word problem.*

However, despite the fact that it is undecidable whether or not a context-free language is the word problem of a group, this problem becomes decidable if the language is deterministic context-free and is given by a deterministic pushdown automaton:

Theorem 31. *The following decision problem is decidable:*
Input: *a deterministic pushdown automaton $M = (Q, \Sigma, \Gamma, \tau, s, A)$.*
Output: *"yes" if $L(M)$ is the word problem of a group; "no" otherwise.*

Proof. If $\epsilon \notin L = L(M)$ then L is not the word problem of a group; so we check first that $\epsilon \in L$ (outputting "no" if that is not the case); we will assume that $\epsilon \in L$ in what follows.

We convert our deterministic pushdown automaton to a reduced context-free grammar Γ such that $L(\Gamma) = L$ and then use the Hotz group construction in Remark 27 to write down a finite presentation $\wp$ of the group $G = H(\Gamma)$.

As in Remark 27, if L is the word problem of a group, then it must be the word problem of G. If W is the word problem of G with respect to the generating set Σ, then the question is whether $L = W$ (in which case L is the word problem of a group) or $L \subset W$ (in which case L is not the word problem of a group).

If L is the word problem of G then, as L is context-free, G must be virtually free. With this is mind, we start a process which we will refer to as Process 1.

Process 1 enumerates the finite-index subgroups of G and enumerates all presentations of the finite-index subgroups, checking each such presentation it generates to see if it is a natural presentation of a free group (i.e. a presentation with no relations). This is a semi-decision process; if G is virtually free, then we will eventually find such a presentation and so know that G is virtually free, but, if G is not virtually free, then this process will not terminate.

At the same time we start Process 2. Process 2 takes the finite presentation $\wp$ and enumerates the words in Σ^* representing the identity element, checking each one it generates for membership of L. If Process 2 ever finds a word which is trivial in the group G but not a member of L then we terminate all the running processes and output "no". (If L were the word problem of a group then it has to be the word problem of G, in which case no word trivial in G could lie outside L.) Process 2 is also a semi-decision process; we continue enumerating words whilst we do not have an output of "no".

Eventually one of these two processes must terminate. If it is Process 1 then we know that the group G is virtually free and we start Process 3. Process 3 uses the presentation $\wp$ of G and its finite-index free subgroup to construct a deterministic pushdown automaton N which accepts the word problem of G; we can then test N for equivalence with M by the theorem of Sénizergues in [21]. We halt all the processes and output the result of the equivalence test as our final output. Note that, if we reach Process 3, then Process 3 will always terminate.

Eventually either Process 2 terminates (and we output "no") or else Process 1 (and therefore Process 3) terminates. Thus we have an algorithm which outputs "yes" if $L(M)$ is the word problem of a group and outputs "no" if it is not, as required. □

Remark 32. As the reader will see, the use of the theorem of Sénizergues concerning the decidability of the equivalence problem for deterministic pushdown automata is a critical component of the proof of Theorem 31. We are also using procedures such as the enumeration of finite-index subgroups of a group searching for one that is a free group. As it is (in general) undecidable as to whether or not a finitely presented group is virtually free, this procedure will not necessarily terminate, and our proof relies on the fact that we can run this in parallel with another semi-decision procedure and that, given our situation, one of these two procedures must terminate. Given that the two procedures we have used will not have computable time complexity in general, we are not claiming any degree of efficiency for this decision procedure, merely that the problem is decidable. □

Acknowledgements. The authors would like to thank the referees for their careful reading of the paper and for their constructive suggestions. The second author would also like to thank Hilary Craig for all her help and encouragement.

References

1. Anisimov, V.A.: The group languages. Kibernetika 4, 18–24 (1971)
2. Berstel, J.: Transductions and Context-free Languages. Teubner (1979)
3. Diekert, V., Möbus, A.: Hotz-isomorphism theorems in formal language theory. Informatique Théorique et Applications 23, 29–43 (1989)
4. Dunwoody, M.J.: The accessibility of finitely presented groups. Inventiones Mathematicae 81, 449–457 (1985)
5. Frougny, C., Sakarovitch, J., Valkema, E.: On the Hotz-group of a context-free grammar. Acta Mathematica 18, 109–115 (1982)
6. Ginsburg, S.: The Mathematical Theory of Context-free Languages. McGraw-Hill (1966)
7. Herbst, T.: On a subclass of context-free groups. Informatique Théorique et Applications 25, 255–272 (1991)
8. Herbst, T., Thomas, R.M.: Group presentations, formal languages and characterizations of one-counter groups. Theoretical Computer Science 112, 187–213 (1993)
9. Higman, G.: Ordering by divisibility in abstract algebras. Proceedings of the London Mathematical Society 2, 326–336 (1952)
10. Holt, D.F., Owens, M.D., Thomas, R.M.: Groups and semigroups with a one-counter word problem. Journal of the Australian Mathematical Society 85, 197–209 (2008)
11. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
12. Hotz, G.: Eine neue invariante für kontext-freie sprachen. Theoretical Computer Science 11, 107–116 (1980)
13. Ibarra, O.H.: Restricted one-counter machines with undecidable universe problems. Mathematical Systems Theory 13, 181–186 (1979)
14. Ito, M.: Algebraic Theory of Automata & Languages. World Scientific Press (2004)
15. Ito, M., Kari, L., Thierrin, G.: Insertion and deletion closure of languages. Theoretical Computer Science 183, 3–19 (1997)
16. Johnson, D.L.: Presentations of Groups. Cambridge University Press (1990)
17. Lakin, S.R., Thomas, R.M.: Space complexity and word problems of groups. Groups-Complexity-Cryptology 1, 261–273 (2009)
18. Muller, D.E., Schupp, P.E.: Groups, the theory of ends, and context-free languages. Journal of Computer and System Sciences 26, 295–310 (1983)
19. Parkes, D.W., Thomas, R.M.: Groups with context-free reduced word problem. Communications in Algebra 30, 3143–3156 (2002)
20. Rampersad, N., Shallit, J., Xu, Z.: The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. Fundamenta Informaticae 116, 223–236 (2012)
21. Sénizergues, G.: $L(A) = L(B)$? Decidability results from complete formal systems. Theoretical Computer Science 251, 1–166 (2001)
22. Valk, R., Vidal-Naquet, G.: Petri nets and regular languages. Journal of Computer and System Sciences 23, 299–325 (1981)

Verification of Reachability Properties for Time Petri Nets

Kais Klai[1,2], Naim Aber[2], and Laure Petrucci[2]

[1] Institut TELECOM SudParis, CNRS UMR Samovar
9 rue Charles Fourier 91011 Evry, France
kais.klai@telecom-sudparis.eu
[2] LIPN, CNRS UMR 7030, Université Paris 13
99 avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
{naim.aber,kais.klai,laure.petrucci}@lipn.univ-paris13.fr

Abstract. This paper deals with verification of reachability properties on Time Petri Nets (TPN). TPNs allow the specification of real-time systems involving timing constraints explicitly. The main challenge of the analysis of such systems is to construct a finite abstraction of the corresponding (infinite) state graph preserving timed properties. Thus, we propose a new finite graph, called Timed Aggregate Graph (TAG), abstracting the behaviour of bounded TPNs with strong time semantics. The main feature the TAG compared to existing approaches is the encoding of the time information within the nodes of this graph. This allows to compute the minimum and maximum elapsed time in every path of the graph. The TAG preserves runs and reachable states of the corresponding TPN which allows for the verification of both event- and state-based properties.

Keywords: Time Petri Nets, Reachability properties, Model Checking.

1 Introduction

Time Petri nets are one of the most used formal models for the specification and the verification of systems where the explicit consideration of time is primordial. The main extensions of Petri nets with time are *time Petri nets* [14] and *timed Petri nets* [18]. In the first, a transition can fire within a time interval whereas, in the second, time durations can be assigned to the transitions; tokens are meant to spend that time as reserved in the input places of the corresponding transitions. Several variants of timed Petri nets exist: time is either associated with places (p-timed Petri nets), with transitions (t-timed Petri nets) or with arcs (a-timed Petri nets) [19]. The same holds for time Petri nets [7]. In [17], the authors prove that p-timed Petri nets and t-timed Petri nets have the same expressive power and are less expressive than time Petri nets. Several semantics have been proposed for each variant of these models. Here we focus on t-time Petri nets, which we simply call TPNs. There are two ways of letting the time elapse in a TPN [17]. The first way, known as the *Strong Time Semantics* (STS), is defined in such a manner that time elapsing cannot disable a transition. Hence, when the upper bound of a firing interval is reached, the transition must be fired. In contrast to that, the *Weak Time Semantics* (WTS) does not make any restriction on the elapsing of time.

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 159–170, 2013.

For real-time systems, dense time model (where time is considered in the domain $\mathbb{R}_{\geq 0}$) is the unique possible option, raising the problem of handling an infinite number of states. In fact, the set of reachable states of the TPN is generally infinite due to the infinite number of time successors a given state could have. Two main approaches are used to treat this state space: region graphs [1] and the state class approach [3]. The other methods [2,20,4,8,5,13,6,9] are either refinements or improvements or derived from these basic approaches. The objective of these representations is to yield a state-space partition that groups concrete states into sets of states with similar behaviour with respect to the properties to be verified. These sets of states must cover the entire state space and must be finite in order to ensure the termination of the verification process. In this work, we propose a new contribution for the abstraction and the verification of timed systems and especially those modelled by bounded TPNs.

This paper is organised as follows: In Section 2, some preliminaries about TPNs and the corresponding semantics are recalled. In Section 3, we define the Timed Aggregate Graph (TAG) associated with a TPN. In Section 4, we propose algorithms for the verification of some usual time properties based on TAGs. In Section 5, we discuss the experimental results obtained with our implementation compared to two well-known tools with respect to the size of the obtained abstraction size. Finally, a conclusion and some perspectives are given in Section 6.

2 Preliminaries and Basic Notations

A t-time Petri net (TPN for short) is a P/T Petri net [16] where a time interval $[t_{\min}; t_{\max}]$ is associated with each transition t.

Definition 1. *A* TPN *is a tuple* $\mathcal{N} = \langle P, T, Pre, Post, I \rangle$ *where:*

- $\langle P, T, Pre, Post \rangle$ *is a* P/T Petri net *where:*
  - P *is a finite set of places;*
  - T *is a finite set of transitions with* $P \cap T = \emptyset$ *;*
  - $Pre : T \longrightarrow \mathbb{N}^P$ *is the backward incidence mapping;*
  - $Post : T \longrightarrow \mathbb{N}^P$ *is the forward incidence mapping;*
- $I : T \longrightarrow \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$ *is the* time interval function *such that:* $I(t) = (t_{\min}, t_{\max})$*, with* $t_{\min} \leq t_{\max}$*, where* $t_{\min}$ *(resp.* $t_{\max}$*) is the earliest (resp. latest) firing time of transition t.*

A *marking* of a TPN is a function $m : P \longrightarrow \mathbb{N}$ where $m(p)$, for a place p, denotes the number of tokens in p. A *marked TPN* is a pair $\mathcal{N} = \langle \mathcal{N}_1, m_0 \rangle$ where $\mathcal{N}_1$ is a TPN and m_0 is a corresponding *initial marking*. A transition t is enabled by a marking m iff $m \geq Pre(t)$ and $Enable(m) = \{t \in T : m \geq Pre(t)\}$ denotes the set of enabled transitions in m. If a transition t_i is enabled by a marking m, then $\uparrow(m, t_i)$ denotes the set of newly enabled transitions [2]. Formally, $\uparrow(m, t_i) = \{t \in T \mid (m - Pre(t_i) + Post(t_i)) \geq Pre(t) \wedge (m - Pre(t_i)) < Pre(t)\}$. If a transition t is in $\uparrow(m, t_i)$, we say that t is newly enabled by the successor of m by firing t_i. Dually, $\downarrow(m, t_i) = \{t \in T \mid (m - Pre(t_i) + Post(t_i)) \geq Pre(t) \wedge (m - Pre(t_i)) \geq Pre(t)\}$ is the set of oldly enabled transitions. The possibly infinite set of reachable markings of $\mathcal{N}$ is denoted $Reach(\mathcal{N})$. If the set $Reach(\mathcal{N})$ is finite we say that $\mathcal{N}$ is bounded.

The semantics of TPNs can be given in terms of Timed Transition Systems (TTS) [12] which are usual transition systems with two types of labels: discrete labels for events (transitions) and positive real labels for time elapsing (delay). States of the TTS are pairs $s = (m, V)$ where m is a marking and $V : T \longrightarrow \mathbb{R}_{\geq 0} \cup \{\bot\}$ a time valuation. If a transition t is enabled in m then $V(t)$ is the elapsed time since t became enabled, otherwise $V(t) = \bot$. Given a state $s = (m, V)$ and a transition t, t is said to be firable in s iff $t \in Enable(m) \wedge V(t) \neq \bot \wedge t_{min} \leq V(t) \leq t_{\max}$.

Definition 2 (Semantics of a TPN). *Let $\mathcal{N} = \langle P, T, Pre, Post, I, m_0 \rangle$ be a marked TPN. The semantics of $\mathcal{N}$ is a TTS $\mathcal{S}_{\mathcal{N}} = \langle Q, s_0, \rightarrow \rangle$ where:*

1. *Q is a (possibly infinite) set of states*
2. *$s_0 = (m_0, V_0)$ is the initial state such that:*

$$\forall t \in T,\ V_0(t) = \begin{cases} 0 & \text{if } t \in Enable(m_0) \\ \bot & \text{otherwise} \end{cases}$$

3. *$\rightarrow \subseteq Q \times (T \cup \mathbb{R}_{\geq 0}) \times Q$ is the discrete and continuous transition relations:*
   (a) *the discrete transition relation:*
   $\forall t \in T :\ (m, V) \xrightarrow{t} (m', V')$ *iff:*

$$\begin{cases} t \in Enable(m) \wedge m' = m - Pre(t) + Post(t) \\ t_{min} \leq V(t) \leq t_{max} \\ \forall t' \in T : V'(t') = \begin{cases} 0 & \text{if } t' \in \uparrow(m, t) \\ V(t') & \text{if } t' \in \downarrow(m, t) \\ \bot & \text{otherwise} \end{cases} \end{cases}$$

   (b) *the continuous transition relation:* $\forall d \in \mathbb{R}_{\geq 0}$, $(m, V) \xrightarrow{d} (m', V')$ *iff:*

$$\begin{cases} \forall t \in Enable(m),\ V(t) + d \leq t_{max} \\ m' = m \\ \forall t \in T : \\ V'(t) = \begin{cases} V(t) + d & \text{if } t \in Enable(m); \\ V(t) & \text{otherwise.} \end{cases} \end{cases}$$

First, the delay transitions respect the STS semantics: an enabled transition must fire within its firing interval unless disabled by the firing of others. Second, a state change occurs either by the firing of transitions or by time elapsing: The firing of a transition may change the current marking while the time elapsing may make some new transitions firable.

Given a TPN $\mathcal{N}$ and the corresponding TTS $\mathcal{S}_{\mathcal{N}}$, a path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$, where $\alpha_i \in (T \cup \mathbb{R}_{\geq 0})$, is a run of $\mathcal{S}_{\mathcal{N}}$ iff $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$ for each $i = 0, 1, \ldots$. The length of a run π can be infinite and is denoted by $\mid \pi \mid$. Without loss of generality, we assume that for each non empty run $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ of a STS corresponding to a TPN, α_i and α_{i+1} are not both in $\mathbb{R}_{\geq 0}$. Then, π can be written, involving the reachable markings of $\mathcal{N}$, as $\pi = m_0 \xrightarrow{(d_1, t_1)} \ldots$ s.t. d_i is the time elapsed at marking m_{i-1} before firing t_i. In order to associate a run π of $\mathcal{S}_{\mathcal{N}}$ with a run of $\mathcal{N}$, denoted $\mathcal{P}(\pi)$, we define

the following projection function, where . denotes the concatenation operator between paths and π^i, for $i = 0, 1 \ldots$, denotes the suffix of π starting at state s_i.

$$\mathcal{P}(\pi) = \begin{cases} s_0.m & \text{if } \mid \pi \mid = 0 \\ s_0.m \xrightarrow{(0,\alpha_1)} . \mathcal{P}(\pi^1) & \text{if } \alpha_1 \in T \\ s_0.m \xrightarrow{(\alpha_1,\alpha_2)} . \mathcal{P}(\pi^2) & \text{if } \alpha_1 \in \mathbb{R}_{\geq 0} \wedge \mid \pi \mid \geq 2 \\ s_0.m \xrightarrow{\alpha_1} . \mathcal{P}(\pi^1) & \text{if } \alpha_1 \in \mathbb{R}_{\geq 0} \wedge \mid \pi \mid = 1 \end{cases}$$

In the following, we define, for a given finite run π of a TPN $\mathcal{N}$, the time elapsed before reaching (resp. firing) a marking (resp. a transition) belonging to this run.

Definition 3. *Let $\mathcal{N}$ be a TPN and let $\pi = m_0 \xrightarrow{(d_1,t_1)} \ldots \xrightarrow{(d_n,t_n)} m_n$ (resp. $\pi = m_0 \xrightarrow{(d_1,t_1)} \ldots \xrightarrow{(d_n,t_n)} m_n \xrightarrow{d_{n+1}}$) be a run of $\mathcal{N}$. The access (resp. firing) time of marking m (resp. transition t) in π, denoted $AT_\pi(m)$ (resp. $FT_\pi(t)$), is defined as follows:*

- $AT_\pi(m_0) = 0$
- $\forall 1 \leq i \leq n,\ AT_\pi(m_i) = FT_\pi(t_i) = \sum_{k=1}^{i} d_k$.

3 Timed Aggregate Graph

In this section, we propose to abstract the reachability state space of a TPN using a new graph called Timed Aggregate Graph (TAG) where nodes are called *aggregates* and are grouping sets of states of a TTS. The key idea behind TAGs is that time information is encoded inside aggregates. It includes the time the system is able to stay in the aggregate as well as a dynamic interval associated with each enabled transition. The first feature allows to encapsulate the delay transitions of the corresponding TTS (the arcs of a TAG are labeled with transitions of the corresponding TPN only), while the second allows to dynamically update the earliest and latest firing times of enabled transitions. It also allows to maintain the relative differences between the firing times of transitions.

Before we formally define the TAG and illustrate how the attributes of an aggregate are computed, let us first formally define aggregates.

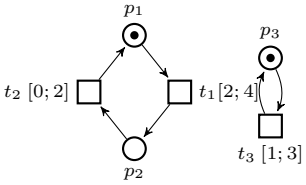
Definition 4 (Timed Aggregate). *A* timed aggregate *associated with a* TPN $\mathcal{N} = \langle P, T, Pre, Post, I \rangle$ *is a 4-tuple $a = (m, E, h, H)$, where:*

- *m is a marking*
- *$E = \{\langle t, \alpha_t, \beta_t \rangle \mid t \in Enable(m),\ \alpha_t \in (\mathbb{Z} \cup \{-\infty\}) \wedge \beta_t \in \mathbb{N} \cup \{+\infty\}\}$ is a set of enabled transitions each asssociated with two time values.*
- *$h = \min_{\langle t,\alpha_t,\beta_t \rangle \in E}(max(0, \alpha_t))$: the minimum time the system can stay in a*
- *$H = \min_{\langle t,\alpha_t,\beta_t \rangle \in E}(\beta_t)$: the maximum time the system can stay in a*

Each aggregate is characterised by three attributes that are computed dynamically: first, a marking m. Second, a set E of enabled transitions, each associated with two time values. For a given enabled transition t, α_t represents the minimum time the system should wait before firing t and β_t represents the maximum time the system can delay the firing of t. Note that, α_t can be negative which means that t can be fired immediately. Otherwise, α_t represents the earliest firing time of t. Starting from this aggregate, the

firing of t can occur between $max(0, \alpha_t)$ and β_t. In the rest of this paper, α_t will be abusively called the *dynamic* earliest firing time of t and β_t its *dynamic* latest firing time. Finally, the h and H attributes represent the minimum and the maximum time, respectively, the system can spend at the current aggregate.

Figure 1 illustrates an example of an aggregate. The associated marking is the marking of the left hand TPN. In this figure, we assume that this aggregate is the initial one. The h (resp. H) attribute corresponds to the minimum earliest firing time (resp. latest firing time) of the enabled transitions. Enabled transitions are associated with their static time intervals.



(a) Time Petri Net

$m = (1, 0, 1)$
$h = 1$
$H = 3$
$E = \{\langle t_1, 2, 4\rangle, \langle t_3, 1, 3\rangle\}$

(b) Aggregate a

Fig. 1. Example of aggregate

The TAG is a labeled transition system where nodes are timed aggregates. It has an initial aggregate, a set of actions (the set of transitions of $\mathcal{N}$) and a transition relation. The initial aggregate is easily computed by considering static information of the TPN.

Definition 5 (Timed Aggregate Graph). *A* TAG *associated with a TPN* $\mathcal{N} = \langle P, T, Pre, Post, I, m_0\rangle$ *is a tuple* $G = \langle \mathcal{A}, T, a_0, \delta\rangle$ *where:*

1. $\mathcal{A}$ *is a set of timed aggregates;*
2. $a_0 = \langle m_0, h_0, H_0, E_0\rangle$ *is the initial timed aggregate s.t.:*
   (a) m_0 *is the initial marking of* $\mathcal{N}$.
   (b) $h_0 = \min_{t \in Enable(m_0)}(t_{\min})$
   (c) $H_0 = \min_{t \in Enable(m_0)}(t_{\max})$
   (d) $E_0 = \{\langle t, t_{\min}, t_{\max}\rangle \mid t \in Enable(m_0)\}$
3. $\delta \subseteq \mathcal{A} \times T \times \mathcal{A}$ *is the transition relation such that:*
   for an aggregate $a = \langle m, h, H, E\rangle$*, a transition* t *s.t.* $\langle t, \alpha_t, \beta_t\rangle \in E$ *and an aggregate* $a' = \langle m', h', H', E'\rangle$*,* $(a, t, a') \in \delta$ *iff the following holds:*
   (a) $m' = m - Pre(t) + Post(t)$
   (b) $\alpha_t \leq H$
   (c) $\forall \langle t', \alpha_{t'}, \beta_{t'}\rangle \in E$,
   $t_{min} > t'_{max} \Rightarrow (t_{min} - \alpha_t) - (t'_{min} - \alpha_{t'}) \geq (t_{min} - t'_{max})$
   (d) $E' = E'_1 \cup E'_2$*, where:*
   - $E'_1 = \bigcup_{t' \in \uparrow(a,t)} \{\langle t', t'_{\min}, t'_{\max}\rangle\}$
   - $E'_2 = \bigcup_{t' \in \downarrow(a,t)} \{\langle t', \alpha_{t'} - H, \beta_{t'} - max(0, \alpha_t)\rangle\}$

   (e) $h' = \min_{\langle t', \alpha_{t'}, \beta_{t'}\rangle \in E'}(max(0, \alpha_{t'}))$
   (f) $H' = \min_{\langle t', \alpha_{t'}, \beta_{t'}\rangle \in E'}(\beta_{t'})$

Given an aggregate $a = \langle m, h, H, E \rangle$ and a transition $t \in T$, t is said to be enabled by a, denoted by $a \xrightarrow{t}$, iff: (1) $\exists (\alpha_t, \beta_t) \in (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{N} \cup \{+\infty\})$ s.t. $(t, \alpha_t, \beta_t) \in E$ and $\alpha_t \leq H$, and, (2) there is no other transition t', enabled by $a.m$, that should be fired before t. In fact, the first condition is not sufficient when t_{min} is greater than t'_{max} for some other transition t'. The firing of t from a leads to a new aggregate $a' = \langle m', h', H', E' \rangle$ whose attributes are computed as follows:

- The elements of E' are processed by taking the transitions that are enabled by m' and computing their earliest and latest firing times depending on their membership to $\uparrow (a, t)$ and $\downarrow (a, t)$. For each transition $t' \in Enable(a'.m)$, if t' is newly enabled, then its dynamic earliest and latest firing times are statically defined by t'_{min} and t'_{max} respectively. Otherwise, let $\langle t', \alpha_{t'}, \beta_{t'} \rangle \in a.E$ and $\langle t', \alpha'_{t'}, \beta'_{t'} \rangle \in a'.E$, then the maximum time elapsed by the system at $a.m$ (i.e., $a.H$) is subtracted from $\alpha_{t'}$ and the minium time is substracted from $\beta_{t'}$. Indeed, more the system can stay in $a.m$ less it can stay in $a'.m$ (and vice versa). Thus, the earliest firing time of t starting from a' is $max(0, \alpha_{t'} - a.H)$ while its latest firing time is $\beta_{t'} - max(0, \alpha_t)$.
- The computation of $a'.h$ (resp. $a'.H$) is ensured by taking the minimum of the dynamic earliest (resp. latest) firing time of enabled transitions.

