Ahmed Bouajjani
Wei-Ngan Chin (Eds.)

LNCS 6252

Automated Technology for Verification and Analysis

8th International Symposium, ATVA 2010
Singapore, September 2010
Proceedings

Springer

Ahmed Bouajjani   Wei-Ngan Chin (Eds.)

Automated Technology for Verification and Analysis

8th International Symposium, ATVA 2010
Singapore, September 21-24, 2010
Proceedings

Springer

Volume Editors

Ahmed Bouajjani
University of Paris 7
LIAFA, Case 7014
75205 Paris cedex 13, France
E-mail: abou@liafa.jussieu.fr

Wei-Ngan Chin
National University of Singapore
Department of Computer Science
13 Computing Drive
Singapore 117417
E-mail: chinwn@comp.nus.edu.sg

Library of Congress Control Number: 2010933601

CR Subject Classification (1998): D.2, D.1, F.3, C.2, D.3, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-15642-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-15642-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

These proceedings contain the papers presented at the 8th Internationl Symposium on Automated Technology for Verification and Analysis held during September 21–24, 2010 in Singapore. The primary objective of the ATVA conferences remains the same: to exchange and promote the latest advances of state-of-the-art research on theoretical and practical aspects of automated analysis, verification and synthesis.

From 72 papers submitted to ATVA 2010 in response to our call for papers, the Program Committee accepted 21 regular papers and 9 tool papers. Each paper received at least three reviews. The Program Committee worked hard to ensure that every submission received a rigorous and fair evaluation, with the final program selected after a 10-day online discussions via the Easychair system.

Our program also included three keynote talks and invited tutorials by Thomas A. Henzinger (IST Austria), Joxan Jaffar (National University of Singapore) and Igor Walukiewicz (CNRS, France). The conference organizers were truly grateful to have such distinguished researchers as keynote speakers for the symposium.

A new feature for the ATVA symposium this year were the two co-located workshops, Infinity 2010 (co-chaired by Yu-Fang Chen and Ahmed Rezine) and PMCW 2010 (co-chaired by Jun Sun and Hai Wang). We are delighted with the expanded scope, interactions and depth that the two workshops helped bring to the symposium.

Many people worked hard and offered their valuable time so generously to make ATVA 2010 successful. First and foremost, we would like to thank all authors who worked hard to complete and submit papers to the conference. The Program Committee members, reviewers and Steering Committee members also deserve special recognition. Without them, a competitive and peer-reviewed international symposium simply cannot take place.

Many individuals offered enthusiastic help to the conference. We are grateful to Jin Song Dong (Conference Chair) and Farn Wang (Steering Committee Chair) for their many invaluable suggestions and advice. We thank Yang Liu (Local Arrangements Chair) and Chunqing Chen (Finance and Registration Chair) for their support on local logistics. We also thank Florin Craciun for his invaluable help in the final proceedings preparation, and Tian Huat Tan for Web support.

We sincerely hope that the readers find the proceedings of ATVA 2010 informative and rewarding.

July 2010

Ahmed Bouajjani
Wei-Ngan Chin

Organization

General Chair

Jin Song Dong	National University of Singapore

Program Chairs

Ahmed Bouajjani	University of Paris, France
Wei-Ngan Chin	National University of Singapore

Steering Committee

E. Allen Emerson	University of Texas, USA
Teruo Higashino	Osaka University, Japan
Oscar H. Ibarra	University of California Santa Barbara, USA
Insup Lee	University of Pennsylvania, USA
Doron A. Peled	Bar Ilan University, Israel
Farn Wang	National Taiwan University, Taiwan
Hsu-Chun Yen	National Taiwan University, Taiwan

Program Committee

Jonathan Billington
Florin Craciun
E. Allen Emerson
Laurent Fribourg
Masahiro Fujita
Jaco Geldenhuys
Patrice Godefroid
Susanne Graf
Teruo Higashino
Franjo Ivančić
Naoki Kobayashi
Orna Kupferman
Marta Kwiatkowska
Insup Lee
Xuandong Li
Brendan Mahony
Andrew Martin
Ken McMillan
Kedar Namjoshi
Hua Phung Nguyen
Richard Paige
Madhusudan Parthasarathy
Doron Peled
Zongyan Qiu
Hiroyuki Seki
Natarajan Shankar
A. Prasad Sistla
Jun Sun
Mandana Vaziri
Helmut Veith
Farn Wang
Hai Wang
Ji Wang
Hsu-Chun Yen
Wang Yi
Sergio Yovine
Wenhui Zhang

Local Organization

Local Chair	Yang Liu (National University of Singapore)
Registration Chair	Chunqing Chen (National University of Singapore)
Web Master	Tian Huat Tan (National University of Singapore)

Sponsoring Institution

National University of Singapore

External Reviewers

Étienne André
Gogul Balakrishnan
Bernard Berthomieu
Aaron Bradley
Phil Brooke
Lei Bu
Thang Bui
Sebastian Burckhardt
Ana Busic
Aziem Chawdhary
Liqian Chen
Zhenbang Chen
Cristina David
Bruno Dutertre
Pontus Ekberg
Constantin Enea
Juan Pablo Galeotti
Guy Gallasch
Malay Ganai
Cristian Gherghina
Amar Gupta
Guanhua He
Andreas Holzer
Zhenjiang Hu
Linh Huynh
Visar Januzaj
Mark Kattenbelt
Gal Katz
Andrew King
Vijay Anand Korthikanti
Maciej Koutny
Ulrich Kuehne
Binh Le
Ton Chanh Le
Quang Loc Le
Lin Liu
Chenguang Luo
Jim McCarthy
Stphane Messika
Kozo Okano
Xiaoyue Pan
David Parker
Corneliu Popeea
Hongyang Qu
Alex Roederer
Sriram Sankaranarayanan
Mihaela Sighireanu
Nishant Sinha
Viorica Sofronie-Stokkermans
Martin Stigge
Yoshiaki Takata
Tho Quan Thanh
Ashutosh Trivedi
Hoang Truong
Krishna K. Venkatasubramanian
Razvan Voicu
Meng Wang
Shaohui Wang
Michael Westergaard
Jianhua Zhao
Florian Zuleger

Table of Contents

Probabilistic Automata on Infinite Words: Decidability and Undecidability Results[⋆]

Krishnendu Chatterjee and Thomas A. Henzinger

IST Austria (Institute of Science and Technology Austria)

Abstract. We consider probabilistic automata on infinite words with acceptance defined by safety, reachability, Büchi, coBüchi, and limit-average conditions. We consider quantitative and qualitative decision problems. We present extensions and adaptations of proofs for probabilistic finite automata and present an almost complete characterization of the decidability and undecidability frontier of the quantitative and qualitative decision problems for probabilistic automata on infinite words.

1 Introduction

Probabilistic automata and decision problems. Probabilistic automata for finite words were introduced in the seminal work of Rabin [Rab63], and have been extensively studied (see the book by [Paz71] on probabilistic automata and the survey of [Buk80]). Probabilistic automata on infinite words have been studied recently in the context of verification [BG05, BBG08]. We consider probabilistic automata on infinite words with acceptance defined by safety, reachability, Büchi, coBüchi, and limit-average conditions. We consider the *quantitative* and *qualitative* decision problems [Paz71, GO09]. The quantitative decision problems ask, given a rational $0 \leq \lambda \leq 1$, whether (a) *(equality)* there is a word with acceptance probability exactly λ; (b) *(existence)* there is a word with acceptance probability greater than λ; and (c) *(value)* for all $\varepsilon > 0$, there is a word with acceptance probability greater than $\lambda - \varepsilon$. The qualitative decision problems are the special cases of the quantitative problems with $\lambda \in \{ 0, 1 \}$. The qualitative and quantitative decision problems for probabilistic automata are the generalization of the emptiness and universality problem for deterministic automata.

Known results for probabilistic automata on infinite words. The decision problems for probabilistic automata on finite words have been extensively studied [Paz71, Buk80]. For probabilistic automata on infinite words it follows from the results of [BBG08] that for the coBüchi acceptance condition, the qualitative equality problem is undecidable and the qualitative existence problem is decidable, whereas for the Büchi acceptance condition, the qualitative equality problem is decidable and the qualitative existence problem is undecidable.

⋆ This research was supported by the European Union project COMBEST and the European Network of Excellence ArtistDesign.

Our results: quantitative decision problems. In [GO09] a simple and elegant proof for the undecidability of the quantitative decision problems for probabilistic *finite* automata was given. In Section 3 we show that the proof of [GO09] can be extended to show that the quantitative decision problems are undecidable for safety, reachability, Büchi, coBüchi, and limit-average conditions. In particular we show the undecidability of the quantitative equality problem for the special classes of probabilistic automata with safety and reachability acceptance conditions. The other undecidability results for quantitative decision problems are obtained by simple extensions of the result for quantitative equality.

Our results: qualitative decision problems. In Section 4 we show that all qualitative decision problems are decidable for probabilistic safety automata. We present a simple adaptation of a proof of [GO09] to give the precise characterization of the decidability and undecidability frontier for all qualitative decision problems for probabilistic reachability, Büchi, and coBüchi automata. We show that for probabilistic limit-average automata, the qualitative value problem is undecidable. The other two problems (qualitative equality and qualitative existence) remain open for probabilistic limit-average automata.

2 Definitions

In this section we present definitions for probabilistic automata, notions of acceptance for infinite words, and the decision problems.

2.1 Probabilistic Automata

Probabilistic automata. A *probabilistic automaton* $\mathcal{A}$ is a tuple $(Q, q_\iota, \Sigma, \mathcal{M})$ that consists of the following components:

1. a finite set Q of states and an initial state q_ι;
2. a finite alphabet Σ;
3. a set $\mathcal{M} = \{ M_\sigma \mid \sigma \in \Sigma \}$ of transition probability matrices M_σ; i.e., for $\sigma \in \Sigma$ we have M_σ is a transition probability matrix. In other words, for all $\sigma \in \Sigma$ the following conditions hold: (a) for all $q, q' \in Q$ we have $M_\sigma(q, q') \geq 0$; and (b) for all $q \in Q$ we have $\sum_{q' \in Q} M_\sigma(q, q') = 1$.

Infinite paths and words. Given a probabilistic automaton $\mathcal{A}$, an infinite path $\pi = (q_0, q_1, q_2, \ldots)$ is an infinite sequence of states such that for all $i \geq 0$, there exists $\sigma \in \Sigma$ such that $M_\sigma(q_i, q_{i+1}) > 0$. We denote by π a path in $\mathcal{A}$, and by Π the set of all paths. For a path π, we denote by $\mathrm{Inf}(\pi)$ the set of states that appear infinitely often in π. An infinite (resp. finite) word is an infinite (resp. finite) sequence of letters from Σ. For a finite word w we denote by $|w|$ the length of w. Given a finite or an infinite word w, we denote w_i as the i-th letter of the word (for a finite word w we assume $i \leq |w|$).

Cones and probability measure. Given a probabilistic automaton $\mathcal{A}$ and a finite sequence $\overline{q} = (q_0, q_1, \ldots, q_n)$ of states, the set $\mathrm{Cone}(\overline{q})$ consists of the set of paths π with prefix $\overline{q}$. Given a word $w \in \Sigma^\omega$, we first define a measure μ^w on cones as follows:

1. $\mu^w(\text{Cone}(q_\iota)) = 1$ and for $q_i \neq q_\iota$ we have $\mu^w(\text{Cone}(q_i)) = 0$;
2. for a sequence $\overline{q} = (q_0, q_1, \ldots, q_{n-1}, q_n)$ we have

$$\mu^w(\text{Cone}(\overline{q})) = \mu^w(\text{Cone}(q_0, q_1, \ldots, q_{n-1})) \cdot M_{w_n}(q_{n-1}, q_n).$$

The probability measure $\mathbb{P}^w_{\mathcal{A}}$ is the unique extension of the measure μ^w to the set of all measurable paths in $\mathcal{A}$.

2.2 Acceptance Conditions

Probabilistic automata on finite words. A probabilistic *finite* automaton consists of a probabilistic automaton and a set F of final states. The automaton runs over finite words $w \in \Sigma^*$ and for a finite word $w = \sigma_0\sigma_1 \ldots \sigma_n \in \Sigma^*$ it defines a probability distribution over Q as follows: let $\delta_0(q_\iota) = 1$ and for $i \geq 0$ we have $\delta_{i+1} = \delta_i \cdot M_{\sigma_i}$. The acceptance probability for the word w, denoted as $\mathbb{P}_{\mathcal{A}}(w)$, is $\sum_{q \in F} \delta_{|w|+1}(q)$.

Acceptance for infinite words. Let $\mathcal{A}$ be a probabilistic automaton and let $F \subseteq Q$ be a set of accepting (or target) states. Then we consider the following functions to assign values to paths.

1. *Safety condition.* The safety condition Safe(F) defines the set of paths in $\mathcal{A}$ that only visits states in F; i.e., $\text{Safe}(F) = \{\, (q_0, q_1, \ldots) \mid \forall i \geq 0.\ q_i \in F \,\}$.
2. *Reachability condition.* The reachability condition Reach(F) defines the set of paths in $\mathcal{A}$ that visits states in F at least once; i.e., $\text{Reach}(F) = \{\, (q_0, q_1, \ldots) \mid \exists i \geq 0.\ q_i \in F \,\}$.
3. *Büchi condition.* The Büchi condition Büchi(F) defines the set of paths in $\mathcal{A}$ that visits states in F infinitely often; i.e., $\text{Büchi}(F) = \{\, \pi \mid \text{Inf}(\pi) \cap F \neq \emptyset \,\}$.
4. *coBüchi condition.* The coBüchi condition coBüchi(F) defines the set of paths in $\mathcal{A}$ that visits states outside F finitely often; i.e., $\text{coBüchi}(F) = \{\, \pi \mid \text{Inf}(\pi) \subseteq F \,\}$.
5. *Limit-average condition.* The limit-average condition is a function $\text{LimitAvg} : \Pi \to \mathbb{R}$ that assigns to a path the long-run average frequency of the accepting states. Formally, for a state $q \in Q$, let $r(q) = 1$ if $q \in F$ and 0 otherwise, then for a path $\pi = (q_0, q_1, \ldots)$ we have

$$\text{LimitAvg}(\pi) = \lim\inf_{k \to \infty} \frac{1}{k} \cdot \sum_{i=0}^{k-1} r(q_i).$$

In sequel, we will consider Reach, Safe, Büchi, coBüchi and LimitAvg as functions from Π to $\mathbb{R}$. Other than LimitAvg, all the other functions only returns boolean values (0 or 1). Given an condition $\Phi : \Pi \to \mathbb{R}$, a probabilistic automaton $\mathcal{A}$ and a word w, we denote by $\mathbb{E}^w_{\mathcal{A}}(\Phi)$ the expectation of the function Φ under the probability measure $\mathbb{P}^w_{\mathcal{A}}$. Given a probabilistic automaton $\mathcal{A}$ and a condition Φ, we use the following notation: $\mathcal{A}(\Phi, w) = \mathbb{E}^w_{\mathcal{A}}(\Phi)$, and if the condition Φ is clear from the context we simply write $\mathcal{A}(w)$. If Φ is boolean, then $\mathcal{A}(\Phi, w)$ is the acceptance probability for the word w for the condition Φ in $\mathcal{A}$.

2.3 Decision Problems

We now consider the quantitative and qualitative decision problems.

Quantitative decision problems. Given a probabilistic automaton $\mathcal{A}$, a condition Φ, and a rational number $0 < \lambda < 1$, we consider the following questions:

1. *Quantitative equality problem.* Does there exist a word w such that $\mathbb{E}^w_{\mathcal{A}}(\Phi) = \lambda$. If Φ is boolean, then the question is whether there exists a word w such that $\mathbb{P}^w_{\mathcal{A}}(\Phi) = \lambda$.
2. *Quantitative existence problem.* Does there exist a word w such that $\mathbb{E}^w_{\mathcal{A}}(\Phi) > \lambda$. If Φ is boolean, then the question is whether there exists a word w such that $\mathbb{P}^w_{\mathcal{A}}(\Phi) > \lambda$. This question is related to emptiness of probabilistic automata: let $\mathcal{L}_{\mathcal{A}}(\Phi, \lambda) = \{w \mid \mathbb{P}^w_{\mathcal{A}}(\Phi) > \lambda\}$ be the set of words with acceptance probability greater than λ; then the set is non-empty iff the answer to the quantitative existence problem is yes.
3. *Quantitative value problem.* Whether the supremum of the values for all words is greater than λ, i.e., $\sup_{w \in \Sigma^\omega} \mathbb{E}^w_{\mathcal{A}}(\Phi) > \lambda$. If Φ is boolean, this question is equivalent to whether for all $\varepsilon > 0$, does there exist a word w such that $\mathbb{P}^w_{\mathcal{A}}(\Phi) > \lambda - \varepsilon$.

Qualitative decision problems. Given a probabilistic automaton $\mathcal{A}$, a condition Φ, we consider the following questions:

1. *Almost problem.* Does there exist a word w such that $\mathbb{E}^w_{\mathcal{A}}(\Phi) = 1$.
2. *Positive problem.* Does there exist a word w such that $\mathbb{E}^w_{\mathcal{A}}(\Phi) > 0$.
3. *Limit problem.* For all $\varepsilon > 0$, does there exist a word w such that $\mathbb{E}^w_{\mathcal{A}}(\Phi) > 1 - \varepsilon$.

If Φ is boolean, then in all the above questions $\mathbb{E}$ is replaced by $\mathbb{P}$.

3 Undecidability of Quantitative Decision Problems

In this section we study the computability of the quantitative decision problems. We show undecidability results for special classes of probabilistic automata for safety and reachability conditions, and all other results are derived from the results on these special classes. The special classes are related to the notion of absorption in probabilistic automata.

3.1 Absorption in Probabilistic Automata

We will consider several special cases with absorption condition and consider some simple equivalences.

Absorbing states. Given a probabilistic automaton $\mathcal{A}$, a state q is *absorbing* if for all $\sigma \in \Sigma$ we have $M_\sigma(q, q) = 1$ (hence for all $q' \neq q$ we have $M_\sigma(q, q') = 0$).

Acceptance absorbing automata. Given a probabilistic automaton $\mathcal{A}$ let F be the set of accepting states. The automaton is *acceptance-absorbing* if all states in F are absorbing. Given an acceptance-absorbing automaton, the following equivalence hold: $\text{Reach}(F) = \text{Büchi}(F) = \text{coBüchi}(F) = \text{LimitAvg}(F)$. Hence our goal would be to show hardness (undecidability) for acceptance-absorbing automata with reachability condition, and the results will follow for Büchi, coBüchi and limit-average conditions.

Absorbing automata. A probabilistic automaton $\mathcal{A}$ is absorbing if the following condition holds: let C be the set of absorbing states in $\mathcal{A}$, then for all $\sigma \in \Sigma$ and for all $q \in (Q \setminus C)$ we have $M_\sigma(q, q') > 0$ for some $q' \in C$. In sequel we will use C for absorbing states in a probabilistic automaton.

3.2 Absorbing Safety Automata

In sequel we write automata (resp. automaton) to denote probabilistic automata (resp. probabilistic automaton) unless mentioned otherwise. We now prove some simple properties of absorbing automata with safety condition.

Lemma 1. *Let $\mathcal{A}$ be an absorbing automaton with C as the set of absorbing states. Then for all words $w \in \Sigma^\omega$ we have $\mathbb{P}^w_{\mathcal{A}}(\mathit{Reach}(C)) = 1$.*

Proof. Let $\eta = \min_{q \in Q, \sigma \in \Sigma} \sum_{q' \in C} M_\sigma(q, q')$ be the minimum transition probability to absorbing states. Since $\mathcal{A}$ is absorbing we have $\eta > 0$. Hence for any word w, the probability to reach C after n steps is at least $1 - (1-\eta)^n$. Since $\eta > 0$, as $n \to \infty$ we have $1 - (1-\eta)^n \to 1$, and hence the result follows. ∎

Complementation of absorbing safety automata. Let $\mathcal{A}$ be an absorbing automaton, with the set F as accepting states, and we consider the safety condition $\text{Safe}(F)$. Without loss of generality we assume that every state in $Q \setminus F$ is absorbing. Otherwise, if a state in $q \in Q \setminus F$ is non-absorbing, we transform it to an absorbing state and obtain an automaton $\mathcal{A}'$. It is easy to show that for the safety condition $\text{Safe}(F)$ the automata $\mathcal{A}$ and $\mathcal{A}'$ coincide (i.e., for all words w we have $\mathcal{A}(\text{Safe}(F), w) = \mathcal{A}'(\text{Safe}(F), w)$). Hence we consider an absorbing safety automaton such that all states in $Q \setminus F$ are absorbing: it follows that every non-absorbing state is an accepting state, i.e., $Q \setminus C \subseteq F$. We claim that for all words w we have

$$\mathbb{P}^w_{\mathcal{A}}(\text{Safe}(F)) = \mathbb{P}^w_{\mathcal{A}}(\text{Reach}(F \cap C)).$$

Since every state in $Q \setminus C \subseteq F$ and all states in C are absorbing, it follows that $\mathbb{P}^w_{\mathcal{A}}(\text{Safe}(F)) \geq \mathbb{P}^w_{\mathcal{A}}(\text{Reach}(F \cap C))$. Since $\mathcal{A}$ is absorbing, for all words w we have $\mathbb{P}^w_{\mathcal{A}}(\text{Reach}(C)) = 1$ and hence it follows that

$$\mathbb{P}^w_{\mathcal{A}}(\text{Safe}(F)) \leq \mathbb{P}^w_{\mathcal{A}}(\text{Reach}(C)) \cdot \mathbb{P}^w_{\mathcal{A}}(\text{Safe}(F) \mid \text{Reach}(C)) = \mathbb{P}^w_{\mathcal{A}}(\text{Reach}(F \cap C)).$$

The complement automaton $\overline{\mathcal{A}}$ is obtained from $\mathcal{A}$ by changing the set of accepting states as follows: the set of accepting states $\overline{F}$ in $\overline{\mathcal{A}}$ is $(Q \setminus C) \cup (C \setminus F)$, i.e., all non-absorbing states are accepting states, and for absorbing states the accepting states are switched. Since $\overline{\mathcal{A}}$ is also absorbing it follows that for all words w we have $\mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Safe}(\overline{F})) = \mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Reach}(C \cap \overline{F})) = \mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Reach}(C \setminus F))$. Since $\overline{\mathcal{A}}$ is absorbing, it follows that for all words we have $\mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Reach}(C)) = 1$ and hence $\mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Safe}(\overline{F})) = \mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Reach}(C \setminus F)) = 1 - \mathbb{P}^w_{\overline{\mathcal{A}}}(\text{Reach}(C \cap F)) = 1 - \mathbb{P}^w_{\mathcal{A}}(\text{Reach}(C \cap F))$. It follows that for all words w we have $\mathcal{A}(\text{Safe}(F), w) = 1 - \overline{\mathcal{A}}(\text{Safe}(\overline{F}), w)$, i.e., $\overline{\mathcal{A}}$ is the complement of $\mathcal{A}$.

Remark 1. It follows from above that an absorbing automaton with safety condition can be transformed to an acceptance-absorbing automaton with reachability condition. So any hardness result for absorbing safety automata also gives a hardness result for acceptance-absorbing reachability automata (and hence also for Büchi, coBüchi, limit-average automata).

3.3 Undecidability for Safety and Acceptance-Absorbing Reachability Automata

Our first goal is to show that the quantitative equality problem is undecidable for safety and acceptance-absorbing reachability automata. The reduction is from the Post Correspondence Problem (PCP). Our proof is inspired by and is an extension of a simple elegant proof of undecidability for probabilistic finite automata [GO09]. We first define the PCP and some related notations.

PCP. Let $\varphi_1, \varphi_2 : \Sigma \to \{\, 0, 1, \ldots, k-1 \,\}^*$, and extended naturally to Σ^* (where $k = |\Sigma|$). The PCP asks whether there is a word $w \in \Sigma^*$ such that $\varphi_1(w) = \varphi_2(w)$. The PCP is undecidable [HU79].

Notations. Let $\psi : \{\, 0, 1, \ldots, k-1 \,\}^* \to [0,1]$ be the function defined as:

$$\psi(\sigma_1\sigma_2\ldots\sigma_n) = \frac{\sigma_n}{k} + \frac{\sigma_{n-1}}{k^2} + \cdots + \frac{\sigma_2}{k^{n-1}} + \frac{\sigma_1}{k^n} \,.$$

For $i \in \{\, 1, 2 \,\}$, let $\theta_i = \psi \circ \varphi_i : \Sigma^* \to [0,1]$. We first prove a property of θ_i, that can be derived from the results of [BMT77].

Lemma 2. *For a finite word $w \in \Sigma^*$ and a letter $\sigma \in \Sigma$ we have*

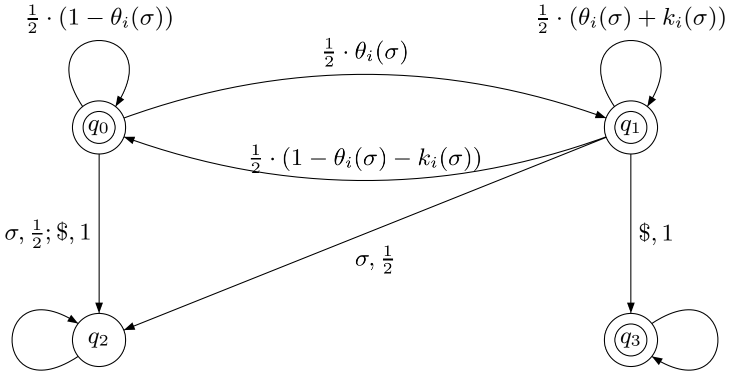
$$\theta_i(w \cdot \sigma) = \theta_i(\sigma) + \theta_i(w) \cdot k_i(\sigma),$$

where $k_i(\sigma) = k^{-|\varphi_i(\sigma)|}$.

Proof. Let $w = \sigma_1\sigma_2\ldots\sigma_n$, and let $\varphi_i(w) = b_1b_2\ldots b_m$ and $\varphi_i(\sigma) = a_1a_2\ldots a_\ell$. Then we have

$$\begin{aligned}
\theta_i(w \cdot \sigma) &= \psi \circ \varphi_i(w \cdot \sigma) \\
&= \psi(b_1b_2\ldots b_m a_1 a_2 \ldots a_\ell) \\
&= \frac{a_\ell}{k} + \cdots + \frac{a_1}{k^\ell} + \frac{b_m}{k^{\ell+1}} + \cdots + \frac{b_1}{k^{\ell+m}} \\
&= \left(\frac{a_\ell}{k} + \cdots + \frac{a_1}{k^\ell}\right) + \frac{1}{k^\ell} \cdot \left(\frac{b_m}{k} + \cdots + \frac{b_1}{k^m}\right) \\
&= \psi \circ \varphi_i(\sigma) + \frac{1}{k^\ell} \cdot (\psi \circ \varphi_i(w)) \\
&= \theta_i(\sigma) + \theta_i(w) \cdot k_i(\sigma).
\end{aligned}$$

The result follows. ∎



$\frac{1}{2}\cdot(1-\theta_i(\sigma))$
$\frac{1}{2}\cdot\theta_i(\sigma)$
$\frac{1}{2}\cdot(\theta_i(\sigma)+k_i(\sigma))$
q_0
q_1
$\frac{1}{2}\cdot(1-\theta_i(\sigma)-k_i(\sigma))$
$\sigma,\frac{1}{2};\$,1$
$\sigma,\frac{1}{2}$
$\$,1$
q_2
q_3

Fig. 1. Absorbing safety automata from PCP instances

The absorbing safety automata from PCP instances. Given an instance of the PCP problem we create two safety automata $\mathcal{A}_i = (Q, \Sigma \cup \{ \$ \}, \mathcal{M}^i, q_\iota, F)$, for $i \in \{ 1, 2 \}$ as follows (we assume $\$ \not\in \Sigma$):

1. *(Set of states and initial state).* $Q = \{ q_0, q_1, q_2, q_3 \}$ and $q_\iota = q_0$;
2. *(Accepting states).* $F = \{ q_0, q_1, q_3 \}$;
3. *(Transition matrices).* The set of transition matrices is as follows:
   (a) the states q_2 and q_3 are absorbing;
   (b) $M^i_\$(q_0, q_2) = 1$ and $M^i_\$(q_1, q_3) = 1$;
   (c) for all $\sigma \in \Sigma$ we have
      i. $M^i_\sigma(q_0, q_2) = M^i_\sigma(q_1, q_2) = \frac{1}{2}$;
      ii. $M^i_\sigma(q_0, q_0) = \frac{1}{2}\cdot(1-\theta_i(\sigma))$; $M^i_\sigma(q_0, q_1) = \frac{1}{2}\cdot\theta_i(\sigma)$;
      iii. $M^i_\sigma(q_1, q_0) = \frac{1}{2}\cdot(1-\theta_i(\sigma)-k_i(\sigma))$; $M^i_\sigma(q_1, q_1) = \frac{1}{2}\cdot(\theta_i(\sigma)+k_i(\sigma))$;

A pictorial description is shown in Fig 1. We will use the following notations: (a) we use $\widehat{\Sigma}$ for $\Sigma \cup \{ \$ \}$; (b) for a word $w \in \widehat{\Sigma}^\omega$ if the word contains a \$, then we denote by $\mathsf{first}(w)$ the prefix w' of w that does not contain a \$ and the first \$ in w occurs immediately after w' (i.e., $w'\$$ is a prefix of w). We now prove some properties of the automata $\mathcal{A}_i$.

1. The automata $\mathcal{A}_i$ is absorbing: since for every state and every letter the transition probability to the set $\{ q_2, q_3 \}$ is at least $\frac{1}{2}$; and q_2 and q_3 are absorbing.
2. Consider a word $w \in \widehat{\Sigma}^\omega$. If the word contains no \$, then the state q_2 is reached with probability 1 (as every round there is at least probability $\frac{1}{2}$ to reach q_2). If the word w contains a \$, then as soon as the input letter is \$, then the set $\{ q_2, q_3 \}$ is reached with probability 1. Hence the following assertion holds: the probability $\mathbb{P}^w_{\mathcal{A}}(\mathrm{Safe}(F))$ is the probability that after the word $w' = \mathsf{first}(w)$ the current state is q_1.

Lemma 3. *For all words $w \in \Sigma^*$, the probability that in the automaton $\mathcal{A}_i$ after reading w (a) the current state is q_1 is equal to $\frac{1}{2^{|w|}}\cdot\theta_i(w)$; (b) the current state is q_0 is equal to $\frac{1}{2^{|w|}}\cdot(1-\theta_i(w))$; and (c) the current state is q_2 is equal to $1-\frac{1}{2^{|w|}}$.*

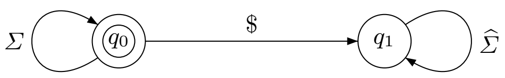


Fig. 2. Safety automaton $\mathcal{A}_3$

Proof. The result follows from induction on length of w, and the base case is trivial. We prove the inductive case. Consider a word $w \cdot \sigma$: by inductive hypothesis the probability that after reading w the current state is q_0 is $\frac{1}{2^{|w|}} \cdot (1 - \theta_i(w))$ and the current state is q_1 is $\frac{1}{2^{|w|}} \cdot \theta_i(w)$, and the current state is q_2 with probability $(1 - \frac{1}{2^{|w|}})$. After reading σ, if the current state is q_0 or q_1, with probability $\frac{1}{2}$ a transition to q_2 is made. Hence the probability to be at q_2 after $w \cdot \sigma$ is $(1 - \frac{1}{2^{|w|+1}})$ and the rest of the probability is to be at either q_0 and q_1. The probability to be at q_1 is

$$\begin{aligned}\frac{1}{2^{|w|+1}} \cdot \big((1 - \theta_i(w)) \cdot \theta_i(\sigma) + \theta_i(w) \cdot (\theta_i(\sigma) + k_i(\sigma)\big) &= \frac{1}{2^{|w|+1}} \cdot (\theta_i(\sigma) + \theta_i(w) \cdot k_i(\sigma)) \\ &= \frac{1}{2^{|w|+1}} \cdot \theta_i(w \cdot \sigma).\end{aligned}$$

The first equality follows by rearranging and the second equality follows from Lemma 2. Hence the result follows. ∎

The following lemma is an easy consequence.

Lemma 4. *For $i \in \{1, 2\}$, for a word $w \in \widehat{\Sigma}^\omega$, (a) if w contains no \$, then $\mathcal{A}_i(w) = 0$; (b) if w contains a \$, let $w' = \mathsf{first}(w)$, then $\mathcal{A}_i(w) = \frac{1}{2^{|w'|}} \cdot \theta_i(w')$.*

Constant automata and random choice automata. For any rational constant ν it is easy to construct an absorbing safety automaton $\mathcal{A}$ that assigns value ν to all words. Given two absorbing safety automata $\mathcal{A}_1$ and $\mathcal{A}_2$, and two non-negative rational numbers β_1, β_2 such that $\beta_1 + \beta_2 = 1$, it is easy to construct an automaton $\mathcal{A}$ (by adding an initial state with initial randomization) such that for all words w we have $\mathcal{A}(w) = \beta_1 \cdot \mathcal{A}_1(w) + \beta_2 \cdot \mathcal{A}_2(w)$. We will use the notation $\beta_1 \cdot \mathcal{A}_1 + \beta_2 \cdot \mathcal{A}_2$ for the automaton $\mathcal{A}$.

Safety automaton $\mathcal{A}_3$. We consider a safety automaton $\mathcal{A}_3$ on the alphabet $\widehat{\Sigma}$ as follows: (a) for a word w without any \$ the acceptance probability is 1; (b) for a word with a \$ the acceptance probability is 0. The automaton is shown in Fig 2 and the only accepting state is q_0. Consider the acceptance-absorbing reachability automaton $\mathcal{A}_4$ as follows: the automaton is same as in Fig 2 with accepting state as q_1: the automaton accepts a word with a \$ with probability 1, and for words with no \$ the acceptance probability is 0.

Theorem 1 (Quantitative equality problem). *The quantitative equality problem is undecidable for probabilistic safety and acceptance-absorbing reachability automata.*

Proof. Consider an automaton $\mathcal{A} = \frac{1}{3} \cdot \mathcal{A}_1 + \frac{1}{3} \cdot (1 - \mathcal{A}_2) + \frac{1}{3} \cdot \mathcal{A}_3$ (since $\mathcal{A}_2$ is absorbing we can complement $\mathcal{A}_2$ and obtain an automaton for $1 - \mathcal{A}_2$). We show that the quantitative equality problem for $\mathcal{A}$ with $\lambda = \frac{1}{3}$ is yes iff the answer to the PCP problem instance is yes. We prove the following two cases.

1. Suppose there is a finite word w such that $\varphi_1(w) = \varphi_2(w)$. Consider the infinite word $w^* = w\$w'$ where w' is an arbitrary infinite word. Then the acceptance probability of w^* in $\mathcal{A}_3$ is 0 (since it contains a \$). Since w^* contains a \$ and $\mathsf{first}(w^*) = w$, by Lemma 4 the acceptance probability of w^* in $\mathcal{A}$ is $\frac{1}{3} + \frac{1}{2^{|w|}} \cdot (\theta_1(w) - \theta_2(w))$. Since $\varphi_1(w) = \varphi_2(w)$ it follows that $\theta_1(w) - \theta_2(w) = 0$. It follows that the acceptance probability of w^* in $\mathcal{A}$ is $\frac{1}{3}$.
2. Consider an infinite word w. If the word contains no \$, then the acceptance probability of w in $\mathcal{A}_1$ and $\mathcal{A}_2$ is 0, and the acceptance probability in $\mathcal{A}_3$ is 1. Hence $\mathcal{A}$ accepts w with probability $\frac{2}{3} > \frac{1}{3}$. If the word w contains a \$, then $\mathcal{A}_3$ accepts w with probability 0. The difference of acceptance probability of w in $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\frac{1}{2^{|w'|}} \cdot (\theta_1(w') - \theta_2(w'))$, where $w' = \mathsf{first}(w)$. Hence the acceptance probability of w in $\mathcal{A}$ is $\frac{1}{3}$ iff $\theta_1(w') = \theta_2(w')$. Hence w' is a witness that $\varphi_1(w') = \varphi_2(w')$.

It follows that there exists a finite word w that is a witness to the PCP instance iff there exists an infinite word w^* in $\mathcal{A}$ with acceptance probability equal to $\frac{1}{3}$.

For acceptance-absorbing reachability automata: consider the same construction as above with $\mathcal{A}_3$ being replaced by $\mathcal{A}_4$, and the equality question for $\lambda = \frac{2}{3}$. Since $\mathcal{A}_1 - \mathcal{A}_2 < 1$, it follows that any witness word must contain a \$, and then the proof is similar as above. ∎

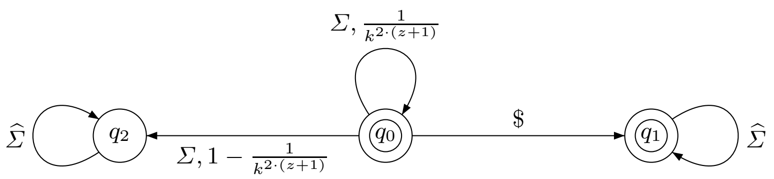
Quantitative existence and value problems. We will now show that the quantitative existence and the quantitative value problems are undecidable for probabilistic absorbing safety automata. We start with a technical lemma.

Lemma 5. *Let us consider a PCP instance. Let* $z = \max_{\sigma \in \Sigma}\{|\varphi_1(\sigma)|, |\varphi_2(\sigma)|\}$. *For a finite word w, if $\theta_1(w) - \theta_2(w) \neq 0$, then $(\theta_1(w) - \theta_2(w))^2 \geq \frac{1}{k^{2\cdot|w|\cdot z}}$.*

Proof. Given the word w, we have $|\varphi_i(w)| \leq |w| \cdot z$, for $i \in \{1, 2\}$. It follows that $\theta_i(w)$ can be expressed as a rational number $\frac{p_i}{q}$, where $q \leq k^{z\cdot|w|}$. It follows that if $\theta_1(w) \neq \theta_2(w)$, then $|\theta_1(w) - \theta_2(w)| \geq \frac{1}{k^{z\cdot|w|}}$. The result follows. ∎

Automaton $\mathcal{A}_5$. Consider the automaton shown in Fig 3: the accepting states are q_0 and q_1. For any input letter in Σ, from the initial state q_0 the next state is itself with probability $\frac{1}{k^{2\cdot(z+1)}}$, and the rest of the probability is to goto q_2. For input letter \$ the next state is q_1 with probability 1. The states q_1 and q_2 are absorbing. Given a word w, if there is no \$ in w, then the acceptance probability is 0; otherwise the acceptance probability is $\frac{1}{k^{2(z+1)\cdot|w'|}}$, where $w' = \mathsf{first}(w)$. Also note that the automaton $\mathcal{A}_5$ is absorbing.

Theorem 2 (Quantitative existence and value problems). *The quantitative existence and value problems are undecidable for probabilistic absorbing safety automata.*



$\Sigma, \frac{1}{k^{2\cdot(z+1)}}$
$\widehat{\Sigma}$
q_2
$\Sigma, 1-\frac{1}{k^{2\cdot(z+1)}}$
q_0
$\$$
q_1
$\widehat{\Sigma}$

Fig. 3. Safety automaton $\mathcal{A}_5$

Proof. Let us first consider the following automata: $\mathcal{B}_1 = \frac{1}{2}\cdot\mathcal{A}_1 + \frac{1}{2}\cdot(1-\mathcal{A}_2)$ and $\mathcal{B}_2 = \frac{1}{2}\cdot\mathcal{A}_2 + \frac{1}{2}\cdot(1-\mathcal{A}_1)$. The Cartesian product of $\mathcal{B}_1$ and $\mathcal{B}_2$ gives us the automaton $\mathcal{B}_3 = \frac{1}{4} - (\mathcal{A}_1 - \mathcal{A}_2)^2$. Finally, let us consider the automaton $\mathcal{B} = \frac{1}{2}\cdot\mathcal{B}_3 + \frac{1}{2}\cdot\mathcal{A}_5$. The automaton $\mathcal{B}$ can be obtained as an absorbing safety automaton. We show that there exists a word (or the sup value of words) is greater than $\frac{1}{8}$ iff the answer to PCP is yes.

1. If the answer to PCP problem is yes, consider the witness finite word w such that $\varphi_1(w) = \varphi_2(w)$. We construct an infinite word w^* for $\mathcal{B}$ as follows: $w^* = w\$w'$ where w' is an arbitrary infinite word. Since the word w^* contains a \$ by Lemma 4 we have the difference in acceptance probability of w^* in $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\frac{1}{2^{|w|}}\cdot(\theta_1(w)-\theta_2(w))$. Since $\varphi_1(w) = \varphi_2(w)$, it follows that $\mathcal{A}_1(w^*) = \mathcal{A}_2(w^*)$. Hence we have $\mathcal{B}(w^*) = \frac{1}{8} + \frac{1}{k^{2\cdot(z+1)\cdot|w|}} > \frac{1}{8}$.
2. We now show that if there is an infinite word with acceptance probability greater than $\frac{1}{8}$ (or the sup over the infinite words of the acceptance probability is greater than $\frac{1}{8}$) in $\mathcal{B}$, then the answer to the PCP problem is yes. Consider an infinite word w for $\mathcal{B}$. If w contains no \$, then $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_5$ accepts w with probability 0, and hence the acceptance probability in $\mathcal{B}$ is $\frac{1}{8}$. Consider an infinite word w that contains a \$. Let $w' = \mathsf{first}(w)$. If $\theta_1(w') \neq \theta_2(w')$, then by Lemma 4 and Lemma 5 we have $(\mathcal{A}_1(w)-\mathcal{A}_2(w))^2 \geq \frac{1}{2^{2|w'|}}\cdot\frac{1}{k^{2\cdot|w'|\cdot z}} \geq \frac{1}{k^{2\cdot|w'|\cdot(z+1)}}$. Since $\mathcal{A}_5(w) = \frac{1}{k^{2\cdot|w'|\cdot(z+1)}}$, it follows that $\mathcal{B}(w) \leq 1/8$. If $\theta_1(w') = \theta_2(w')$ (which implies $\varphi_1(w') = \varphi_2(w')$), then $\mathcal{B}(w) = \frac{1}{8} + \frac{1}{k^{2\cdot|w'|\cdot(z+1)}} > \frac{1}{8}$.