According to Definition 5, the dynamic earliest firing time of a transition can decrease infinitely which could lead to an infinite state space TAG. Thus, an equivalence relation allowing to identify equivalent aggregates has been introduced in [11]. This equivalence relation is used in the construction of a TAG so that each newly built aggregate is not explored as long as an already built equivalent aggregate has been. Moreover, in [11], we have established that the TAG, built under this equivalence relation is finite when the corresponding TPN is bounded. We also demonstrated that the TAG is an exact representation of the reachability state space of a TPN. For each path in the TAG (resp. in the TPN) it is possible to find a path in the TPN (resp. TAG) involving the same sequence of transitions and where the time elapsed within a given state is between the minimum and the maximum stay time of the corresponding aggregate.

Figure 2 illustrates the TAG corresponding to the TPN of Figure 1.

For the verification of time properties, an abstraction-based approach should allow the computation of the minimum and maximum elapsed time over any path.

Definition 6. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta \rangle$ be the corresponding TAG. Let $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$ be a path in G.*

- *For each aggregate a_i (for $i = 0 \ldots n$), $MinAT_\pi(a_i)$ (resp. $MaxAT_\pi(a_i)$) denotes the minimum (resp. maximum) elapsed time between a_0 and a_i. In particular, $MinAT(a_0) = 0$ and $MaxAT(a_0) = a_0.H$.*
- *For each transition t_i (for $i = 1 \ldots n$), $MinFT_\pi(t_i)$ (resp. $MaxFT_\pi(t_i)$) denotes the minimum (resp. maximum) elapsed time before firing t_i.*

Proposition 1. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta \rangle$ be the corresponding TAG. Let $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$ be a path in G. We denote by α_{i_t} (resp. β_{i_t}) the dynamic earliest (resp. latest) firing time of a transition t at aggregate a_i, for $i = 1 \ldots n$. Then, $\forall i = 1 \ldots n$, the following holds:*

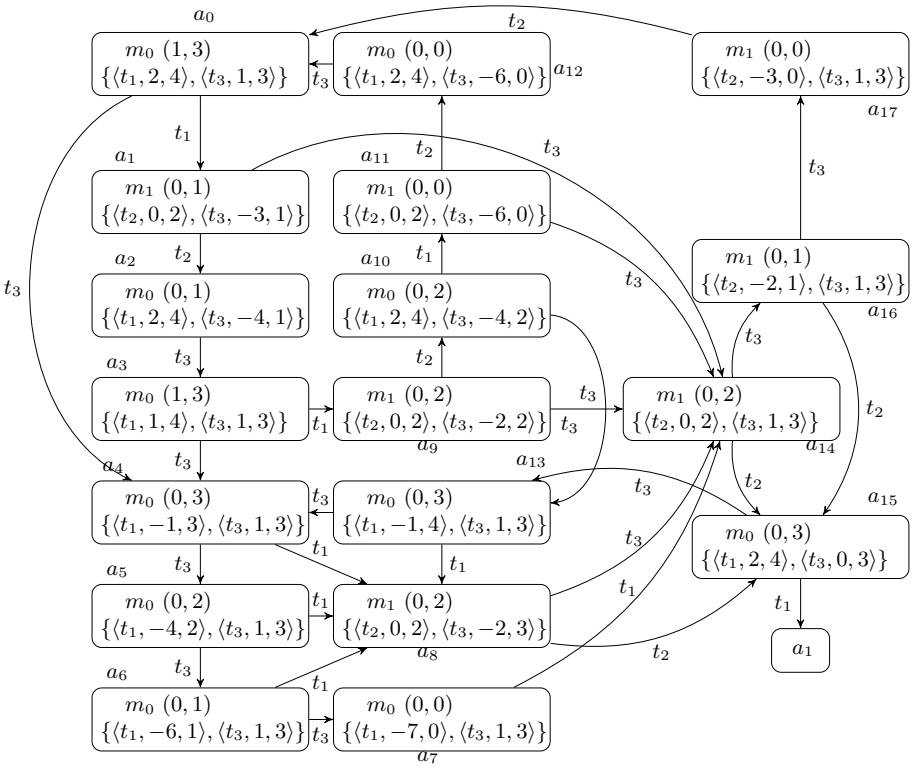


Fig. 2. The TAG of Fig. 1

- *Minimum and maximum access time*
  - $MinAT_\pi(a_i) = MinAT_\pi(a_{i-1}) + \max(0, \beta_{i-1_{t_i}} - (t_{i_{max}} - t_{i_{min}}))$
  - $MaxAT_\pi(a_i) = MaxAT(a_{i-1}) +$
    $\min(\min_{t\in\uparrow(a_{i-1},t_i)}(t_{\min}), \min_{t\in\downarrow(a_{i-1},t_i)}(\beta_{i-1_t} - a_{i-1}.H))$
- *Minimum and maximum firing time*
  - $MinFT_\pi(t_i) = MinAT_\pi(a_i)$
  - $MaxFT_\pi(t_i) = MaxAT_\pi(a_{i-1})$

4 Checking Time Reachability Properties

Our ultimate goal is to be able, by browsing the TAG associated with a TPN, to check timed reachability properties. For instance, we might be interested in checking whether some state-based property φ is satisfyied within a time interval $[d, D)$, with $d \in \mathbb{R}_{\geq 0}$ and $D \in (\mathbb{R}_{\geq 0} \cup \infty)$, starting from the initial marking. The following usual reachability properties belong to this category.

1. $\exists\Diamond_{[d;D]}\varphi$: There exists a path starting from the initial state, consuming between d and D time units and leading to a state that satisfies φ.

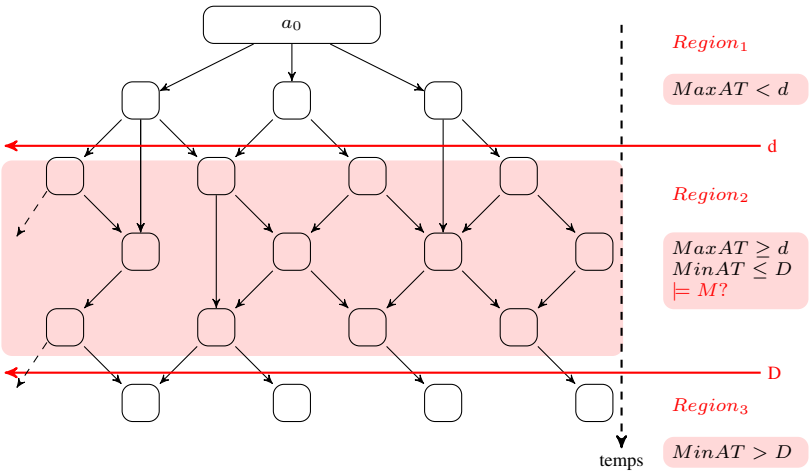


Fig. 3. Reachability analysis on the TAG

2. $\forall\Box_{[d;D]}\varphi$: For all paths starting from the initial state, all the states, that are reached after d and before D time units, satisfy φ.

3. $\forall\Diamond_{[d;D]}\varphi$: For all paths starting from the initial state, there exists a state in the path, reached after d and before D time units that satisfies φ.

4. $\exists\Box_{[d;D]}\varphi$: There exists a path from the initial state where all the states, that are reached after d and before D time units, satisfy φ.

Because of lack of space, we do not give the detailed algorithms for checking the above formulae, but give the main intuition.

In order to check any of the above properties, we propose on-th-fly approach where the TAG is represented as a tree which is partitioned into three regions (see. Figure 3). The first region ($Region_1$) contains the aggregates that are reachable strictly before d time units. The second region ($Region_2$) contains the aggregates that are reachable between d and D time units and the last region contains the aggregates that are reachable strictly after D time units. In case $D = \infty$ $Region_3$ is empty. By doing so, the verification algorithms behave as follows: only aggregates belonging to $Region_2$ are analysed with respect to φ. $Region_1$ must be explored in order to compute the maximal and minimum access time of the traversed aggregates, but $Region_3$ is never explored. In fact, as soon as an aggregate is proved to belong to $Region_3$ the exploration of the current path is stopped.

For instance checking the fromula number 1 is reduced to the search of an aggregate a in $Region_2$ that satisfies φ. As soon as such an aggregate is reached the checking algorithm stops the exploration and returns *true*. When, all the aggregates of $Region_2$ are explored (none satisfies φ) the checking algorithm return *false*. Dually, the formula number 2 is proved to be unsatisfied as soon as an aggregate in $Region_2$ that do not

satisfy φ is reached. When all the aggregates of $Region_2$ are explored (each satisfies φ) the checking algorithm return *true*.

Checking formulae number 3 and 4 is slightly more complicated. In fact, checking formula number 3 is reduced to check if, along any path in $Region_2$, there exists at least one aggregate satisfying φ. As soon as a path in $Region_2$ is completely explored without encountering an aggregate satisfying φ, the exploration is stopped and the checking algorithm returns *false*. Otherwise, it returns *true*. Finally, checking formula 4 is reduced to check that there exists a path in $Region_2$ such that all the aggregates belonging to this path satisfy φ. This formula is proved to be true as soon as such a path is found. Otherwise, when all the paths of $Region_2$ are explored (none satisfies the desired property), the checking algorithm returns *false*.

A similar approach can be trivially imagined for event-based approaches.

5 Experimental Results

The efficiency of the verification of timed reachability properties is closely linked with the size of the explored structure to achieve this verification. Thus, it was important to first check that the TAG is a suitable/reduced abstraction before performing verification on it. Our approach for building TAG-TPN was implemented in a prototype tool (written in C++), and used for experiments in order to validate the size of the graphs generated by the approach (note that the prototype was not optimised for time efficiency yet, therefore no timing figures are given in this section). All results reported in this section have been obtained on 2.8 gigahertz Intel with four gigabytes of RAM. The implemented prototype allowed us to have first comparison with existing approaches with respect to the size of obtained graphs. We used the TINA tool to build the SCGs, ROMEO tool for the ZBGs and our tool for the TAGs. We tested our approach on several TPN models and we report here the obtained results. The considered models are representative of the characteristics that may have a TPN, such as: concurrency, synchronisation, disjoint firing intervals and infinite firing bounds. The two first models (Figure 4(a) and Figure 4(b)) are two parametric models where the number of processes can be increased. In Figure 4(a), the number of self loops ($p_n \rightarrow t_n \rightarrow p_n$ is increased while in Figure 4(b) the number of processes, whose behavior is either local, by transition t_i, or global by synchronization with all the other processes, by transition t_0, is increased.

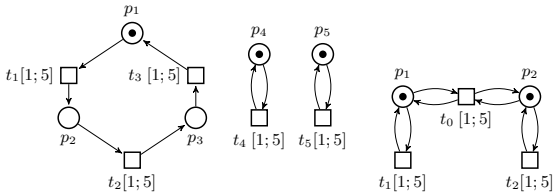
In addition to these two illustrative examples, we used two well known other parametric TPN models. The first one [10] represents a composition of producer/consumer models. The second (adapted from [15]) is the Fischer's protocol for mutual exclusion.

Table1 reports the results obtained with the SCG, the ZBG and the TAG-TPN approaches, in terms of graph size number of nodes/number of edges). The obtained results for the producers/consumers models show that the TAG yields better abstraction (linear order) than the SCG and the ZBG approaches. Each time a new module of producer/consumer is introduced, the size of graphs increases for all three approaches. However, the SAG achieves a better performance than the two other approaches. For the TPN of Figure 4(a), the obtained results show that the size of the TAG exponentially increases when the the parallelism occur in the structure of TPN. This is also the case also for the ZBG and the SCG methods, and we can see that our method behaves better

Table 1. Experimentation results

Parameters	SCG (with Tina) (nodes / arcs)	ZBG (with Romeo) (nodes / arcs)	TAG-TPN (nodes / arcs)
Nb. prod/cons	TPN model of producer/consumer		
1	34 / 56	34 / 56	34 / 56
2	748 / 2460	593 / 1 922	407 / 1 255
3	4 604 / 21891	3 240 / 15 200	1 618 / 6 892
4	14 086 / 83 375	9 504 / 56 038	3 972 / 20 500
5	31 657 / 217 423	20 877 / 145 037	8 175 / 48 351
6	61 162 / 471 254	39 306 / 311 304	15 157 / 99 539
7	107 236 / 907 708	67 224 / 594 795	26 113 / 186 363
8	175 075 / 1 604 319	107 156 / 1 044 066	42 503 / 324 600
9	270 632 / 2 655 794	161 874 / 1 718 104	66 103 / 534 055
10	400 648 / 4 175 413	234 398 / 2 687 147	99 036 / 839 011
Nb. self-loops	TPN example with concurrency (Figure 4(a))		
1	39 / 72	40 / 74	39 / 72
2	471 / 1 296	472 / 1 299	354 / 963
3	6 735 / 25 056	6 736 / 25 060	2 745 / 9 888
4	119 343 / 563 040	119 344 / 563 045	19 488 / 87 375
5	2 546 679 / 14 564 016	? / ?	130 911 / 701 748
Nb. processes	TPN example with synchronization (Figure 4(b))		
1	1 / 2	2 / 4	1 / 2
2	13 / 35	14 / 38	13 / 35
3	157 / 553	158 / 557	118 / 409
4	2 245 / 10 043	2 246 / 10 048	915 / 3 909
5	3 9781 / 21 7681	39 782 / 217 687	6 496 / 33 071
6	848 893 / 5 495 603	848 894 / 5 495 610	43 637 / 258 051
7	? / ?	? / ?	282 514 / 1.90282e+06
Nb. processes	Fischer protocol		
1	4 / 4	4 / 4	4 / 4
2	18 / 29	19 / 32	20 / 32
3	65 / 146	66 / 153	80 / 171
4	220 / 623	221 / 652	308 / 808
5	727 / 2 536	728 / 2 615	1 162 / 3 645
6	2 378 / 9 154	2 379 / 10 098	4 274 / 15 828
7	7 737 / 24 744	7 738 / 37 961	15 304 / 66 031
8	25 080 / 102 242	25 081 / 139 768	53 480 / 265 040

when we increment the self-loop structures in the model. The ZBG's and the SCG's execution have aborted due to a lack of memory when the number of self-loops was equal to 5. The number of edges of the obtained graphs follows the same proportion. In the synchronisation pattern exmaple, our approache behaves well as well. Indeed, with, 1, 2 and 3 processes, the sizes of the obtained graphs are almost similar with the three approaches. But, from 4 synchronised processes, the size of the SCGs and the ZBGs increase exponentially, leading to a state explosion with 7 processes, whereas the TAGs have been computed successfully with 7 processes (and even more). The Fischer protocol model is the only model where our approach leads to relatively bad results



(a) concurrency (b) synchronisation

Fig. 4. TPN models used in the experiments

(although the difference with the two other approaches is linear). Our first explanation is that, in case of disjoint firing intervals, the abstraction can be weak in some cases. In fact, when a transition t is enabled by an aggregate t and there exists a trasnition t', not enabled by a, s.t. $t_{min} > t'_{max}$, a is considered non equivalent (while it could be) to all aggregates where the earliest firing time of t is not the same. However, it could be that t and t' are never enabled simultanously. We think that taking into account some structural properties of the model could allow to refine our abstraction.

The experimental results show (in most cases) an important gain in performances in terms of graph size (nodes/arcs) compared to the SCG and the ZBG approaches for the tested examples. This promises performant verification approaches based on the TAG .

6 Conclusion

We proposed adapted algorithms for reachability analysis of time properties based on a new finite abstraction of the TPN state space. Unlike, the existing approaches, our abstraction can be directly useful to check both state and event-based logic properties. Our ultimate goal is to use the TAG traversal algorithm for the verification of timed reachability properties expressed in the *TCTL* logic. Several issues have to be explored in the future: We first have to implement and experiment our verification algorithms. Second, we believe that the size of the TAG can be further reduced while preserving time properties without necessarily preserving all the paths of the underlying TPN. We also plan to design and implement model checking algorithms for full *TCTL* logic.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Berthomieu, B., Diaz, M.: Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE Trans. Software Eng. 17(3), 259–273 (1991)
3. Berthomieu, B., Menasche, M.: An Enumerative Approach for Analyzing Time Petri Nets. In: IFIP Congress, pp. 41–46 (1983)
4. Berthomieu, B., Vernadat, F.: State Class Constructions for Branching Analysis of Time Petri Nets. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 442–457. Springer, Heidelberg (2003)

5. Berthomieu, B., Vernadat, F.: Time Petri Nets Analysis with TINA. In: QEST, pp. 123–124 (2006)
6. Boucheneb, H., Gardey, G., Roux, O.H.: TCTL Model Checking of Time Petri Nets. J. Log. Comput. 19(6), 1509–1540 (2009)
7. Boyer, M., Roux, O.H.: Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 63–82. Springer, Heidelberg (2007)
8. Gardey, G., Roux, O.H., Roux, O.F.: Using Zone Graph Method for Computing the State Space of a Time Petri Net. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 246–259. Springer, Heidelberg (2004)
9. Hadjidj, R., Boucheneb, H.: Improving state class constructions for CTL* model checking of time Petri nets. STTT 10(2), 167–184 (2008)
10. Hadjidj, R., Boucheneb, H.: On-the-fly TCTL model checking for time Petri nets. Theor. Comput. Sci. 410(42), 4241–4261 (2009)
11. Klai, K., Aber, N., Petrucci, L.: To appear in a new approach to abstract reachability state space of time petri nets. In: TIME, Lecture Notes in Computer Science. Springer (2013)
12. Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995)
13. Lime, D., Roux, O.H.: Model Checking of Time Petri Nets Using the State Class Timed Automaton. Discrete Event Dynamic Systems 16(2), 179–205 (2006)
14. Merlin, P.M., Farber, D.J.: Recoverability of modular systems. Operating Systems Review 9(3), 51–56 (1975)
15. Penczek, W., Pólrola, A., Zbrzezny, A.: SAT-Based (Parametric) Reachability for a Class of Distributed Time Petri Nets. T. Petri Nets and Other Models of Concurrency 4, 72–97 (2010)
16. Petri, C.A.: Concepts of net theory. In: MFCS 1973, pp. 137–146. Mathematical Institute of the Slovak Academy of Sciences (1973)
17. Pezzè, M., Young, M.: Time Petri Nets: A Primer Introduction. Tutorial Presented at the Multi-Workshop on Formal Methods in Performance Evaluation and Applications (1999)
18. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Cambridge, MA, USA (1974)
19. Sifakis, J.: Use of Petri nets for performance evaluation. Acta Cybern. 4, 185–202 (1980)
20. Yoneda, T., Ryuba, H.: CTL model checking of time Petri nets using geometric regions (1998)

Branching-Time Model Checking Gap-Order Constraint Systems

Richard Mayr and Patrick Totzke

University of Edinburgh, UK

Abstract. We consider the model checking problem for Gap-order Constraint Systems (GCS) w.r.t. the branching-time temporal logic CTL, and in particular its fragments **EG** and **EF**. GCS are nondeterministic infinitely branching processes described by evolutions of integer-valued variables, subject to Presburger constraints of the form $x - y \geq k$, where x and y are variables or constants and $k \in \mathbb{N}$ is a non-negative constant. We show that **EG** model checking is undecidable for GCS, while **EF** is decidable. In particular, this implies the decidability of strong and weak bisimulation equivalence between GCS and finite-state systems.

1 Introduction

Counter machines [Min67] extend a finite control-structure with unbounded memory in the form of counters that can hold arbitrarily large integers (or natural numbers), and thus resemble basic programming languages. However, almost all behavioral properties, e.g., reachability and termination, are undecidable for counter machines with two or more counters [Min67]. For the purpose of formal software verification, various formalisms have been defined that approximate counter machines and still retain the decidability of some properties. E.g., Petri nets model weaker counters that cannot be tested for zero, and have a decidable reachability problem [May84].

Gap-order constraint systems [Boz12, BP12] are another model that approximates the behavior of counter machines. They are nondeterministic infinitely branching processes described by evolutions of integer-valued variables, subject to Presburger constraints of the form $x - y \geq k$, where x and y are variables or constants and $k \in \mathbb{N}$ is a non-negative constant. Unlike in Petri nets, the counters can be tested for zero, but computation steps still have a certain type of monotonicity that yields a decidable reachability problem. In fact, control-state reachability is decidable even for the more general class of constrained multiset rewriting systems [AD06].

Previous Work. Beyond reachability, several model checking problems have been studied for GCS and related formalisms. The paper [Cer94] studies Integral Relational Automata (IRA), a model that is subsumed by GCS. It is shown that CTL model checking of IRA is undecidable in general, but the existential and universal fragments of CTL^* remain decidable for IRA. Termination and safety

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 171–182, 2013.

properties for GCS were studied in [Boz12, BP12]. In particular, LTL model checking is PSPACE complete for GCS. Moreover, model checking GCS is decidable for the logic $ECTL^*$, but undecidable for $ACTL^*$, which are the existential and universal fragments of CTL^* respectively. These fragments do not allow arbitrary nesting of path quantifiers and negation and are therefore orthogonal to **EF** and **EG**, which allow nesting of negation and operators EF/EG but forbid the respective other operator.

Our contribution. We study the decidability of model checking problems for GCS with fragments of computation-tree logic (CTL), namely **EF** and **EG**. While general CTL model checking of GCS is undecidable (even for the weaker model of IRA [Cer94]), we show that it is decidable for the logic **EF**. On the other hand, model checking is undecidable for the logic **EG**. An immediate consequence of our decidability result for **EF** is that strong and weak bisimulation equivalence are decidable between GCS and finite-state systems.

2 Gap-Order Constraint Systems

Let $\mathbb{Z}$ and $\mathbb{N}$ denote the sets of integers and non-negative integers. A *labeled transition system* (LTS) is described by a triple $T = (V, Act, \longrightarrow)$ where V is a (possibly infinite) set of states, Act is a finite set of action labels and $\longrightarrow \subseteq V \times Act \times V$ is the labeled transition relation. We use the infix notation $s \xrightarrow{a} s'$ for a transition $(s, a, s') \in \longrightarrow$, in which case we say T makes an *a-step* from s to s'. For a set $S \subseteq V$ of states and $a \in Act$ we define the set of a-predecessors by $Pre_a(S) = \{s' | s' \xrightarrow{a} s \in S\}$ and let $Pre^*(S) = \{s' | s' \longrightarrow^* s \in S\}$. A state in a LTS is often referred to as a *process*.

We fix a finite set Var of *variables* ranging over the integers and a finite set $Const \subseteq \mathbb{Z}$ of constants. Let Val denote the set of variable *evaluations* $\nu : Var \to \mathbb{Z}$. To simplify the notation, we will sometimes extend the domain of evaluations to constants, where they behave as the identity, i.e., $\nu(c) = c$ for all $c \in \mathbb{Z}$.

Definition 1 (Gap-Constraints). *A* gap clause *over* $(Var, Const)$ *is an inequation of the form*

$$(x - y \geq k) \tag{1}$$

where $x, y \in Var \cup Const$ *and* $k \in \mathbb{Z}$. *A clause is called* positive *if* $k \in \mathbb{N}$. *A (positive) gap* constraint *is a finite conjunction of (positive) gap clauses. A* gap formula *is an arbitrary boolean combination of gap clauses.*

An evaluation $\nu : Var \to \mathbb{Z}$ *satisfies the clause* $\mathcal{C} : (x - y) \geq k$ *(write* $\nu \models \mathcal{C}$*) if it respects the prescribed inequality. That is,*

$$\nu \models (x - y) \geq k \iff \nu(x) - \nu'(y) \geq k. \tag{2}$$

We define the satisfiability of arbitrary gap formulae inductively in the usual fashion and write $Sat(\varphi) = \{\nu \in Val \mid \nu \models \varphi\}$ *for the set of evaluations that satisfy the formula* φ. *In particular, an evaluation satisfies a gap constraint iff it satisfies all its clauses. A set* $S \subseteq Val$ *of evaluations is called* gap-definable *if there is a gap formula* φ *with* $S = Sat(\varphi)$.

We will consider processes whose states are described by evaluations and whose dynamics is described by stepwise changes in these variable evaluations, according to positive gap-order constraints.

Let $Var' = \{x' | x \in Var\}$ be the set of primed copies of the variables. These new variables are used to express constraints on how values can change when moving from one evaluation to another: x' is interpreted as the next value of variable x. A *transitional* gap-order clause (-constraint, -formula) is a gap-order clause (-constraint, -formula) with variables in $Var \cup Var'$.

For evaluations $\nu : Var \to \mathbb{Z}$ and $\nu' : Var \to \mathbb{Z}$ we define the combined evaluation $\nu \oplus \nu' : Var \cup Var' \to \mathbb{Z}$ of variables in $Var \cup Var'$ by

$$\nu \oplus \nu'(x) = \begin{cases} \nu(x), & \text{if } x \in Var \\ \nu'(x), & \text{if } x \in Var'. \end{cases} \tag{3}$$

Transitional gap-clauses can be used as conditions on how evaluations may evolve in one step. For instance, ν may change to ν' only if $\nu \oplus \nu' \models \varphi$ for some gap-clause φ.