It follows from above that the quantitative existence and the quantitative value problems are undecidable for absorbing safety (and hence also for acceptance-absorbing reachability) automata. ∎

Corollary 1. *Given any rational* $0 < \lambda < 1$*, the quantitative equality, existence, and value problems are undecidable for probabilistic absorbing safety and acceptance-absorbing reachability automata.*

Proof. We have shown in Theorem 1 and Theorem 2 that the problems are undecidable for specific constants (e.g., $\frac{1}{3}, \frac{1}{8}$ etc.). We show below given the problem is undecidable for a constant $0 < c < 1$, the problem is also undecidable for any given rational $0 < \lambda < 1$. Given an automaton $\mathcal{B}$ we consider two cases:

1. If $\lambda \leq c$, then consider the automaton $\mathcal{A} = \frac{\lambda}{c}\cdot\mathcal{B} + (1-\frac{\lambda}{c})\cdot 0$. The quantitative decision problems for constant c in $\mathcal{B}$ is exactly same as the quantitative decision problems for $\mathcal{A}$ with λ.

2. If $\lambda \geq c$, then consider the automaton $\mathcal{A} = \frac{1-\lambda}{1-c} \cdot \mathcal{B} + \frac{\lambda-c}{1-c} \cdot 1$. The quantitative decision problems for c in $\mathcal{B}$ is exactly same as the quantitative decision problems for $\mathcal{A}$ with λ.

The desired result follows. ∎

Corollary 2 (Quantitative decision problems). *Given any rational* $0 < \lambda < 1$*, the quantitative equality, existence and value problems are undecidable for probabilistic safety, reachability, Büchi, coBüchi and limit-average automata.*

4 (Un-)Decidability of Qualitative Decision Problems

In this section we show that the qualitative decision problems are decidable for safety automata, and the limit problem is undecidable for reachability, Büchi, coBüchi and limit-average automata. The result is inspired from the proof of [GO09] that shows that the limit problem is undecidable for finite automata. If the finite automata construction of [GO09] were acceptance absorbing, the result would have followed for acceptance-absorbing reachability automata (and hence also for Büchi, coBüchi and limit-average automata). However, the undecidability proof of [GO09] for finite automata constructs automata that are not acceptance-absorbing. Simply changing the accepting states of the automata constructed in [GO09] to absorbing states does not yield the undecidability for accepting absorbing automata. We show that the construction of [GO09] can be adapted to prove undecidability of the limit problem for acceptance-absorbing automata with reachability condition. We first present a simple proof that for safety condition the almost and limit problem coincide.

Lemma 6. *Given an automaton with a safety condition the answer to the limit problem is yes iff the answer to the almost problem is yes.*

Proof. If the answer to the almost problem is yes, then trivially the answer to the limit problem is yes (since a witness for almost problem is a witness for all $\varepsilon > 0$ for the limit problem). We now show the converse is true. Consider an automaton $\mathcal{A}$ with ℓ states, and let $\eta > 0$ be the minimum non-zero transition probability in $\mathcal{A}$. We assume that the answer to the limit problem is yes and show that the answer to the almost problem is also yes. Consider $\varepsilon = \eta^{2^\ell}$, and let w be a word such that $\mathcal{A}(w) > 1-\varepsilon$. For a position i of w let $S_i = \{q \in Q \mid \delta_{|w_i|+1}(q) > 0\}$ be the set of states that have positive probability in the distribution of states after reading the prefix of length i of w. If a path in $\mathcal{A}$ leaves the set F of accepting states within k steps, then it leaves with probability at least η^k. Since w ensures the safety condition with probability at least $1 - \varepsilon = 1 - \eta^{2^\ell}$ it follows that for all $1 \leq i \leq 2^\ell$ we have $S_i \subseteq F$ (i.e., each S_i upto 2^ℓ is a subset of the accepting states). There must exist $0 \leq n < m \leq 2^\ell$ such that $S_n = S_m$. Consider the word $w^* = w_1 w_2 \cdots w_{n-1}(w_n w_{n+1} w_{m-1})^\omega$. The word w^* ensures the following: let $S_i^* = \{\, q \in Q \mid \delta_{|w_i^*|+1}(q) > 0 \,\}$, then for all $0 \leq i \leq n$ we have $S_i = S_i^*$, and for all $i \geq n$ we have $S_i^* \subseteq \bigcup_{j=n}^m S_i$. It follows that for all $i \geq 0$ we have $S_i^* \subseteq F$ and hence $\mathcal{A}(w^*) = 1$. The result follows. ∎

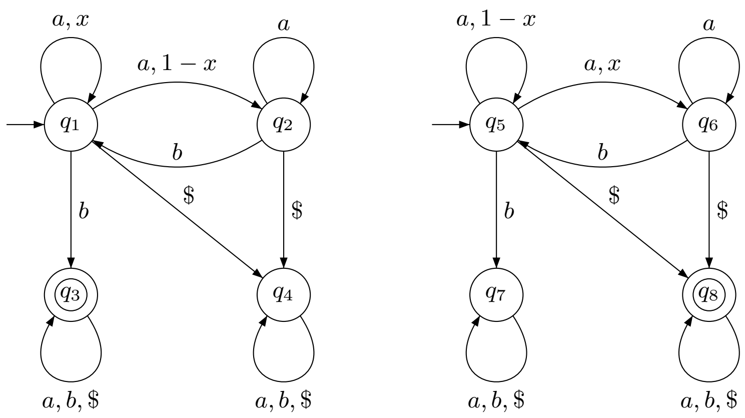


Fig. 4. Automata $\mathcal{A}_1$ (on left) and $\mathcal{A}_2$ (on right)

Qualitative decision problems for safety conditions. The above lemma shows that for safety condition the answers to the almost and the limit problem coincide. The almost and positive problem for safety condition can be solved by reduction to partial observable Markov decision processes (POMDPs). The decidability of the almost and positive problem for POMDPs with safety condition is known in literature [BGG09]. It follows that the qualitative decision problems are decidable for safety condition.

Lemma 7. *Consider the acceptance-absorbing automaton $\mathcal{A} = \frac{1}{2}\mathcal{A}_1 + \frac{1}{2} \cdot \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are shown in Fig 4. The following assertion hold: for all $\varepsilon > 0$, there exists a word w such that $\mathcal{A}(w) > 1 - \varepsilon$ iff $x > \frac{1}{2}$.*

Proof. Given $\varepsilon > 0$, a word w to be accepted with probability at least $1 - \frac{\varepsilon}{2}$ both $\mathcal{A}_1$ and $\mathcal{A}_2$ must accept it with probability at least $1 - \varepsilon$. Conversely, given a word w if it is accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$ with probability at least $1 - \varepsilon$, then $\mathcal{A}$ accepts it with probability at least $1 - \varepsilon$. Hence we show that for all $\varepsilon > 0$ there exist words w such that both $\mathcal{A}_1(w) \geq 1 - \varepsilon$ and $\mathcal{A}_2(w) \geq 1 - \varepsilon$ iff $x > \frac{1}{2}$.

We first observe that a word w to be accepted in $\mathcal{A}_1$ can have a \$ only after having reached its absorbing accepting state (since for all other states the input letter \$ leads to the non-accepting absorbing state q_4). A word w to be accepted in $\mathcal{A}_2$ must have a \$, and the \$ must occur when the current state is either q_5 or q_6. A word to be accepted in $\mathcal{A}_1$ must contain a b and the b must occur when the current state is q_1. Hence it suffices to consider words of the form $w^* = ww'$, where w' is an arbitrary infinite word and w is a finite word of the form $a^{n_0}ba^{n_1}b \dots a^{n_i}b\$$. Consider a word $a^n b$, for $n \geq 0$: the probability to reach the state q_3 from q_1 is x^n, and the probability to reach the absorbing non-accepting state q_7 from q_5 is $(1-x)^n$. Consider a word $w = a^{n_0}ba^{n_1}b \dots a^{n_i}b\$$: (a) the probability of reaching q_3 from q_1 in $\mathcal{A}_1$ is $1 - \prod_{k=0}^{i}(1 - x^{n_i})$; (b) the probability of reaching q_7 from q_5 in $\mathcal{A}_2$ is

$$(1-x)^{n_1} + (1-(1-x)^{n_1}) \cdot (1-x)^{n_2} + \cdots + \prod_{k=0}^{i-1}(1-(1-x)^{n_k}) \cdot (1-x)^{n_i}$$
$$= 1 - \prod_{k=0}^{i}(1-(1-x)^{n_i}).$$

If $x \le \frac{1}{2}$, then $x \le 1-x$, and hence $1-\prod_{k=0}^{i}(1-x^{n_i}) \le 1-\prod_{k=0}^{i}(1-(1-x)^{n_i})$. It follows that the acceptance probability of w in $\mathcal{A}_1$ is less than the rejection probability in $\mathcal{A}_2$, hence $\mathcal{A}(w) \le \frac{1}{2}$. It follows that if $x < \frac{1}{2}$, then for all words w we have $\mathcal{A}(w) \le \frac{1}{2}$.

To complete the proof we show that if $x > \frac{1}{2}$, then for all $\varepsilon > 0$, there exist words w such that both $\mathcal{A}_1(w) \ge 1-\varepsilon$ and $\mathcal{A}_2(w) \ge 1-\varepsilon$. Our witness words w^* will be of the following form: let $w = a^{n_0}ba^{n_1}b \ldots a^{n_i}b$ and we will have $w^* = w\$^\omega$. Our witness construction closely follows the result of [GO09]. Given the word w, the probability to reach q_3 from q_1 is

$$L_1 = 1 - \prod_{k=0}^{i}(1-x^{n_i});$$

hence $\mathcal{A}_1$ accepts w^* with probability L_1. Given the word w, since the last letter is a b, with probability 1 the current state is either q_7 or q_5. The probability to reach q_7 from q_5 for w is given by

$$L_2 = (1-x)^{n_1} + (1-(1-x)^{n_1}) \cdot (1-x)^{n_2} + \cdots + \prod_{k=0}^{i-1}(1-(1-x)^{n_k}) \cdot (1-x)^{n_i}$$
$$\le \sum_{k=0}^{i}(1-x)^{n_i} \qquad \text{since } \prod_{k=0}^{j}(1-(1-x)^{n_k}) \le 1 \text{ for all } j \le i.$$

Thus the acceptance probability for w in $\mathcal{A}_2$ is at least $1-L_2$. Hence given $\varepsilon > 0$, our goal is to construct a sequence $(n_0, n_1, n_2, \ldots, n_i)$ such that $L_1 \ge 1-\varepsilon$ and $L_2 \le \varepsilon$. For $\varepsilon > 0$, it suffices to construct an infinite sequence $(n_k)_{k\in\mathbb{N}}$ such that

$$\prod_{k\ge 0}(1-x)^{n_k} = 0; \quad \text{and} \quad \sum_{k\ge 0}(1-x)^{n_k} \le \varepsilon.$$

Then we can construct a finite sequence $(n_0, n_1, \ldots, n_i)$ of numbers such that we have both

$$\prod_{k=0}^{i}(1-x)^{n_k} \le \varepsilon; \quad \text{and} \quad \sum_{k=0}^{i}(1-x)^{n_k} \le 2\cdot\varepsilon,$$

as desired. Since $\prod_{k\ge 0}(1-x)^{n_k} = 0$ iff $\sum_{k=0}^{\infty} x^{n_k} = \infty$, we construct sequence $(n_k)_{k\in\mathbb{N}}$ such that $\sum_{k=0}^{\infty} x^{n_k} = \infty$, and $\sum_{k\ge 0}(1-x)^{n_k} \le \varepsilon$. Let $(n_k)_{k\in\mathbb{N}}$ be a sequence such that $n_k = \ln_x(\frac{1}{k}) + J$, where J is a suitable constant (to be chosen later and depends on ε and x). Then we have

$$\sum_{k\ge 0} x^{n_k} = x^J \cdot \sum_{k\ge 0}\frac{1}{k} = \infty.$$

On the other hand, we have

$$1 - x = x^{\ln_x(1-x)} = x^{\frac{\ln(1-x)}{\ln x}}$$

Since $x > \frac{1}{2}$ we have $\beta = \frac{\ln(1-x)}{\ln x} > 1$, i.e., there exists $\beta > 1$ such that $1-x = x^\beta$. Hence

$$\sum_{k\geq 0}(1-x)^{n_k} = \sum_{k\geq 0} x^{\beta\cdot n_k} = x^{\beta\cdot J}\cdot \sum_{k\geq 0} x^{\beta\cdot \ln_x(\frac{1}{k})} = x^{\beta\cdot J}\cdot \sum_{k\geq 0}\frac{1}{k^\beta}.$$

Since the above series converges, for all $\varepsilon > 0$, there exists a J such that $\sum_{k\geq 0}(1-x)^{n_k} \leq \varepsilon$. This completes the proof. ∎

Almost and limit problem do not coincide. In contrast to probabilistic safety automata where the almost and limit question coincide, the answers are different for probabilistic acceptance-absorbing reachability automata. In the automaton $\mathcal{A}$ above if $x > \frac{1}{2}$, the answer to the limit question is yes. It follows from the proof above that if $\frac{1}{2} < x < 1$, then though the answer to the limit question is yes, the answer to the almost question is no. The almost and positive problem for reachability automata is decidable by reduction to POMDPs with reachability objective. We show that the limit question is undecidable. The proof uses the above lemma and the reduction technique of [GO09].

Reduction for undecidability. Given an acceptance-absorbing reachability automaton $\mathcal{B}$, we construct an automaton as shown in Fig 5: we assume that $\sharp$ does not belong to the alphabet of $\mathcal{B}$, and in the picture the dashed transitions from $\mathcal{B}$ on $\sharp$ are from accepting states, and the solid transitions are from non-accepting states. We call the automaton as $\mathcal{A}^*$. We claim the following: the answer to the limit problem for the automaton $\mathcal{A}^*$ is yes iff there exists a word w that is accepted in $\mathcal{B}$ with probability greater than $\frac{1}{2}$.

1. Suppose there is a word w^* such that $\mathcal{B}(w^*) = y > \frac{1}{2}$. Let $\eta = y - \frac{1}{2}$. There is a finite prefix w of w^* such that the probability to reach an accepting state in $\mathcal{B}$ given w is $x = y - \frac{\eta}{2} > \frac{1}{2}$. Given $\varepsilon > 0$ and x as defined $y - \frac{\eta}{2}$, we have $x > \frac{1}{2}$, and hence by Lemma 7 there exists a sequence $(n_0, n_1, \dots, n_k)$ such that $(a^{n_0}ba^{n_1}b\dots a^{n_k}b)(\$)^\omega$ is accepted in $\mathcal{A}$ with probability at least $1-\varepsilon$. It follows that the automaton $\mathcal{A}^*$ accepts $\big((aw\sharp)^{n_0}b(aw\sharp)^{n_1}\dots b(aw\sharp)^{n_k}b\big)(\$)^\omega$ with probability at least $1-\varepsilon$. It follows that the answer to the limit problem is yes.
2. If for all words w we have $\mathcal{B}(w) \leq 1/2$, then we show that the answer to the limit problem is no. Let $x = \sup_{w\in\Sigma^\omega}\mathcal{B}(w) \leq \frac{1}{2}$. Consider a word $\widehat{w} = \big((aw_0\sharp)^{n_0}b(aw_1\sharp)^{n_1}\dots b(aw_2\sharp)^{n_k}b\big)(\$)^\omega$: given this word the probability to reach q_3 in q_1 given $\widehat{w}$ is at most

$$1 - \prod_{k=0}^{i}(1-x^{n_i});$$

and the probability to reach q_7 from q_5 given $\widehat{w}$ is at least

$$\begin{aligned}(1-x)^{n_1} + (1-(1-x)^{n_1})\cdot(1-x)^{n_2} + \cdots + \prod_{k=0}^{i-1}(1-(1-x)^{n_k})\cdot(1-x)^{n_i} \\ = 1 - \prod_{k=0}^{i}(1-(1-x)^{n_i}).\end{aligned}$$

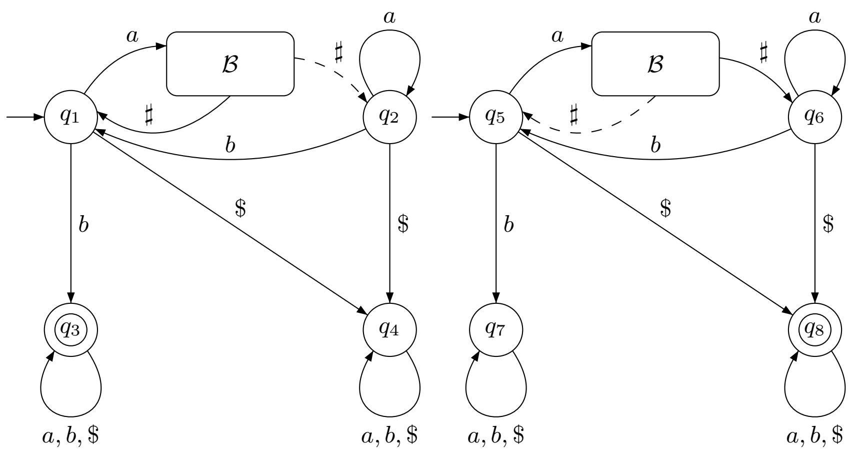


Fig. 5. Limit undecidability construction

The argument is similar to the proof of Lemma 7. It follows from the argument of Lemma 7 that the word is accepted with probability at most $\frac{1}{2}$ in $\mathcal{A}^*$. Thus it follows that for all words w we have $\mathcal{A}^*(w) \leq \frac{1}{2}$. Since the quantitative existence problem is undecidable for acceptance absorbing reachability automata (Theorem 2), it follows that the limit problem is also undecidable.

Table 1. Decidability and undecidability results for probabilistic automata

	Positive	Almost	Limit	Quantitative Equality
Safety	Dec.	Dec.	Dec.	Undec.
Reachability	Dec.	Dec.	Undec.	Undec.
Büchi	Undec.	Dec.	Undec.	Undec.
coBüchi	Dec.	Undec.	Undec.	Undec.
Limit-average	??	??	Undec.	Undec.

Undecidability. From the above proof it follows that the limit problem is undecidable for acceptance absorbing reachability automata, and hence also for Büchi, coBüchi, and limit-average condition. This gives us the Theorem 3. For other qualitative questions the results are as follows: (A) We argued that for safety condition that almost and limit problem coincide, and the decision problems can be answered from the solutions of partial-observable MDPs (POMDPs) [BGG09]. (B) For reachability condition the almost and positive problem can be answered through POMDPs ([BBG08]). (C) For Büchi condition it was shown in [BBG08] that the positive problem is undecidable and the almost problem is decidable. (D) For coBüchi condition it was shown in [BBG08] that the almost problem is undecidable and the positive problem is decidable. (E) For limit-average condition the decidability of the positive and almost problem remains open. The results are summarized in Table 1 (the results for quantitative existence and quantitative value problem is the same as for quantitative equality problem).