Definition 2. *A* Gap-Order Constraint System *(GCS) is given by a finite set of transitional gap-clauses together with a labeling function. Formally, a GCS is a tuple* $\mathcal{G} = (Var, Const, Act, \Delta, \lambda)$ *where* $Var, Const, Act$ *are finite sets of variables, constants and action symbols,* Δ *is a finite set of* positive *transitional gap-order constraints over* $(Var, Const)$ *and* $\lambda : \Delta \to Act$ *is a labeling function. Its operational semantics is given by an infinite LTS with states Val where*

$$\nu \xrightarrow{a} \nu' \iff \nu \oplus \nu' \models \mathcal{C} \tag{4}$$

for some constraint $\mathcal{C} \in \Delta$ *with* $\lambda(\mathcal{C}) = a$*. For a set* $M \subseteq Val$ *of evaluations we write* $Pre_{\mathcal{C}}(M)$ *for the set* $\{\nu \mid \exists \nu' \in M. \nu \oplus \nu' \models \mathcal{C}\}$ *of* $\mathcal{C}$*-predecessors.*

Observe that a constraint $(x - 0 \geq 0) \ \wedge \ (0 - x \geq 0)$ is satisfied only by evaluations assigning value 0 to variable x. Similarly, one can test if an evaluation equates two variables. Also, it is easy to simulate a finite control in a GCS using additional variables.[1] What makes this model computationally non-universal is the fact that we demand *positive* constraints: while one can easily demand an increase or decrease of variable x by *at least* some offset $k \in \mathbb{N}$, one cannot demand a difference of *at most* k (nor exactly k).

Example 1. Consider the GCS with variables $\{x, y\}$ and single constant $\{0\}$ with two constraints $\Delta = \{\mathcal{C}X, \mathcal{C}Y\}$ for which $\lambda(\mathcal{C}X) = a$ and $\lambda(\mathcal{C}Y) = b$.

$$\mathcal{C}X = ((x - x' \geq 1) \wedge \ (y' - y \geq 0) \ \wedge \ (y - y' \geq 0) \ \wedge \ (x' - 0 \geq 0)) \tag{5}$$

$$\mathcal{C}Y = ((y - y' \geq 1) \ \wedge \ (x' - x \geq 0) \ \wedge \ (y' - 0 \geq 0)). \tag{6}$$

[1] In fact, [BP12, Boz12] consider an equivalent notion of GCS that explicitly includes a finite control.

This implements a sort of lossy countdown where every step strictly decreases the tuple (y, x) lexicographically: $\mathcal{C}X$ induces a-steps that decrease x while preserving the value of y and $\mathcal{C}Y$ induces b-steps that increase x arbitrarily but have to decrease y at the same time. The last clauses in both constraints ensure that x and y never change from a non-negative to a negative value.

In the sequel, we allow ourselves to abbreviate constraints for the sake of readability. For instance, the constraint $\mathcal{C}X$ in the previous example could equivalently be written as $(x > x' \geq 0) \ \wedge \ (y = y')$.

3 Branching-Time Logics for GCS

We consider (sublogics of) the branching-time logic CTL over processes defined by gap-order constraint systems, where atomic propositions are gap-clauses. The denotation of an atomic proposition $\mathcal{C} = (x - y \geq k)$ is $[\![\mathcal{C}]\!] = Sat(\mathcal{C})$, the set of evaluations satisfying this constraint. Well-formed CTL formulae are inductively defined by the following grammar, where $\mathcal{C}$ ranges over the atomic propositions and $a \in Act$ over the action symbols.

$$\psi ::= \mathcal{C} \mid true \mid \neg\psi \mid \psi \wedge \psi \mid \langle a \rangle \psi \mid EF\psi \mid EG\psi \mid E(\psi U \psi) \tag{7}$$

For the semantics, let $Paths^{\omega}(\nu_0)$ be the set of infinite derivations

$$\pi = \nu_0 \xrightarrow{a_0} \nu_1 \xrightarrow{a_1} \nu_2 \ldots \tag{8}$$

of $\mathcal{G}$ starting with evaluation $\nu_0 \in Val$ and $\pi(i) = \nu_i$ the ith evaluation ν_i on π. Similarly, we write $Paths^{*}(\nu_0)$ for the set of finite paths from ν_0 and let $Paths(\nu_0) = Paths^{\omega}(\nu_0) \cup Paths^{*}(\nu_0)$. The denotation of composite formulae is defined in the standard way.

$$[\![\mathcal{C}]\!] = Sat(\mathcal{C}) \tag{9}$$
$$[\![true]\!] = Val \tag{10}$$
$$[\![\neg\psi]\!] = Val \setminus [\![\psi]\!] \tag{11}$$
$$[\![\psi_1 \wedge \psi_2]\!] = [\![\psi_1]\!] \cap [\![\psi_2]\!] \tag{12}$$
$$[\![\langle a \rangle \psi]\!] = Pre_a([\![\psi]\!]) \tag{13}$$
$$[\![EF\psi]\!] = \{\nu \mid \exists \pi \in Paths^{*}(\nu).\ \exists i \in \mathbb{N}.\ \pi(i) \in [\![\psi]\!]\} \tag{14}$$
$$[\![EG\psi]\!] = \{\nu \mid \exists \pi \in Paths^{\omega}(\nu).\ \forall i \in \mathbb{N}.\ \pi(i) \in [\![\psi]\!]\} \tag{15}$$
$$[\![E(\psi_1 U \psi_2)]\!] = \{\nu \mid \exists \pi \in Paths(\nu).\ \exists i \in \mathbb{N}.\ \pi(i) \in [\![\psi_2]\!] \wedge \forall j < i.\pi(j) \in [\![\psi_1]\!]\} \tag{16}$$

We use the usual syntactic abbreviations $false = \neg true$, $\psi_1 \vee \phi_2 = \neg(\neg\psi_1 \wedge \neg\phi_2)$.

The sublogics **EF** and **EG** are defined by restricting the grammar (7) defining well-formed formulae: **EG** disallows subformulae of the form $E(\psi_1 U \psi_2)$ and $EF\psi$ and in **EF**, no subformulae of the form $E(\psi_1 U \psi_2)$ or $EG\psi$ are allowed. The *Model Checking Problem* is the following decision problem.

INPUT: A GCS $G = (Var, Const, Act, \Delta, \lambda)$, an evaluation $\nu : Var \to \mathbb{Z}$ and a formula ψ.
QUESTION: $\nu \models \psi$?

Cerans [Cer94] showed that general CTL model checking is undecidable for gap-order systems. This result holds even for restricted CTL without *next* operators $\langle a \rangle$. In the following section we show a similar undecidability result for the fragment **EG**. On the other hand, model checking GCS with the fragment **EF** turns out to be decidable; cf. Section 5.

4 Undecidability of EG Model Checking

Theorem 1. *The model checking problem for* ***EG*** *formulae over GCS is undecidable.*

Proof. By reduction from the halting problem of deterministic 2-counter Minsky Machines (2CM). 2-counter machines consist of a deterministic finite control, including a designated halting state *halt*, and two integer counters that can be incremented and decremented by one and tested for zero. Checking if such a machine reaches the halting state from an initial configuration with control-state *init* and counter values $x_1 = x_2 = 0$ is undecidable [Min67].

Given a 2CM M, we will construct a GCS together with an initial evaluation ν_0 and a **EG** formula ψ such that $\nu_0 \models \psi$ iff M does not halt.

First of all, observe that we can simulate a finite control of n states using one additional variable *state* that will only ever be assigned values from 1 to n. To do this, let $[p] \leq n$ be the index of state p in an arbitrary enumeration of the state set. Now, a transition $p \longrightarrow q$ from state p to q introduces the constraint $(state = [p] \wedge state' = [q])$. We will abbreviate such constraints by $(p \longrightarrow q)$ in the sequel and simply write p to mean the clause $(state = [p])$.

We use two variables x_1, x_2 to act as integer counters. Zero-tests can then directly be implement as constraints $(x_1 = 0)$ or $(x_2 = 0)$. It remains to show how to simulate increments and decrements by exactly 1. Our GCS will use two auxiliary variables y, z and a new state *err*. We show how to implement increments by one; decrements can be done analogously.

Consider the x_1-increment $p \overset{x_1 = x_1 + 1}{\longrightarrow} q$ that takes the 2CM from state p to q and increments the counter x_1. The GCS will simulate this in two steps, as depicted in Figure 1 below. The first step can arbitrarily increment x_1 and will remember (in variable y) the old value of x_1. The second step does not change any values and just moves to the new control-state. However, incrementing by more than one in the first step enables an extra move to the error state *err* afterwards. This error-move is enabled if one can assign a value to variable z that is strictly in between the old and new value of x_1, which is true iff the increment in step 1 was not faithful. The incrementing transition of the 2CM is thus translated to the following three constraints.

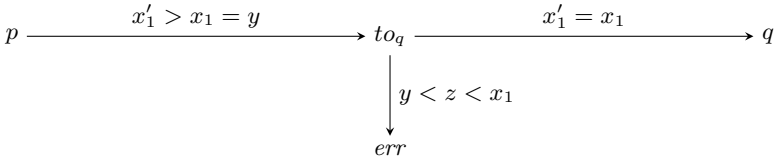


Fig. 1. Forcing faithful simulation of x_1-increment. All steps contain the additional constraint $x_2' = x_2$, which is not shown, to preserve the value of the other counter x_2.

$$(p \longrightarrow to_q) \wedge (x_1' > x_1 = y) \wedge (x_2' = x_2) \tag{17}$$
$$(to_q \longrightarrow q) \wedge (x_1' = x_1) \wedge (x_2 = x_2) \tag{18}$$
$$(to_q \longrightarrow err) \wedge (y < z < x_1). \tag{19}$$

If we translate all operations of the 2CM into the GCS formalism as indicated above, we end up with an overapproximation of the 2CM that allows runs that faithfully simulate runs in the 2CM but also runs which 'cheat' and possibly increment or decrement by more than one and still don't go to state *err* in the following step.

We enforce a faithful simulation of the 2CM by using the formula that is to be checked, demanding that the error-detecting move is never enabled. The GCS will only use a unary alphabet $Act = \{a\}$ to label constraints. In particular, observe that the formula $\langle a \rangle err$ holds in every configuration which can move to state *err* in one step. Now, the **EG** formula

$$\phi = EG(\neg halt \ \wedge \ \neg \ \langle a \rangle err) \tag{20}$$

asserts that there is an infinite path which never visits state *halt* and along which no step to state *err* is ever enabled. This means φ is satisfied by evaluation $\nu_0 = \{state = [init], x_1 = x_2 = y = z = 0\}$ iff there is a faithful simulation of the 2CM from initial state *init* with both counters set to 0 that never visits the halting state. Since the 2CM is deterministic, there is only one way to faithfully simulate it and hence $\nu_0 \models \psi$ iff the 2CM does not halt. □

5 Decidability of EF Model Checking

Let us fix sets *Var* and *Const* of variables and constants, respectively. We will use an alternative characterization of gap-constraints called *monotonicity graphs* (MG) which are finite graphs with nodes $Var \cup Const$.[2]

Monotonicity graphs can be used to represent sets of evaluations. We show that so represented sets are effectively closed under all logical connectors allowed in **EF**, and one can thus evaluate a formula bottom up.

[2] These were called *Graphose Inequality Systems* in [Cer94].

Definition 3 (Monotonicity Graphs). *A* monotonicity graph *(MG) over* $(Var, Const)$ *is a finite, directed graph* $M = (V, E)$ *with nodes* $V = Var \cup Const$ *and edges labeled by elements of* $\mathbb{Z} \cup \{-\infty, \infty\}$.

An evaluation $\nu : Var \to \mathbb{Z}$ *satisfies* M *if for every edge* $(x \xrightarrow{k} y)$ *it holds that* $\nu(x) - \nu(y) \geq k$. *We write* $\nu \models M$ *in this case and let* $Sat(M)$ *denote the set of evaluations satisfying* M.

Let $M(x, y) \in \{-\infty, \infty\} \cup \mathbb{Z}$ *denote the least upper bound of the sums of the labels on any path from node* x *to node* y. *The* closure $|M|$ *is the unique complete monotonicity graph with edges* $x \xrightarrow{M(x,y)} y$ *for all* $x, y \in Var \cup Const$.

The degree *of* M *is the smallest* $K \in \mathbb{N}$ *such that* $k = -\infty$ *or* $k \geq -K$ *for all edge labels* k *in* M. *It is the negation of the smallest finite negative label or* 0 *if no such label exists.*

The following lemma states some basic properties of monotonicity graphs that can easily be verified.

Lemma 1.

1. $Sat(M) = \emptyset$ *for any monotonicity graph* M *that contains an edge labeled by* ∞ *or some cycle with positive weight sum.*
2. $|M|$ *is polynomial-time computable from* M *and* $Sat(M) = Sat(|M|)$.
3. *If we fix sets* Var, $Const$ *of variables and constants then for any gap-constraint* $\mathcal{C}$ *there is a unique MG* $M_{\mathcal{C}}$ *containing an edge* $x \xrightarrow{k} y$ *iff there is a clause* $x - y \geq k$ *in* $\mathcal{C}$, *for which* $Sat(M_{\mathcal{C}}) = Sat(\mathcal{C})$.

The last point of this lemma states that monotonicity graphs and gap-constraints are equivalent formalisms. We thus talk about *transitional* monotonicity graphs over $(Var, Const)$ as those with nodes $Var \cup Var' \cup Const$ and call a MG *positive* if it has degree 0. We further define the following operations on MG.

Definition 4. *Let* M, N *be monotonicity graphs over* Var, $Const$ *and* $V \subseteq Var$.

- *The* restriction $M|_V$ *of* M *to* V *is the maximal subgraph of* M *with nodes* $V \cup Const$.
- *The* projection $Proj(M, V) = |M|_V$ *is the restriction of* M*'s closure to* V.
- *The* intersection $M \otimes N$ *is the MG that contains and edge* $x \xrightarrow{k} y$ *if* k *is the maximal label of any edge from* x *to* y *in* M *or* N.
- *The* composition $G \circ M$ *of a* transitional *MG* G *and* M *is obtained by consistently renaming variables in* M *to their primed copies, intersecting the result with* G *and projecting to* $Var \cup Const$. $G \circ M := Proj(M_{[Var \mapsto Var']} \otimes G, Var)$.

These operations are surely computable in polynomial time. The next lemma states important properties of these operations; see also [Cer94, BP12].

Lemma 2.

1. $Sat(Proj(M, V)) = \{\nu|_V : \nu \in Sat(M)\}$.
2. $Sat(M \otimes N) = Sat(M) \cap Sat(N)$

3. $Sat(G \circ M) = \{\nu \mid \exists \nu' \in Sat(M).\ \nu \oplus \nu' \in Sat(G)\} = Pre_G(M)$.
4. *If M has degree n and G is a transitional MG of degree 0, then $G \circ M$ has degree $\leq n$.*

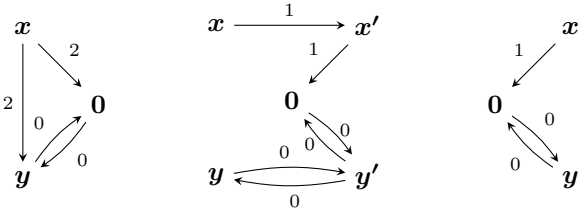
We will use monotonicity graphs to finitely represent sets of evaluations. To that end, let us call a set $S \subseteq Val$ *MG-definable* if there is a finite set $\{M_0, M_1, \ldots, M_k\}$ of MG such that

$$S = \bigcup_{0 \leq i \leq k} Sat(M_i). \tag{21}$$

Call S MG^n-definable if there is such a set of MG with degree $\leq n$.

Example 2. The monotonicity graph on the left below corresponds to the contraint $\mathcal{C}X$ in Example 1. On the right we see its closure (where edges labeled by $-\infty$ are omitted). Both have degree 0.

Let us compute the $\mathcal{C}X$-predecessors of the set $S = \{\nu \mid \nu(x) > \nu(y) = 0\}$ which is characterized by the single MG on the right below.



If we rename variables x and y to x' and y' and intersect the result with $M_{\mathcal{C}X}$ we get the MG in the middle. We project into $Var \cup Const$ by computing the closure and restricting the result accordingly. This leaves us with the MG on the left, which characterizes the set $Pre_{\mathcal{C}X}(S) = \{\nu \mid \nu(x) \geq 2 \ \wedge \ \nu(y) = 0\}$ as expected.

We have seen how to construct a representation of the $\mathcal{C}$-predecessors $Pre_{\mathcal{C}}(S)$ and thus $Pre_a(S)$ for MG-definable set S, gap-constraints $\mathcal{C}$ and actions $a \in Act$. The next lemma is a consequence of Lemma 1, point 3 and asserts that we can do the same for complements.

Lemma 3. *The class of MG-definable sets is effectively closed under complements.*

Proof. By Lemma 1 we can interpret a set $\mathcal{M} = \{M_0, M_1, \ldots, M_k\}$ as gap-formula in DNF. One can then use De Morgan's laws to propagate negations to atomic propositions, which are gap-clauses $x - y \geq k$ for which the negation is expressible as gap-clause $y - x \geq -(k+1)$. It remains to bring the formula into DNF again, which can be interpreted as set of MG. □

Observe that complementation potentially constructs MG with increased degree. This next degree is bounded by the largest finite edge-label in the current graph plus one, but nevertheless, an increase of degree cannot be avoided. The classes of MG^n-definable sets are therefore not closed under complement.

Example 3. The set $S = \{\nu \mid \nu(x) > \nu(y) = 0\}$ from the previous example corresponds to the gap-formula $\varphi_S = (x - 0 \geq 1) \wedge\ (0 - y \geq 0)\ \wedge\ (y - 0 \geq 0)$. Its complement is characterized by the set $S^c = \{(0 \xrightarrow{-2} x), (y \xrightarrow{-1} 0), (0 \xrightarrow{-1} y)\}$, which contains a MG of degree 2.

It remains to show that we can compute $Pre^*(S)$ for MG-definable sets S. We recall [Cer94] the following partial ordering on monotonicity graphs and its properties.

Definition 5. *Let M, N be MG over $(Var, Const)$. We say M* covers *N (write $N \sqsubseteq M$) if for all $x, y \in Var \cup Const$ it holds that $N(x, y) \leq M(x, y)$.*

Lemma 4.

1. *If $N \sqsubseteq M$ then $Sat(N) \supseteq Sat(M)$.*
2. *$\sqsubseteq$ is a WQO on $\mathbf{MG}^n$ for every fixed $n \in \mathbb{N}$.*

Note that point 1 states that a $\sqsubseteq$-bigger MG is more restrictive and hence has a smaller denotation. Also note that $\sqsubseteq$ is **not** a well ordering on the set of all MG, because edges may be labeled with arbitrary integers (and hence ever smaller negative ones).

Lemma 5. *Let S be a MG^n-definable set of evaluations. Then $Pre^*(S)$ is MG^n-definable and a representation of $Pre^*(S)$ can be computed from a representation of S.*

Proof. It suffices to show the claim for a set characterized by a single monotonicity graph M because $Pre^*(S \cup S') = Pre^*(S) \cup Pre^*(S')$. Assume that M has degree n.

We proceed by exhaustively building a tree of MG, starting in M. For every node N we compute children $G \circ N$ for all of the finitely many transitional MG G in the system. Point 4) of the Lemma 2 guarantees that all intermediate representations have degree $\leq n$. By Lemma 4, point 2, any branch eventually ends in a node that covers a previous one and Lemma 4, point 1 allows us to stop exploring such a branch. We conclude that $Pre^*(M)$ can be characterized by the finite union of all intermediate MG. □

Finally, we are ready to prove our main result.

Theorem 2. ***EF*** *Model checking is decidable for Gap-order constraint systems. Moreover, the set* $[\![\psi]\!]$ *of evaluations satisfying an* ***EF****-formula* ψ *is effectively gap-definable.*

Proof. We can evaluate a formula bottom up, representing the sets satisfying subformulae by finite sets of MG. Atomic propositions are either *true* or gap-clauses and can thus be written directly as MG. For composite formulae we use the properties that gap-definable sets are effectively closed under intersection (Lemma 2) and negation (Lemma 3), and that we can compute representations of $Pre_a(S)$ and $Pre^*(S)$ for MG-definable sets S by Lemmas 2 and 5.

The key observation is that although negation (i.e., complementing) may increase the degree of the intermediate MG, this happens only finitely often in the bottom up evaluation of an **EF** formula. Computing representations for modalities $\langle a \rangle$ and EF does not increase the degree. □

6 Applications

We consider labeled transition systems induced by GCS. In a weak semantics, one abstracts from non-observable actions modeled by a dedicated action $\tau \in Act$. The *weak step* relation $\Longrightarrow$ is defined by

$$\stackrel{\tau}{\Longrightarrow} = \stackrel{\tau}{\longrightarrow}^*, \text{ and for } a \neq \tau, \qquad \stackrel{a}{\Longrightarrow} = \stackrel{\tau}{\longrightarrow}^* \cdot \stackrel{a}{\longrightarrow} \cdot \stackrel{\tau}{\longrightarrow}^*.$$

Bisimulation and weak bisimulation are semantic equivalences in van Glabbeeks linear time – branching time spectrum [Gla01], which are used to compare the behavior of processes. Their standard co-inductive definition is as follows.

Definition 6. *A binary relation* $R \subseteq V^2$ *on the states of a labeled transition system is a* bisimulation *if* sRt *implies that*

1. *for all* $s \stackrel{a}{\longrightarrow} s'$ *there is a* t' *such that* $t \stackrel{a}{\longrightarrow} t'$ *and* $s'Rt'$*, and*
2. *for all* $t \stackrel{a}{\longrightarrow} t'$ *there is a* s' *such that* $s \stackrel{a}{\longrightarrow} s'$ *and* $s'Rt'$*.*

Similarly, R *is a* weak bisimulation *if in both conditions above* $\longrightarrow$ *is replaced by* $\Longrightarrow$*. (Weak) bisimulations are closed under union, so there exist unique maximal bisimulation* $\sim$ *and weak bisimulation* $\approx$ *relations, which are equivalences on* V*.*

Let the maximal (weak) bisimulation between two LTS with state sets S *and* T *be the maximal (weak) bisimulation in their union projected into* $(S \times T) \cup (T \times S)$*.*

The *Equivalence Checking Problem* is the following decision problem.

INPUT: Given LTS $T_1 = (V_1, Act, \longrightarrow)$ and $T_2 = (V_2, Act, \longrightarrow)$, states $s \in V_1$ and $t \in V_2$ and an equivalence R.
QUESTION: sRt?

In particular, we are interested in checking strong and weak bisimulation between processes of GCS and finite systems. Note that the decidability of weak bisimulation implies the decidability of the corresponding strong bisimulation as $\sim$ and $\approx$ coincide for LTS without τ labels.

We recall (see e.g. [KJ06, JKM98]) that finite systems admit characteristic formulae up to weak bisimulation in **EF**.

Theorem 3. *Let $T_1 = (V_1, Act, \longrightarrow)$ be an LTS with finite state set V_1 and $T_2 = (V_2, Act, \longrightarrow)$ be an arbitrary LTS. For every state $s \in V_1$ one can construct an **EF**-formula ψ_s such that $t \approx s \iff t \models \psi_s$ for all states $t \in V_2$.*

The following is a direct consequence of Theorems 3 and 2.

Theorem 4. *For every GCS $\mathcal{G} = (Var, Const, Act, \Delta, \lambda)$ and every LTS $T = (V, Act, \longrightarrow)$ with finite state set V, the maximal bisimulation $\approx$ between $T_{\mathcal{G}}$ and T is effectively gap-definable.*

Proof. By Theorems 3 we can compute, for every state s of T, a characteristic formula ψ_s that characterizes the set of evaluations $\{\nu \mid \nu \approx s\} = [\![\psi_s]\!]$. By Theorem 2 these sets are MG- and thus gap-definable. Since the class of gap-definable sets is effectively closed under finite unions and $\approx = \bigcup_{s \in V} [\![\psi_s]\!]$, the result follows. □

Considering that gap-formulae are particular formulae of Presburger Arithmetic, we know that gap-definable sets have a decidable membership problem. Theorem 4 thus implies the decidability of equivalence checking between GCS processes and finite systems w.r.t. strong and weak bisimulation.

7 Conclusion

We have shown that model checking gap-order systems with the logic **EG** is undecidable while the problem remains decidable for the logic **EF**. An immediate consequence of the latter result is the decidability of strong and weak bisimulation checking between GCS and finite systems.

The decidability of **EF** model checking is shown by using finite sets of monotonicity graphs or equivalently, gap-formulae to represent intermediate results in a bottom-up evaluation. This works because the class of arbitrary gap-definable sets is effectively closed under union and complements and one can compute finite representations of $Pre(S)$ and $Pre^*(S)$ for gap-definable sets S.

Our decidability result relies on a well-quasi-ordering argument to ensure termination of the fixpoint computation for $Pre^*(S)$, and therefore does not yield any meaningful upper complexity bound.

Interesting open questions include determining the exact complexity of model checking GCS with respect to **EF**. We also plan to investigate the decidability and complexity of checking behavioral equivalences like strong and weak bisimulation between two GCS processes as well as checking (weak) simulation preorder and trace inclusion.