Theorem 3 (Qualitative problem). *The following assertions hold: (a) the positive problem is decidable for probabilistic safety, reachability, and coBüchi automata, and is undecidable for probabilistic Büchi automata; (b) the almost problem is decidable for probabilistic safety, reachability, and Büchi automata, and is undecidable for probabilistic coBüchi automata; and (c) the limit problem is decidable for probabilistic safety automata and is undecidable for probabilistic reachability, Büchi, coBüchi and limit-average automata.*

References

[BBG08] Baier, C., Bertrand, N., Größer, M.: On decision problems for probabilistic Büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)

[BG05] Baier, C., Größer, M.: Recognizing omega-regular languages with probabilistic automata. In: LICS, pp. 137–146 (2005)

[BGG09] Bertrand, N., Genest, B., Gimbert, H.: Qualitative determinacy and decidability of stochastic games with signals. In: LICS. IEEE, Los Alamitos (2009)

[BMT77] Bertoni, A., Mauri, G., Torelli, M.: Some recursive unsolvable problems relating to isolated cutpoints in probabilistic automata. In: Salomaa, A., Steinby, M. (eds.) ICALP 1977. LNCS, vol. 52, pp. 87–94. Springer, Heidelberg (1977)

[Buk80] Bukharaev, R.G.: Probabilistic automata. Jorunal of Mathematical Sciences 13, 359–386 (1980)

[GO09] Gimbert, H., Oualhadj, Y.: Automates probabilistes: problémes décidables et indécidables. In: Rapport de Recherche RR-1464-09 LaBRI (Conference version to appear ICALP 2010) (2009)

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, Reading (1979)

[Paz71] Paz, A.: Introduction to probabilistic automata, Computer science and applied mathematics. Academic Press, London (1971)

[Rab63] Rabin, M.O.: Probabilistic automata. Information and Control 6, 230–245 (1963)

Abstraction Learning

Joxan Jaffar, Jorge Navas, and Andrew Santosa

National University of Singapore

A state-of-the-art approach for proving safe programs is *CounterExample-Guided Abstraction Refinement* (*CEGAR*) which performs a "abstraction-verification-refinement" cycle by starting with a coarse abstract model that is then refined repeatedly whenever a counterexample is encountered. In this work, we present a dual approach which starts with a concrete model of the program but progressively abstracts away details but only when these are known to be irrelevant. We call this concept *Abstraction Learning* (AL). In order to deal with loops, our algorithm is encapsulated in an iterative deepening search where, because of a particular depth bound, abstraction is forced upon loops. This abstraction will correspond to the strongest loop invariant we can discover. As in CEGAR, this abstraction is of a speculative nature: if the proof is unsuccessful, the abstraction is removed and we initiate a new attempt using a new depth bound.

A key difference between AL and CEGAR is that AL detects more *infeasible paths* in the state-space traversal phase. We argue that this key feature poses unique benefits, and demonstrate the performance of our prototype implementation against the state-of-the-art BLAST system.

Synthesis: Words and Traces

Igor Walukiewicz

LaBRI (Université de Bordeaux - CNRS)

Abstract. The problem of synthesising a reactive system is discussed. The most standard instance of this problem ask to construct a finite input-output automaton satisfying a given regular specification. During fifty years since its introduction by Church, numerous extensions of the initial formulation have been considered. One particularly challenging case is that of distributed synthesis where a construction of a network of input/output automata is required.

1 General Context

Synthesis is about constructing a system from a given specification. For example: constructing of a circuit realizing a given boolean function. This case is easy until one adds constraints on placement of gates, etc. In some other settings the task is impossible from the very beginning: there is no algorithm constructing from an arithmetic formula a program realising the specified I/O function. Yet some, severe, restrictions of this problem are decidable, e.g., when considering only formulas of Presburger arithmetic. Here we will be interested in an extension of the first example to infinite, reactive behaviour.

The starting point of this research is the setting proposed by A. Church more than half a century ago [6]. He considered devices that transform an infinite sequence of input bits into an infinite sequence of output bits. The device is required to work "on-line": for each input bit read, it should produce an output bit. Church asked for an algorithm that constructs such a device from a given specification. The specification language he considered is monadic second-order logic (MSOL) over natural numbers with order, $\langle \mathbb{N}, \leq \rangle$. In this case a specification is a formula $\varphi(X,Y)$, where X and Y stand for subsets of $\mathbb{N}$, or equivalently, infinite sequences of bits. So the formula defines a desired relation between the input sequence X and the output sequence Y.

The problem of Church is fundamentally different from decidability of MSOL theory of $\langle \mathbb{N}, \leq \rangle$. Satisfiability of the formula $\forall X.\exists Y.\varphi(X,Y)$ is just a necessary condition for the existence of a required device: not only for every input sequence there should exist a good output sequence, but moreover this sequence should be produced "on-line". Indeed, while Büchi has shown decidability of MSOL theory of $\langle \mathbb{N}, \leq \rangle$ in 1960 [3], the solution to the synthesis problem came almost a decade later [2,19,20].

At the end of the eighties, Ramadge and Wonham introduced the theory of control of discrete event systems [21,9,4]. In this theory we start with a finite automaton, also called a plant, that defines all possible sequential behaviours of the system. The goal is to find for a given plant another finite automaton, called controller, such that the synchronous product of the plant and the controller satisfies desired properties: MSOL properties of the language of the product.

This kind of synthesis problem would be interesting neither from theoretical nor from practical point of view if there were no additional restrictions on controllers. In the most standard form, some events of the plant are declared to be uncontrollable, and some others to be unobservable. The controller is not allowed to block uncontrollable events, and is not supposed to see unobservable events. More precisely these restrictions are determined by two subsets A_{unc} and A_{uobs} of the alphabet of events with the associated requirement that:

(C) For every state q of the controller, and for every uncontrollable event $a \in A_{unc}$, there is a transition from q labelled by a.

(O) For every state q of the controller, and for every unobservable event $a \in A_{uobs}$, if there is a transition from q labelled by a then this transition is a loop over q.

Ramadge and Wonham setting is more general than Church formulation. Interestingly though, the tools developed for the Church problem are sufficient to solve this case too. One important lesson here is that synthesis is ultimately about branching properties: properties of trees rather than properties of sequences. Once MSOL theory of trees is well understood, the rest is relatively easy.

At present, Ramadge and Wonham setting as described above is well established. Starting from there, many extensions have been studied: richer automata models, richer specification languages, introduction of time constraints, quantitative constraints, . . . One of the most challenging and promising directions is the extension of the framework to the distributed case. Here, one puts restrictions on the form of synthesised device: it should be distributed into several modules, each with limited capacities of observing the plant.

2 Distributed Synthesis

In a distributed system one can have multiple processes. The system specifies possible interactions between the processes and the environment, as well as, the interactions between the processes themselves. The synthesis problem is to find a program for each of the processes such that the overall behaviour of the system satisfies a given specification.

The problem can be modeled by a game with incomplete information. In such a game we have a team of controllers playing against a single player representing environment. Finding a program for each controller is then equivalent to computing a distributed winning strategy for each of the controllers. In general, multiplayer games with incomplete information are undecidable [17,16]. For similar reasons the distributed control problem is also undecidable in most cases [18,11,10,12,14,1]. Thanks to these works we understand some sources for undecidability, but we do not have the whole picture yet. It is fair to say that the examples leading to undecidability can be qualified as unrealistic. It would be very interesting to refine the setting to rule out these examples, but no satisfactory proposal is known at present.

One important attempt to get a decidable framework of distributed synthesis is to change the way information is distributed in the system. In the case above, every controller sees only its inputs and its outputs. In order to deduce some information about the global state of the system a controller can use only his knowledge about the architecture and the initial state of the system. In particular, controllers are not permitted to pass additional information during communication. It is clear though that when we allow some transfer of information during communication, we give more power to controllers.

Pushing the idea of sharing information to the limit, we obtain a model where two processes involved in a communication share all the information they have about the global state of the system [8]. This point of view is not as unrealistic as it may seem at the first glance. It is rooted in the theory of traces that studies finite communicating automata with this kind of information transfer. A fundamental result of Zielonka [22,7] implies that in fact there is a bound on the size of additional information that needs to be transferred during communication. In our terms, the theory of traces considers the case of synthesis for closed systems, i.e., systems without environment. For the distributed synthesis with environment, some decidability results for some special cases are known [8,13,15,5]. Moreover, similarly to Zielonka's Theorem, these results give a bound on additional information that needs to be transferred. The decidability of the general case is open. Interestingly, the general case can be formulated as an extension of the Ramadge and Wonham setting from words, that is linear orders, to special partial orders called Mazurkiewicz traces.

References

1. Arnold, A., Walukiewicz, I.: Nondeterministic controllers of nondeterministic processes. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata. Texts in Logic and Games, vol. 2, pp. 29–52. Amsterdam University Press, Amsterdam (2007)
2. Büchi, J., Landweber, L.: Solving sequential conditions by finite state strategies. Trans. Amer. Math. Soc. 138, 367–378 (1969)
3. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. 1960 Int. Congr. for Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)
4. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers, Dordrecht (1999)
5. Chatain, T., Gastin, P., Sznajder, N.: Natural specifications yield decidability for distributed synthesis of asynchronous systems. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 141–152. Springer, Heidelberg (2009)
6. Church, A.: Applications of recursive arithmetic to the problem of cricuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic, vol. I, pp. 3–50. Cornell Univ., Ithaca (1957)
7. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific, Singapore (1995)

8. Gastin, P., Lerman, B., Zeitoun, M.: Distributed games with causal memory are decidable for series-parallel systems. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 275–286. Springer, Heidelberg (2004)
9. Kumar, R., Garg, V.K.: Modeling and control of logical discrete event systems. Kluwer Academic Pub., Dordrecht (1995)
10. Kupferman, O., Vardi, M.: Synthesizing distributed systems. In: Proc. 16th IEEE Symp. on Logic in Computer Science (2001)
11. Madhusudan, P., Thiagarajan, P.: Distributed control and synthesis for local specifications. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001)
12. Madhusudan, P., Thiagarajan, P.: A decidable class of asynchronous distributed controllers. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 145–160. Springer, Heidelberg (2002)
13. Madhusudan, P., Thiagarajan, P.S., Yang, S.: The MSO theory of connectedly communicating processes. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 201–212. Springer, Heidelberg (2005)
14. Mohalik, S., Walukiewicz, I.: Distributed games. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 338–351. Springer, Heidelberg (2003)
15. Muscholl, A., Walukiewicz, I., Zeitoun, M.: A look at the control of asynchronous automata. In: Perspectives in Concurrency Theory – Festschrift for P.S. Thiagarajan, pp. 356–371. Universities Press (2008)
16. Peterson, G., Reif, J., Azhar, S.: Lower bounds for multiplayer noncooperative games of incomplete information. Comput. Math. Appl. 41, 957–992 (2001)
17. Peterson, G.L., Reif, J.H.: Multi-person alternation. In: Proc. IEEE FOCS, pp. 348–363 (1979)
18. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31th IEEE Symposium Foundations of Computer Science (FOCS 1990), pp. 746–757 (1990)
19. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the AMS 141, 1–23 (1969)
20. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. American Mathematical Society, Providence (1972)
21. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77(2), 81–98 (1989)
22. Zielonka, W.: Notes on finite asynchronous automata. RAIRO–Theoretical Informatics and Applications 21, 99–135 (1987)

Promptness in ω-Regular Automata

Shaull Almagor[1], Yoram Hirshfeld[2], and Orna Kupferman[1]

[1] Hebrew University, School of Engineering and Computer Science, Jerusalem, Israel
[2] Tel Aviv University, School of Mathematical Science, Tel Aviv, Israel

Abstract. Liveness properties of on-going reactive systems assert that something good will happen eventually. In satisfying liveness properties, there is no bound on the "wait time", namely the time that may elapse until an eventuality is fulfilled. The traditional "unbounded" semantics of liveness properties nicely corresponds to the classical semantics of automata on infinite objects. Indeed, acceptance is defined with respect to the set of states the run visits infinitely often, with no bound on the number of transitions taken between successive visits.

In many applications, it is important to bound the wait time in liveness properties. Bounding the wait time by a constant is not always possible, as the bound may not be known in advance. It may also be very large, resulting in large specifications. Researchers have studied *prompt eventualities*, where the wait time is bounded, but the bound is not known in advance. We study the automata-theoretic counterpart of prompt eventually. In a *prompt-Büchi* automaton, a run r is accepting if there exists a bound k such that r visits an accepting state every at most k transitions. We study the expressive power of nondeterministic and deterministic prompt-Büchi automata, their properties, and decision problems for them. In particular, we show that regular nondeterministic prompt Büchi automata are exactly as expressive as nondeterministic co-Büchi automata.

1 Introduction

A specification of a reactive system describes the required on-going behaviors of the system. Specifications can be viewed as ω-regular languages and are traditionally classified into *safety* and *liveness* properties [1]. Intuitively, safety properties assert that nothing bad will ever happen during the execution of the system, and liveness properties assert that something good will happen eventually. In satisfying liveness properties, there is no bound on the "wait time", namely the time that may elapse until an eventuality is fulfilled.

In many applications, it is important to bound the wait time in liveness properties. Bounding the wait time by a constant changes the specification from a liveness property to a safety property. For example, if we bound the wait time in the specification "every request is eventually granted" and replace it by the specification "every request is granted within k transitions" for some fixed k, then we end up with a safety property – we never want to come across a request that is not granted within the next k transitions. While the safety property is much stronger, it involves two drawbacks. First, the bound k needs to be known in advance, which is not the case in many applications. For example, it may depend on the system, which may not yet be known, or it may change, if the system changes. Second, the bound may be large, causing the state-based description of the specification (e.g., an automaton for it) to be large too. Thus, the common practice is to use liveness properties as an abstraction of such safety properties.

Several earlier work suggested and studied an alternative semantics to eventually properties. The semantics, termed *finitary fairness* in [3], *bounded fairness* in [9], and *prompt eventually* in [15], does not suffer from the above two drawbacks and is still more restrictive than the standard semantics. In the alternative semantics, the wait time is bounded, but the bound is not known in advance. Consider, for example, the computation $\pi = \mathit{req}.\mathit{grant}.\mathit{req}.\neg\mathit{grant}.\mathit{grant}.\mathit{req}.(\neg\mathit{grant})^2.\ \mathit{grant}.\ \mathit{req}.(\neg\mathit{grant})^3.\mathit{grant}\ldots$, in which the wait time to a grant increases in an unbounded (yet still finite) manner. While π satisfies the liveness property "every request is eventually granted", it does not satisfy its prompt variant. Indeed, there is no bound k such that π satisfies the property "every request is granted within k transitions".

The traditional "unbounded" semantics of liveness properties nicely corresponds to the classical semantics of automata on infinite objects. Indeed, acceptance is defined with respect to the set of states the run visits infinitely often, with no bound on the number of transitions taken between successive visits. The correspondence in the semantics is essential in the translation of temporal-logic formulas to automata, and in the automata-theoretic approach to *specification, verification*, and *synthesis* of nonterminating systems [17,19]. The automata-theoretic approach views questions about systems and their specifications as questions about languages, and reduces them to automata-theoretic problems like containment and emptiness.

In this paper we introduce and study a prompt semantics for automata on infinite words, by means of *prompt-Büchi* automata. In a Büchi automaton, some of the states are designated as accepting states, and a run is accepting iff it visits states from the accepting set infinitely often [7]. Dually, in a *co-Büchi* automaton, a run is accepting iff it visits states outside the accepting set only finitely often. In a prompt-Büchi automaton, a run is accepting iff there exists a bound k such that the number of transitions between successive visits to accepting states is at most k. More formally, if $\mathcal{A}$ is a prompt-Büchi automaton with a set α of accepting states, then an infinite run $r = r_1, r_2, ...$ of $\mathcal{A}$ is accepting iff there exists a bound $k \in \mathbb{N}$ such that for all $i \in \mathbb{N}$ it holds that $\{r_i, ..., r_{i+k-1}\} \cap \alpha \neq \emptyset$. We consider both nondeterministic (NPBWs, for short) and deterministic (DPBW, for short) prompt-Büchi automata.

It is not hard to see that if $\mathcal{A}$ is a Büchi automaton, then the language of the automaton obtained by viewing $\mathcal{A}$ as a prompt Büchi automaton is contained in the union of all the safety languages contained in the language of $\mathcal{A}$.

The relation between safety languages and promptness has been studied already in [20]. There, the authors define *bound automata*, which are parameterized by a bound k, and a run is accepting if for all $i \in \mathbb{N}$ it holds that

$\{r_{ik}, ..., r_{(i+1)k-1}\} \cap \alpha \neq \emptyset$. Thus, bound automata are similar to prompt automata, except that one restricts attention to windows that start in positions 0 mod k. It is shown in [20] that eventhough the language of $\mathcal{A}$ with parameter k is a safety language, the automaton $\mathcal{A}$ can be used in order to generate an infinite sequence of safety properties. As stated in [3], the union of all safety languages that are recognized by $\mathcal{A}$ with parameter k, for some k, need not be ω-regular. We prove that the language of the prompt-Büchi automaton need not be ω-regular as well. In fact, Büchi and prompt-Büchi automata are incomparable in terms of expressive power. Indeed, no NPBW can recognize the ω-regular language $(a^*b)^\omega$ (infinitely many occurrences of b), whereas no nondeterministic Büchi automaton (NBW, for short) can recognize the language L_b, where $w \in \{a,b\}^\omega$ is in L_b if there exists a bound $k \in \mathbb{N}$ such that all the subwords of w of length k have at least one occurrence of b. Note that L_b is recognized by a two-state DPBW that goes to an accepting state whenever b is read. Note that when an NPBW runs on a word, it guesses and verifies the bound k with which the run is going to be accepting. The bound k may be bigger than the number of states, and still the NPBW has to somehow count and check whether an accepting state appears at least once every k transitions. We would like to understand how this ability of NPBWs influences their expressive power and succinctness.

We start by investigating prompt Büchi automata in their full generality and show that while both NPBWs and DPBWs are closed under intersection and not closed under complementation, only NPBWs are closed under union. Also, NPBWs are strictly more expressive than DPBWs. We then focus on *regular*-NPBWs, namely languages that can be recognized by both an NPBW and an NBW. We first show that NPBWs are NBW-type: if the language of a given NPBW $\mathcal{A}$ is regular, then $\mathcal{A}$ is also an NBW for the language. From a theoretical point of view, our result implies that if a union of safety languages is ω-regular, then the promptness requirement can be removed from every NPBW that recognizes it. From a practical point of view, our result implies that there is no state blow-up in translating NPBWs to NBWs, when possible. On the other hand, we show that there are NBWs that recognize languages that can be recognized by an NPBW, but an NPBW for them requires an automaton with a different structure. Thus, if we add the promptness requirement to an NBW, we may restrict its language, even if this language is a union of safety properties.

Our main result shows that regular-NPBWs are as expressive as nondeterministic co-Büchi automata (NCWs). To show this, we first prove that counting to unbounded bounds is not needed, and that the distance between successive visits in accepting states can be bounded by 3^{n^2}, where n is the number of states of the automaton. Technically, the bound follows from an analysis of equivalence classes on Σ^* and an understanding that increasingly long subwords that skip visits in accepting states must contain equivalent prefixes. It is easy to show that the existence of the global 3^{n^2} bound implies that the language is NCW-recognizable. The global bound suggests a translation to NCW that is not optimal. We use results on the translation of NBWs to NCWs [6] in order to show an alternative translation, with a blow up of only $n2^n$. We also describe a matching lower bound. It follows that regular-NPBW are exponentially more succinct than NCWs. The equivalence with NCWs also gives immediate results about the closure properties of regular-NPBWs.

Finally, we study decision problems for prompt automata. We show that the problem of deciding whether a prompt automaton is regular is PSPACE-complete for NPBW and is NLOGSPACE-complete for DPBW. The same bounds hold for the universality and the containment problems. The main challenge in these results is the need to complement the NPBW. We show how we can circumvent the complementation, work, instead, with an automaton that approximates the complementing one, in the sense it accepts only words with a fixed bound, and still solve the regularity, universality, and containment problems.

Related work. The work in [9,15] studies the prompt semantics from a temporal-logic prospective. In [9], the authors study an eventuality operator parameterized by a bound (see also [2]), and the possibility of quantifying the bound existentially. In [15], the authors study the logic PROMPT-LTL, which extends LTL by a prompt-eventuality operator $\mathbf{F_p}$. A system S satisfies a PROMPT-LTL formula ψ if there is a bound $k \in \mathbb{N}$ such that S satisfies the LTL formula obtained from ψ by replacing all $\mathbf{F_p}$ by $\mathbf{F}^{\leq k}$. Thus, there should exist a bound, which may depend on the system, such that prompt eventualities are satisfied within this bounded wait time. It is shown in [15] that the realizability and the model-checking problems for PROMPT-LTL have the same complexity as the corresponding problems for LTL, though the algorithms for achieving the bounds are technically more complicated. Note that the definition of prompt Büchi automata corresponds to the semantics of the $\mathbf{F_p}$ operator of PROMPT-LTL. Thus, given an LTL formula ψ, we can apply to ψ the standard translation of LTL to NBW [19], and end up with an NPBW for the PROMPT-LTL formula obtained from ψ by replacing its eventualities by prompt ones.

The work in [8,10] studies the prompt semantics in ω-regular games. The games studied are finitary parity and finitary Streett games. It is shown in [8] that these games are determined and that the player whose objective is to generate a computation that satisfies the finitary parity or Streett condition has a memoryless strategy. In contrast, the second player may need infinite memory. In [10] it is shown that the problem of determining the winner in a finitary parity game can be solved in polynomial time.[1]

The closest to our work here is [4,5], which introduced the notion of promptness to Monadic Second Order Logic (MSOL) and ω-regular expressions. In [5], the authors introduced ωBS-regular expressions, which extend ω-regular expressions with two new operators B and S – variants of the Kleene star operator. The semantics of the new operators is that $(r^B)^\omega$, for a regular expression r, states that words in $L(r)$ occur globally with a bounded length, and $(r^S)^\omega$ states that

[1] The computations of games with a single player correspond to runs of a nondeterministic automaton. Games, however, do not refer to languages, and indeed the problems we study are very different from these studied in [8,10].

words in $L(r)$ occur with a strictly increasing length. Thus, ωBS-regular expressions are clearly more expressive than NPBWs. In [4], the author studies the properties of a prompt extension to MSOL. The contribution in [4,5] is orthogonal to our contribution here. From a theoretical point of view, [5,4] offer an excellent and exhaustive study of the logical aspects of promptness, in terms of closure properties and the decidability of the satisfiability problem for the formalisms and their fragments, where the goal is to give a robust formalism and then study its computational properties. Indeed, the algorithmic aspects of the studied formalisms are not appealing: the complexity of the decidability problem is much higher than that of NPBWs, and the model of automata that is needed in order to solve these problems is much more complex than NPBW (the automata are equipped with counters, and the translation of expressions to them is complicated). Our contribution, on the other hand, focuses on the prompt variant of the Büchi acceptance condition. As such, it does not attempt to present a robust promptness formalism but rather to study NPBWs in depth. As our results show, NPBWs are indeed much simpler than the robust formalisms, making them appealing also from a practical point of view.

Due to the lack of space, most proofs are omitted and can be found in the full version in the authors' URLs.

2 Preliminaries

Given an alphabet Σ, a word over Σ is a (possibly infinite) sequence $w = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots$ of letters in Σ. For $x \in \Sigma^*$ and $y \in \Sigma^* \cup \Sigma^\omega$, we say that x is a *prefix* of y, denoted $x \preceq y$, if there is $z \in \Sigma^* \cup \Sigma^\omega$ such that $y = x \cdot z$. If $z \neq \epsilon$ then x is a *strict prefix* of y, denoted $x \prec y$. For an infinite word w and indices $0 \leq k \leq l$, let $w[k..l] = \sigma_k \cdots \sigma_l$ be the infix of w between positions k and l.

An *automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $\alpha \subseteq Q$ is an acceptance condition. We define several acceptance conditions below. Intuitively, $\delta(q, \sigma)$ is the set of states that $\mathcal{A}$ may move into when it is in the state q and it reads the letter σ. The automaton $\mathcal{A}$ may have several initial states and the transition function may specify many possible transitions for each state and letter, and hence we say that $\mathcal{A}$ is *nondeterministic*. In the case where $|Q_0| = 1$ and for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| = 1$, we say that $\mathcal{A}$ is *deterministic*. The transition function extends to sets of states and to finite words in the expected way, thus for $x \in \Sigma^*$, the set $\delta(S, x)$ is the set of states that $\mathcal{A}$ may move into when it is in a state in S and it reads the finite word x. Formally, $\delta(S, \epsilon) = S$ and $\delta(S, x \cdot \sigma) = \bigcup_{q \in \delta(S,x)} \delta(q, \sigma)$. We abbreviate $\delta(Q_0, x)$ by $\delta(x)$, thus $\delta(x)$ is the set of states that $\mathcal{A}$ may visit after reading x. A *run* $r = r_1, r_2, r_3, \ldots$ of $\mathcal{A}$ on an infinite word $w = \sigma_1 \cdot \sigma_2 \cdots \in \Sigma^\omega$ is an infinite sequence of states such that $r_1 \in Q_0$, and for every $i \geq 1$, we have that $r_{i+1} \in \delta(r_i, \sigma_i)$. Note that while a deterministic automaton has a single run on an input word, a nondeterministic automaton may have several runs on an input word. We sometimes refer to r as a word in Q^ω or as a function from the set of prefixes of w to the states of $\mathcal{A}$ (that is, for a prefix $x \preceq w$, we have $r(x) = r_{|x|+1}$). Acceptance is defined with respect to the set $\mathit{inf}(r)$ of states that the run r visits infinitely often. Formally, $\mathit{inf}(r) = \{q \in Q \mid \text{for infinitely many } i \geq 1, \text{ we have } r_i = q\}$.

As Q is finite, it is guaranteed that $\mathit{inf}(r) \neq \emptyset$. The run r is *accepting* if it satisfies the acceptance condition α. A run r satisfies a *Büchi* acceptance condition α if $\mathit{inf}(r) \cap \alpha \neq \emptyset$. That is, r visits α infinitely often. Dually, r satisfies a *co-Büchi* acceptance condition α if $\inf(r) \subseteq \alpha$. That is, r visits α almost always. Note that the latter is equivalent to $\mathit{inf}(r) \cap (Q \setminus \alpha) = \emptyset$

We now define a new acceptance condition, *prompt-Büchi*, as follows: A run $r = r_1, r_2, \ldots$ satisfies a *prompt-Büchi* acceptance condition α if there is $k \geq 1$ such that for all $i \geq 1$ there is $j \in \{i, i+1, \ldots, i+k-1\}$ such that $r_j \in \alpha$. That is, in each block of k successive states, the run r visits α at least once. We say that r is *accepting with a bound k*. It is easy to see that requiring the bound to apply eventually (instead of always) provides an equivalent definition. Thus, equivalently, a run r satisfies prompt-Büchi acceptance condition α if there is $k \geq 1$ and $n_0 \in \mathbb{N}$ such that for all $i \geq n_0$ there is $j \in \{i, i+1, \ldots, i+k-1\}$ such that $r_j \in \alpha$. We say that r is *accepting with eventual bound k*. Given a prompt-Büchi accepting run with bound k and eventual bound k', note that it always holds that $k' \leq k$, and that a strict inequality is possible.

An automaton *accepts* a word if it has an accepting run on it. The *language* of an automaton $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts. We also say that $\mathcal{A}$ *recognizes* the language $L(\mathcal{A})$. For two automata $\mathcal{A}$ and $\mathcal{A}'$ we say that $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent* if $L(\mathcal{A}) = L(\mathcal{A}')$.

We denote the classes of automata by acronyms in $\{D, N\} \times \{B, PB, C\} \times \{W\}$. The first letter stands for the branching mode of the automaton (deterministic or nondeterministic); the second letter stands for the acceptance-condition type (Büchi, Prompt-Büchi, or co-Büchi); the third letter indicates that the automaton runs on words. For example, DPBW stands for deterministic prompt-Büchi automaton. We say that a language L is in a class γ if L is γ-recognizable; that is, L can be recognized by an automaton in the class γ.

Given two classes γ and η we say that γ is *more expressive* than η if every η-recognizable language is also γ-recognizable. If γ is not more expressive than η and η is not more expressive than γ, we say that γ and η are *incomparable*. Different classes of automata have different expressive power. In particular, NBWs recognize all ω-regular languages, while NCWs are strictly less expressive [18].

3 Properties of Prompt Languages

In this section we study properties of the prompt-Büchi acceptance condition. As discussed in Section 1, the class of NPBW-recognizable languages is incomparable with that of ω-regular languages. Similar results were shown in [3], in the context of finitary fairness in and safety languages.

Theorem 1. NPBWs *and* NBWs *are incomparable.*

The fact that NPBWs and NBWs are incomparable implies we cannot borrow the known closure properties of ω-regular languages to the classes NPBW and

DPBW. In Theorem 2 below, we study closure properties. As detailed in the proof, for the cases of NPBW union as well as NPBW and DPBW intersection, the known constructions for NBW and DBW are valid also for the prompt setting. To show non-closure, we define, the following language. Let $\Sigma = \{a, b\}$. For a letter $\sigma \in \Sigma$, let $L_\sigma = \{w \in \{a, b\}^\omega :$ there exists $k \in \mathbb{N}$ such that for all$i \in \mathbb{N}$ we have $\sigma \in \{w_i, w_{i+1}, ..., w_{i+k}\}\}$. Recall that the language L_b is in NPBW but is not ω-regular. It is easy to see that L_σ can be recognized by a DPBW with two states (that goes to an accepting state whenever σ is read). We show, however, that the union of L_a and L_b cannot be recognized by a DPBW and that no NPBW exists for its complementation. These results also imply that NPBWs are strictly more expressive than DPBWs.

Theorem 2

1. NPBWs *are, but* DPBWs *are not closed under union.*
2. NPBWs *and* DPBWs *are closed under intersection.*
3. NPBWs *and* DPBWs *are not closed under complementation.*
4. NPBWs *are strictly more expressive than* DPBWs.

4 Regular NPBW

We have seen in Section 3 that NPBWs and NBWs are incomparable. In this section we study regular prompt languages, namely languages that are both NPBW and NBW recognizable. We use *reg*-NPBW and *reg*-DPBW to denote the classes NBW $\cap$ NPBW and NBW $\cap$ DPBW, respectively. For an automaton $\mathcal{A}$, we denote by $\mathcal{A}_B, \mathcal{A}_P$, and $\mathcal{A}_C$ the automaton $\mathcal{A}$ when referred to as an NBW, NPBW, and NCW, respectively.

4.1 Typeness

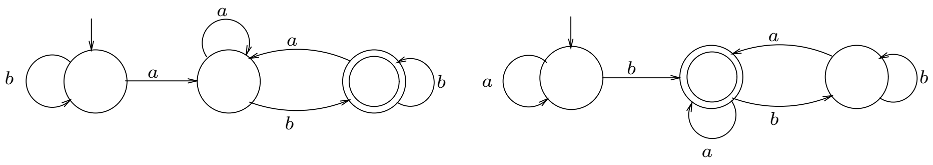
Given two types of automata γ_1 and γ_2, we say that a γ_1 is γ_2-type if for every automaton $\mathcal{A}$ in the class γ_1, if $\mathcal{A}$ has an equivalent automaton in the class γ_2, then there exists an equivalent automaton in the class γ_2 on the same structure as $\mathcal{A}$ (that is, only changing the acceptance condition). Typeness was studied in [12,13], and is useful, as it implies that a translation between the two classes does not involve a blowup and is very simple. In this section we study typeness for NPBWs and NBWs.

Theorem 3. NPBW *are* NBW*-type: if $\mathcal{A}$ is an automaton such that $L(\mathcal{A}_P)$ is ω-regular, then $L(\mathcal{A}_P) = L(\mathcal{A}_B)$.*

Proof. Since both $L(\mathcal{A}_P)$ and $L(\mathcal{A}_B)$ are ω-regular, so is their difference, and thus there is an NBW $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A}_B) \setminus L(\mathcal{A}_P)$. We prove that $L(\mathcal{A}') = \emptyset$. Assume by way of contradiction that $L(\mathcal{A}') \neq \emptyset$. Then $\mathcal{A}'$ accepts also a periodic word, thus there are $u, v \in \Sigma^*$ such that $u \cdot v^\omega \in L(\mathcal{A}_B) \setminus L(\mathcal{A}_P)$. Let r be an accepting run of $\mathcal{A}_B$ on $u \cdot v^\omega$. Thus, $inf(r) \cap \alpha \neq \emptyset$. Hence, there are indices $i < j$ such that r visits α between reading $u \cdot v^i$ and $u \cdot v^j$, and such that $r(uv^i) = r(uv^j)$. Then, however, we can pump r to an accepting run of $\mathcal{A}_P$ on $u \cdot (v^i v^{i+1} \cdots v^{j-1})^\omega = u \cdot v^\omega$, contradicting the fact that $u \cdot v^\omega \notin L(\mathcal{A}_P)$. $\square$

Theorem 4. NBWs *are not* NPBW-*type: there exists an automaton $\mathcal{A}$ such that $L(\mathcal{A}_B)$ is* NPBW-*recognizable, but there is no* NPBW *$\mathcal{A}'$ with the same structure as $\mathcal{A}$ such that $L(\mathcal{A}') = L(\mathcal{A}_B)$.*

Proof. Consider the automaton $\mathcal{A}$ below. Note that $L(\mathcal{A}_B) = \{a, b\}^\omega \setminus \{a^\omega, b^\omega\}$. It is easy to construct a four-state NPBW for $L(\mathcal{A}_B)$. On the other hand, it is not hard to see that all possibilities to define an acceptance condition on top of the structure of $\mathcal{A}$ results in an NPBW whose language is different. □



4.2 Reg-NPBW=NCW

In this section we prove that reg-NPBWs are as expressive as NCWs. We show that while in the deterministic case, an equivalent DCW can always be defined on the same structure, the nondeterministic case is much more complicated and reg-NPBW are exponentially more succinct than NCWs.

We start with the deterministic case. As detailed in the proof, a DPBW $\mathcal{A}_P$ that recognizes a regular language can be translated to an equivalent DCW by making the set α of accepting states maximal. That is, by adding to α all states that do not increase the language of the DPBW. Intuitively, it follows from the fact that if a run of the obtained DCW does not get trapped in α, and α is maximal, then there must exists a state $q \notin \alpha$ that is visited infinitely often along the run. The word that witnesses the fact that q cannot be added to α can then be pumped to a word showing that $L(\mathcal{A}_P) \neq L(\mathcal{A}_B)$, contradicting the regularity of $\mathcal{A}_P$.

Theorem 5. *Let $\mathcal{A}$ be a reg*-DPBW*, then there exists a* DCW *$\mathcal{B}$ on the same structure as $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.*

Proof. Consider the DPBW $\mathcal{A}_P$. For simplicity, we assume that α is maximal (that is, no states can be added to α without increasing the language. Clearly, we can turn a non-maximal accepting condition to an equivalent maximal one). We claim that $L(\mathcal{A}_C) = L(\mathcal{A}_P)$.

It is easy to see that $L(\mathcal{A}_C) \subseteq L(\mathcal{A}_P)$. Indeed, if w is in $L(\mathcal{A}_C)$, then the run of $\mathcal{A}$ on w eventually gets stuck in α, and so $w \in L(\mathcal{A}_P)$. For the other direction, consider a word $w \in L(\mathcal{A}_P)$ and assume by way of contradiction that $w \notin L(\mathcal{A}_C)$. Let r be the accepting run of $\mathcal{A}$ on w, and let $q \in \alpha$ and $q' \notin \alpha$ be states that are visited infinitely often in r. Since r is prompt-accepting and not co-Büchi accepting, such q and q', which are reachable from each other, exist. Let $v \in \Sigma^*$ be a word leading from the initial state of $\mathcal{A}$ to q' and let $u \in \Sigma^*$ be a word leading from q' back to itself via q. Note that since $q \in \alpha$, then $v \cdot u^\omega$ is accepted by $\mathcal{A}$ (with a $\max\{|v|, |u|\}$ bound).

Recall that α is maximal. Thus, adding q' to α would result in an automaton whose language strictly contains $L(\mathcal{A}_P)$. Hence, there exists $z \in \Sigma^*$, such that z leads from q' to itself and $v \cdot z^\omega$ is not in $L(\mathcal{A}_P)$. Thus, the run on z from q' is a cycle back to q' that visits no states in α.

For all $k \geq 1$, the word $w_k = v \cdot (z^k \cdot u)^\omega$ is accepted by $\mathcal{A}$. Indeed, the run r of $\mathcal{A}$ on w_k first gets to q' after reading v. Then, whenever the run reads a $z^k u$ subword, it traverses a loop from q' to itself k times while reading z^k, and traverses a loop that visits α while reading u. Since the cycle traversed while reading u occurs every $|z^k u|$ letters (that is, every fixed number of letters), this run is accepting.

Denote the number of states in $\mathcal{A}$ by n and let $k > n$. In every z-block there are l_1 and l_2 such that r visits the same state after it reads z^{l_1} and z^{l_2} in this block. Formally, for all $i \geq 0$ there are $0 \leq l_1 < l_2 \leq k$ such that $r(v \cdot (z^k \cdot u)^i \cdot z^{l_1}) = r(v \cdot (z^k \cdot u)^i \cdot z^{l_2})$. This means we can pump the z-blocks of w_k to a word $w = v \cdot z^{i_1} \cdot u \cdot z^{i_2} \cdot u \cdot z^{i_3} \cdot u \cdots$, with $i_1 < i_2 < i_3 < \cdots$, such that $w \in L(\mathcal{A}_B)$. By Theorem 3, we know that $L(\mathcal{A}_B) = L(\mathcal{A}_P)$. Hence, also $w \in L(\mathcal{A}_P)$. Let k be the bound with which w is accepted by $\mathcal{A}_P$. Since there is $j \geq 1$ such that $i_j > \max\{k, n\}$, we can conclude, as in the proof of Theorem 1, that the word $w' = v \cdot z^{i_1} \cdot u \cdot z^{i_2} \cdots u \cdot z^{i_j} \cdot u \cdot z^\omega$ is accepted by $\mathcal{A}_P$. However, the run r' of $\mathcal{A}$ on w' has $r'(v \cdot z^{i_1} \cdot u \cdot z^{i_2} \cdots u \cdot z^{i_j} \cdot u) = q'$. Then, reading the z^ω suffix, the run r' does not visit α, implying that $\mathcal{A}_P$ does not accept w', and we have reached a contradiction. $\square$

We now proceed to study the (much harder) nondeterministic case. Consider an automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$. For $u \in \Sigma^*$ and $q, q' \in Q$, we say that $q \to_u^+ q'$ if $q' \in \delta(q, u)$ and there is a run of $\mathcal{A}$ from q to q' that reads u and visits α. Similarly, we say that $q \to_u^- q'$ if $q' \in \delta(q, u)$ but all runs of $\mathcal{A}$ from q to q' that read u do not visit α.

We define a relation $\equiv_\mathcal{A} \subseteq \Sigma^* \times \Sigma^*$, where $u \equiv_\mathcal{A} v$ if for all $q, q' \in Q$, we have that $q \to_u^+ q'$ iff $q \to_v^+ q'$ and $q \to_u^- q'$ iff $q \to_v^- q'$. Intuitively, $u \equiv_\mathcal{A} v$ if for all states q of $\mathcal{A}$, reading u from q has exactly the same effect (by means of reachable states and visits to α) as reading v from q.

It is easy to see that $\equiv_\mathcal{A}$ is reflexive, symmetric, and transitive. We can characterize each equivalence class $[u]$ of $\equiv_\mathcal{A}$ by a function $f_u : Q \to \{+, -, \perp\}^Q$, where for all $q, q' \in Q$, we have $f_u(q)(q') = +$ iff $q \to_u^+ q'$, $f_u(q)(q') = -$ iff $q \to_u^- q'$ and $f_u(q)(q') = \perp$ iff $q' \notin \delta(q, u)$. Since there are only $3^{|Q|^2}$ such functions, we have the following.

Lemma 1. *The relation $\equiv_\mathcal{A}$ is an equivalence relation with at most $3^{(|Q|^2)}$ equivalence classes.*

Lemma 2. *Consider an NPBW $\mathcal{A}$. Let m denote the number of equivalence classes of $\equiv_\mathcal{A}$. Consider a word $w \in \Sigma^m$. There exist $s \in \Sigma^*$ and $z \in \Sigma^+$ such that $s \prec z \preceq w$, and $s \equiv_\mathcal{A} z$.*

Proof. Since $\equiv_{\mathcal{A}}$ has m equivalence classes, every set of $m+1$ words must contain at least two $\equiv_{\mathcal{A}}$-equivalent words. In particular, this holds for the set $\{\epsilon, w[0..1], w[0..2], \ldots, w[0..m]\}$. □

Theorem 6. *Consider an automaton $\mathcal{A}$. Let m denote the number of equivalence classes of $\equiv_{\mathcal{A}}$. If $L(\mathcal{A}_P)$ is ω-regular, then every word in $L(\mathcal{A}_P)$ has an accepting run with an eventual bound of $2m$.*

Proof. Since $\mathcal{A}_P$ is ω-regular, then, by Theorem 3, we have that $L(\mathcal{A}_P) = L(\mathcal{A}_B)$. Consider a word $w \in L(\mathcal{A}_P)$. We prove that there is a run that accepts w with bound at most $2m$. Let $w = b_1 \cdot b_2 \cdot b_3 \cdots$ be a partition of w to blocks of length m; thus, for all $i \geq 1$, we have that $b_i \in \Sigma^m$. We define w' by replacing each block b_i by a new block b'_i, of length strictly greater than m, defined as follows. For $i \geq 1$, let $h_i = b_1 \cdots b_{i-1}$ and $h'_i = b'_1 \cdots b'_{i-1}$. Thus, h_i and h'_i are the prefixes of w and w', respectively, that consist of the first $i-1$ blocks. Note that $h_1 = h'_1 = \epsilon$.

Assume we have already defined h'_i. We define b'_i as follows: By Lemma 2, for every $i \geq 1$ there exist $s_i \in \Sigma^*$ and $z_i \in \Sigma^+$ such that $s_i \prec s_i \cdot z_i \preceq b_i$ and $s_i \equiv_{\mathcal{A}} s_i \cdot z_i$. Let t_i in Σ^* be such that $b_i = s_i \cdot z_i \cdot t_i$. Now, we define $b'_i = s_i \cdot (z_i)^i \cdot t_i$. Thus, b'_i is obtained from b_i by pumping an infix of it that lies between two prefixes that are $\equiv_{\mathcal{A}}$-equivalent.