References

[AD06] Abdulla, P.A., Delzanno, G.: Constrained multiset rewriting. In: Proc. AVIS 2006, 5th Int. Workshop on on Automated Verification of InfiniteState Systems (2006)

[Boz12] Bozzelli, L.: Strong termination for gap-order constraint abstractions of counter systems. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 155–168. Springer, Heidelberg (2012)

[BP12] Bozzelli, L., Pinchinat, S.: Verification of gap-order constraint abstractions of counter systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 88–103. Springer, Heidelberg (2012)

[Cer94] Cerans, K.: Deciding properties of integral relational automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 35–46. Springer, Heidelberg (1994)

[Gla01] van Glabbeek, R.J.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, ch. 1, pp. 3–99. Elsevier (2001)

[JKM98] Jančar, P., Kučera, A., Mayr, R.: Deciding bisimulation-like equivalenceswith finite-state processes. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 200–211. Springer, Heidelberg (1998)

[KJ06] Kučera, A., Jančar, P.: Equivalence-checking on infinite-state systems: Techniques and results. TPLP 6(3), 227–264 (2006)

[May84] Mayr, E.W.: An algorithm for the general petri net reachability problem. SIAM J. Comput. 13(3), 441–460 (1984)

[Min67] Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River (1967)

Constructing Minimal Coverability Sets

Artturi Piipponen and Antti Valmari

Department of Mathematics
Tampere University of Technology
P.O. Box 553, FI–33101 Tampere, Finland
{artturi.piipponen,antti.valmari}@tut.fi

Abstract. This publication addresses two bottlenecks in the construction of minimal coverability sets of Petri nets: the detection of situations where the marking of a place can be converted to ω, and the manipulation of the set A of maximal ω-markings that have been found so far. For the former, a technique is presented that consumes very little time in addition to what maintaining A consumes. It is based on Tarjan's algorithm for detecting maximal strongly connected components of a directed graph. For the latter, a data structure is introduced that resembles BDDs and Covering Sharing Trees, but has additional heuristics designed for the present use. Results from initial experiments are shown. They demonstrate significant savings in running time and varying savings in memory consumption compared to an earlier state-of-the-art technique.

Keywords: coverability set, Tarjan's algorithm, antichain data structure.

1 Introduction and Notation

The background of this work would be very difficult to introduce without first making certain notions precise. Therefore, this section is an interleaving of definitions and the discussion of the background.

A very well-known form of Petri nets is *place/transition net* $(P, T, W, \hat{M})$. It consists of a set P of *places*, set T of *transitions* (such that $P \cap T = \emptyset$), function $W : (P \times T) \cup (T \times P) \mapsto \mathbb{N}$ of *weights* and the *initial marking* $\hat{M}$. In this publication P and T are finite. A *marking* M is a vector of $|P|$ natural numbers. A transition t is *enabled at* M, denoted with $M\,[t\rangle$, if and only if $M(p) \geq W(p,t)$ for every $p \in P$. Then t may *occur* yielding the marking M' such that $M'(p) = M(p) - W(p,t) + W(t,p)$ for every $p \in P$. This is denoted with $M\,[t\rangle\,M'$. It is also said that t is *fired* at M yielding M'. The notation is extended to sequences of transitions in the natural way. A marking M' is *reachable from* M if and only if there is $\sigma \in T^*$ such that $M\,[\sigma\rangle\,M'$.

The set of *reachable markings* (that is, markings that are reachable from the initial marking) of a finite place/transition net is not necessarily finite. However, there always is a finite *coverability set* of certain kind of extended markings that can be used for some of the same purposes as the set of reachable markings is often used [6]. We call them *ω-markings*. An ω-marking is a vector of $|P|$

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 183–195, 2013.

elements of the set $\mathbb{N} \cup \{\omega\}$, where ω intuitively denotes "unbounded". The enabledness and occurrence rules of transitions are extended to ω-markings with the conventions that for every $i \in \mathbb{N}$, $\omega \geq i$ and $\omega + i = \omega - i = \omega$.

We say that M' *covers* M if and only if $M(p) \leq M'(p)$ for every $p \in P$. We say that M is *a limit* of a set $\mathcal{M}$ of ω-markings if and only if $\mathcal{M}$ contains $M_0 \leq M_1 \leq \ldots$ such that for every $p \in P$, either $M(p) = \omega$ and $M_i(p)$ grows without limit as i grows, or there is i such that $M(p) = M_i(p) = M_{i+1}(p) = \ldots$.

A coverability set is any set $\mathcal{M}$ that satisfies the following conditions:

1. Every reachable marking M is covered by some $M' \in \mathcal{M}$.
2. Every $M \in \mathcal{M}$ is a limit of reachable markings.

A coverability set is not necessarily finite. Indeed, the set of reachable markings is a coverability set. Fortunately, there is a unique *minimal coverability set* that is always finite [4]. It has no other coverability set as a subset, and no ω-marking in it is covered by another ω-marking in it. It consists of the maximal elements of the set of the limits of the set of reachable markings.

The construction of coverability sets resembles the construction of the set of reachable markings but has additional features. A central idea is that if $M_0 \,[\sigma\rangle\, M \,[t\rangle\, M' > M_0$, then the sequence σt can occur repeatedly without limit, making the markings of those p grow without limit that have $M'(p) > M_0(p)$, while the remaining p have $M'(p) = M_0(p)$. The limit of the resulting markings is M'', where $M''(p) = \omega$ if $M'(p) > M_0(p)$ and $M''(p) = M_0(p)$ otherwise.

Roughly speaking, instead of storing M' and remembering that $M \,[t\rangle\, M'$, most if not all algorithms store M'' and remember that $M \,-t\!\xrightarrow{\omega}\, M''$, where $M \,-t\!\xrightarrow{\omega}\, M''$ denotes that M'' was obtained by firing t at M and then possibly adding ω-symbols to the result. However, this is not precisely true for four reasons.

First, the algorithms need not remember that $M \,-t\!\xrightarrow{\omega}\, M''$. It suffices to remember that there is t such that $M \,-t\!\xrightarrow{\omega}\, M''$. Second, instead of $M_0 \,[\sigma\rangle\, M$ the algorithms use $M_0 \,-\sigma\!\xrightarrow{\omega}\, M$, because they only have access to the latter.

Third, after firing t at M, an algorithm may use more than one M_0 and σ that have $M_0 \,-\sigma\!\xrightarrow{\omega}\, M$ to add ω-symbols. The *pumping operation* assigns ω to those $M'(p)$ that have $M_0(p) < M'(p) < \omega$. It may be triggered when the algorithm detects that the *pumping condition* holds with M_0. It holds with M_0 when there is $\sigma \in T^*$ such that $M' > M_0 \,-\sigma\!\xrightarrow{\omega}\, M$, where M' has been obtained by firing t at M and then doing zero or more pumping operations. The algorithms in [9] and this publication never fail to do the pumping operation when the pumping condition holds, but this is not necessarily true of all algorithms.

Fourth, after firing $M \,[t\rangle\, M'$ and doing zero or more pumping operations, an algorithm may reject the resulting M', if it is covered by some already stored ω-marking M''. The intuition is that whatever M' could contribute to the minimal coverability set, is also contributed by M''. So M' need not be investigated.

Whether $M \,-t\!\xrightarrow{\omega}\, M'$ holds depends on not just M, t, and M', but also on what the algorithm has done before trying t at M. So the precise meaning of the notation $M \,-t\!\xrightarrow{\omega}\, M'$ depends on the particular algorithm. The meaning used in this publication will be made precise in Section 2.

Some ideas for speeding up the construction of minimal coverability sets have been suggested [4,5,7]. However, [9] gave both theoretical, heuristic, and experimental evidence that a straightforward approach is very competitive, when ω-markings are constructed in depth-first or so-called most tokens first order and pumping conditions are always detected when possible. Nevertheless, as was pointed out in [9], performance measurements must be taken with more than one grain of salt. The running time of an algorithm may depend dramatically on the order in which the transitions are listed in the input, and sorting the transitions according to a natural heuristic does not eliminate this effect.

The algorithm in [9] maintains the set A of maximal ω-markings that have been constructed so far, and most others maintain something similar (but not necessarily precisely the same). ("A" stands for "antichain".) At a low level, the most time-consuming operations in [9] — and probably also in most, if not all, other minimal coverability set construction algorithms — are the manipulation of A and the detection of pumping conditions. In this publication we present a significant improvement to both.

The overall structure of our new algorithm is presented in Section 2. The detection of the pumping condition involves finding out that $M_0 - \sigma \xrightarrow{\omega} M$. Section 3 describes how *Tarjan's algorithm* for detecting maximal strongly connected components of a directed graph [8,1] can be harnessed to convert this otherwise expensive test to only consume constant time. A data structure that improves the efficiency of maintaining A is introduced in Section 4. It uses ideas from BDDs [2] and covering sharing trees [3], and has heuristics designed for coverability sets. An additional optimisation is discussed in Section 5. Section 6 presents some performance measurements without and with using the new ideas.

2 Overall Algorithm

Figure 1 shows the new minimal coverability set construction algorithm of this publication in its basic form. Variants of it will be discussed in Section 6.

Lines 1, 3–6, 13–16, 18, 20–24, and 27 implement most of the coverability set construction algorithm of [9]. Let us discuss them in this section. The remaining lines may be ignored until they are discussed in later sections.

The set A contains the maximal ω-markings that have been found so far. Upon termination it contains the result of the algorithm. Its implementation will be discussed in Section 4. In addition to what is explicit in Fig. 1, the call on line 22 may remove elements from A in favour of a new element M' that strictly covers them. We will discuss this in detail later.

The set F is a hash table. ω-Markings are added to it at the same time as to A, but they are never removed from it. So always $A \subseteq F$. The attributes of an ω-marking such as $M.\mathsf{tr}$ (discussed soon) are stored in F and not in A. That is, F contains records, each of which contains an ω-marking and some additional information. The reason is that, as we will see later, some information on an ω-marking may remain necessary even after it has been removed from A. Like in [9], F is also used to implement an optimisation that will be discussed together with

```
1   F := {M̂}; A := {M̂}; W.push(M̂); M̂.tr := first transition
2   S.push(M̂); M̂.ready := false; n_f := 1; M̂.index := 1; M̂.lowlink := 1
3   while W ≠ ∅ do
4       M := W.top; t := M.tr; if t ≠ nil then M.tr := next transition endif
5       if t = nil then
6           W.pop
7           activate transitions as discussed in Section 5
8           if M.lowlink = M.index then
9               while S.top ≠ W.top do S.top.ready := true; S.pop endwhile
10          else if W ≠ ∅ then
11              W.top.lowlink := min{W.top.lowlink, M.lowlink}
12          endif
13          go to line 3
14      endif
15      if ¬M [t⟩ then go to line 3 endif
16      M' := the ω-marking such that M [t⟩ M'
17      if M' ≤ M then passivate t; go to line 3 endif
18      if M' ∈ F then
19          if ¬M'.ready then M.lowlink := min{M.lowlink, M'.lowlink} endif
20          go to line 3
21      endif
22      Cover-check(M', A)      // only keep maximal — may update A and M'
23      if M' is covered then go to line 3 endif
24      F := F ∪ {M'}; A := A ∪ {M'}; W.push(M'); M'.tr := first transition
25      S.push(M'); M'.ready := false
26      n_f := n_f + 1; M'.index := n_f; M'.lowlink := n_f
27  endwhile
```

Fig. 1. A coverability set algorithm that uses Tarjan's algorithm and some heuristics

lines 18 and 20. Hash tables are very efficient, so F does not cause significant extra cost.

For efficiency, instead of the common recursive implementation, depth-first search is implemented with the aid of a stack which is called W (for work-set). The elements of W are pointers to records in F. Each ω-marking M has an attribute tr that points to the next transition that should be tried at M.

The algorithm starts on lines 1 and 2 by putting the initial marking to all data structures. Roughly speaking, lines 3 to 27 try each transition t at each encountered ω-marking M in depth-first order. (This is not strictly true, because heuristics that are discussed later may prematurely terminate the processing of M and may cause the skipping of some transitions at M.) If M has untried transitions, line 4 picks the next, otherwise lines 6–13 that implement backtracking are executed. Lines 7–12 will be discussed later.

If the picked transition t is disabled at the current ω-marking M, then it is rejected on line 15. Otherwise t is fired at M on line 16. Lines 17 and 19 will be discussed later. If M' has already been encountered, it is rejected on lines 18 and 20. This quick rejection of M' is useful, because reaching the same ω-marking

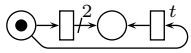
again is expected to be very common, because $M\,[t_1t_2\rangle\,M_{12}$ and $M\,[t_2t_1\rangle\,M_{21}$ imply that $M_{12} = M_{21}$. Without lines 18 and 20, M' would be rejected on line 23, but after consuming more time. Line 18 is also needed because of line 19.

The call Cover-check(M', A) first checks whether M' is covered by any ω-marking in A. If yes, then M' is rejected on line 23.

In the opposite case, Cover-check checks whether the pumping condition holds with any $M_0 \in A$. (In [9], the pumping condition was detected for $M_0 \in F$. Theorem 4 will tell why it suffices to use A instead.) If yes, it changes $M'(p)$ to ω for the appropriate places p. Cover-check also removes from A those ω-markings that the updated M' covers strictly. When M is removed, $M.\mathsf{tr}$ is set to nil, so that even if the algorithm backtracks to M in the future, no more transitions will be fired at it. The addition of ω-symbols makes M' grow in the "$\leq$" ordering and may thus make the pumping condition hold with some other M_0. Cover-check continues until there is no $M_0 \in A$ with which the pumping condition holds but the pumping operation has not yet been done. The details of Cover-check will be discussed in later sections.

If M' was not covered, its updated version is added to the data structures on lines 24–26. This implements the entering to M' in the depth-first search.

It is the time to make the notation $M\,{-t\!\xrightarrow{\omega}}\,M'$ precise. It denotes that t was fired at M on line 16 resulting in some M'' such that $M'' \not\leq M$, and either $M'' \in F$ held on line 18 (in which case $M' = M''$), or M'' was transformed to M' on line 22 and then added to F on line 24.

Thus $M\,{-t\!\xrightarrow{\omega}}\,M'$ is either always false or becomes true during the execution of the algorithm. Even if $M \in F$ and $M\,[t\rangle$, it may be that there never is any M' such that $M\,{-t\!\xrightarrow{\omega}}\,M'$. This is the case if M is removed from A and $M.\mathsf{tr}$ is set to nil before t is tried at M, or if the result of trying t at M is rejected on line 17 or 23. With the following Petri net, the latter happens although $M \in A$ when the algorithm has terminated: 

.

The correctness of this approach has been proven in detail in [9]. Intuitively, every ω-marking that is put into F is a limit of reachable markings, because for each p, $M\,{-t\!\xrightarrow{\omega}}\,M'$ either mimics $M\,[t\rangle\,M'$, copies ω from $M(p)$ to $M'(p)$, or sets $M'(p)$ to ω as justified by some pumping condition. Pumping operations make progress towards termination. The algorithm does not terminate prematurely, because each time when something is rejected or passivated, something else is kept or remains active that makes at least the same contribution to the final A.

The following details are essential for this publication.

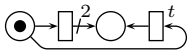
Lemma 1. *For every ω-markings M and M', $p \in P$, $t \in T$, and $\sigma \in T^*$,*

1. *If $M\,{-\sigma\!\xrightarrow{\omega}}\,M'$ and $M(p) = \omega$, then $M'(p) = \omega$.*
2. *Assume that $M\,{-t\!\xrightarrow{\omega}}\,M'$ and $M(p) < \omega = M'(p)$. After constructing the edge $M\,{-t\!\xrightarrow{\omega}}\,M'$, the algorithm does not backtrack from M' before it has investigated all M'' that have $\sigma \in T^*$ such that $M'\,{-\sigma\!\xrightarrow{\omega}}\,M''$.*
3. *For every $M_0 \in F$, there is $M_0' \in A$ such that $M_0 \leq M_0'$.*
4. *Every M constructed by the algorithm is a limit of $M_1, M_2, \ldots$ such that there are $\sigma_1, \sigma_2, \ldots$ such that $\hat{M}\,[\sigma_i\rangle\,M_i$ for $i \geq 1$.*

In depth-first search, an ω-marking M is *white* if it has not been found (that is, $M \notin F$); *grey* if it has been found but not backtracked from (that is, $M \in W$); and *black* if it has been backtracked from. If M is black and $M - t \xrightarrow{\omega} M'$, then M' is grey or black. The grey ω-markings M_i^{g} (where $M_0^{\mathsf{g}} = \hat{M}$) and the $M_{i-1}^{\mathsf{g}} - t_i \xrightarrow{\omega} M_i^{\mathsf{g}}$ via which they were first found constitute a path from the initial to the current ω-marking.

Lemma 1(1) follows trivially from the transition firing rule. It implies that if M has been found and has ω in some place where the current ω-marking M_{c} does not have ω, then there is no path from M to M_{c}. As a consequence, M and all its descendants are black. They remain black from then on, because a black ω-marking no longer changes colour. This implies Lemma 1(2).

Inspired by the above, we say that M is *ripe* if and only if $M \in F$ and either the algorithm has terminated, or after finding M there has been an instant of time such that for some place p, $M_{\mathsf{c}}(p) < \omega = M(p)$, where M_{c} was the current ω-marking at that time. All descendants of all ripe ω-markings are black.

The following lemma says that the "future" of each ripe ω-marking has been fully covered. This result does not immediately follow from the fact that the descendants of each ripe ω-marking are black, because any such descendant may have been rejected in favour of another ω-marking that strictly covers it (cf. 

). For the same reason, the lemma does not promise that its M'_n is obtained via the sequence $M'_0 - t_1 \cdots t_n \xrightarrow{\omega} M'_n$.

Lemma 2. *If M'_0 is ripe and $M'_0 \geq M_0 \, [t_1 \cdots t_n\rangle \, M_n$, then there is M'_n such that M'_n is ripe and $M'_n \geq M_n$. A similar claim holds for $M_0 - t_1 \cdots t_n \xrightarrow{\omega} M_n$.*

Proof. To prove the first claim, consider the moment when M'_0 became ripe. We use induction on $1 \leq i \leq n$. By Lemma 1(3), there was $M''_{i-1} \in A$ such that $M'_{i-1} \leq M''_{i-1}$. The algorithm had tried t_i at M''_{i-1}. If the result was kept, it qualifies as M'_i, otherwise it was rejected because it was covered by an ω-marking that qualifies as M'_i. By Lemma 1(1), M'_i has ω-symbols in at least the same places as M'_{i-1}. So M'_i is ripe and $M'_i \geq M_i$.

The above proof referred to a certain moment in time to ensure that $M''_{i-1} \in A$. Later $M''_{i-1} \in A$ may cease to hold, but what was proven remains valid. We point out for the sequel that if $M'_{i-1}(p) < \omega$ then $M'_i(p) - M'_{i-1}(p) \geq M_i(p) - M_{i-1}(p)$, because the firing of t_i has the same effect to the ω-marking of p in both cases, and the possible additional operations may not reduce $M'_i(p)$.

With $M_{i-1} - t_i \xrightarrow{\omega} M_i$, there may be $p_1, \ldots, p_k$ such that $M_{i-1}(p_j) < \omega = M_i(p_j)$. Given M'_{i-1}, we apply induction on $1 \leq j \leq k$ to obtain an M'_i that has the required properties. Let $M_{i,j}$ be the ω-marking just after the algorithm made $M_i(p_j) = \omega$. So $M_{i-1} \, [t_i\rangle \, M_{i,0}$ and $M_{i,k} = M_i$. The first claim yields $M'_{i,0}$. Let σ_j be the sequence that justified converting $M_{i,j-1}$ to $M_{i,j}$. There are $\dot{M}_{i,j}$ and $\ddot{M}_{i,j}$ such that $M_{i,j-1} \, [\sigma_j\rangle \, \dot{M}_{i,j} \, [\sigma_j\rangle \, \ddot{M}_{i,j}$. The first claim can be applied to this sequence, yielding $\dot{M}'_{i,j}$ and $\ddot{M}'_{i,j}$. If $\dot{M}'_{i,j}(p) < \omega$, then $\ddot{M}'_{i,j}(p) - \dot{M}'_{i,j}(p) \geq \ddot{M}_{i,j}(p) - \dot{M}_{i,j}(p) \geq 0$. Therefore, $\dot{M}'_{i,j} \leq \ddot{M}'_{i,j}$. Because of the use of A in the proof of the first claim, $\dot{M}'_{i,j} \not< \ddot{M}'_{i,j}$. So $\ddot{M}'_{i,j} = \dot{M}'_{i,j}$, implying $\ddot{M}'_{i,j}(p) = \omega$ if

$\ddot{M}_{i,j}(p) > \dot{M}_{i,j}(p)$, that is, if $M_{i,j}(p) = \omega > M_{i,j-1}(p)$. With the remaining p, $\ddot{M}'_{i,j}(p) \geq M'_{i,j-1}(p) \geq M_{i,j-1}(p) = M_{i,j}(p)$. These yield $M_{i,j} \leq \ddot{M}'_{i,j}$. So $\ddot{M}'_{i,j}$ qualifies as the $M'_{i,j}$. Choosing $M'_i = M'_{i,k}$ completes the proof of step i. □

3 Constant-Time Reachability Testing

A *maximal strongly connected component* or *strong component* of a directed graph (V, E) is a maximal set of vertices $V' \subseteq V$ such that for any two vertices u and v in V', there is a path from u to v. The strong components constitute a partition of V. Tarjan's algorithm [8,1] detects strong components in time $O(|V| + |E|)$. It is based on depth-first search of the graph. It is slower than depth-first search only by a small constant factor.

In our case, V consists of all ω-markings that are encountered during the construction of the minimal coverability set, that is, those that are (eventually) stored in F. The edges are defined by $(M, M') \in E$ if and only if there is $t \in T$ such that $M -t\xrightarrow{\omega} M'$. This notion, and thus also V and E, depends on the order in which transitions are picked on lines 1, 4, and 24 in Fig. 1. Fortunately, this does not confuse Tarjan's algorithm, because an edge is introduced either when the algorithm is ready to investigate it or not at all.

In Fig. 1, Tarjan's algorithm is represented via lines 2, 8–12, 19, and 25–26. In addition to W, it uses another stack, which we call S. Also its elements are pointers to records in F.

Tarjan's algorithm also uses two attributes on each ω-marking called index and lowlink. The index is a running number that the ω-marking gets when it is encountered for the first time. It never changes afterwards. The lowlink is the smallest index of any ω-marking that is known to belong to the same strong component as the current ω-marking. When backtracking and when encountering an ω-marking that has already been visited and is in the same strong component with the current ω-marking, the lowlink value is backward-propagated and the smallest value is kept. The lowlink value is not backward-propagated from ω-markings that belong to already completed strong components.

Each ω-marking is pushed to S when it is found and popped from S when its strong component is ready, and it never returns to S. Presence in S is tested quickly via an attribute ready that is updated when S is manipulated.

The following is the central invariant property of Tarjan's algorithm:

Lemma 3. *Let $M_0 \in F$. There is a path from M_0 to the M of Fig. 1 if and only if $M_0 \in S$. If $M_0 \notin S$, then every ω-marking to which there is a path from M_0 is neither in S nor in W.*

Cover-check(M', A) has to find each M_0 such that $M_0 \in A$ and $M_0 < M'$, because they have to be removed from A. When it has found such an M_0, it checks whether $M_0.\mathsf{ready} = \mathsf{false}$, that is, whether $M_0 \in S$. This is a constant-time test that reveals whether there is a path from M_0 to M'. In this way Cover-check detects each valid pumping condition where $M_0 \in A$ with a constant amount of additional effort per removed element of A.

If ω-symbols are added to M', then the checking is started again from the beginning, because the updated M' may cover strictly elements of A that the original M' did not cover strictly. Also they have to be removed and checked against the pumping condition. When Cover-check terminates, there are no unused instances of the pumping condition where $M_0 \in A$, and A no longer contains ω-markings that are strictly covered by M'.

This method only detects the cases where $M_0 \in A$, while [9] uses $M_0 \in F$. Fortunately, the following theorem tells that it does not make a difference.

Theorem 4. *The algorithm in Fig. 1 constructs the same ω-markings as it would if F were used instead of A in the pumping conditions.*

Proof. In this case the pumping condition is $M' > M_0 - \sigma \xrightarrow{\omega} M\, [t\rangle\, M_\#$, where $\sigma \in T^*$ and M' has been made from $M_\#$ by replacing the contents of zero or more places by ω. By Lemma 3, from each ω-marking in S and from no ω-marking in $F \setminus S$ there is a path to M. The pumping condition triggers the updating of M' to M'' such that for every $p \in P$, either $M_0(p) = M'(p) = M''(p)$ or $M_0(p) < M'(p) \leq M''(p) = \omega$.

We prove the claim by induction. We show that in every pumping operation, A causes (at least) the same updates as F, the induction assumption being that also in the previous times A caused the same updates as F. At least the same updates implies precisely the same updates, because $A \subseteq F$.