For runs r and r' of $\mathcal{A}$ on w and w', respectively, we say that r and r' are matching if for all $i \geq 0$, the run r visits α when it reads the block b_i iff the run r' visits α when it reads the block b'_i. We prove that every run r of $\mathcal{A}$ on w induces a matching run r' of $\mathcal{A}$ on w', and, dually, every run r' of $\mathcal{A}$ on w' induces a matching run r of $\mathcal{A}$ on w.

Consider a run r of $\mathcal{A}$ on w. For all $i \geq 1$, recall that $b_i = s_i \cdot z_i \cdot t_i$, where $s_i \equiv_{\mathcal{A}} s_i \cdot z_i$. It is easy to see (by induction on j) that for all $j \geq 1$, the latter implies that $s_i \cdot z_i \equiv_{\mathcal{A}} s_i \cdot (z_i)^j$. In particular, $s_i \cdot z_i \equiv_{\mathcal{A}} s_i \cdot (z_i)^i$. We define the behavior of r' on b'_i as follows. By the definition so far, $r'(h'_i) = r(h_i)$. We rely on the fact that $s_i \cdot z_i \equiv_{\mathcal{A}} h'_i s_i \cdot (z_i)^i$ and define r' so that $r'(h'_i \cdot s_i \cdot (z_i)^i) = r(h_i \cdot s_i \cdot z_i)$. Also, r' visits α when it reads $s_i \cdot (z_i)^i$ iff r visits α when it reads $s_i \cdot z_i$. On t_i we define r' to be identical to r. It is easy to see that r and r' are matching. In a similar way, every run r' of $\mathcal{A}$ on w' induces a matching run r of $\mathcal{A}$ on w.

Recall that $w \in L(\mathcal{A}_P)$. Therefore, there is a run r of $\mathcal{A}$ on w that visits α infinitely often, or, equivalently, visits α in infinitely many blocks. Hence, the matching run r' of $\mathcal{A}$ on w' also visits α in infinitely many blocks, and $w' \in L(\mathcal{A}_B) = L(\mathcal{A}_P)$.

Since $w \in L(\mathcal{A}_P)$, there is a run r' on w' that is accepting with some bound $k \geq 1$. For every $i > k$, the block b'_i contains the infix $(z_i)^i$, for $z_i \neq \epsilon$, and is therefore of length at least k. Hence, the run r' visits α when it reads the the block b'_i. Let r be a run of $\mathcal{A}$ on w that matches r'. Since r and r' are matching, the run r visits α when it reads the block b_i, for all $i > k$. Since $|b_i| = m$, the longest α-less window in r after block i is of length $2m-1$, thus r is accepting with an eventual bound of $2m$. Hence, w is accepted by a run that has a $2m$ eventual bound. □

Theorem 6, together with Lemma 1, induce a translation of reg-NPBW to NCW: the NCW can guess the location in which the eventual bound k becomes valid and then stays in an accepting region as long as an accepting state of the NPBW is visited at least once every k transition. Since an NCW can be translated to an NPBW with eventual bound 1, we can conclude with the following.

Theorem 7. *reg-*NPBWs *are as expressive as* NCWs*.*

The construction used in the proof of Theorem 7 involves a blow-up that depends on the number $3^{(|Q|^2)}$ of equivalence classes, and is thus super-exponential. We now show that while we can do better, an exponential blow-up can not be avoided.

Theorem 8. *The tight blow-up in the translation of reg-*NPBW *to* NCW *is $n2^n$, where n is the number of states of the* NPBW*.*

Proof. Consider a reg-NPBW $\mathcal{A}$ with n states. From Theorem 7, we know that $L(\mathcal{A}_P)$ is NCW-recognizable. From Theorem 3, we know that $L(\mathcal{A}_P) = L(\mathcal{A}_B)$. Thus, $\mathcal{A}_B$ is an NBW whose language is NCW recognizable. Hence, by [6], there exists an NCW $\mathcal{B}$ with at most $n2^n$ states equivalent to $\mathcal{A}_B$, and hence also to $\mathcal{A}_P$. Moreover, it can be shown that the family of languages with which the $n2^n$ lower bound was proven in [6] can be defined by means of reg-NPBWs, rather than NBWs, implying a matching lower bound. □

4.3 Properties of Reg-NPBW and Reg-DPBW

The fact that reg-NPBW =NCW immediately implies that closure properties known for NCWs can be applied to reg-NPBW. For reg-DPBW, we present the corresponding constructions. The fact that only reg-DPBW are closed under complementation also implies that reg-NPBW are more expressive than reg-DPBW.

Theorem 9

1. *reg-*NPBW *are closed under finite union and intersection, and are not closed under complementation.*
2. *reg-*DPBW *are closed under finite union, intersection and complementation.*
3. *reg-*NPBW *are strictly more expressive than reg-*DPBW*.*

5 Decision Problems

In this section we study three basic decision problems for prompt automata: regularity (given $\mathcal{A}_P$, deciding whether $L(\mathcal{A}_P)$ is ω-regular) universality (given $\mathcal{A}_P$, deciding whether $L(\mathcal{A}_P) = \Sigma^\omega$), and containment (given $\mathcal{A}_P$ and an NBW $\mathcal{A}$, deciding whether $L(\mathcal{A}) \subseteq L(\mathcal{A}_P)$). Note that the nonemptiness problem for an NPBW $\mathcal{A}_P$ can ignore the promptness and solve the nonemptiness of $\mathcal{A}_B$ instead. The other problems, however, require new techniques. The main challenge solving them has to do with the fact that we cannot adopt solutions from the regular setting, as these involve complementation. Instead, we use *approximated determinization* of NPBW: determinization that restricts attention to words accepted with some fixed eventual bound. We show that we can approximately determinize NPBWs and that we can use the approximating automaton in order to solve the three problems.

We first show that in the deterministic setting, the three problems can be solved by analyzing the structure of the automaton. To see the idea behind the algorithms, consider a reachable state $q \in Q$ such that q is reachable from itself both by a cycle that intersects α and by a cycle that does not intersect α. The existence of such a state implies the existence of words $x, u, v \in \Sigma^*$ such that the word $x(uv^i)^\omega$ is accepted by $\mathcal{A}_P$ for all $i \in \mathbb{N}$, but $w = xuvuv^2uv^3 \cdots$ satisfies $w \in L(\mathcal{A}_B) \setminus L(\mathcal{A}_P)$. Thus, $\mathcal{A}_P$ is regular iff no such state exists, which can be checked in NLOGSPACE. As detailed in the proof, the algorithms for universality and containment follow similar arguments.

Theorem 10. *The regularity, universality, and containment problems for* DPBW *are NLOGSPACE-complete.*

We continue to the nondeterministic setting and start with the construction of the deterministic co-Büchi approximating automaton. The DCW $\mathcal{D}$ that approximates the NPBW $\mathcal{A}$ has a parameter k and it accepts exactly all words that are accepted in $\mathcal{A}$ with eventual bound k. Essentially, $\mathcal{D}$ follows the subset construction of $\mathcal{A}$, while keeping, for each reachable state q, the minimal number of transitions along which q is reachable from a state in α in some run of $\mathcal{A}$. If this number exceeds k, then $\mathcal{D}$ regards it as ∞, meaning that this state cannot be part of a run that has already reached the suffix in which the eventual bound applies. A run of $\mathcal{D}$ is accepting if eventually it visits only subsets that contain at least one state with a finite bound. Indeed, a run of $\mathcal{D}$ can get stuck in such states iff there is a run of $\mathcal{A}$ that visits α every at most k transitions. We note that a similar construction is described in [20], for the translation of a bound automaton with parameter k to a deterministic Büchi automaton.

Theorem 11. *Let $\mathcal{A}$ be an* NPBW *with n states. For each $k \in \mathbb{N}$ there exists a* DCW *$\mathcal{D}$ with at most $(k+1)^n$ states such that $L(\mathcal{D}) = \{w : w$ is accepted by $\mathcal{A}$ with eventual bound $k\}$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ and let $k \in \mathbb{N}$. We define $\mathcal{D} = \langle \Sigma, Q', \delta', Q'_0, \alpha' \rangle$ as follows. The state space Q' is the set of partial functions $d : Q \to \{0, ..., k-1, \infty\}$. For a state $d \in Q'$, we say that a state $q \in Q$ is *d-safe* if $d(q) \in \{0, 1, ..., k-1\}$. Let *safe*$(d) = \{q : q$ is d-safe$\}$ be the set of all states that are d-safe. When $\mathcal{A}$ reads a word w, a state $d \in Q'$ keeps track of all the runs of $\mathcal{A}$ on w in the following manner. For each state $q \in Q$, if $d(q)$ is undefined, then q is not reachable in all runs of $\mathcal{A}$ on the prefix of the input word read so far. Otherwise, $d(q)$ is the minimal number of transition that some run of $\mathcal{A}$ on w that visits q has made since visiting α. If this number is greater than k, that is, no run of $\mathcal{A}$ that visits q has visited α in the past k transitions, then $d(q) = \infty$. Intuitively, this means that runs that visit q are still not in the suffix in which α is visited promptly. As we shall prove below, some run has reached a "good" suffix, iff $\mathcal{D}$ can get trapped in states that contain at least one safe state.

We now define $\mathcal{D}$ formally. The initial state is $Q'_0 = \{d_0\}$ such that for all $q \in Q_0$, we have $d_0(q) = 0$. The set of accepting states is $\alpha' = \{d : \mathit{safe}(d) \neq \emptyset\}$. Before we define the transition function, we define an addition operator on $\{0, 1, ..., k-1, \infty\}$ as follows. For all $i \in \{1, ..., k-1, \infty\}$, we define $i+1 = i+1$ if $i < k-1$ and $i+1 = \infty$ if $i \in \{k-1, \infty\}$.

For technical convenience, we associate with a state d the set S_d of states on which d is defined. We say that d is *lost* if for all $q \in S_d$, we have $d(q) = \infty$. For $d \in Q'$ and $\sigma \in \Sigma$ we define δ' as follows. If d is lost, then $\delta'(d, \sigma)(q') = 0$ for all $q' \in \delta(S_d, \sigma)$. Otherwise we define $\delta'(d, \sigma) = d'$ as follows. For $q' \in \delta(\mathit{safe}(d), \sigma) \cap \alpha$, we have $d'(q') = 0$. For $q' \in \delta(S_d, \sigma) \cap (Q \setminus \alpha)$ we have $d'(q') = \min\{d(q) + 1 : q' \in \delta(q, \sigma) \text{ and } q \in S_d\}$. Finally, for $q' \in (\delta(S_d, \sigma) \setminus \delta(\mathit{safe}(d), w_i)) \cap \alpha$ we have $d'(q') = \infty$. Intuitively, $\mathit{safe}(d)$ is the set of states that are still within the k bound in some run of $\mathcal{A}$. Thus, d' checks which of these states indeed visit α after reading σ, and resets them to 0. Otherwise it increases the counter. If all states reached ∞, then the eventual bound k was not yet in effect. We "take note" of this by not visiting α, and then d' resets to 0, giving the eventual bound a new chance. One may notice that the sets S_d are exactly the subset construction of $\mathcal{A}$. Thus, essentially, d is a labeling function of the subset construction. $\square$

We refer to $\mathcal{D}$ as the *k-approximating DCW of $\mathcal{A}$*. While the deterministic DCW constructed in Theorem 11 accepts only words that are accepted in the original NPBW with a fixed eventual bound, Theorem 6 enables us to use such a DCW in the process of deciding properties of the NPBW. In particular, for a regular-NPBW $\mathcal{A}$, Theorem 6 implies that a DCW constructed with bound $2 \cdot 3^{n^2}$ is equivalent to $\mathcal{A}$. Recall that NPBWs are NBW-type. By [6], an NBW whose language is DCW-recognizable can be translated to a DCW with 3^n states. Hence we have the following.

Theorem 12. *Let $\mathcal{A}$ be a regular* NPBW*. There exists a* DCW *with at most* 3^n *states such that $L(\mathcal{D}) = L(\mathcal{A})$.*

In fact, as we show below, the deterministic approximating automaton is helpful for deciding all three problems, even when applied to non-regular NPBWs.

Theorem 13. *The regularity, universality, and containment problems for* NPBWs *are PSPACE-complete.*

Proof. We prove here the upper bounds. For the lower bounds, we use a reduction from NFW universality to NPBW universality, and a reduction from NPBW universality to NPBW regularity. Since universality can be reduced to containment, PSPACE-hardness for all the three problems follow.

We first prove that the universality problem is in PSPACE. Let $\mathcal{A}$ be an NPBW. Let $\mathcal{D}$ be the DCW defined in Theorem 11 with bound $2 \cdot 3^{n^2}$. We claim that $L(\mathcal{A}_P) = \Sigma^\omega$ iff $L(\mathcal{D}) = \Sigma^\omega$. For the first direction, if $L(\mathcal{A}_P) = \Sigma^\omega$, then in particular $L(\mathcal{A}_P)$ is in reg-NPBW. From Theorem 12 we get that $L(\mathcal{D}) = L(\mathcal{A}_P)$, so $L(\mathcal{D}) = \Sigma^\omega$. For the other direction, observe that it is always true that $L(\mathcal{D}) \subseteq L(\mathcal{A}_P)$. Thus, if $L(\mathcal{D}) = \Sigma^\omega$, then $L(\mathcal{A}_P) = \Sigma^\omega$. Since the number of states in $\mathcal{D}$ is $(2 \cdot 3^{n^2} + 1)^n$ and the universality problem for DCWs is in NLOGSPACE, a PSPACE upper bound follows.

We proceed to prove that the regularity problem is in PSPACE. Consider the automaton $\mathcal{D}$ constructed in Theorem 12. If $L(\mathcal{A}_P) = L(\mathcal{A}_B)$ then $L(\mathcal{A}_P)$ is ω-regular and thus $L(\mathcal{D}) = L(\mathcal{A}_P) = L(\mathcal{A}_B)$. On the other hand, if $L(\mathcal{D}) = L(\mathcal{A}_P)$ then $L(\mathcal{A}_P)$ is ω-regular and by Theorem 3 we have that $L(\mathcal{A}_P) = L(\mathcal{A}_B)$. Thus, deciding whether $L(\mathcal{A}_P) = L(\mathcal{A}_B)$ is equivalent to deciding whether $L(\mathcal{D}) = L(\mathcal{A}_B)$. Note, however, that by Theorem 11 it is always true that $L(\mathcal{D}) \subseteq L(\mathcal{A}_P) \subseteq L(\mathcal{A}_B)$. Thus, it is sufficient to check whether $L(\mathcal{A}_B) \subseteq L(\mathcal{D})$, which can be done in PSPACE.

It is left to prove that the containment problem is in PSPACE. We describe a PSPACE algorithm for deciding whether $L(\mathcal{B}) \subseteq L(\mathcal{A}_P)$. Let m and n be the number of states in $\mathcal{B}$ and $\mathcal{A}$, respectively. First, check whether $L(\mathcal{B}) \subseteq L(\mathcal{A}_B)$. If $L(\mathcal{B}) \not\subseteq L(\mathcal{A}_B)$ return no. Otherwise, construct, per Theorem 11, a $2k$-approximating DCW, with $k = (m+1) \cdot 3^{n^2}$. Next, check whether $L(\mathcal{B}) \subseteq L(\mathcal{D})$. If so, return yes. Otherwise, return no.

It is not hard to see that the algorithm can be implemented in PSPACE. We now prove its correctness. Since it is always true that $L(\mathcal{D}) \subseteq L(\mathcal{A}_P) \subseteq L(\mathcal{A}_B)$, then it is easy to see that if $L(\mathcal{B}) \subseteq L(\mathcal{D})$ then $L(\mathcal{B}) \subseteq L(\mathcal{A}_P)$, and if $L(\mathcal{B}) \not\subseteq L(\mathcal{A}_B)$ then $L(\mathcal{B}) \not\subseteq L(\mathcal{A}_P)$. It remains to show that if $L(\mathcal{B}) \subseteq L(\mathcal{A}_B)$ but $L(\mathcal{B}) \not\subseteq L(\mathcal{D})$ then $L(\mathcal{B}) \not\subseteq L(\mathcal{A}_P)$.

Recall that $L(\mathcal{D}) = \{w : w$ is accepted by $\mathcal{A}_P$ with eventual bound of at most $2k\}$. We claim that if $L(\mathcal{B}) \subseteq L(\mathcal{A}_P)$ then $L(\mathcal{B}) \subseteq L(\mathcal{D})$. Thus, we show that if $w \in L(\mathcal{B})$ then w is accepted by $\mathcal{A}_P$ with eventual bound of at most $2k$. The proof of this claim follows the reasoning in the proof of Theorem 6.

Let $w \in L(\mathcal{B})$. Since $L(\mathcal{B}) \subseteq L(\mathcal{A}_P)$ then $w \in L(\mathcal{A}_P)$. Assume by way of contradiction that w is accepted by $\mathcal{A}_P$ with an eventual bound greater than $2k$. We divide w into blocks of length k, so $w = b_1 \cdot b_2 \cdots$ such that $|b_i| = k$. Let s and r be accepting runs of $\mathcal{B}$ and $\mathcal{A}_P$ on w, respectively. Thus, there are infinitely many blocks b_i such that s visits $\alpha_{\mathcal{B}}$ when reading b_i. Since r has an eventual bound greater than $2k$, then there are infinitely many blocks b_i such that r does not visit $\alpha_{\mathcal{A}}$ when reading b_i. Since we took $k = (m+1) \cdot 3^{n^2}$, then in every block $b_i = x_1 \cdots x_k$ there exist two indices $j_1 < j_2$ such that $x_1 \cdots x_{j_1} \equiv_{\mathcal{A}} x_1 \cdots x_{j_2}$ and $s(x_{j_1}) = s(x_{j_2})$. We can now pump the infix $x_{j_1} \cdots x_{j_2}$ such that both r and s are not affected outside the pumped infix. Now, similarly to the proof of Theorem 6, we can pump infinitely many blocks b_i such that the run r is no longer prompt-accepting and s is still accepting. We end up with a word w' that is accepted by $\mathcal{B}$. By our assumption, w' is accepted by $\mathcal{A}_P$, and thus has a prompt accepting run r' of $\mathcal{A}_P$. We can shrink the pumped blocks of w' back to w such that the respective shrinking of s' is a prompt accepting run of w with eventual bound of at most $2k$, which leads to a contradiction. ☐

In [14], the authors describe a PSPACE model-checking algorithm for Prompt-LTL. Our PSPACE containment algorithm completes the picture and implies that all prompt properties given by an NPBW can be model-checked in PSPACE.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. IPL 21, 181–185 (1985)
2. Alur, R., Etessami, K., La Torre, S., Peled, D.: Parametric temporal logic for model measuring. ACM Transactions on Computational Logic 2(3), 388–407 (2001)
3. Alur, R., Henzinger, T.A.: Finitary fairness. In: Proc. 9th IEEE Symp. on Logic in Computer Science, pp. 52–61 (1994)
4. Bojanczyk, M.: A bounding quantifier. In: Proc. 13th Annual Conf. of the European Association for Computer Science Logic, pp. 41–55 (2004)
5. Bojańczyk, M., Colcombet, T.: Bounds in ω-regularity. In: Proc. 21st IEEE Symp. on Logic in Computer Science, pp. 285–296 (2006)
6. Boker, U., Kupferman, O.: Co-ing Büchi made tight and helpful. In: Proc. 24th IEEE Symp. on Logic in Computer Science, pp. 245–254 (2009)
7. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. Int. Congress on Logic, Method, and Philosophy of Science, pp. 1–12 (1962)
8. Chatterjee, K., Henzinger, T.A.: Finitary winning in ω-regular games. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 257–271. Springer, Heidelberg (2006)
9. Derahowitz, N., Jayasimha, D.N., Park, S.: Bounded fairness. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 304–317. Springer, Heidelberg (2004)
10. Horn, F.: Faster algorithms for finitary games. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 472–484. Springer, Heidelberg (2007)
11. Jones, N.D.: Space-bounded reducibility among combinatorial problems. Journal of Computer and Systems Science 11, 68–75 (1975)
12. Krishnan, S.C., Puri, A., Brayton, R.K.: Deterministic ω-automata vis-a-vis deterministic Büchi automata. In: Du, D.-Z., Zhang, X.-S. (eds.) ISAAC 1994. LNCS, vol. 834, pp. 378–386. Springer, Heidelberg (1994)
13. Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. IJFCS 17(4), 869–884 (2006)
14. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
15. Kupferman, O., Piterman, N., Vardi, M.Y.: From liveness to promptness. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 406–419. Springer, Heidelberg (2007)
16. Kupferman, O., Vardi, M.Y.: Verification of fair transition systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 372–382. Springer, Heidelberg (1996)
17. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton Univ. Press, Princeton (1994)
18. Landweber, L.H.: Decision problems for ω–automata. Mathematical Systems Theory 3, 376–384 (1969)
19. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)
20. Margaria, T., Sistla, A.P., Steffen, B., Zuck, L.D.: Taming interface specifications. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 548–561. Springer, Heidelberg (2005)

Using Redundant Constraints for Refinement

Eugene Asarin[1], Thao Dang[2], Oded Maler[2], and Romain Testylier[2]

[1] LIAFA,
CNRS and Université Paris Diderot
Case 7014 75205 Paris Cedex 13, France
[2] VERIMAG,
2 avenue de Vignate,
38610 Gières, France

Abstract. This paper is concerned with a method for computing reachable sets of linear continuous systems with uncertain input. Such a method is required for verification of hybrid systems and more generally embedded systems with mixed continuous-discrete dynamics. In general, the reachable sets of such systems (except for some linear systems with special eigenstructures) are hard to compute exactly and are thus often over-approximated. The approximation accuracy is important especially when the computed over-approximations do not allow proving a property. In this paper we address the problem of refining the reachable set approximation by adding redundant constraints which allow bounding the reachable sets in some critical directions. We introduce the notion of directional distance which is appropriate for measuring approximation effectiveness with respect to verifying a safety property. We also describe an implementation of the reachability algorithm which favors the constraint-based representation over the vertex-based one and avoids expensive conversions between them. This implementation allowed us to treat systems of much higher dimensions. We finally report some experimental results showing the performance of the refinement algorithm.