Let the pumping condition hold such that $M_0 \in F$. If $M_0 \in A$, then the induction step holds trivially. From now on $M_0 \notin A$.

Lemma 1(3) yields $M_0' \in A$ such that $M_0 < M_0'$. It was found after M_0, because otherwise M_0 would have been rejected on line 23. Because $M_0 - \sigma \xrightarrow{\omega} M$ now, M_0 is now in S. So M_0 was in S when M_0' was found. At that moment there was a path from M_0 to what was then M, that is, there is ρ such that $M_0 - \rho \xrightarrow{\omega} M_0'$. So the pumping condition held with $M_0 \in F$. By the induction assumption, ω-symbols were added to M_0'. Therefore, for every $p \in P$, either $M_0'(p) = M_0(p)$ or $M_0'(p) = \omega$.

Return to the moment when ω-symbols are added to M'. If $M_0' \in S$, there is a path from M_0' to M. If $M_0'(p) = \omega$, then also $M'(p) = \omega$ by Lemma 1(1). We already saw that if $M_0'(p) \neq \omega$, then $M_0'(p) = M_0(p)$. So the pumping condition holds with M_0' and causes precisely the same result as M_0 causes.

The case remains where $M_0' \notin S$. If there is M'' such that $M_0' - \sigma \xrightarrow{\omega} M''$, then $M'' \neq M$. Furthermore, $M'' \geq M$, because $M_0 < M_0'$, and $M_0 - \rho \xrightarrow{\omega} M_0'$ implies that ω-symbols are added to (or are already in) at least the same places along $M_0' - \sigma \xrightarrow{\omega} M''$ as along $M_0 - \sigma \xrightarrow{\omega} M$. So $M'' > M$. But that is a contradiction with the fact that M is the current ω-marking.

So there are σ_i, t_i, σ_i', and M_i' such that $\sigma_i t_i \sigma_i' = \sigma$, $M_0' - \sigma_i \xrightarrow{\omega} M_i'$, but t_i was not tried at M_i' or the result of trying it was rejected. We discuss the case that t_i was not tried. The other case is similar.

Failure to try t_i implies that there was M_i'' such that $M_i' < M_i''$. If M_i is the ω-marking such that $M_0 - \sigma_i \xrightarrow{\omega} M_i$, then $M_i \leq M_i'$. Thus M_i'' was found after t_i was fired at M_i but before M_i' was backtracked from. Because M_i is in S now, it

was in S when M_i'' was found. So there was a path from M_i to M_i'', triggering the pumping condition. There is at least one p such that $M_i(p) \leq M_i'(p) < M_i''(p)$. Therefore, M_i'' has more ω-symbols than M_i'. So M_i'' is ripe. Lemma 2 says that M is covered by some ripe M_n'', which is a contradiction with the fact that M is the current ω-marking. □

4 A Data Structure for Maximal ω-Markings

This section presents a data structure for maintaining A. It has been inspired by Binary Decision Diagrams [2] and Covering Sharing Trees [3]. However, ω-markings are only added one at a time. So we are not presenting a symbolic approach. The purpose of using a BDD-like data structure is to facilitate fast detection of situations where an ω-marking covers another. The details of the data structure have been designed accordingly. We will soon see that they make certain heuristics fast.

We call the M' on line 22 of Fig. 1 the *new* ω-marking, while those stored in A are *old*. Cover-check first uses the data structure to detect if M' is covered by any old ω-marking. If yes, then nothing more needs to be done. In the opposite case, Cover-check then searches for old ω-markings that are covered by M'. By the first search, they are strictly covered. This search cannot be terminated when one is found, because Cover-check has to remove all strictly covered ω-markings from A and use them in the pumping test. Therefore, finding the first one quickly is less important than finding quickly the first old ω-marking that covers M'. As a consequence, the data structure has been primarily optimised to detect if any old ω-marking covers the new one, and secondarily for detecting covering in the opposite order.

Let $\underline{M}(p) = M(p)$ if $M(p) < \omega$ and $\underline{M}(p) = 0$ otherwise. Let $\overline{M}(p) = 1$ if $M(p) = \omega$ and $\overline{M}(p) = 0$ otherwise. In this section we assume without loss of generality that $P = \{1, 2, \ldots, |P|\}$.

The data structure consists of $|P| + 1$ layers. The topmost layer is an array of pointers that is indexed with the total number of ω-symbols in an ω-marking, that is, $\sum_{p=1}^{|P|} \overline{M}(p)$. This number can only be in the range from 0 to $|P|$, so a small array suffices. An array is more efficient than the linked lists used at lower layers. The pointer at index w leads to a representation of the set of ω-markings in A that have w ω-symbols each.

Layer $|P|$ consists of $|P| + 1$ linked lists, one for each total number of ω-symbols. Each node v in the linked list number w contains a value $v.m$, a pointer to the next node in the list, and a pointer to a representation of those ω-markings in A that have $\sum_{p=1}^{|P|} \overline{M}(p) = w$ and $\sum_{p=1}^{|P|} \underline{M}(p) = v.m$. The list is ordered in decreasing order of the m values, so that the ω-markings that have the best chance of covering M' come first.

Let $1 \leq \ell < |P|$. Each node v on layer ℓ contains two values $v.w$ and $v.m$, a link to the next node on the same layer, and a link to a node on layer $\ell - 1$. Of course, this last link is nil if $\ell = 1$. The node represents those ω-markings in A that have $\sum_{p=1}^{\ell} \overline{M}(p) = v.w$, $\sum_{p=1}^{\ell} \underline{M}(p) = v.m$, and the places greater than

ℓ have the unique ω-markings determined by the path that leads to v, as will be discussed below. If more than one path leads to v, then v represents more than one subset of A. They are identical with respect to the contents of the places from 1 to ℓ, but differ on at least one place above ℓ. The lists on these layers are ordered primarily in increasing order of the w values and secondarily in increasing order of the m values.

Like in BDDs, nodes with identical values and next-layer pointers are fused. To be more precise, when a node is being created, it is first checked whether a node with the desired contents already exists, and if yes, it is used instead. A specific hash table makes it fast to find existing nodes based on their contents.

Because every ω-marking that is in A is also in F, it has an explicit representation there. As a consequence, unlike with typical applications of BDDs, the storing of dramatically big numbers of ω-markings is not possible. As was mentioned above, the goal is not to do symbolic construction of ω-markings. Even so, the fusing of identical nodes pays off. Otherwise, for each ω-marking, A would use a whole node on layer 1 and additional partially shared nodes on other layers, while F represents the ω-marking as a dense vector of bytes. So A would use much more memory for representing each ω-marking than F uses.

Consider the checking whether $M' \leq M$, where M' is the new and M is any old ω-marking. After entering a node v at level $\ell - 1$, $M(\ell)$ is computed as $u.w - v.w$ and $u.m - v.m$, where u is the node at level ℓ from which level $\ell - 1$ was entered. If $M'(\ell) > M(\ell)$, then the traversal backtracks to u. This is because, thanks to the ordering of the lists, both v and the subsequent nodes in the current list at level $\ell - 1$ correspond to too small a marking in $M(\ell)$.

On the other hand, M may be rejected also if $\sum_{p=1}^{\ell-1} \overline{M}'(p) > \sum_{p=1}^{\ell-1} \overline{M}(p)$. To quickly detect this condition, an array wsum is pre-computed such that $\mathsf{wsum}[\ell] = \sum_{p=1}^{\ell} \overline{M}'(p)$:

$\mathsf{wsum}[1] := \overline{M}'(1)$
for $p := 2$ **to** $|P|$ **do** $\mathsf{wsum}[p] := \mathsf{wsum}[p-1] + \overline{M}'(p)$ **endfor**

This pre-computation introduces negligible overhead. The condition becomes $\mathsf{wsum}[\ell - 1] > v.w$, which is a constant time test. If this condition is detected, layer $\ell - 2$ is not entered from the current node, but the scanning of the list on layer $\ell - 1$ is continued.

The current node v (but not the current list) is rejected also if $\mathsf{wsum}[\ell - 1] = v.w$ and $\mathsf{msum}[\ell - 1] > v.m$, where $\mathsf{msum}[\ell] = \sum_{p=1}^{\ell} \underline{M}'(p)$. There also is a third pre-computed array mmax with $\mathsf{mmax}[\ell]$ being the maximum of $\underline{M}(1)$, $\underline{M}(2)$, $\ldots$, $\underline{M}(\ell)$. It is used to reject v when

$$\mathsf{wsum}[\ell-1] < v.w \quad \text{and} \quad (v.w - \mathsf{wsum}[\ell-1]) \cdot \mathsf{mmax}[\ell-1] + v.m < \mathsf{msum}[\ell-1] \ .$$

The idea is that considering the places from 1 to $\ell - 1$, each extra ω-symbol in M covers at most $\mathsf{mmax}[\ell - 1]$ ordinary tokens in M'. There is thus a fast heuristic for each of the cases $\mathsf{wsum}[\ell-1] < v.w$, $\mathsf{wsum}[\ell-1] = v.w$, and $\mathsf{wsum}[\ell-1] > v.w$. These heuristics are the reason for storing $\sum_{p=1}^{\ell} \overline{M}(p)$ and $\sum_{p=1}^{\ell} \underline{M}(p)$ into the node instead of $M(\ell)$.

Consider the situation where none of the above heuristics rejects v. Then layer $\ell - 2$ is entered from v. If it turns out that M' is covered, then the search need not be continued. In the opposite case, it is marked into v that layer $\ell - 2$ was tried in vain. If v is encountered again during the processing of the same M', this mark is detected and v is not processed further. To avoid the need of frequently resetting these marks, the mark is a running number that is incremented each time when the processing of a new M' is started. The marks are reset only when this running number is about to overflow the range of available numbers. This trick is from [9].

Similar heuristics are used for checking whether the new ω-marking strictly covers any ω-marking in A. The biggest difference is that now the search cannot be stopped when such a situation is found, as has been explained above. The condition "strictly" need not be checked, because if $M' \in A$, then M' is rejected by the first search or already on line 20. The third heuristic mentioned above is not used, because information corresponding to $\mathsf{mmax}[\ell]$ cannot be obtained cheaply for ω-markings in A. It would not necessarily be unique, and it would require an extra field in the record for the nodes.

5 Transition Removal Optimisation

By Lemma 1(1), if $M -t\xrightarrow{\omega} M'$ and there is p such that $M(p) < M'(p) = \omega$, then the ω-marking of p will remain ω until the algorithm backtracks to M. If the firing of a transition does not increase the ω-marking of any place, that is, if $M -t\xrightarrow{\omega} M'$ and $M' \leq M$, then t is *useless*. Lines 22 and 23 would reject M', had it not already been done on line 17. A transition that is not originally useless in this sense becomes useless, if ω-symbols are added to each p such that $W(p,t) < W(t,p)$.

Lines 7 and 17 implement an additional optimisation based on these facts. The "first transition" and "next transition" operations in Fig. 1 pick the transitions from a doubly linked list which we call the *active list*. When t that is in the active list has become useless, that is detected on Line 17. Then t is linked out from the active list and inserted to a singly linked list that starts at $\mathsf{passive}[c]$, where c is the number of locations in W where ω-symbols have been added, and $\mathsf{passive}$ is an array of size $|P|+1$. There also is a similarly indexed array toW such that $\mathsf{toW}[c]$ points to the most recent location in W where ω-symbols have been added. The forward and backward links of t still point to the earlier successor and predecessor of t in the active list. This operation takes constant time.

From then on, t is skipped at no additional cost until the algorithm backtracks to M. This moment is recognized from the current top of W getting below $\mathsf{toW}[c]$. Then all transitions from $\mathsf{passive}[c]$ are removed from there and linked back to their original places in the active list, and c is decremented. Because each passive list is manipulated only at the front, it releases the transitions in opposite order to in which they were inserted to it. This implies that the original ordering of the active list is restored when transitions are linked back to it, and the "next transition" operation is not confused. Also the linking back is constant time per transition.

Table 1. Initial measurements with some versions of the new algorithm

	mesh2x2		mesh3x2		AP13a		smallT5x2		largeT2x15x2		
$\|A\|$ $\|F\|$	256	316	6400	7677	1245	1281	31752	31752	32768	32768	
$\|S\|$		316		7677		65		50		32768	
$\approx$ [9]	4	5	23	696	739	718	60	66	467	3151	3182	1736	7121	7142	3328
no node fusion	2	2	70	46	48	1389	49	56	4393	67	72	4058	231	237	20736
basic new	3	4	24	52	68	732	57	62	850	80	89	1738	248	273	4224
no tr. rem.	4	4	24	64	76	732	56	60	850	81	91	1738	261	270	4224
partial F	3	4	22	55	61	672	54	71	401	152	162	4	250	258	3968

If the check on line 17 were removed, the algorithm would still reject M', but in the worst case that might happen much later in Cover-check. This heuristic is very cheap and may save time by rejecting M' early. Unfortunately, in our initial experiments (Section 6) it did not perform well.

6 Experiments and Conclusions

In this section we present some of the first experiments that we have made with an implementation of our new algorithm. To make comparison of running times reasonable, we compare the new implementation to a slightly improved version of the implementation in [9] (better hash function, etc.). Both have been written in the same programming language (C++) and were executed on the same computer.

The Petri net mesh2x2 is the heaviest example from [5,7]. It has been included to point out that the examples from [5,7] are not challenging enough for testing the new implementation. Mesh3x2 is a bigger version of it. AP13a has been modified from users.cecs.anu.edu.au/~thiebaux/benchmarks/petri/ by Henri Hansen, to present a somewhat bigger challenge. SmallT5x2 was designed for this publication, to have a small S and offer many possibilities for fusing nodes in the representation of A. LargeT2x15x2 had precisely the opposite design goal.

The results are in Table 1. The second row shows the final sizes of the sets A and F, assuming that ω-markings are never removed from F. The third row shows the maximal sizes of S. The sizes of W are not shown, because always $|W| \leq |S|$.

The next five rows show results for various implementations. "$\approx$ [9]" was explained above. "Basic new" is the algorithm described in this publication. "No node fusion" is otherwise the same as "Basic new", but nodes in the data structure for A that have the same values and next-layer pointers are not fused. "No tr. rem." is otherwise the same as "Basic new", but the optimization discussed in Section 5 is not in use. "Partial F" is otherwise the same as "Basic new", but when an ω-marking is removed from S it is also removed from F. If such an ω-marking is constructed anew, it is covered by some ω-marking in A, and thus will be rejected on line 23 at the latest.

For each implementation and Petri net, the first two numbers report the shortest and longest running times for five identical measurements in milliseconds.

The third number is the amount of memory consumed, measured in kilobytes (1024 bytes), assuming that memory is reserved for A, S, and the base table of F only as needed. For W, the same amount of memory was reserved as for S. These numbers are theoretical in the sense that the implementation did not use dynamically growing arrays in reality. We plan to fix this defect in future measurements.

To protect against programming errors, we checked for each Petri net that every version returned the same A.

All new versions are significantly faster than the one in [9] excluding AP13A where all versions are roughly equally fast, and mesh2x2 that is too small for a meaningful comparison. Although precise comparison is not possible, the results on mesh2x2 make it obvious that the new implementation outperforms the one in [7]. The memory consumptions of the four new versions relate to each other as one would expect. Compared to [9] whose A was a doubly linked list of the same records that F used, the new versions consume much more memory except when the fusion of nodes and the removal of ω-markings from F have a big effect. While [9] uses two additional pointers per ω-marking to represent A, the new versions have the complicated structure described in Section 4. Furthermore, S was absent from [9].

Acknowledgements. We thank the anonymous reviewers for their effort.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
3. Delzanno, G., Raskin, J.-F., Van Begin, L.: Covering Sharing Trees: A Compact Data Structure for Parameterized Verification. Software Tools for Technology Transfer 5(2-3), 268–297 (2004)
4. Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674, pp. 210–243. Springer, Heidelberg (1993)
5. Geeraerts, G., Raskin, J.-F., Van Begin, L.: On the Efficient Computation of the Minimal Coverability Set of Petri Nets. International Journal of Foundations of Computer Science 21(2), 135–165 (2010)
6. Karp, R.M., Miller, R.E.: Parallel Program Schemata. Journal of Computer and System Sciences 3(2), 147–195 (1969)
7. Reynier, P.-A., Servais, F.: Minimal Coverability Set for Petri Nets: Karp and Miller Algorithm with Pruning. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 69–88. Springer, Heidelberg (2011)
8. Tarjan, R.E.: Depth-First Search and Linear Graph Algorithms. SIAM J. Computing 1(2), 146–160 (1972)
9. Valmari, A., Hansen, H.: Old and New Algorithms for Minimal Coverability Sets. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 208–227. Springer, Heidelberg (2012) (Extended version has been accepted to Fundamenta Informaticae)

On the Complexity of Counter Reachability Games*

Julien Reichert

LSV, ENS Cachan, France
reichert@lsv.ens-cachan.fr

Abstract. Counter reachability games are played by two players on a graph with labelled edges. Each move consists in picking an edge from the current location and adding its label to a counter vector. The objective is to reach a given counter value in a given location. We distinguish three semantics for counter reachability games, according to what happens when a counter value would become negative: the edge is either disabled, or enabled but the counter value becomes zero, or enabled. We consider the problem of deciding the winner in counter reachability games and show that, in most cases, it has the same complexity under all semantics. Surprisingly, under one semantics, the complexity in dimension one depends on whether the objective value is zero or any other integer.

1 Introduction

Counter reachability games are played by two players, a Reacher and an Opponent, on a counter system. Such a system is represented by a labelled directed graph (Q, E), where Q is a finite set of locations and $E \subseteq Q \times \mathbb{Z}^d \times Q$ is a set of edges. The integer d is the dimension of the system. We associate to a counter system a vector of d counters, which is updated when an edge (q, v, q') is taken by adding v to it. The locations are partitioned into a set Q_1 of Reacher locations and a set Q_2 of Opponent locations. A configuration in a counter system is a pair (location, counter vector).

A play is an infinite sequence $(q_0, v_0)(q_1, v_1) \cdots \in (Q \times \mathbb{Z}^d)^\omega$, starting at a given initial location q_0 with the initial counter vector v_0. At any stage i, the owner of the location q_i chooses an edge (q_i, v, q_{i+1}), then the next configuration is $(q_{i+1}, v_i + v)$. The objective is given by a subset C of $Q \times \mathbb{Z}^d$: Reacher wins every play that reaches a configuration in C. Here, we deal with cases where it is equivalent to consider only subsets C that are singletons.

In many works on counter systems, there are only nonnegative counter values, e.g., in vector addition systems with states (VASS, in short) [1], an edge is disabled whenever it would make a counter become negative. In energy games [2,3], the objective is to bound counter values, especially with 0 as lower bound.

* This work is supported by the french Agence Nationale de la Recherche, REACHARD (grant ANR-11-BS02-001).

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 196–208, 2013.

In order to capture common behaviours around zero, we consider three semantics for counter systems:

- $\mathbb{Z}$ *semantics*: A counter can have any value in $\mathbb{Z}$.
- VASS *semantics*: An edge is disabled if taking it would make any counter value become negative.
- *non-blocking* VASS *semantics*: Every time an edge is taken, negative values are replaced by 0.

The decision problem associated to a counter reachability game is to determine whether Reacher has a winning strategy. We study decidability and complexity of this problem under the three semantics. Most of our results assume that the set of edges is restricted to a subset of $Q \times \{-1, 0, 1\}^d \times Q$; we call this the *short-range* property and we say that counter systems are *short-ranged*. Any counter system can be transformed into a short-ranged counter system at the cost of an exponential blowup, by splitting the edges with labels not in $\{-1, 0, 1\}^d$. However, we need to be careful when we deal with reachability issues, because a run in the short-ranged counter system visits configurations that the corresponding run of the first counter system does not visit.

We prove in Section 3 that the decision problem is undecidable for reachability games on counter systems of dimension two under the $\mathbb{Z}$ semantics, by an adaptation of the undecidability proof for reachability games on VASS of dimension two in [4].

We prove in Section 4 that the decision problem is PSPACE-complete for reachability games on short-ranged counter systems of dimension one under the $\mathbb{Z}$ semantics when the objective is $(q_f, 0)$, and under the non-blocking VASS semantics when the objective is $(q_f, 1)$. The proof is based on mutual reductions from the decision problem for reachability games on short-ranged counter systems of dimension one under the VASS semantics when the objective is $(q_f, 0)$, which has been proved PSPACE-complete in [4]. The case of a reachability games on short-ranged counter systems of dimension one under the non-blocking VASS semantics when the objective is $(q_f, 0)$ is considered separately. Surprisingly, the decision problem is then in P.

Without the short-range property, we have an immediate EXPSPACE upper bound for counter reachability games in dimension one. There are at least two particular cases of counter reachability games for which the decision problem is EXPTIME-hard in dimension one: countdown games [5] and robot games [6]. To the best of our knowledge, it is not known whether counter reachability games in dimension one are in EXPTIME.

2 Definitions

When we write "positive" or "negative", we always mean "strictly positive" or "strictly negative". We write $-\mathbb{N}$ for the set of nonpositive integers.

A *counter system* is a directed graph (Q, E), where Q is a finite set of *locations* and $E \subseteq Q \times \mathbb{Z}^d \times Q$ is a finite set of *edges*, with $d \in \mathbb{N} \setminus \{0\}$. The vector in $\mathbb{Z}^d$

is called the *label* of an edge. A *configuration* in a counter system is a pair (q, x), where $q \in Q$ and $x \in \mathbb{Z}^d$. A *run* of a counter system (Q, E) is an infinite sequence $r = (q_0, x_0)(q_1, x_1)\dots$ starting from an arbitrary initial configuration $(q_0, x_0) \in Q \times \mathbb{Z}^d$ and such that $(q_i, x_{i+1} - x_i, q_{i+1}) \in E$ for every $i \in \mathbb{N}$. A counter system has the *short-range property* if the integers in the labels of the edges are always in $\{-1, 0, 1\}$.

A *counter reachability game* is played by two players, a *Reacher* and an *Opponent*, on a counter system (Q, E). We partition the set of locations into $Q_1 \uplus Q_2$; Reacher owns Q_1, and Opponent owns Q_2. In our figures, we use $\bigcirc$ to represent Reacher locations, $\square$ to represent Opponent locations and $\diamond$ when the owner of the location does not matter.

A *play* is represented by an infinite path of configurations that players form by moving a token on (Q, E) and updating a counter as follows. At the beginning, the token is at a location q_0 and the counter is initialized with x_0, hence the initial configuration is (q_0, x_0). If the token is at $p \in Q_1$, then Reacher chooses an edge (p, v, q), otherwise Opponent chooses. The token is moved to q, the counter is updated to $x + v$, and the configuration $(q, x + v)$ is appended to the play. There is a special configuration, called the *objective* of the game, such that Reacher wins every play that visits the objective.

A *play prefix* starting from the configuration (q_0, x_0) is a finite sequence $(q_0, x_0)(q_1, x_1)\dots(q_k, x_k)$ of configurations in the underlying counter system. A *strategy* for a player is a function that takes as argument a play prefix and returns an edge that is available from the end of the play prefix. Given an configuration (q_0, x_0), two strategies s_1 and s_2 for the players, the *outcome* of these strategies from the configuration is the play starting at (q_0, x_0) and obtained when each player always chooses edges according to his strategy. A strategy s is *winning* for a player, from a given configuration, if he wins the outcome of s with any strategy of the other player from the configuration. A configuration (q_0, x_0) in the game is *winning* if Reacher has a winning strategy from (q_0, x_0). The decision problem associated to a counter reachability game is to determine whether Reacher has a winning strategy from a configuration in input.

A *Vector Addition System with States* (VASS, in short) is a counter system where the vectors in the configurations are always nonnegative. In order to maintain this property, an edge in a VASS is disabled if a counter would then become negative. A *non-blocking* VASS is a counter system where every negative counter value is replaced by 0.

We introduce a notation for the decision problems that we deal with, and we write Reach-$semantics^1_d(x_f)$ with the following parameters: a subscript d for the dimension, an argument x_f for the counter value in the objective and a superscript 1 to point out, if present, when the system is short-ranged. The counter value in the objective is also optional. We omit the location in the objective, because only the counter value is relevant here. For example, let us look at two notations that appear in the next two sections.

– The problem of deciding the winner on a counter system of dimension two with an arbitrary objective is denoted by Reach-CS_2.

– The problem of deciding the winner on a short-ranged non-blocking VASS of dimension one with 1 as objective value is denoted by Reach-NBVASS$^1_1(1)$.

3 Counter Reachability Games in Dimension Two or More

3.1 Reduction from VASS to General Counter Systems

We present a construction that we use in this section to prove undecidability of counter reachability games in dimension two, and in the next section to give lower complexity bounds.

In order to show the reduction from VASS to general counter systems, we simulate in the winning condition the deactivation of edges in VASS, which makes the difference to the $\mathbb{Z}$ semantics. We here denote by 0_d the d-dimensional vector $(0, \ldots, 0)$.

Proposition 1. Reach-VASS$_d(0_d)$ *reduces to* Reach-CS$_d(0_d)$ *in polynomial time for any dimension* d.

Proof. Let (Q, E) be a VASS of dimension d, let (q_0, x_0) and (q_f, x_f) be configurations of (Q, E). We consider the reachability game on (Q, E) where the objective is (q_f, x_f).

The following hypothesis makes most proofs of this work simpler, without loss of generality. We assume that q_f is a Reacher location. Else, we could simply create a Reacher location q'_f that has only one outgoing edge to q_f with label (0) and choose as objective (q'_f, x_f).

We want to build a general counter system on which Reacher has a winning strategy from a particular configuration if, and only if, he has a winning strategy from (q_0, x_0) in the VASS. The key property is that each player must be able to win whenever his adversary makes a counter value become negative. We can then simulate the VASS semantics.

In order to have this property, let (Q', E') be a counter system with locations $Q' = Q \cup \{\text{test}_e \mid e = (p, v, q) \in E, v \notin \mathbb{N}^d\} \cup \{\text{check}, \text{check}_1, \ldots, \text{check}_d\}$, where Reacher owns Q_1, the check locations and exactly the locations test_e for which the source of e belongs to Opponent in (Q, E). The set of edges E' is obtained from E, first by splitting every edge $e = (p, v, q)$ such that $v \notin \mathbb{N}^d$ into two edges (p, v, test_e) and $(\text{test}_e, 0, q)$, and second by adding moves from every location test_e to the new locations of Q', as depicted in Figures 1 and 2.

More precisely, E' is the union of the following sets of edges, where $(x)_{i,d}$ is the vector with x as i^{th} component and 0 everywhere else:

– $\{(p, v, q) \in E \mid v \in \mathbb{N}^d\}$;
– $\{(p, v, \text{test}_e), (\text{test}_e, (0), q) \mid e = (p, v, q) \in E, v \notin \mathbb{N}^d\}$;
– $\{(\text{test}_e, (0), \text{check}) \mid e = (p, v, q) \in E, p \in Q_1\}$;
– $\{(\text{test}_e, (0), \text{check}_i) \mid e = (p, v, q) \in E, p \in Q_2, 1 \leq i \leq d\}$;
– $\{(\text{check}, (-1)_{i,d}, \text{check}) \mid 1 \leq i \leq d\}$;

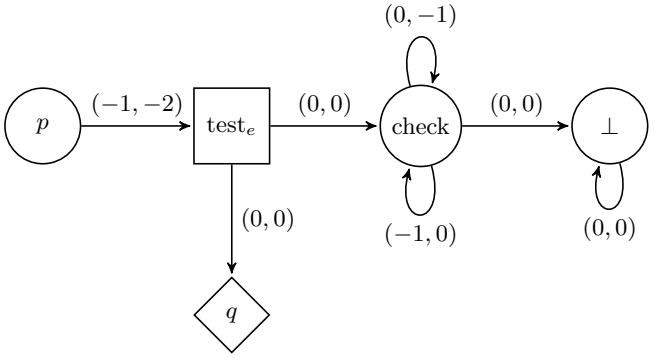


Fig. 1. Gadget to replace an edge $e = (p, (-1, -2), q)$ from a Reacher location in the reduction from Reach-VASS$_2$$((0, 0))$ to Reach-CS$_2$$((0, 0))$

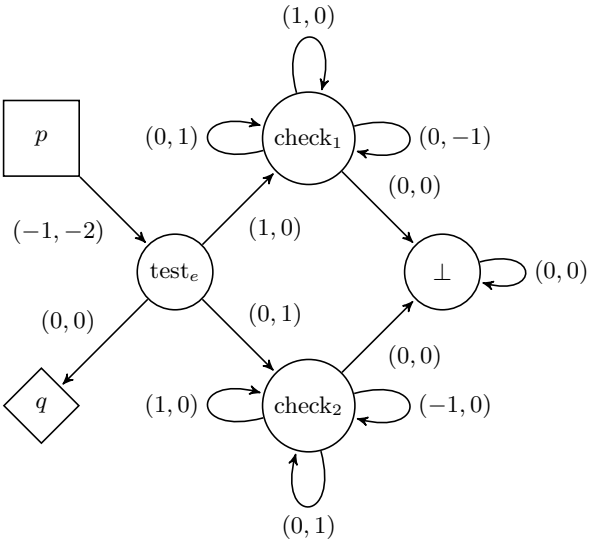


Fig. 2. Gadget to replace an edge $e = (p, (-1, -2), q)$ from an Opponent location in the reduction from Reach-VASS$_2$$((0, 0))$ to Reach-CS$_2$$((0, 0))$

- $\{(\text{check}_i, (-1)_{j,d}, \text{check}_i) \mid 1 \leq j \leq d, j \neq i\}$;
- $\{(\text{check}_i, (1)_{j,d}, \text{check}_i) \mid 1 \leq j \leq d\}$;
- $\{(q_f, -x_f, \bot)\} \cup \{(p, 0, \bot) \mid p \in \{\bot, \text{check}, \text{check}_1, \ldots, \text{check}_d\}\}$.

The objective of the counter reachability game is $(\bot, (0, \ldots, 0))$. Hence, in the location check, Reacher has a winning strategy if, and only if, every counter is nonnegative, and in the location check_i, Reacher has a winning strategy if, and only if, the i^{th} counter, which has been incremented when the play reached check_i, is nonpositive. Consequently, as soon as a player makes a counter become

negative, his adversary has a winning strategy by going to a check location. If all counters remain positive, then Reacher has a winning move once the play visits the objective of the game on (Q, E), and only in this case.

The reduction is polynomial: we have $|Q'| \leq d + 2 + |Q| + |E|$ and $|E'| \leq (d+2)|E| + 2d(d+1) + 2$. Moreover, the short-range property is preserved when it holds for the reduced VASS, provided that the objective in the VASS is a vector that contains only values in $\{-1, 0, 1\}$. □

3.2 Undecidability of Counter Reachability Games on VASS

The following proposition rephrases Proposition 4 from [4].

Theorem 2 ([4]). *Let (Q, E) be a short-ranged* VASS *of dimension two. Consider a reachability game on (Q, E) with $Q_Z \times (((\{0\} \times \mathbb{N}) \cup (\mathbb{N} \times \{0\}))$ as objective, where $Q_Z \subseteq Q$. The problem of deciding the winner of this game is undecidable.*

To apply Proposition 1, there must be only one configuration in the objective.

Proposition 3. *Let (Q, E) be a* VASS *of dimension two. Consider a reachability game on (Q, E) with $Q_Z \times ((\{0\} \times \mathbb{N}) \cup (\mathbb{N} \times \{0\}))$ as objective, where $Q_Z \subseteq Q$. We can build a* VASS *(Q', E') such that Reacher wins the reachability game on (Q, E) if, and only if, he wins the reachability game on (Q', E') with objective $(\bot, (0, 0))$, where $\bot \in Q' \setminus Q$.*

Proof. We suppose that Q_Z contains Reacher locations only. This is without loss of generality as in the proof of Proposition 1. Let $Q' = Q \cup \{\emptyset_1, \emptyset_2, \bot\}$, and let

$$\begin{aligned} E' = E &\cup \{(q, (0,0), \emptyset_1), (q, (0,0), \emptyset_2) \mid q \in Q_Z\} \\ &\cup \{(\emptyset_1, (-1,0), \emptyset_1), (\emptyset_2, (0,-1), \emptyset_2)\} \\ &\cup \{(\emptyset_1, (0,0), \bot), (\emptyset_2, (0,0), \bot), (\bot, (0,0), \bot)\}. \end{aligned}$$

Note that the short-range property is preserved. If Reacher has a winning strategy in the game on (Q, E), then he can follow the same strategy on (Q', E') and reach a configuration where the location is in Q_Z and one of the two counters is zero. At this point, he can go to the location where he resets the second counter and, after that, go to $\bot$ and win. Conversely, if Reacher has a winning strategy in the game on (Q', E'), then he can enforce that the play visits $Q_Z \times (((\{0\} \times \mathbb{N}) \cup (\mathbb{N} \times \{0\}))$, as this is the only possibility to reach a location $\emptyset_i$ with the $(3-i)^{\text{th}}$ counter at zero and, after that, to reach the objective. □

Theorem 4. Reach-CS_2^1 *is undecidable.*

Proof. We make two successive reductions from the decision problem of Theorem 2 using Propositions 3 and 1. □

4 Counter Reachability Games in Dimension One

In [4], counter reachability games are played on short-ranged VASS, where the winning condition in the one-dimensional case is to reach $Q_Z \times \{0\}$ for a given subset Q_Z of Q. It can be seen as the objective $(\bot, 0)$, once we add a gadget that permits Reacher to go from any location in Q_Z to $\bot$ without any further modification of the counter value. The decision problem is PSPACE-complete in general and it is in P when $Q_Z = Q$.

In this section, we establish mutual reductions between the decision problem for counter reachability games under the three semantics in dimension one. The complexity classes follow from the reductions.

4.1 Relative Integers Semantics

We recall that Proposition 1 implies that there is a polynomial-time reduction from Reach-VASS$^1_1(0)$ to Reach-CS$^1_1(0)$, hence Reach-CS$^1_1(0)$ is PSPACE-hard.

The main idea of the construction in this section is to simulate, with nonnegative integers only, a counter value in $\mathbb{Z}$. For this purpose, we use two copies of the set of locations and explain how to move from one copy to another.

Theorem 5. Reach-CS$^1_1(0)$ *is* PSPACE-*complete.*

Proof. We reduce Reach-CS$^1_1(0)$ to Reach-VASS$^1_1(0)$ in polynomial time. Consider a reachability game on a short-ranged counter system (Q, E), where the objective is $(q_f, 0)$, with $q_f \in Q_1$. Note that when the objective counter value is not 0, we can always shift initial and objective value in a general counter system.

Let $Q_+ = \{q_+ \mid q \in Q\}$ and $Q_- = \{q_- \mid q \in Q\}$ be two copies of Q, and let Q_E be the set $\{q_e \mid \exists p, q \in E, v \in \{\pm 1\}, e = (p, v, q) \in E\}$. We build the short-ranged VASS (Q', E'), where $Q' = Q_+ \cup Q_- \cup Q_E \cup \{\text{no}, \bot\}$ is partitioned into $Q'_1 = \{q_+, q_- \mid q \in Q_1\} \cup \{q_e \in Q_E \mid e \in Q_2 \times \{0, \pm 1\} \times Q\} \cup \{\text{no}, \bot\}$ and Q'_2. The set of edges E' contains two copies of E, i.e., edges (p_+, v, q_+) and $(p_-, -v, q_-)$ for each edge $(p, v, q) \in E$. The other edges of E' are used to move between Q_+ and Q_- via the new locations of Q_E, as depicted in Figures 3 and 4.

More precisely, E' is the union of the following sets of edges:

- $\{(p_+, v, q_+), (p_-, -v, q_-) \mid (p, v, q) \in E\}$;
- $\{(p_-, 0, q_e), (q_e, 0, \bot), (q_e, 1, q_+) \mid e = (p, 1, q) \in E, p \in Q'_1\}$;
- $\{(p_+, 0, q_e), (q_e, 0, \bot), (q_e, 1, q_-) \mid e = (p, -1, q) \in E, p \in Q'_1\}$;
- $\{(p_-, 0, q_e), (q_e, -1, \text{no}), (q_e, 1, q_+) \mid e = (p, 1, q) \in E, p \in Q'_2\}$;
- $\{(p_+, 0, q_e), (q_e, -1, \text{no}), (q_e, 1, q_-) \mid e = (p, -1, q) \in E, p \in Q'_2\}$;
- $\{(\text{no}, -1, \text{no}), (\text{no}, 0, \bot), (q_{f,+}, 0, \bot), (q_{f,-}, 0, \bot), (\bot, 0, \bot)\}$.

The VASS (Q', E') is designed such that a play in it corresponds to a play in the counter system (Q, E). Hence, a configuration $(q, x) \in Q \times -\mathbb{N}$ in (Q, E) is associated to the configuration $(q_-, -x) \in Q_- \times \mathbb{N}$ in (Q', E'). That is why the labels of the edges between locations in Q_- are the opposite of the labels of the edges in Q.

The objective of the game on (Q', E') is $(\bot, 0)$. In fact, Reacher loses whenever a play reaches $\bot$ with another counter value. Furthermore, if a player makes a move to a location q_e in Q_E and the counter value is not 0, then his adversary, who owns q_e, has a winning move. □

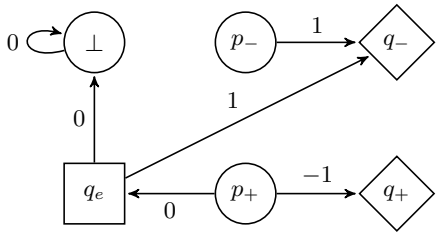


Fig. 3. Gadget to replace an edge $e = (p, -1, q)$ from a Reacher location in the reduction from Reach-$\mathrm{CS}_1^1(0)$ to Reach-$\mathrm{VASS}_1^1(0)$

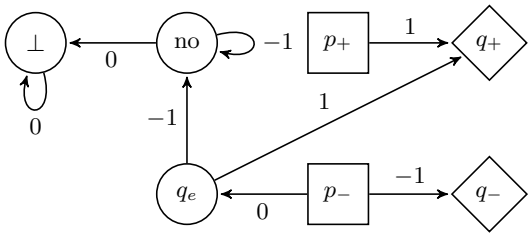


Fig. 4. Gadget to replace an edge $e = (p, 1, q)$ from an Opponent location in the reduction from Reach-$\mathrm{CS}_1^1(0)$ to Reach-$\mathrm{VASS}_1^1(0)$

A consequence of Theorem 5 is that Reach-CS_1 is in EXPSPACE: It suffices to split every edge with another label than -1, 0 or 1. However, we do not know yet whether EXPSPACE is an optimal upper bound, but we have the following lower bound.

Theorem 6 ([5,6]). Reach-CS_1 *is* EXPTIME*-hard.*

This lower bound is inherited from countdown games [5] and robot games [6], which we can express as counter reachability games.

4.2 Non-blocking VASS Semantics

When we simulate a game on a non-blocking VASS, we need, like for VASS, to handle the behaviour around the value 0. The idea is the following: For every edge labelled by -1 in a short-ranged non-blocking VASS, there are two choices for Opponent in the VASS: decrement the counter or leave it unchanged, depending on whether it is positive or zero. The winning condition is designed

so that Reacher has a checking move that makes him win whenever Opponent chooses the wrong move, e.g., he leaves the counter unchanged whereas he should decrement it. Moreover, Opponent wins if Reacher abuses his checking move.

Theorem 7. Reach-NBVASS$^1_1(1)$ *is* PSPACE-*complete.*

*Proof (*PSPACE-*hardness).* We reduce Reach-NBVASS$^1_1(1)$ to Reach-VASS$^1_1(0)$ in polynomial time. Consider a reachability game on a short-ranged non-blocking VASS (Q, E), where the objective is $(q_f, 1)$, with the assumption that $q_f \in Q_1$. Let Q_E be the set $\{q_e, q_e^{>0}, q_e^{=0} \mid e \in E \cap (Q \times \{-1\} \times Q)\}$. We build the short-ranged VASS (Q', E'), where $Q' = Q \cup Q_E \cup \{\text{no}, \bot\}$ is partitioned into $Q'_1 = Q_1 \cup \{q_e^{>0}, q_e^{=0} \mid e \in E\} \cup \{\text{no}, \bot\}$ and Q'_2. The set of edges is

$$\begin{aligned} E' = &\{(p, v, q) \mid (p, v, q) \in E, v \in \{0, 1\}\} \\ &\cup \{(p, 0, q_e), (q_e, 0, q_e^{>0}), (q_e, 0, q_e^{=0}), (q_e^{=0}, 0, q), (q_e^{>0}, -1, q), \\ &\quad (q_e^{>0}, 0, \bot), (q_e^{=0}, -1, \bot) \mid e = (p, -1, q) \in E\} \\ &\cup \{(\text{no}, -1, \text{no}), (\text{no}, 0, \bot), (q_f, -1, \bot), (\bot, 0, \bot)\}. \end{aligned}$$

Intuitively, every time a play visits an edge with a decrement in (Q', E'), Opponent has to guess whether the counter value is zero or positive, and move accordingly to an intermediate location, where Reacher can move to the actual target of the edge in (Q, E) or to a checking module where the game ends.

The objective of the game on (Q', E') is $(\bot, 0)$. As we can see in Figure 5, Reacher has a winning strategy in every location $q_e^{=0}$ when the counter value is positive, and in every location $q_e^{>0}$ when the counter value is zero. □

In the construction for the reverse reduction, when a player chooses any edge with a negative label and the counter value is less than the value that should be subtracted, then the adversary of this player has a winning move. Whereas this is no problem in a non-blocking VASS, such an edge would be forbidden in a VASS.

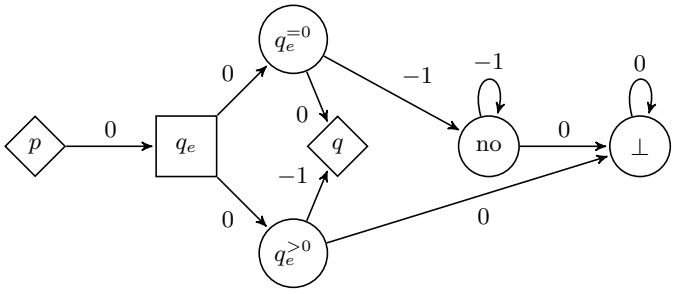


Fig. 5. Gadget to replace an edge $e = (p, -1, q)$ in the reduction from Reach-NBVASS$^1_1(1)$ to Reach-VASS$^1_1(0)$

*Proof (*PSPACE*-membership).* We show a polynomial-time reduction, that preserves the short-range property, from Reach-VASS$_1(0)$ to Reach-NBVASS$_1(1)$. Consider a reachability game on a VASS (Q, E), where the objective is $(q_f, 0)$, with $q_f \in Q_1$. Let Q_E be the set $\{q_e \mid e \in E \cap (Q \times (\mathbb{Z} \setminus \mathbb{N}) \times Q)\}$. We build the non-blocking VASS (Q', E'), where $Q' = Q \cup Q_E \cup \{\text{no}_\text{R}, \text{no}_\text{O}, \bot\}$, $Q'_1 = Q_1 \cup \{q_e \in Q_E \mid e \in Q_2 \times \mathbb{Z} \times Q\} \cup \{\text{no}_\text{R}, \text{no}_\text{O}, \bot\}$, $Q'_2 = Q' \setminus Q'_1$, and E' is obtained from E by splitting every edge (p, v, q) such that $v \in -\mathbb{N}$ into two edges $(p, 0, q_e)$ and (q_e, v, q) and by adding an edge from every location q_e to the "no"-location that corresponds to the owner of p, as well as additional edges between no_O, no_R and $\bot$, as depicted in the Figures 6 and 7.

More precisely, E' is the union of the sets of edges:

- $\{(p, v, q) \mid (p, v, q) \in E, x \in \mathbb{N}\}$;
- $\{(p, 0, q_e), (q_e, v, q) \mid e = (p, v, q) \in E, x < 0\}$;
- $\{(q_e, x + 1, \text{no}_\text{R}) \mid e = (p, v, q) \in E, v < 0, p \in Q_1\}$;
- $\{(q_e, x + 1, \text{no}_\text{O}) \mid e = (p, v, q) \in E, v < 0, p \in Q_2\}$;
- extra edges $\{(\text{no}_\text{R}, -1, \text{no}_\text{R}), (\text{no}_\text{R}, 0, \bot), (\text{no}_\text{O}, 1, \bot), (q_f, 1, \bot), (\bot, 0, \bot)\}$.

The non-blocking VASS (Q', E') is designed such that a play in it corresponds to a play in the VASS (Q, E). Let us consider a location $q_e \in Q_E$, for an edge (p, v, q) in E. Note that $v < 0$ and that the owner of q_e is not the owner of p. In the play on the VASS, the edge (p, v, q) can only be taken if the counter value is at least $-v$. If a player goes to q_e, i.e., simulates the choice of the edge (p, v, q),

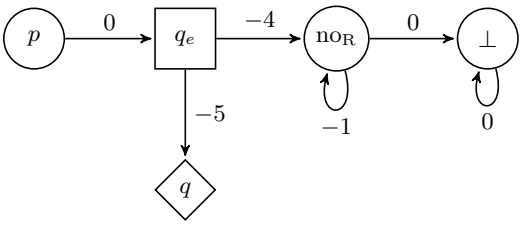


Fig. 6. Gadget to replace an edge $e = (p, -5, q)$ from a Reacher location in the reduction from Reach-VASS$^1_1(0)$ to Reach-NBVASS$^1_1(1)$

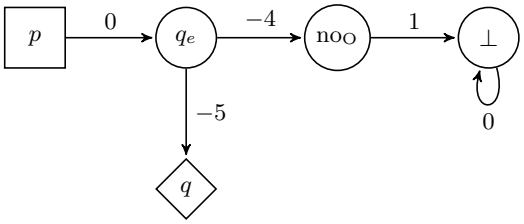


Fig. 7. Gadget to replace an edge $e = (p, -5, q)$ from an Opponent location in the reduction from Reach-VASS$^1_1(0)$ to Reach-NBVASS$^1_1(1)$

his adversary should win whenever the counter value is less than $-v$, by going to a "no"-location, as we can see in the Figures 6 and 7. □

4.3 The Case of Zero-Reachability on Non-blocking VASS

For non-blocking VASS, we prove that the set of winning configurations is downward closed when the reachability objective is $(q_f, 0)$ for a given q_f. Hence, to decide whether Reacher has a winning strategy, we compute for all locations the maximal initial value for which the pair (location, value) is winning and we look at the initial configuration.

Lemma 8. *Let (Q, E) be a non-blocking* VASS*. Consider a reachability game on (Q, E), where the objective is $(q_f, 0)$, where $q_f \in Q$. If the initial configuration (q_0, x) is winning, then every configuration (q_0, x') for $x' < x$ is winning.*

Proof. Let (q_0, x) be a winning configuration, and let s be a winning strategy for Reacher from (q_0, x). Consider any strategy s' for Opponent. The outcome of the strategies s and s' from (q_0, x) is a play π that Reacher wins, i.e., the play π eventually visits $(q_f, 0)$. Now, let us look at the outcome of the strategies s and s' from (q_0, x') for $x' < x$. It is a play π' that visits the same locations as π, and no edge is disabled because of the semantics of a non-blocking VASS. Moreover, the counter value in π' is after each move less than or equal to the counter value in the corresponding move of π. In particular, π' eventually visits q_f with counter value 0, hence Reacher wins. □

Algorithm 1 determines the winner of a reachability game on a non-blocking VASS when the objective counter value is 0. Its time complexity is exponential in the initial counter value. Accordingly, we call it only with 0 as initial counter value in the proof of Theorem 9.

Theorem 9. Reach-NBVASS$_1^1(0)$ *is in* P.

Proof. According to Lemma 8, we just need to compute for every location $q \in Q$ the maximal value x_m such that (q, x_m) is winning. We even do more: First, we compute the set Q_Z of locations from which Reacher has a winning strategy with initial counter value 0. For this purpose, we use the previous algorithm, and here the time complexity is polynomial. Second, we build the VASS (Q', E'), where $Q' = Q_Z \cup \{\bot\}$ and E' is the union of $E \cap (Q_Z \times \mathbb{Z} \times Q_Z)$ and of $\{(q, 1, \bot) \mid (q, v, q') \in E, q \in Q_Z, q' \notin Q_Z\} \cup \{(\bot, 0, \bot)\}$. In (Q', E'), the value 0 can only be reached in a location that belongs to Q_Z. Consider the reachability game on (Q', E'), where the objective is $Q \times 0$, like defined in [4]; deciding the winner in this game is in P. Moreover, Reacher has a winning strategy if, and only if, he has a strategy in Q to reach $(q, 0)$ for any $q \in Q_Z$, hence to reach $(q_f, 0)$. Indeed, if a play visits a location outside of Q_Z, then Opponent has a winning strategy. We conclude that deciding the winner of the reachability game is in P too. □

Algorithm 1. Solves Reach-NBVASS$_1(0)$

Data: A non-blocking VASS (Q, E), a location q_f, and a configuration (q_0, x_0)
Result: Does Reacher have a winning strategy to reach $(q_f, 0)$ from (q_0, x_0)?
begin
  Create a table M_q with $q \in Q$ as indices initialized to $-\infty$;
  $M_{q_f} \leftarrow 0$;
  repeat
    foreach $e = (q, v, q')) \in E$ **do**
      $M_q \leftarrow \max(M_q, M'_q - v)$
  until *a fixpoint is reached or* $M_{q_0} \geq x_0$;
  if $M_{q_0} \geq x_0$ **then return** *true*;
  else return *false*;

Note that we need the short-range property for our non-blocking VASS, else the algorithm could still require exponential time. For example, consider that there is an edge from q_0 to q_f with label 2^n and a self-loop on q_f with label -1. The algorithm would need $2^n + 1$ iterations to conclude that $(q_0, 0)$ is a winning configuration, whereas the size of the non-blocking VASS is linear in n because of the binary encoding.