1 Introduction

Hybrid systems, that is, systems exhibiting both continuous and discrete dynamics, have been recognized as a high-level model appropriate for embedded systems, since this model can describe, within a unified framework, the logical part and the continuous part of an embedded system. Due to the safety critical features of many embedded applications, formal analysis is a topic of particular interest. Recently, much effort has been devoted to the development of automatic analysis methods and tools for hybrid systems, based on formal verification.

Reachability analysis is a central problem in formal verification. The importance of this problem can be seen in its growing literature. For hybrid systems with simple continuous dynamics that have piecewise constant derivatives, their reachable sets can be computed using linear algebra. This is a basis for the verification algorithms implemented in a number of tools, such as UPPAAL [19], KRONOS [25], HYTECH [13] and PHAVER [9]. For hybrid systems with more complex continuous dynamics described by differential or difference equations, the reachability problem is much more difficult, and a major ingredient in designing a reachability analysis algorithm for such hybrid systems is an efficient method to handle their continuous dynamics. Hybrid systems can be seen as a combination of continuous and discrete dynamics; their verification thus requires, in addition to computing reachable sets of continuous dynamics, performing operations specific for discrete behavior, such as Boolean operations over reachable sets. It is worth noting first that *exact computation* of reachable sets of linear continuous systems is in general difficult. It was proved in [21] that this is possible for a restricted class with special eigenstructures.

In this paper we revisit the problem of approximating reachable sets of linear continuous systems, which was investigated in [3,8]. The approximation accuracy is important especially when the computed over-approximations do not allow proving a property. We propose a method for refining a reachable set approximation by adding redundant constraints to bound it in some critical directions. We also introduce the notion of directional distance which is appropriate for measuring approximation effectiveness with respect to verifying a safety property. Although in this paper we focus only on methods for accurately approximating reachable sets of linear continuous systems, it is important to note that we use convex polyhedra to represent reachable sets, the treatment of discrete dynamics in a hybrid system can thus be done using available algorithms for Boolean operations over these polyhedra. The methods presented in this paper can therefore be readily integrated in any existing polyhedron-based algorithm for verification of hybrid automata with linear continuous dynamics, such as [2].

The reachability problem for linear systems has been a topic of intensive research over the past decade. We defer a discussion on related work to the last section. The paper is organized as follows. We start with a brief description of the technique for approximating reachable sets using optimal control [24,3]. We then present two methods for refining approximations by using redundant constraints. One method can reduce approximation error globally while the other is more local and guided by the property to verify. We also describe an efficient implementation of these methods and some experimental results.

2 Preliminaries

Notation and Definitions. Let $\mathbf{x}$, $\mathbf{y}$ be two points in $\mathbb{R}^n$ and X, Y be two subsets of $\mathbb{R}^n$. We denote by $\langle \mathbf{x}, \mathbf{y} \rangle$ the scalar product of $\mathbf{x}$ and $\mathbf{y}$. The Hausdorff semi-distance from X to Y is defined as $h_+(X,Y) = \sup_{\mathbf{x} \in X} \inf_{\mathbf{y} \in Y} \{||\mathbf{x} - \mathbf{y}||\}$ where $||\cdot||$ is the Euclidian norm. The Hausdorff distance between X and Y is $h(X,Y) = \sup \{h_+(X,Y), h_+(Y,X)\}$.

We now consider a discrete-time linear system described by the following equation

$$\mathbf{x}(k+1) = A\mathbf{x}(k) + \mathbf{u}(k) \tag{1}$$

where $\mathbf{x} \in \mathcal{X}$ is the state of the system and $\mathbf{u} \in U$ is the input. The input set U is a bounded convex polyhedron in $\mathbb{R}^n$. We assume a set $\mathcal{U}$ of admissible input functions consisting of functions of the form $\boldsymbol{\mu} : \mathbb{Z}^+ \to U$. Note that any system $\mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{v}(k)$ where $\mathbf{v}(k) \in \mathcal{V} \subset \mathbb{R}^m$ can be transformed into a system of the form (1) by letting $\mathbf{u} = B\mathbf{v}$ and defining the input set $U = \{\mathbf{u} \mid \mathbf{u} = B\mathbf{v} \ \wedge \ \mathbf{v} \in \mathcal{V}\}$.

A trajectory of (1) starting from a point $\mathbf{y} \in \mathcal{X}$ under a given input $\boldsymbol{\mu} \in \mathcal{U}$ is the solution $\xi_{\boldsymbol{\mu}}(\mathbf{y}, k)$ of (1) with the initial condition $\mathbf{x}(0) = \mathbf{y}$, that is $\forall k > 0 \ \xi_{\boldsymbol{\mu}}(\mathbf{y}, k+1) = A\xi_{\boldsymbol{\mu}}(\mathbf{y}, k) + \boldsymbol{\mu}(k)$.

Let $\mathcal{I}$ be a subset of $\mathcal{X}$. The set of states reachable from $\mathcal{I}$ at a time point k under a given input $\boldsymbol{\mu} \in \mathcal{U}$, denoted by $Post_{\boldsymbol{\mu}}(\mathcal{I}, k)$, is the set of states visited at time k by all the trajectories starting from points in $\mathcal{I}$ under $\boldsymbol{\mu}$: $Post_{\boldsymbol{\mu}}(\mathcal{I}, k) = \bigcup\limits_{\mathbf{y} \in \mathcal{I}} \xi_{\boldsymbol{\mu}}(\mathbf{y}, k)$. The set of all states reachable from $\mathcal{I}$ at time point k is $Post(\mathcal{I}, k) = \bigcup\limits_{\boldsymbol{\mu} \in \mathcal{U}} Post_{\mathcal{I}, \boldsymbol{\mu}}(k)$.

Reachability Problem. The reachability problem we address in this paper is stated as follows. Given a set $\mathcal{I} \subset \mathcal{X}$ of initial states, we want to compute the reachable set $Post(\mathcal{I}, k)$ of the system (1) up to some time $k = k_{max}$.

2.1 Reachability Algorithm

With a view to safety verification, our goal is to obtain conservative approximations. To this end, we make use of the technique, which was first suggested in [24] and then applied in [3,8,2] for hybrid systems. This technique is based on the Maximum Principle in optimal control [14]. In the following, we recap the main idea of this technique.

We assume that the initial set is a bounded convex polyhedron described as the intersection of a set $\mathcal{H} = \{H(\mathbf{a}_1, \mathbf{y}_1), \ldots, H(\mathbf{a}_m, \mathbf{y}_m)\}$ of m half-spaces: $\mathcal{I} = \bigcap\limits_{i=0}^{m} H(\mathbf{a}_i, \mathbf{y}_i)$ where each half-space $H(\mathbf{a}_i, \mathbf{y}_i) \in \mathcal{H}$ is defined as $H(\mathbf{a}_i, \mathbf{y}_i) = \{\mathbf{x} \mid \langle \mathbf{a}_i, \mathbf{x} \rangle \leq \langle \mathbf{a}_i, \mathbf{y}_i \rangle = \gamma_i\}$. We say that $\langle \mathbf{a}_i, \mathbf{x} \rangle \leq \langle \mathbf{a}_i, \mathbf{y}_i \rangle$ is the linear constraint associated with $H(\mathbf{a}_i, \mathbf{y}_i)$. For simplicity, we sometimes use $H(\mathbf{a}_i, \mathbf{y}_i)$ to refer to both the half-space and its associated constraint. The polyhedron defined by the intersection of the half-spaces of $\mathcal{H}$ is denoted by $Poly(\mathcal{H})$. We emphasize that in the above description of $\mathcal{I}$, the set of associated constraints may not be minimal, in the sense that it may contain redundant constraints. The vector $\mathbf{a}_i$ is called the *outward normal* of the half-space $H(\mathbf{a}_i, \mathbf{y}_i)$ and $\mathbf{y}_i$ is an arbitrary point of $Poly(\mathcal{H})$ on the boundary of the half-space $H(\mathbf{a}_i, \mathbf{y}_i)$, which we call a *supporting point* of $H(\mathbf{a}_i, \mathbf{y}_i)$.

By the Maximum Principle [14], for every half-space $H(\mathbf{a}, \mathbf{y})$ of $\mathcal{I}$ with the outward normal $\mathbf{a}$ and a supporting point $\mathbf{y}$, there exists an input $\boldsymbol{\mu}^* \in \mathcal{U}$ such that computing the successors of $H(\mathbf{a}, \mathbf{y})$ under $\boldsymbol{\mu}^*$ is sufficient to derive a tight polyhedral approximation of $Post(\mathcal{I}, k)$. It can be proved that the evolution of the normal of each $H(\mathbf{a}, \mathbf{y})$ is governed by the adjoint system whose dynamics is described by the following equation:

$$\boldsymbol{\lambda}(k+1) = -A^T \boldsymbol{\lambda}(k). \tag{2}$$

The solution to (2) with the initial condition $\boldsymbol{\lambda}(0) = \mathbf{a}$ is denoted by $\lambda_{\mathbf{a}}(k)$. The following proposition shows that one can compute *exactly* the reachable set from a half-space.

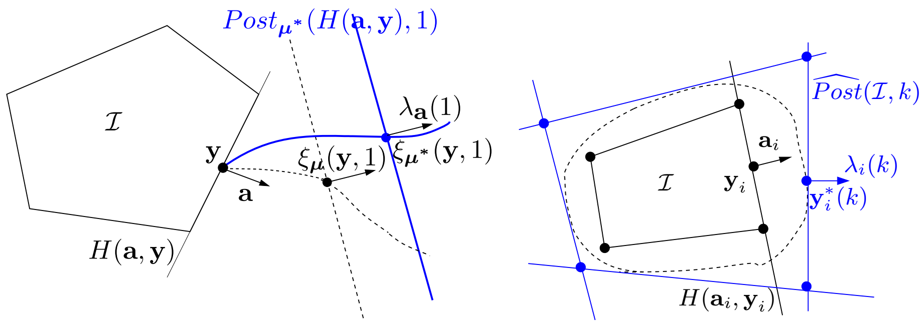
Proposition 1. *Let $H(\mathbf{a}, \mathbf{y}) = \{\mathbf{x} \mid \langle \mathbf{a}, \mathbf{x}\rangle \leq \langle \mathbf{a}, \mathbf{y}\rangle\}$ be a half-space with the normal $\mathbf{a}$ and a supporting point $\mathbf{y}$. Let $\lambda_{\mathbf{a}}(k)$ be the solution to (2) with the initial condition $\boldsymbol{\lambda}(0) = \mathbf{a}$. Let $\boldsymbol{\mu}^*(k) \in \mathcal{U}$ be an input such that for every $k \geq 0$*

$$\boldsymbol{\mu}^*(k) \in \arg\max\{\langle \lambda_{\mathbf{a}}(k), \mathbf{u}\rangle \mid \mathbf{u} \in U\}.$$

Let $\xi_{\boldsymbol{\mu}^*}(\mathbf{y}, k)$ be the solution to (1) under the above input $\boldsymbol{\mu}^*$ with the initial condition $\mathbf{x}(0) = \mathbf{y}$. Then, the reachable set from $H(\mathbf{a}, \mathbf{y})$ is the half-space with the normal $\lambda_{\mathbf{a}}(k)$ and the supporting point $\xi_{\boldsymbol{\mu}^*}(\mathbf{y}, k)$:

$$Post(H(\mathbf{a}, \mathbf{y}), k) = Post_{\boldsymbol{\mu}^*}(H(\mathbf{a}, \mathbf{y}), k) = \{\mathbf{x} \mid \langle \lambda_{\mathbf{a}}(k), \mathbf{x}\rangle \leq \langle \lambda_{\mathbf{a}}(k), \xi_{\boldsymbol{\mu}^*}(\mathbf{y}, k)\rangle\}. \tag{3}$$

We remark that the evolution of the normal $\lambda_{\mathbf{a}}(k)$ is independent of the input; therefore for all $\boldsymbol{\mu} \in \mathcal{U}$ the boundaries of the half-spaces $Post_{\boldsymbol{\mu}}(H(\mathbf{a}, \mathbf{y}), k)$ are parallel to each other, as shown in Figure 1. ■



$Post_{\boldsymbol{\mu}^*}(H(\mathbf{a}, \mathbf{y}), 1)$
$\mathcal{I}$
$\mathbf{y}$
$\mathbf{a}$
$\lambda_{\mathbf{a}}(1)$
$\xi_{\boldsymbol{\mu}}(\mathbf{y}, 1)$
$\xi_{\boldsymbol{\mu}^*}(\mathbf{y}, 1)$
$H(\mathbf{a}, \mathbf{y})$
$\widehat{Post}(\mathcal{I}, k)$
$\mathbf{a}_i$
$\mathcal{I}$
$\mathbf{y}_i$
$\lambda_i(k)$
$\mathbf{y}_i^*(k)$
$H(\mathbf{a}_i, \mathbf{y}_i)$

Fig. 1. Left figure: The solid and the dotted curves are the trajectories $\xi_{\boldsymbol{\mu}^*}(\mathbf{y}, k)$ under $\boldsymbol{\mu}^*$ and $\xi_{\boldsymbol{\mu}}(\mathbf{y}, k)$ under $\boldsymbol{\mu}$. At the first step, the reachable set $Post_{\boldsymbol{\mu}^*}(H(\mathbf{a}, \mathbf{y}), 1)$ is the half-space determined by the normal $\lambda_{\mathbf{a}}(1)$ and the supporting point $\xi_{\boldsymbol{\mu}^*}(\mathbf{y}, 1)$. **Right figure**: Over-approximating the exact reachable set $Post(\mathcal{I}, k)$ (shown in dotted line) by the polyhedron $\widehat{Post}(\mathcal{I}, k)$.

The following proposition [24,8] shows that one can *conservatively* approximate the reachable set from the convex polyhedron $\mathcal{I} = \bigcap_{i=0}^{m} H(\mathbf{a}_i, \mathbf{y}_i)$.

Proposition 2. *For every $i \in \{1, \ldots, m\}$, let ${\boldsymbol{\mu}^*}_i(k)$ be an admissible input such that for every $k \geq 0$ ${\boldsymbol{\mu}^*}_i(k) \in \arg\max\{\langle \lambda_{\mathbf{a}_i}(k), \mathbf{u}\rangle \mid \mathbf{u} \in U\}$. Then,*

$$Post(\mathcal{I}, k) \subseteq \bigcap_{i=0}^{m} \{\mathbf{x} \mid \langle \lambda_{\mathbf{a}_i}(k), \mathbf{x}\rangle \leq \langle \lambda_{\mathbf{a}_i}(k), \xi_{{\boldsymbol{\mu}^*}_i}(\mathbf{y}_i, k)\rangle\}.$$

Proposition 2 provides the following scheme to over-approximate $Post(\mathcal{I}, k)$ by $\widehat{Post}(\mathcal{I}, k)$. For brevity we denote $\mathbf{y}_i^*(k) = \xi_{\boldsymbol{\mu^*}_i}(\mathbf{y}_i, k)$ and $\lambda_i(k) = \lambda_{\mathbf{a}_i}(k)$.

$$\lambda_i(k+1) = -A^T\lambda_i(k);\ \ \lambda_i(0) = \mathbf{a}_i, \tag{4}$$

$$\mathbf{y}_i^*(k+1) = A\mathbf{y}_i^*(k) + \boldsymbol{\mu^*}_i(k);\ \ \mathbf{y}_i^*(0) = \mathbf{y}_i, \tag{5}$$

$$\boldsymbol{\mu^*}_i(k) \in \arg\max\ \{\langle\lambda_i(k), \mathbf{u}\rangle \mid \mathbf{u} \in U\}. \tag{6}$$

Algorithm 1. Over-approximating $Post_{\mathcal{I}}(k)$

```
for $i = 1, \ldots, m$ do
    Compute $\lambda_i(k)$ by solving (4).
    Compute $\boldsymbol{\mu^*}_i(k) = \arg\max\ \{\langle\boldsymbol{\lambda}_i(k), \mathbf{u}\rangle \mid \mathbf{u} \in U\}$.
    Compute $\mathbf{y}_i^*(k)$ by solving (5) with $\boldsymbol{\mu^*}_i(k)$ obtained in the previous step.
end for
$\widehat{Post}(\mathcal{I}, k) = \bigcap_{i=1}^m\ \{\mathbf{x} \mid \langle\lambda_i(k), \mathbf{x}\rangle \leq \langle\lambda_i(k), \mathbf{y}_i^*(k)\rangle\}$.
```

Note that if the input set U is a bounded convex polyhedron then $\boldsymbol{\mu^*}_i(k)$ can be selected at one of its vertices at every time point k. The last step consists in intersecting all the half-spaces defined by the normal vectors $\lambda_i(k)$ and the points $\mathbf{y}_i^*(k)$ to obtain the convex polyhedron $\widehat{Post}(\mathcal{I}, k)$, which is an over-approximation of $Post(\mathcal{I}, k)$ (see Figure 1 for an illustration of the algorithm).

2.2 Approximation Accuracy

As a metric for our approximations, we use the Hausdorff distance introduced in the beginning of Section 2. This is a good measure for differences between sets. The approximation error is defined as: $\epsilon_k = h(\widehat{Post}(\mathcal{I}, k), Post(\mathcal{I}, k))$.

Proposition 3. *For a linear system $\mathbf{x}(k+1) = A\mathbf{x}(k) + \mathbf{u}(k)$ where the set of input function is a singleton, the equality in Proposition 2 is achieved, that is the set $\widehat{Post}(\mathcal{I}, k) = Post(\mathcal{I}, k)$.*

To prove this result, we notice that the image of the intersection of two sets X and Y by a function g satisfies $g(X \cap Y) \subseteq g(X) \cap g(Y)$. In particular, if g is injective, $g(X \cap Y) = g(X) \cap g(Y)$. Indeed, the solutions of deterministic systems are injective since they do not cross one another. On the contrary, the solutions of non-deterministic systems are generally not injective (since from different initial states, different inputs may lead the system to the same state). This implies that for linear systems with uncertain input, Algorithm 1 produces only an over-approximation of this set. ■

Proposition 4. *If a half-space H supports the initial set $\mathcal{I}$, that is its boundary contains a point in $\mathcal{I}$, then for every $k > 0$ the half-space $Post(H, k)$ computed as shown in the formula (3) supports the exact reachable set $Post(\mathcal{I}, k)$.*

The proof of this can be straightforwardly established from the fact that the supporting point $\mathbf{y}^*(k) = \xi_{\boldsymbol{\mu}^*}(\mathbf{y}, k)$ of each $Post_k(H)$ is indeed a point in the exact reachable set from $\mathcal{I}$ since it is computed as a successor from a point in $\mathcal{I}$ under an admissible input function. ■

From the above results, to improve the approximation accuracy one can use additional half-spaces in the description of the initial set, such that with respect to the initial set their associated constraints are redundant, but under the system's dynamics they evolve into new half-spaces which can be useful for bounding the approximation in some directions. In order to significantly reduce the approximation error which is measured by the Hausdorff distance between the approximate and the exact sets, it is important to find the area when the difference between these sets is large. In the following, we propose two methods for refining the reachable set approximation by dynamically adding constraints.

3 Refinement Using Sharp Angles

For simplicity we use $\mathcal{H} = \{H_1, \ldots, H_m\}$ with $H_i = H(\mathbf{a}_i, \mathbf{y}_i) = \{\mathbf{x} \mid \langle \mathbf{a}_i, \mathbf{x}\rangle \leq \langle \mathbf{a}_i, \mathbf{y}_i\rangle = \gamma_i\}$ to denote the half-spaces of the reachable set at the k^{th} step. As mentioned earlier, the constraints to add should not change the polyhedron and thus must be redundant. Another requirement is that the corresponding half-spaces should be positioned where the approximation error is large. The over-approximation error indeed occurs mostly around the intersections of the half-spaces. In addition, this error is often large when the angle between two adjacent half-spaces, which can be determined from their normal vectors, is smaller than some threshold σ. We call them *adjacent half-spaces with sharp angle.*

Indeed, when two adjacent half-spaces form a sharp angle, the area near their intersection is elongated, which causes a large difference between the polyhedral approximation and the actual reachable set. Hence, in order to better approximate the exact boundary, one needs to use more approximation directions.