5 Conclusion

In this paper, we studied three simple semantics for games on counter systems, and compared the complexity of reachability problems. In dimension two, every problem that we considered is undecidable. In dimension one, the decision problems associated to the counter value 0 are in P for the case of the non-blocking VASS semantics and PSPACE-complete for the two other semantics, when the counter system is short-ranged. Without this property, which guarantees that the set of all visited counter values is an interval, the complexity is not settled yet, to the best of our knowledge, and lies between EXPTIME and EXPSPACE.

Acknowledgement. The author would like to thank Dietmar Berwanger and Laurent Doyen for proposing the topic and for helping to organize the paper, and Marie van den Bogaard for patient reading and checking of the proofs.

References

1. Karp, R.M., Miller, R.E.: Parallel program schemata. The Journal of Computer and System Sciences 3(2), 147–195 (1969)
2. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)

3. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011)
4. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
5. Jurdziński, M., Laroussinie, F., Sproston, J.: Model checking probabilistic timed automata with one or two clocks. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 170–184. Springer, Heidelberg (2007)
6. Arul, A., Reichert, J.: The complexity of robot games on the integer line. In: Proceedings of the 11th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL). EPTCS, vol. 117, pp. 132–146 (2013)

Completeness Results for Generalized Communication-Free Petri Nets with Arbitrary Edge Multiplicities

Ernst W. Mayr and Jeremias Weihmann

Technische Universität München, 85748 Garching, Germany
{mayr,weihmann}@informatik.tu-muenchen.de
http://www14.in.tum.de/personen/index.html.en

Abstract. We investigate gcf-Petri nets, a generalization of communication-free Petri nets allowing arbitrary edge multiplicities, and characterized by the sole restriction that each transition has at most one incoming edge. We use canonical firing sequences with nice properties for gcf-PNs to show that the RecLFS, (zero-)reachability, covering, and boundedness problems of gcf-PNs are in PSPACE. By showing, how PSPACE-Turing machines can be simulated by gss-PNs, a subclass of gcf-PNs where additionally all transitions have at most one outgoing edge, we ultimately prove the PSPACE-completess of these problems for gss/gcf-PNs. Last, we show PSPACE-hardness as well as a doubly exponential space bound for the containment and equivalence problems of gss/gcf-PNs.

1 Introduction

In [12], Mayr proposed a non-primitive recursive algorithm for the general Petri net reachability problem, thus proving its decidability. For many restricted Petri net classes, a better complexity of the reachability problem can be shown. However, the nets of most Petri net classes for which the complexity of the reachability problem could be refined are subject to the restriction that all edges from places to transitions have multiplicity one. Well known examples of such nets with NP-complete reachability problems are communication-free Petri nets (cf-PNs/BPP-PNs), [3, 18], conflict-free Petri nets [7] and normal as well as sinkless Petri nets [8] (for the latter two, the promise problem variation of the reachability problem was considered). Remarkable examples for Petri net classes with general edge multiplicities and matching lower and upper bounds for the reachability problem are single-path Petri nets [6] (PSPACE-complete) and reversible Petri nets [13] (EXPSPACE-complete). For a more comprehensive overview, the reader is referred to [4].

Our ultimate goal is to gain insight into how general edge multiplicities influence the complexity of the reachability problem and several other classical problems. In this paper, we investigate a generalization of communication-free Petri nets. A cf-PN is a Petri net such that each transition has exactly one input place, connected by an edge with multiplicity one. Cf-PNs are closely related to

P.A. Abdulla and I. Potapov (Eds.): RP 2013, LNCS 8169, pp. 209–221, 2013.

Basic Parallel Processes defined in [1, 2] as well as to context-free (commutative) grammars [4, 10]. We call our generalization generalized communication-free Petri nets (gcf-PNs). The nets of this class are characterized by a single topological constraint, namely, that each transition has at most one input place, connected by an edge with arbitrary multiplicity.

For cf-PNs, tight bounds for the reachability problem are known. Esparza [3] showed NP-completeness while Yen [18] gave an alternative proof for NP-membership, based on canonical firing sequences. Both proofs (implicitly) rely on the fact that the RecLFS problem (recognize legal firing sequence, see [17]) is decidable in polynomial time due to a very easily checkable criterion. (The problem RecLFS asks if a given Parikh vector is enabled at some given marking.) For gcf-PNs, no such criterion exists (under the assumption P $\neq$ PSPACE) since the problem is PSPACE-complete as shown in Section 3.

In Section 3, we show PSPACE-hardness for the RecLFS, the reachability, the covering, and the boundedness problems of generalized S-Systems (gss-PNs) which are a subclass of gcf-PNs where each transition has at most one incoming and at most one outgoing edge, each with arbitrary edge multiplicity. This is interesting because almost all the problems considered in this paper have very low complexity for S-Systems (e.g., they are always bounded, the reachability problem is decidable in polynomial time [5], etc.). Furthermore, the covering, and the boundedness problems of cf-PNs are known to be NP-complete, and linear time (on RAMs), respectively [15]. Then, we derive canonical permutations of firing sequences of gcf-PNs, and use them to show PSPACE-completeness for the RecLFS, the reachability, and the covering problems of gcf-PNs.

In Section 4, we show the existence of canonical firing sequences that have stronger properties than the firing sequences obtained by canonical permutations. These canonical firing sequences resemble those given in [18] for cf-PNs. We use them to show PSPACE-completeness for the boundedness problem of gcf-PNs, and that the equivalence and containment problems of gcf-PNs are PSPACE-hard as well as decidable in doubly exponential space.

Due to space limitations, we provide detailed proofs for the lemmata and theorems in the technical report [14]. In this paper, we give proof sketches and the essential proof ideas. An exception is Lemma 6 where we derive the most central result, the existence of canonical permutations, for which a full proof is provided here.

2 Preliminaries

$\mathbb{Z}$, $\mathbb{N}_0$, and $\mathbb{N}$ denote the set of all integers, all nonnegative integers, and all positive integers, respectively, while $[a,b] = \{a, a+1, \ldots, b\} \subsetneqq \mathbb{Z}$, and $[k] = [1,k] \subsetneqq \mathbb{N}$. For two vectors $u, v \in \mathbb{Z}^k$, we write $u \geq v$ if $u_i \geq v_i$ for all $i \in [k]$, and $u > v$ if $u \geq v$ and $u_i > v_i$ for some $i \in [k]$. When k is understood, $\boldsymbol{a}$ denotes, for a number $a \in \mathbb{Z}$, the k-dimensional vector with $\boldsymbol{a}_i = a$ for all $i \in [k]$.

A Petri net N is a 3-tuple (P, T, F) where P is a finite set of n places, T is a finite set of m transitions with $P \cap T = \emptyset$, and $F : P \times T \cup T \times P \to \mathbb{N}_0$

is a flow function. Throughout this paper, n and m will always refer to the number of places resp. transitions of the Petri net under consideration, and $W = \max\{F(p,t), F(t,p) \mid p \in P, t \in T\}$ to the largest value of its flow function. Usually, we assume an arbitrary but fixed order on P and T, respectively. With respect to this order on P, we can consider an n-dimensional vector v as a function of P, and, abusing the notation, write $v(p)$ for the entry of v corresponding to place p. Analogously, we write $v(t)$ in context of an m-dimensional vector and a transition t.

A marking μ (of N) is a vector of $\mathbb{N}_0^n$. A pair $(N, \mu^{(0)})$ such that $\mu^{(0)}$ is a marking of N is called a marked Petri net, and $\mu^{(0)}$ is called its initial marking. We will omit the term "marked" if the presence of a certain initial marking is clear from the context.

For a transition $t \in T$, ${}^\bullet t$ ($t^\bullet$, resp.) is the preset (postset, resp.) of t and denotes the set of all places p such that $F(p,t) > 0$ ($F(t,p) > 0$, resp.). Analogously, the sets ${}^\bullet p$ and $p^\bullet$ of transitions are defined for the places $p \in P$. A Petri net (P, T, F) is a *generalized communication-free Petri net* (gcf-PN) if $|{}^\bullet t| \leq 1$ for all $t \in T$. A gcf-PN is a *generalized S-System Petri net* (gss-PN) if additionally $|t^\bullet| \leq 1$ for all $t \in T$.

A Petri net naturally corresponds to a directed bipartite graph with edges from P to T and vice versa such that there is an edge from $p \in P$ to $t \in T$ (from t to p, resp.) labelled with w if $0 < F(p,t) = w$ (if $0 < F(t,p) = w$, resp.). The label of an edge is called its multiplicity. If a Petri net is visualized, places are usually drawn as circles and transitions as bars. If the Petri net is marked by μ, then, for each place p, the circle corresponding to p contains $\mu(p)$ so called tokens.

For a Petri net $N = (P, T, F)$ and a marking μ of N, a transition $t \in T$ can be applied at μ producing a vector $\mu' \in \mathbb{Z}^n$ with $\mu'(p) = \mu(p) - F(p,t) + F(t,p)$ for all $p \in P$. The transition t is enabled at μ or in (N, μ) if $\mu(p) \geq F(p,t)$ for all $p \in P$. We say that t is fired at marking μ if t is enabled and applied at μ. If t is fired at μ, then the resulting vector μ' is a marking, and we write $\mu \xrightarrow{t} \mu'$. Intuitively, if a transition is fired, it first removes $F(p,t)$ tokens from p and then adds $F(t,p)$ tokens to p.

An element σ of T^* is called a transition sequence, and $|\sigma|$ denotes its length. For the empty transition sequence $\sigma = ()$, we define $\mu \xrightarrow{\sigma} \mu$. For a nonempty transition sequence $\sigma = t_1 \cdots t_k$, $t_i \in T$, we write $\mu^{(0)} \xrightarrow{\sigma} \mu^{(k)}$ if there are markings $\mu^{(1)}, \ldots, \mu^{(k-1)}$ such that $\mu^{(0)} \xrightarrow{t_1} \mu^{(1)} \xrightarrow{t_2} \mu^{(2)} \ldots \xrightarrow{t_k} \mu^{(k)}$. We write $\sigma_{(i,j)}$ for the subsequence $\sigma_i \cdot \sigma_{i+1} \cdots \sigma_j$, and $\sigma_{(i)}$ for the prefix of length i of σ, i.e., $\sigma_{(i)} = \sigma_{(1,i)}$.

A Parikh vector Φ, also known as firing count vector, is simply an element of $\mathbb{N}_0^m$. The Parikh map $\Psi : T^* \to \mathbb{N}_0^m$ maps each transition sequence σ to its Parikh image $\Psi(\sigma)$ where $\Psi(\sigma)(t) = k$ for a transition t if t appears exactly k times in σ. Note that each Parikh vector Φ is the Parikh image of some transition sequence. Furthermore, we write $t \in \Phi$ if $\Phi(t) > 0$, and $t \in \sigma$ if $t \in \Psi(\sigma)$. For a transition sequence $\sigma \in T^*$, we define ${}^\bullet\sigma = \bigcup_{t \in \sigma} {}^\bullet t$. $\Psi_{\text{first}}(\sigma)$ is the Parikh vector such that, for all transitions t, $\Psi_{\text{first}}(\sigma)(t) = 1$ if ${}^\bullet \bar{t} \neq {}^\bullet t$ for all transitions $\bar{t}$ in front

of the first occurrence of t in σ, and $\Psi_{\text{first}}(\sigma)(t) = 0$ otherwise. For $\sigma, \tau \in T^*$, $\sigma \dot{-} \tau \in T^*$ is obtained by deleting the first $\min\{\Psi(\sigma)(t), \Psi(\tau)(t)\}$ occurences of each transition t from σ.

If there is a marking μ' with $\mu \xrightarrow{\sigma} \mu'$, then we say that σ (the Parikh vector $\Psi(\sigma)$, resp.) is enabled at μ and leads from μ to μ'. For a marked Petri net $(N, \mu^{(0)})$, we call a transition sequence that is enabled at $\mu^{(0)}$ a firing sequence. A marking μ is called reachable if $\mu^{(0)} \xrightarrow{\sigma} \mu$ for some σ. The reachability set $\mathcal{R}(N, \mu^{(0)})$ of $(N, \mu^{(0)})$ consists of all reachable markings. We say that a marking μ can be covered if there is a reachable marking $\mu' \geq \mu$.

The displacement $\Delta : \mathbb{N}_0^m \to \mathbb{Z}^n$ maps Parikh vectors $\Phi \in \mathbb{N}_0^m$ onto the change of tokens at the places $p_1, \ldots, p_n$ when applying transition sequences with Parikh image Φ. That is, we have $\Delta(\Phi)(p) = \sum_{t \in T} \Phi(t) \cdot (F(t,p) - F(p,t))$ for all places p. Accordingly, we define the displacement $\Delta(\sigma)$ of a transition sequence σ by $\Delta(\sigma) := \Delta(\Psi(\sigma))$.

A Parikh vector or a transition sequence having nonnegative displacement at all places is called a nonnegative loop since, if it is fired at some marking, the loop can immediately be fired again at the resulting marking. A nonnegative loop having positive displacement at some place p is a positive loop (for p). A nonnegative loop with displacement 0 at all places is a zero-loop. For a marking μ, a transition sequence σ, and a subset $S \subseteq P$ of places, we define $\max(\mu, S) := \max_{p \in S} \mu(p)$, and $\max(\mu) := \max(\mu, P)$, as well as $\max(\mu, \sigma, S) := \max_{i \in [0, |\sigma|]} \max(\mu + \Delta(\sigma_{(i)}), S)$, and $\max(\mu, \sigma) := \max(\mu, \sigma, P)$.

The wipe-extension $\mathcal{P}^- = (P, T^-, F^-)$ of a Petri net $\mathcal{P} = (P, T, F)$ is obtained from $\mathcal{P}$ by introducing, for each place $p_i \in P$, a transition t_i^- with $F^-(p_i, t_i^-) = 1$.

Some marked Petri nets have reachability sets that are semilinear. A set $S \subseteq \mathbb{N}_0^n$ is semilinear, if there are a $k \in \mathbb{N}_0$ and linear sets $L_1, \ldots, L_k \subseteq \mathbb{N}_0^n$ such that $S = \bigcup_{i \in [k]} L_i$. A set $L \subseteq \mathbb{N}_0^n$ is linear, if there are $\ell \in \mathbb{N}_0$ and vectors $b, p_1, \ldots, p_\ell \in \mathbb{N}_0^n$ such that $L = \{b + \sum_{i \in [\ell]} a_i p_i \mid a_i \in \mathbb{N}_0, i \in [\ell]\}$. The vector b is the constant vector of L, while the vectors p_i are the periods of L. A semilinear representation of a semilinear set S is a set consisting of k pairs $(b_i, \{p_{i,1}, \ldots, p_{i,\ell_i}\})$, $i \in [k]$, for some $k \in \mathbb{N}_0$, such that $S = \bigcup_{i \in [k]} L_i$ where $L_i = \{b_i + \sum_{j \in [\ell_i]} a_{i,j} p_{i,j} \mid a_{i,j} \in \mathbb{N}_0, j \in [\ell_i]\}$. If two Petri nets allow the construction of semilinear representations of the respective reachability sets within a certain space bound, then many problems are decidable that are undecidable for Petri nets in general, and space bounds can be given as well. We will use this well known approach for the containment and the equivalence problems.

Throughout this paper we use a succinct encoding scheme. Every number is encoded in binary representation. A Petri net is encoded as an enumeration of places $p_1, \ldots, p_n$ and transitions $t_1 \ldots, t_m$ followed by an enumeration of the edges with their respective edge weight. A vector of $\mathbb{N}_0^k$ is encoded as a k-tuple. If we regard a tuple as an input (e.g. a marked Petri net), then it is encoded as a tuple of the encodings of the particular components. $\text{size}(\mathcal{P})$ denotes the encoding size of a marked Petri net $\mathcal{P}$. Analogously, $\text{size}(\mathcal{P}, \mu)$ is the encoding size of $\mathcal{P}$ together with an additional marking μ.

In this paper, we study the following problems for gcf-PNs.

- RecLFS: Given a gcf-PN $\mathcal{P}$ and a Parikh vector Φ, is Φ enabled in $\mathcal{P}$?
- Reachability: Given a gcf-PN $\mathcal{P}$ and a marking μ, is μ reachable in $\mathcal{P}$?
- Zero-Reachability: Given a gcf-PN $\mathcal{P}$, is the empty marking reachable in $\mathcal{P}$?
- Covering: Given a gcf-PN $\mathcal{P}$ and a marking μ, is μ coverable in $\mathcal{P}$?
- Boundedness: Given a gcf-PN $\mathcal{P}$, is there, for each $k \in \mathbb{N}$, a reachable marking μ with $\max(\mu) \geq k$?
- Containment: Given two gcf-PNs $\mathcal{P}$ and $\mathcal{P}'$, is $\mathcal{R}(\mathcal{P}) \subseteq \mathcal{R}(\mathcal{P}')$?
- Equivalence: Given two gcf-PNs $\mathcal{P}$ and $\mathcal{P}'$, is $\mathcal{R}(\mathcal{P}) = \mathcal{R}(\mathcal{P}')$?

We remark that the input size of a problem instance consists of the encodings of all entities that are declared as being "given" in the respective problem statement.

3 Canonical Permutations, and the RecLFS, (Zero-)Reachability, and Covering Problems

In this section, we first show PSPACE-completeness of the RecLFS problem. Then, we describe a procedure that, given a gcf-PN $\mathcal{P} = (P, T, F, \mu^{(0)})$, and a firing sequence σ with $\mu^{(0)} \xrightarrow{\sigma} \mu$, produces a permutation σ' of σ enabled at $\mu^{(0)}$ such that every marking reached while firing σ' has encoding size polynomial in $\text{size}(\mathcal{P}, \mu)$. We use these sequences to decide the reachability, and the covering problems in polynomial space, proving their PSPACE-completeness.

Lemma 1. *The RecLFS, the zero-reachability, the reachability, the covering, and the boundedness problems of gss-PNs are PSPACE-hard.*

Proof (Please note that, as indicated in the introduction, most of the proofs give the ideas. Fully detailed proofs are available in [14]). The proof is based on a generic reduction from each language $L \in$ PSPACE to each of the problems of interest mentioned in the lemma. We use the existence of a PSPACE-Turing machine M with certain properties deciding an arbitrary language $L \in$ PSPACE. Our logspace reduction maps the given word x to a gss-PN $\mathcal{P}$ and to a Parikh vector or a marking, corresponding to M and x. $\mathcal{P}$ simulates M in such a way that the Parikh Vector is enabled or the marking can be reached if and only if M accepts x. □

Theorem 1. *The RecLFS problem of general Petri nets is PSPACE-complete, even if restricted to gss-PNs.*

Proof. The PSPACE-hardness of the RecLFS problem is shown in Lemma 1. Now observe that we can guess the order in which the transitions of the given Parikh vector Φ can be fired. Each marking obtained when firing this sequence has encoding size polynomial in the size of the Petri net and Φ. □

Next, we propose four essential lemmata for the construction of canonical permutations of firing sequences in gcf-PNs.

Lemma 2. *Let σ be a firing sequence of a gcf-PN $(N, \mu^{(0)})$. If a transition $t \in \Psi_{first}(\sigma_{(i+1,|\sigma|)})$ is enabled at $\mu^{(0)} + \Delta(\sigma_{(i)})$, then $\sigma_{(i)} \cdot t \cdot (\sigma_{(i+1,|\sigma|)} \dot{-} t)$ is a firing sequence.*

Proof. We can shift a transition which first consumes tokens of a place p to the front of the sequence, given that the initial marking has enough tokens for the transition. Iteratively applying this argument yields the lemma. □

Lemma 3. *Let (P, T, F) be a gcf-PN, σ a transition sequence, and μ, μ' markings with $\mu + \Delta(\sigma) = \mu'$ and $\mu(p), \mu'(p) \geq W$ for all $p \in {}^\bullet\sigma$. Then, there is a permutation of σ enabled at μ (and leading to μ').*

Proof. The proof uses induction over the length of σ. Using Lemma 2, we generate a permutation $\tilde{\sigma} \cdot \bar{\sigma}$ of σ such that $\tilde{\sigma}$ is enabled at the marking with W tokens at all places of S and $\Delta(\tilde{\sigma})(p) \in [-W, -1]$ for all $p \in {}^\bullet\bar{\sigma}$. Applying the induction hypothesis to $\bar{\sigma}$ and $\tilde{\sigma}$ yields permutations $\bar{\sigma}'$ and $\tilde{\sigma}'$ with $\mu \xrightarrow{\bar{\sigma}' \cdot \tilde{\sigma}'} \mu'$. □

Lemma 4. *Let $\mathcal{P} = (P, T, F)$ be a gcf-PN with largest edge multiplicity W, and $S \subseteq P$ a subset of places. Further, let $\sigma = \sigma_1 \cdots \sigma_k$, $\sigma_i \in T$, be a transition sequence of $\mathcal{P}$ with $\mu^{(0)} \xrightarrow{\sigma_1} \mu^{(1)} \ldots \mu^{(k-1)} \xrightarrow{\sigma_k} \mu^{(k)}$ such that*

(a) ${}^\bullet\sigma \subseteq S$,
(b) $\mu^{(i-1)}({}^\bullet\sigma_i) = \max(\mu^{(i-1)}, S)$ *for all $i \in [k]$ (i.e., each transition removes tokens from a place of S with the maximum number of tokens), and*
(c) $\max(\mu^{(k)}, S) > \max(\mu^{(0)}, S) + 2|S|W$.

Then, for some $i \in [1, k-1]$, the suffix $\sigma_{(i,k)}$ is a positive loop.

Proof. By (c), there is an interval $[x, y] \subsetneq [\max(\mu^{(0)}, S), \max(\mu^{(k)}, S)]$ of size $2W$ such that $\mu^{(k)}(p) \notin [x, y]$ for all places $p \in S$. Let $i \in [0, k-1]$ be the smallest index such that $\max(\mu^{(j)}, S) \geq x + W$ for all $j \in [i, k]$.

For all $p \in S$ with $\mu^{(i)}(p) \in [x, y]$ we have $\mu^{(k)}(p) > b$. By (a), (b) and the choice of i, the numbers of tokens of these places will never be below x at all $\mu^{(j)}$ with $j \in [i, k]$. Additionally, the numbers of tokens at all other places are monotonically increasing from $\mu^{(i)}$ to $\mu^{(k)}$. Hence, $\sigma_{(i+1,k)}$ is a positive loop. □

Lemma 5. *Let $N = (P, T, F)$ be a Petri net with n places and m transitions, and let W be the largest edge multiplicity of N. Then, there is a finite set $\mathcal{H}(N) = \{\Phi^{(1)}, \ldots, \Phi^{(k)}\} \subsetneq \mathbb{N}_0^m$ of nonnegative loops of N such that each loop of $\mathcal{H}(N)$ consists of at most $(1 + (n+m)W)^{n+m}$ transitions, and such that, for each nonnegative loop Φ of N, there are $a_1, \ldots, a_k \in \mathbb{N}_0$ with $\Phi = a_1\Phi^{(1)} + \ldots + a_k\Phi^{(k)}$.*

Proof. We can formulate the set of all nonnegative loops as the set of solutions of an appropriately formulated system of linear diophantine inequalities. Using Theorem 1 of [16], we obtain the result. □

Using these lemmata, we can show that firing sequences have canonical permutations with nice properties.

Lemma 6. *There is a constant c such that, for each gcf-PN $\mathcal{P} = (P, T, F, \mu^{(0)})$ and each firing sequence σ leading from $\mu^{(0)}$ to μ, there is a permutation φ of σ leading from $\mu^{(0)}$ to μ, and satisfying* $\max(\mu^{(0)}, \varphi) \leq (2nmW + \max(\mu^{(0)}) + \max(\mu))^{c(n+m)}$.

Proof. Let $\mathcal{P} = (P, T, F, \mu^{(0)})$ be a gcf-PN, and σ a firing sequence leading to some marking μ^σ. We define two special levels $\ell_{\text{big}} := \max\{W, \max(\mu^{(0)}), \max(\mu^\sigma) + 1\}$ and $\ell_{\text{fire}} := \ell_{\text{big}} + W$. Additionally, for $i \in [0, n]$, we define the levels $\ell_i := \ell_{\text{fire}} + W + i \cdot (\max\{(1 + (n+m)W)^{n+m}, 2n\} + 1)W$. A place p is *big* at a marking μ if $\mu(p) \geq \ell_{\text{big}}$, and *firing* if $\mu(p) \geq \ell_{\text{fire}}$.

Consider the following invariants for two transition sequences $\tilde{\sigma}$ and $\bar{\sigma}$:

(i) $\tilde{\sigma} \cdot \bar{\sigma}$ is a permutation of σ with $\mu^{(0)} \xrightarrow{\tilde{\sigma}} \mu^{\tilde{\sigma}} \xrightarrow{\bar{\sigma}} \mu^\sigma$,
(ii) $\max(\mu^{(0)}, \tilde{\sigma}) \leq \ell_n$, and
(iii) if there are $b \geq 1$ big places at $\mu^{\tilde{\sigma}}$, then $\max(\mu^{\tilde{\sigma}}) \leq \ell_{b-1}$.

For $\tilde{\sigma} = ()$ and $\bar{\sigma} = \sigma$, these invariants are obviously satisfied. Assume $|\tilde{\sigma}| < |\sigma|$, and that $\tilde{\sigma}$ and $\bar{\sigma}$ satisfy the invariants. We show how to extend $\tilde{\sigma}$ at the end to a longer transition sequence $\tilde{\sigma}^{\text{new}}$ and obtain a corresponding sequence $\bar{\sigma}^{\text{new}}$ such that $\tilde{\sigma}^{\text{new}}$ and $\bar{\sigma}^{\text{new}}$ again satisfy the invariants.

First, consider the case that there are no firing places at $\mu^{\tilde{\sigma}}$. Then, we set $\tilde{\sigma}^{\text{new}} := \tilde{\sigma} \cdot \bar{\sigma}_{(1)}$, and $\bar{\sigma}^{\text{new}} := \bar{\sigma}_{(2,|\bar{\sigma}|)}$. $\tilde{\sigma}^{\text{new}}$ and $\bar{\sigma}^{\text{new}}$ obviously satisfy property (i). For (ii) and (iii) notice that, for each big place p of $\mu^{\tilde{\sigma}} + \Delta(\bar{\sigma}_{(1)})$, we have $(\mu^{\tilde{\sigma}} + \Delta(\bar{\sigma}_{(1)}))(p) \leq \mu^{\tilde{\sigma}}(p) + W < \ell_{\text{fire}} + W = \ell_0$.

Next, consider the case that there are firing places at $\mu^{\tilde{\sigma}}$. Let S be the set of big places at $\mu^{\tilde{\sigma}}$ and $b = |S| \geq 1$ their number. The number of tokens of a big place $p^* \in S$ as a function of time is illustrated in (a) of Figure 1. We initialize an empty transition sequence $\alpha \leftarrow ()$, as well as $\bar{\sigma}' \leftarrow \bar{\sigma}$. As long as there is a firing place $p \in S$ at $\mu^{\tilde{\sigma}} + \Delta(\alpha)$, we select the transition $t \in \Psi_{\text{first}}(\bar{\sigma}')$ with $p = {}^\bullet t$, and set $\alpha \leftarrow \alpha \cdot t$, as well as $\bar{\sigma}' \leftarrow \bar{\sigma}' \dot{-} t$. Notice that t must exist since $\bar{\sigma}'$ must reduce the number of tokens at p in order to reach $\mu^\sigma(p)$. By Lemma 2, $\tilde{\sigma} \cdot \alpha \cdot \bar{\sigma}'$ is a firing sequence with $\mu^{(0)} \xrightarrow{\tilde{\sigma}} \mu^{\tilde{\sigma}} \xrightarrow{\alpha} \mu^\alpha \xrightarrow{\bar{\sigma}'} \mu^\sigma$, and α is nonempty since $\mu^{\tilde{\sigma}}$ has a firing place, see (b) of Figure 1.

Now, consider the nonnegative loop Φ with the largest component sum such that $\Phi \leq \Psi(\alpha)$. Using Lemma 5, we decompose Φ into short nonnegative loops $\Phi^{(1)}, \ldots, \Phi^{(k)}$, each with component sum at most $(1 + (n+m)W)^{n+m}$. Since $\mu^{\tilde{\sigma}}(p) \geq W$ for all $p \in S$ and ${}^\bullet t \in S$ for all $t \in \Phi^{(j)}$, $j \in [k]$, we can use Lemma 3 to find transition sequences $\tau^{(1)}, \ldots, \tau^{(k)}$ with $\Psi(\tau^{(j)}) = \Phi^{(j)}$, $j \in [k]$, such that $\tau := \tau^{(1)} \cdots \tau^{(k)}$ is enabled at $\mu^{\tilde{\sigma}}$. Let $\mu^{\tilde{\sigma}} \xrightarrow{\tau} \mu^\tau$. For each $p \in S$, we observe $\Delta(\Phi)(p) < W$. To see this, assume $\Delta(\Phi)(p) \geq W$. By the maximality of Φ, $\Psi(\alpha) - \Phi$ doesn't contain a transition t with $p = {}^\bullet t$. Therefore, $\Delta(\alpha)(p) = \Delta(\Phi)(p) + \Delta(\Psi(\alpha) - \Phi)(p) \geq W$. But then, $\mu^{\tilde{\sigma}}(p) + \Delta(\alpha)(p) \geq \ell_{\text{big}} + W = \ell_{\text{fire}}$, a contradiction to the fact that no place of S is firing. Since all $\tau^{(j)}$ are nonnegative loops, we obtain $\Delta(\tau^{(1)} \cdots \tau^{(j)})(p) \leq W$ for all $p \in S$ and $j \in [k]$. Furthermore, $|\tau^{(j)}| < (1 + (n+m)W)^{n+m}$ implies $\Delta(\tau^{(j)}_{(i)})(p) \leq (1 + (n+m)W)^{n+m}W$ for all $i \in [|\tau^{(j)}|]$ and $p \in P$. We obtain $\max(\mu^{\tilde{\sigma}} + \Delta(\tau^{(1)} \cdots \tau^{(j-1)}), \tau^{(j)}, S) \leq$

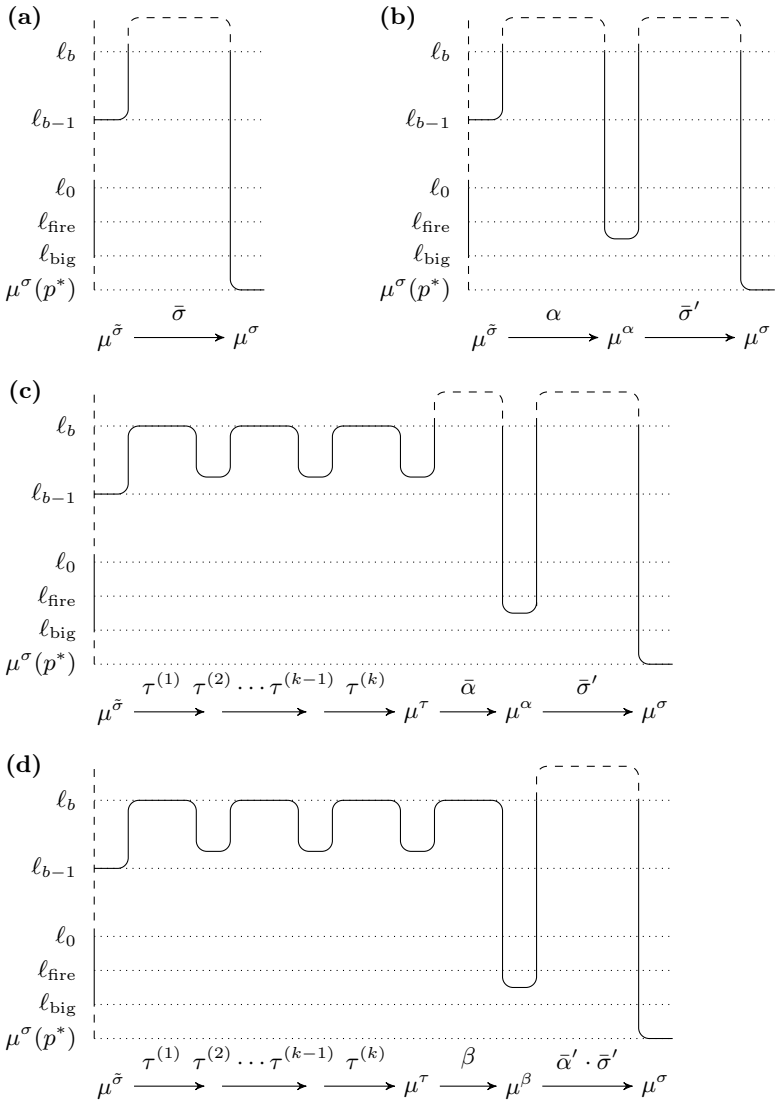


Fig. 1. (a)–(d) illustrate the development of the number of tokens at a place p^* which is big at $\mu^{\tilde{\sigma}}$ during certain steps of the permutation procedure described in Lemma 6. The number of tokens is bounded from above by the respective curve. The number of big places at $\mu^{\tilde{\sigma}}$ is b. Dashed lines symbolize that the number of tokens can become arbitrarily big.

$\ell_{b-1} + W + (1 + (n+m)W)^{n+m}W \leq \ell_b$ for all $j \in [k]$, and thus our first important intermediate result of the proof: $\max(\mu^{\tilde{\sigma}}, \tau, S) \leq \ell_b$.

In other words, the token numbers of places of S at all markings obtained while firing τ at $\mu^{\tilde{\sigma}}$ are at most ℓ_b.

We now consider $\Psi(\alpha) - \Phi$. Observing $\mu^{\tau}(p) \geq \mu^{\tilde{\sigma}}(p) \geq W$ and $\mu^{\sigma}(p) \geq W$ for all $p \in S$, and ${}^{\bullet}\bar{\alpha} \subseteq S$ for some transition sequence $\bar{\alpha}$ with $\Psi(\bar{\alpha}) = \Psi(\alpha) - \Phi$, we use Lemma 3 to find a transition sequence $\bar{\alpha}$ with $\Psi(\bar{\alpha}) = \Psi(\alpha) - \Phi$ that is enabled at μ^{τ}, see (c) of Figure 1.

We initialize another empty transition sequence $\beta \leftarrow ()$, as well as $\bar{\alpha}' \leftarrow \bar{\alpha}$. As long as there is a firing place of S at $\mu^{\tau} + \Delta(\beta)$, we select a place $p \in S$ with $\max(\mu^{\tau} + \Delta(\beta), S) = (\mu^{\tau} + \Delta(\beta))(p)$ and the transition $t \in \Psi_{\mathrm{first}}(\bar{\alpha}')$ with $p = {}^{\bullet}t$, and set $\beta \leftarrow \beta \cdot t$, as well as $\bar{\alpha}' \leftarrow \bar{\alpha}' \dot{-} t$. It is important to note the difference of this selection procedure compared to the one before. Here, we select a place of S with the largest number of tokens. Also note that β is nonempty since $\mu^{\tilde{\sigma}}$ has a firing place in S and $\mu^{\tau} \geq \mu^{\tilde{\sigma}}$. Let $\mu^{\beta} := \mu^{\tau} + \Delta(\beta)$. By Lemma 2, we observe $\mu^{\tau} \xrightarrow{\beta} \mu^{\beta}$, and $\bar{\alpha}'$ is enabled at μ^{β}. In total, we have $\mu^{\tilde{\sigma}} \xrightarrow{\tau} \mu^{\tau} \xrightarrow{\beta} \mu^{\beta} \xrightarrow{\bar{\alpha}' \cdot \bar{\sigma}'} \mu^{\sigma}$.

We observe $\max(\mu^{\tau}, S) = \max(\mu^{\tilde{\sigma}} + \Delta(\tau), S) \leq \max(\mu^{\tilde{\sigma}}, S) + W \leq \ell_{b-1} + W$. Now, for the sake of contradiction, assume that $\max(\mu^{\tau}, \beta, S) > \ell_b$. Then, $\max(\mu^{\tau} + \Delta(\beta_{(i)}), S) > \ell_b \geq \ell_{b-1} + W + 2nW \geq \max(\mu^{\tau}, S) + 2nW$ for some $i \in [|\beta|]$. But then, Lemma 4 implies that β contains a positive loop, a contradiction to the maximality of Φ. Therefore, $\max(\mu^{\tau}, \beta, S) \leq \ell_b$. We merge τ and β and obtain the nonempty transition sequence $\gamma := \tau \cdot \beta$.

Our observations can now be summarized as our second important intermediate result, also see (d) of Figure 1:

$$\mu^{\tilde{\sigma}} \xrightarrow{\gamma} \mu^{\beta} \xrightarrow{\bar{\alpha}' \cdot \bar{\sigma}'} \mu^{\sigma},\ |\gamma| > 0,\ \max(\mu^{\tilde{\sigma}}, \gamma, S) \leq \ell_b, \text{ and } \max(\mu^{\beta}, S) < \ell_{\mathrm{fire}}.$$

As the last step, consider the smallest $j \in [|\gamma|]$ such that the number of big places at $\mu^{\tilde{\sigma}} + \Delta(\gamma_{(j)})$ is at least $b+1$. If such a j does not exist, set $j := |\gamma|$. Now define $\tilde{\sigma}^{\mathrm{new}} := \tilde{\sigma} \cdot \gamma_{(j)}$, as well as $\bar{\sigma}^{\mathrm{new}} := \gamma_{(j+1,|\gamma|)} \cdot \bar{\alpha}' \cdot \bar{\sigma}'$. Observe that $\tilde{\sigma}^{\mathrm{new}}$ is longer than $\tilde{\sigma}$, and, together with $\bar{\sigma}^{\mathrm{new}}$, satisfies the invariants (i)–(iii). In particular, if there is still a big place at the end of the step, then every place that is big at some time during the step is also big at the end of it.

By iteratively applying this procedure, we obtain a permutation φ of σ such that $\mu^{(0)} \xrightarrow{\varphi} \mu^{\sigma}$ and $\max(\mu^{(0)}, \varphi) \leq \ell_n$, i.e., all markings obtained while firing φ contain at most ℓ_n tokens at each place. Note that if one of the values n, m, W is 0, then only the initial marking $\mu^{(0)}$ is reachable. Therefore, we can choose an appropriate constant c such that $\ell_n \leq (2nmW + \max(\mu^{(0)}) + \max(\mu))^{c(n+m)}$ for all possible inputs as defined at the beginning. □

We can use Lemma 6 to show that the reachability and the covering problems of gcf-PNs are PSPACE-complete.

Theorem 2. *The zero-reachability, the reachability, and the covering problems of gcf-PNs are PSPACE-complete, even if restricted to gss-PNs.*

Proof. The PSPACE-hardness of the RecLFS problem is shown in Lemma 1. By Lemma 6, we can guess a firing sequence to a reachable marking such that all intermediately observed markings have size polynomial in the input. Furthermore,

we can use the wipe-extension of the Petri net to reduce the covering problem to the reachability problem. □

4 Canonical Firing Sequences, and the Boundedness, Containment, and Equivalence Problems

The canonical permutation obtained in Section 3 is, by itself, not strong enough to show the membership of the boundedness problem in PSPACE or to yield algorithms deciding the containment and equivalent problems. Therefore, our first objective in this section is to distill a strong form of canonical firing sequences from canonical permutations.

Lemma 7. *There is a constant $c > 0$ such that, for each reachable marking μ of a gcf-PN $\mathcal{P} = (N, \mu^{(0)})$, there are transition sequences ξ, $\bar{\xi}$, $\alpha^{(1)}, \ldots, \alpha^{(k)}$, $\tau^{(1)}, \ldots, \tau^{(k)}$ for some $k \leq n \cdot \max(\mu)$ having the following properties.*

(a) $\xi = \alpha^{(1)} \cdot \tau^{(1)} \cdot \alpha^{(2)} \cdot \tau^{(2)} \cdots \alpha^{(k)} \cdot \tau^{(k)}$ *is a firing sequence leading from $\mu^{(0)}$ to μ.*
(b) $\bar{\xi} = \alpha^{(1)} \cdot \alpha^{(2)} \cdots \alpha^{(k)}$ *is fireable with $|\bar{\xi}| \leq (2nmW + \max(\mu^{(0)}))^{cn(n+m)}$.*
(c) *Each $\tau^{(i)}$, $i \in [k]$, is a positive loop with $|\tau^{(i)}| \leq (2nmW + \max(\mu^{(0)}))^{cn(n+m)}$ enabled at some marking μ^* with $\max(\mu^*) \leq (2nmW + \max(\mu^{(0)}))^{c(n+m)}$ and $\mu^* \leq \mu^{(0)} + \Delta(\alpha^{(1)} \cdot \alpha^{(2)} \cdots \alpha^{(i)})$.*

Proof. Consider the wipe-extension $\mathcal{P}^- = (P, T^-, F^-, \mu^{(0)})$ of $\mathcal{P}$. Each firing sequence σ of $\mathcal{P}$ can be extended by transitions $T^- \setminus T$ yielding a firing sequence σ' of $\mathcal{P}^-$ leading to the empty marking. By Lemma 6 there is a permutation of φ of σ' which intermediately only touches markings whose token numbers are at most exponential in the size of only $\mathcal{P}^-$.We partition φ into subsequences $\varphi^{(1)}, \ldots, \varphi^{(\ell)}$ which witness all markings which can potentially enable a zero-loop contained in φ. From these subsequences, we iteratively cut out all zero-loops which don't contain a zero-loop themselves, and store them for later use. Now, we discard all zero-loops which don't contain transitions of $T^- \setminus T$ since they are also zero-loops in $\mathcal{P}$, and therefore not needed. Let L denote the set of zero-loops that are kept. We remove all transitions of $T^- \setminus T$ from the sequences $\varphi^{(i)}$, $i \in [\ell]$, and all $\tau \in L$. The positive loops $\tau \in L$ constitute, appropriately numbered, the loops $\tau^{(j)}$ while an appropriate partition of $\varphi^{(1)} \cdots \varphi^{(\ell)}$ yields the sequences $\alpha^{(1)} \ldots \alpha^{(k)}$. The bound on the length of these sequences follows from the iterative removal of all zero-loops, and from the fact that each loop that was cut out, didn't contain a zero-loop itself. □

We call the sequence $\bar{\xi}$ the *backbone* of the canonical sequence under consideration. Using canonical firing sequences as constructed in Lemma 7, we can show the following lemma.

Lemma 8. *There is a constant c such that, for each gcf-PN $\mathcal{P} = (P, T, F, \mu^{(0)})$, $\mathcal{P}$ is unbounded if and only if there is a reachable marking μ with $\max(\mu) \geq \max(\mu^{(0)}) + \delta + 1$ if and only if there is a reachable marking μ with $\max(\mu) \in [\max(\mu^{(0)}) + \delta + 1, \max(\mu^{(0)}) + 2\delta + 1]$ where $\delta = (2nmW + \max(\mu^{(0)}))^{cn(n+m)} \cdot W$.*

Proof. The proof can be found in [14]. □

We can now prove the following theorem.

Theorem 3. *The boundedness problem of gcf-PNs is PSPACE-complete, even if restricted to gss-PNs.*

Proof. Since the PSPACE-hardness was shown in Lemma 1, it remains to be shown that it is in PSPACE. By Lemma 8, we have to check if a reachable marking μ as defined in the lemma exists. Hence, in order to check if $\mathcal{P}$ is unbounded, we guess μ in polynomial time, and check in polynomial space if μ is reachable by using Theorem 2. □

In the following, we show a doubly exponential space upper bound for the containment and the equivalence problems.

Lemma 9. *Given a gcf-PN $\mathcal{P} = (P, T, F, \mu^{(0)})$, we can construct a semilinear representation of $\mathcal{R}(\mathcal{P})$ in doubly exponential time in* size$(\mathcal{P})$.

Proof. Let $\mathcal{P}$ and $\mathcal{P}'$ be the gcf-PNs of interest. We consider all possible backbones of canonical firing sequences of $\mathcal{P}$. Each of these backbones $\bar{\xi}$ constitutes its own linear set, where the constant vector is the marking reached by the backbone, and the set of periods is the set of the displacements of all short positive loops enabled at some marking obtained while firing the backbone. Here, we use (c) of Lemma 7 to find all short loops enabled at a small marking μ^*, i.e., we compute all relevant periods before we start enumerating all relevant backbones. Lemma 7 ensures that the constructed semilinear representation represents $\mathcal{R}(\mathcal{P})$. □

Theorem 4. *The containment and the equivalence problems of gcf-PNs are PSPACE-hard and decidable in doubly exponential space, even if restricted to gss-PNs.*

Proof. The idea for the lower bound is to extend the given gss-PN $\mathcal{P}$ to a net $\mathcal{P}'$ in which all markings are reachable if and only if $\mathcal{P}$ is unbounded. Using this net, we can answer the boundedness problem by asking if $\mathcal{R}(\mathcal{P}^*) \subseteq \mathcal{R}(\mathcal{P}')$ (or $\mathcal{R}(\mathcal{P}^*) = \mathcal{R}(\mathcal{P}')$) where $\mathcal{P}^*$ is a gss-PN in which all markings are reachable. The upper bound for our problems is implied by Lemma 9, and bounds of [9] or [11] for semilinear representations. □

Our construction is similar to that given in [15] for cf-PNs which uses results of [18], and yields a semilinear representation of the reachability set of cf-PNs having single exponential encoding size, implying single exponential space algorithms for the containment and equivalence problems. The difference in the encoding sizes of these semilinear representation between cf-PNs and gcf-PNs does not result from the slight differences in the canonical firing sequences themselves (in fact, our canonical sequence can also be used to generate the semilinear representations for cf-PNs in single exponential time), rather, it results from the following.

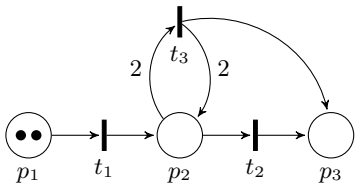


Fig. 2. The firing sequences $t_1t_1t_2t_2$ and $t_1t_2t_1t_2$ have the same Parikh image but only the first sequence intermediately enables the positive loop t_3

For cf-PNs, we used that each nonnegative loop that is intermediately enabled by some backbone can be partitioned into suitable nonnegative loops which are intermediately enabled by every other backbone with the same Parikh image. Therefore, it is sufficient to only consider one of these backbones. This results in a single exponential number of relevant backbones, and therefore in a single exponential number of linear sets, each of single exponential size. However, the same strategy fails in the case of gcf-PNs since the order of the transitions is much more relevant for gcf-PNs than for cf-PNs: firing transitions in a certain order can intermediately enable loops that cannot be partitioned further and that are not intermediately enabled by firing the same transitions in some other order. This is illustrated in Figure 2. Hence, to improve the doubly exponential space bound for the equivalence problem, some other or a refined approach will have to be found.

References

1. Christensen, S.: Distributed bisimularity is decidable for a class of infinite state-space systems. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 148–161. Springer, Heidelberg (1992)
2. Christensen, S., Hirshfeld, Y., Moller, F.: Bisimulation equivalence is decidable for basic parallel processes. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 143–157. Springer, Heidelberg (1993)
3. Esparza, J.: Petri nets, commutative context-free grammars, and basic parallel processes. Fundamenta Informaticae 31(1), 13–25 (1997)
4. Esparza, J., Nielsen, M.: Decibility issues for Petri nets - a survey. Journal of Informatik Processing and Cybernetics 30(3), 143–160 (1994)
5. Ha, L.M., Trung, P.V., Duong, P.T.H.: A polynomial-time algorithm for reachability problem of a subclass of Petri net and chip firing games. In: 2012 IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), pp. 1–6 (2012)
6. Howell, R.R., Jancar, P., Rosier, L.E.: Completeness results for single-path Petri nets. Information and Computation 106(2), 253–265 (1993)
7. Howell, R.R., Rosier, L.E.: Completeness results for conflict-free vector replacement systems. Journal of Computer and System Sciences 37(3), 349–366 (1988)

8. Howell, R.R., Rosier, L.E., Yen, H.-C.: Normal and sinkless Petri nets. In: Csirik, J.A., Demetrovics, J. (eds.) FCT 1989. LNCS, vol. 380, pp. 234–243. Springer, Heidelberg (1989)
9. Huynh, D.T.: The complexity of semilinear sets. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 324–337. Springer, Heidelberg (1980)
10. Huynh, D.T.: Commutative grammars: The complexity of uniform word problems. Information and Control 57(1), 21–39 (1983)
11. Huynh, D.T.: A simple proof for the sum upper bound of the inequivalence problem for semilinear sets. Elektronische Informationsverarbeitung und Kybernetik, 147–156 (1986)
12. Mayr, E.W.: An algorithm for the general Petri net reachability problem. In: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC 1981, pp. 238–246. ACM, New York (1981)
13. Mayr, E.W., Meyer, A.R.: The complexity of the word problems for commutative semigroups and polynomial ideals. Advances in Mathematics 46(3), 305–329 (1982)
14. Mayr, E.W., Weihmann, J.: Completeness Results for Generalized Communication-free Petri Nets with Arbitrary Edge Multiplicities. Technical Report TUM-I1335, Institut für Informatik, TU München (Jul 2013)
15. Mayr, E.W., Weihmann, J.: Results on equivalence, boundedness, liveness, and covering problems of BPP-Petri nets. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 70–89. Springer, Heidelberg (2013)
16. Pottier, L.: Minimal solutions of linear diophantine systems: bounds and algorithms. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 162–173. Springer, Heidelberg (1991)
17. Taoka, S., Watanabe, T.: Time complexity analysis of the legal firing sequence problem of Petri nets with inhibitor arcs. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E89-A, 3216–3226 (2006)
18. Yen, H.-C.: On reachability equivalence for BPP-nets. Theoretical Computer Science 179(1-2), 301–317 (1997)

Author Index