A constraint to add can be determined as follows. Its normal vector $\boldsymbol{\lambda}_n$ can be defined by a linear combination of $\boldsymbol{\lambda}_l$ and $\boldsymbol{\lambda}_j$: $\boldsymbol{\lambda}_n = w_1\boldsymbol{\lambda}_l + w_2\boldsymbol{\lambda}_j$ where $w_1 > 0$, $w_2 > 0$. We next determine a supporting point of the constraint. To this end, we find a point on the facet in the intersection between H_l and H_j by solving the following linear programming problem:

$$\begin{gathered}\min \langle \boldsymbol{\lambda}_n, \mathbf{x}\rangle \\ \text{s.t. } \forall q \in \{1, \ldots, m\} : q \neq l \ \wedge q \neq j \wedge \langle \boldsymbol{\lambda}_q, \mathbf{x}\rangle \leq \gamma_q \\ \langle \boldsymbol{\lambda}_l, \mathbf{x}\rangle = \gamma_l \\ \langle \boldsymbol{\lambda}_j, \mathbf{x}\rangle = \gamma_j\end{gathered} \tag{7}$$

The solution $\mathbf{x}^*$ of the above LP problem yields the new constraint $\langle \lambda_n, \mathbf{x}\rangle \leq \boldsymbol{\lambda}_n \mathbf{x}^*$, which can be used for refinement purposes.

It is important to note that while sharp angles between half-spaces are useful to identify the areas where the approximation error might be large, sharp angles can also be formed when the system converges to some steady state. In this case, "curving" the approximate set does not significantly improve the accuracy, because their normal vectors under the system's dynamics become very close to each other, and we choose not to add constraints in this case.

Dynamical Refinement. In the above we showed how to determine redundant constraints for refinement. The refinement process can be done dynamically as follows. During the computation, if at the k^{th} step, the sharp angle criterion alerts that the approximation error might be large, we move $r \leq k$ steps backwards and add constraints for each pair of adjacent half-spaces with sharp angles. The computation is then resumed from the $(k-r)^{th}$ step.

An important remark is that the larger r is, the more significant accuracy improvement is achieved, at the price of more computation effort. Indeed, when we backtrack until the first step, the half-spaces corresponding to the added constraints actually support the boundary of the initial set. Thus, by Proposition 4, their successors also support the exact reachable set from $\mathcal{I}$, which guarantees *approximation tightness*. On the contrary, if $r < k$, it follows from the above LP problem (7) that the added constraints only support the approximate reachable set and their boundaries therefore are not guaranteed to contain a truly reachable point.

Figure 4 illustrates the improvement in accuracy achieved by this method for a 2-dimensional linear system whose matrix is a Jordan block[1]. The colored parts correspond to the approximation error which is reduced by adding constraints using the sharp angle criterion.

4 Refinement Using Critical Directions

A good compromise between accuracy and computation time depends on the problems to solve and the available computation budget. In this section, we discuss a refinement procedure specific for safety verification problems. As we have seen earlier, in order to guarantee a desired approximation error bound, measured using the Hausdorff distance between the approximate and the exact sets, one needs to assure that their difference does not exceed the bound in all directions. Nevertheless, when reachability analysis is used to prove a safety property, this measure is not appropriate for characterizing the potential of reaching a unsafe zone. In this case, one is more interested in computing an approximation that may not be precise (with respect to the Hausdorff distance to the exact set) but enables deciding whether the exact set intersects with the unsafe set. To this end, we use a measure, called *directional distance*.

4.1 Directional Distance

Given two convex sets A and B in $\mathbb{R}^n$, a Boolean function $contact(A, B) = true$ if and only if the following two conditions are satisfied: (1) $Int(A) \cap Int(B) = \emptyset$; (2) $\delta A \cap \delta B \neq \emptyset$ where $Int(A)$ is the interior of A and δA is its boundary.

[1] A Jordan block is a matrix whose main diagonal is filled with a fixed number and all the entries which are directly above and to the right of the main diagonal are 1.

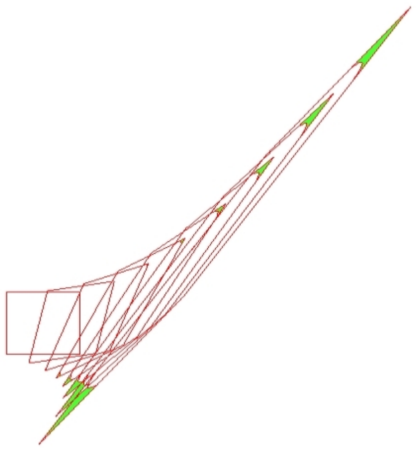


Fig. 2. Adding constraints can reduce approximation error (which is the colored areas in the figure)

Intuitively, *contact*(A, B) is true if and only if A and B intersect with each other and, in addition, they intersect only on their boundaries.

If $A \cap B \neq \emptyset$, the directional distance between A and B is defined as

$$\rho(A, B) = \inf_{\mathbf{p} \in \mathbb{R}^n} \{||\mathbf{p}|| : contact(A, B + \mathbf{p})\}.$$

where $||\mathbf{p}||$ denotes the Euclidian length of the vector $\mathbf{p}$, $B + \mathbf{p} = \{\mathbf{x} + \mathbf{p} \mid \mathbf{x} \in B\}$ is the result of translating the set B by the vector $\mathbf{p}$.

If A and B are in contact, $\rho(A, B) = 0$. Note that if the sets A and B are overlapping, the above $\rho(A, B) > 0$ measures the "depth" of the intersection of the two sets. To generalize this definition to overlapping sets, we need to distinguish this case from the case where A and B do not intersect. To do so, we use a signed version of the directional distance that has positive values if the two sets are non-overlapping and negative values otherwise.

Definition 1. *Given two convex sets A and B in $\mathbb{R}^n$, the signed directional distance between A and B is defined as*

$$\rho_s(A, B) = \begin{cases} \rho(A, B) & \text{if} A \cap B \neq \emptyset, \\ -\rho(A, B) & \text{otherwise.} \end{cases}$$

This directional distance in two and three dimensions has been used in robotics for collision detection [20]. The advantages of using the signed directional distance for the safety verification problem are the following:

- It measures the potential and the degree of collision between two moving objects, which, in the context of reachability computation, provides useful information on the necessity of refining reachable set approximations.
- For convex polyhedral sets, it can be estimated efficiently, as we will show in the sequel.

4.2 Signed Directional Distance Estimation and Refinement Algorithm

The signed directional distance between two convex polyhedra can be computed using existing algorithms (such as in [7]). However, these algorithms are specific for two and three dimensions and require complete polyhedral boundary descriptions (such as their vertices and facets). In high dimensions these descriptions are very expensive to compute, which will be discussed more in Section 5. We therefore focus on the problem of estimating the signed distance using only constraint descriptions of polyhedra. Our solution can be summarized as follows. The underlying idea is based on the relation between the signed directional distance $\rho_s(A, B)$ and the Minkowski difference $A \ominus B$. The latter is defined as follows:

$$A \ominus B = \{\mathbf{b} - \mathbf{a} \; | \mathbf{a} \in A \wedge \mathbf{b} \in B\}.$$

Intuitively, the Minkowski difference contains the translation directions that can bring A into contact with B. Its relation with the signed distance that we exploit is expressed by: $\rho_s(A, B) = \rho_s(\mathbf{0}, A \ominus B)$ where $\mathbf{0}$ is the singleton set that contains the origin of $\mathbb{R}^n$. Again, the vertex description of the Minkowski difference set can be expensive to compute, we resort to a constraint description of this set that can be efficiently computed. To this end, we consider a particular set of translation vectors corresponding to moving the half-space of a face e of A in the direction of its normal $\boldsymbol{\lambda}_e$ by a distance d_e so that the half-space touches B. Then, it can be proved that the half-space $H_e = \{\mathbf{x} \; : \; \langle \boldsymbol{\lambda}_e, \mathbf{x} \rangle \leq d_e\}$ contains at least one face of the exact Minkowski difference $A \ominus B$. Let $\mathcal{H}_e$ be the set of all such half-spaces H_e. This implies that $A \ominus B \subseteq Poly(\mathcal{H}_e) = \bigcap_{\forall e} \{\mathbf{x} \; : \; \langle \boldsymbol{\lambda}_e, \mathbf{x} \rangle \leq d_e\}$. In two or three dimensions, it is possible to obtain the exact constraint description of $A \ominus B$ by considering other translation types (such as an edge of A moves along an edge of B). Nevertheless, this computation requires the vertex descriptions of the polyhedra and we thus omit it.

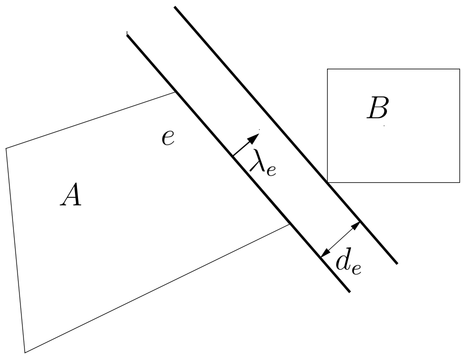


Fig. 3. Estimation of the directional distance

Since $Poly(\mathcal{H}_e)$ is an over-approximation of $A \ominus B$, it can be proved that

$$\begin{cases} \rho_s(\mathbf{0}, Poly(\mathcal{H}_e)) \leq \rho_s(A, B) & \text{if} A \cap B = \emptyset, \\ \rho_s(\mathbf{0}, Poly(\mathcal{H}_e)) \geq \rho_s(A, B) & \text{otherwise.} \end{cases}$$

The distance $\rho_s(\mathbf{0}, Poly(\mathcal{H}_e))$ can then be easily determined, since it is exactly the largest value of all d_e. We use $\rho_s(\mathbf{0}, Poly(\mathcal{H}_e))$ as an estimate of the signed directional distance between A and B.

It is important to note that this estimate is conservative regarding its utility as a critical situation alert. Indeed, if the two sets do not overlap, the result is smaller than the exact separating distance; otherwise, its absolute value is larger than the exact penetration distance. It is important to note again that the above estimation, which does not involve expensive vertex computation, is time-efficient.

In the context of safety verification, the set A plays the role of the reachable set and B the unsafe set. The constraints of A corresponding to the largest values of d_e are called *critical* because they are closest to B with respect to the directional distance measure. Their identification is part of the above computation of the signed directional distance between A and B.

Even if the angle between a critical constraint and one of its adjacent constraints does not satisfy the sharp angle criterion, we still refine around their intersection. The refinement can then be done using the same method for adjacent half-spaces with sharp angles, described in the previous section.

4.3 Refinement Using Constraints from the Safety Specification

Let $\Lambda_{\bar{\mathcal{B}}}$ be the set of normal vectors of the complement of each half-space of the unsafe set $\mathcal{B}$. In many cases, intersection of the reachable set and $\mathcal{B}$ can be tested more easily if the reachable set description contains a constraint whose normal vector coincides with a vector in $\Lambda_{\bar{\mathcal{B}}}$. Hence, for each direction $\boldsymbol{\lambda} \in \Lambda_{\bar{\mathcal{B}}}$, the predecessors of $\boldsymbol{\lambda}$ by the adjoint system can be used to define constraints to

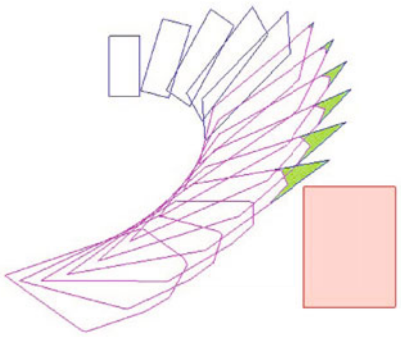


Fig. 4. Refinement using critical directions on a 2-dimensional example

add, again by solving the LP problem (7). The refinement using the predecessors of such directions is needed when the reachable set is close to the unsafe set. The refinement procedure using critical directions is summarized in Algorithm 2. In this algorithm, $\mathcal{H}^0$ is the set of constraints of the initial polyhedron $\mathcal{I}$. The function $dir(\mathcal{H}_c^k)$ returns the set of normal vectors of the constrains in $\mathcal{H}_c^k$. The function $AddConstraints(Poly(\mathcal{H}^{k-r}), \Lambda^{k-r})$ adds in $\mathcal{H}^{k-r}$ the new constraints with normal vectors in Λ^{k-r} that support $Poly(\mathcal{H}^{k-r})$.

Algorithm 2. Refinement Using Critical Directions

```
$\mathcal{H}^1 = \widehat{Post}_1(\mathcal{H}^0)$                                   /* One step computation */
$k = 1$
while $k \leq k_{max}$ do
    if $\rho_s(\mathcal{H}^k, \mathcal{B}) \leq \eta$ then
        $\mathcal{H}_c^k = criticalConstraints(\mathcal{H}^k, \mathcal{B})$   /* $H_c^k$ is the set of critical constraints */
                                                  /* $r \leq k$ is the number of backtracking steps */
        $\Lambda_c^{k-r} = Pre_r(dir(\mathcal{H}_c^k))$   /* Retrieve r-step predecessors of the normal vectors
        of $\mathcal{H}_c^k$
                                                  which were computed at the $(k-r)^{th}$ step */
        $\Lambda_b^{k-r} = Pre_r(\Lambda_{\bar{\mathcal{B}}})$              /* Predecessors of the normal vectors of $\mathcal{B}$ */
        $\Lambda^{k-r} = \Lambda_b^{k-r} \cup \Lambda_c^{k-r}$
        $\mathcal{H}^{k-r} = AddingConstraints(Poly(\mathcal{H}^{k-r}), \Lambda^{k-r})$
        $k = k - r$
    end if
    $\mathcal{H}^{k+1} = \widehat{Post}_1(\mathcal{H}^k)$                           /* One step computation */
    $k = k++$
end while
```

Figure 4 shows the result obtained for a 2-dimensional system in Jordan block form using the above algorithm. The rectangle on the right is the unsafe set. When the reachable set is close to the unsafe set, the algorithm backtracks a number of steps and adds new constraints. This refinement allows approximating more precisely the actual reachable set near the bad set and thus proving that the system does not enter the unsafe set. The colored zones are the parts of the over-approximation error eliminated by the added constraints. It can be seen from the figure that the aprroximation is refined only in the critical zones near the unsafe set.

5 Implementation and Experimental Results

We emphasize that in our development so far the algorithms use the constraint description and do not require the vertex description of polyhedra. Indeed, the transformation from a constraint description to a vertex description is known as vertex enumeration and the inverse transformation is known as facet enumeration. To show the computational complexity of these problems, we mention the algorithm published in [4] which finds m_v vertices of a polyhedron defined by a non-degenerate system of m inequalities in n dimensions (or, dually, the facets of the convex hull of m points in n dimensions, where each facet contains exactly n given points) in time $O(mnm_v)$ and $O(mn)$ space.

From our experience in using polyhedra for reachability computation for continuous and hybrid systems, we noticed that many operations (such as, the convex-hull) are extremely time-consuming, especially in high dimensions. Degeneracy of sets, such as flatness, which occurs frequently in reachable set computation, is also another important factor that limits the scalabibility of existing polyhedron-based algorithms. It is fair to say that they can handle relatively well systems of dimensions only up to 10. This therefore motivated a lot of research exploiting other set representations, which will be discussed later.

On the other hand, when trying to solve a specific verification problem, it is not always necessary to maintain both the vertex and the constraint descriptions of polyhedra. Indeed, for many tasks in a verification process, vertex enumeration can be avoided, such as in the algorithms we presented so far. We have implemented the above described algorithms of reachability computation with refinement for linear continuous systems and this implementation enabled us to handle continuous systems of dimensions much higher than what can be treated by typical polyhedron-based reachability analysis algorithms, such as [2].

In the following, we present some experimental results obtained using this implementation. To evaluate the performance of our methods, we generated a set of linear systems in Jordan block form in various dimensions up to 100 with the values in the diagonal are all equal to (-0.8). The input set $U = [-0.1, 0.1]^n$

Table 1. Computation time for 100 steps on some linear systems in Jordan block form using the implementation with vertex computation

dim n	Final number of added constraints	Computation time in seconds
2	32	0.4
5	59	9.14
10	10	150.93
20	–	–
50	–	–
100	–	–

Table 2. Computation time for 100 steps on the same linear systems in Jordan block form using the constraint-based implementation

dim n	Final number of added constraints	Computation time in seconds
2	32	0.48
5	59	0.76
10	36	2.22
20	38	3.67
50	94	42.07
100	197	335.95

and the initial set $\mathcal{I} = [-1, 1]^n$ are boxes (whose number of constraints equal to $2n$). The threshold for the sharp angle criterion is $\sigma = 60$ degrees.

To show the advantage of the constraint-based implementation, we also tested an implementation using vertex computation on the same examples. In this implementation with vertex computation, the constraint adjacency information can be directly derived and the constraints to add are easier to compute. However, the cost for vertex computation is high, which limited the application of this implementation to the examples of dimensions only up to 10, as shown in Table 1. For the example in 10 dimensions, we had to fix a smaller maximal number of constraints to add, in order to produce the result in a feasible computation time.

The constraint-based implementation is more time-efficient and thus allows us to handle systems of higher dimensions, as shown in Table 2.

6 Summary and Related Work

There have been other works on computing reachable sets of linear continuous systems. The treatment of uncertain inputs can be done by Minkowski sum (such as in the approaches using zonotopes [12,1], ellipsoids [17,5,18], or parallelotopes [15]). This can also be handled by optimization (such as in [16,6,22]). On the other hand, various set representations have been investigated. The classes of set representations with special shapes that have been used for reachability computations include oriented hyper-rectangles [23], zonotopes [10,12,1], ellipsoids [17,5,18], parallelotopes [15]. Compared to general convex polyhedra, their manipulation can be more efficient, but they are less appropriate for approximating sets with complex geometric forms.

To our knowledge, the most scalable algorithm is the recently developed zonotope-based algorithms [12,1]. The main advantage of zonotopes is that some important operations can be performed very efficiently over zonotopes, such as the Minkowski sum and linear transformations. This allows the zonotope-based algorithms to handle continuous systems of dimensions up to a few hundreds of variables. However, a major difficulty that comes with this set representation is that intersection of zonotopes is hard, which limits the extension of this approach to hybrid systems. Similarly, support functions [11] are a representation for general convex sets, on which the Minkowski sum and linear transformations can be efficiently computed. However, using support functions, set intersection is also difficult, which is an obstacle towards lifting the scalability of the associated algorithms to hybrid systems. In the context of hybrid systems verification, the main advantage of convex polyhedra, in our opinion, is their approximation power. In addition, using well-established techniques for linear programming solving and algorithmic geometry, polyhedral manipulation for specific tasks in verification can be optimized, which was demonstrated in this paper.

Concerning approximation refinement, our method is similar to the well-known counter-example based refinement approaches in the idea of guiding the refinement process using the previously explored behaviors. However, to the best of our knowledge, the idea of using redundant constraints for refinement purposes is new. Another novelty in our results is the use of the directional distance to measure approximation effectiveness in proving safety properties and to guide the refinement process.

The results of this paper open many promising research directions. The application of this method to hybrid systems is straighforward since the polyhedral operations for treating discrete dynamics can be computed using available algorithms. However, a challenge in this task is to go beyond the dimension limits of existing polyhedral computation algorithms, by exploiting the structure and the specificity of hybrid systems as well as of the verification problems. In addition, exploring the Minkowski difference between the reachable set and the unsafe set would allow a better measure of critical proximity of the reachable set under the dynamics of the system.

References

1. Althoff, M., Stursberg, O., Buss, M.: Reachability Analysis of Nonlinear Systems with Uncertain Parameters using Conservative Linearization. In: CDC 2008 (2008)
2. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 365–370. Springer, Heidelberg (2002)
3. Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise linear dynamical systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 21–31. Springer, Heidelberg (2000)
4. Avis, D., Fukuda, K.: A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. Discrete and Computational Geometry 8(1), 295–313 (1992)
5. Botchkarev, O., Tripakis, S.: Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 73–88. Springer, Heidelberg (2000)
6. Chutinan, A., Krogh, B.H.: Computational Techniques for Hybrid System Verification. IEEE Trans. on Automatic Control 48, 64–75 (2003)
7. Cameron, S.A., Culley, R.K.: Determining the Minimum Translational Distance between Two Convex Polyhedra. Proceedings of International Conference on Robotics and Automation 48, 591–596 (1986)
8. Dang, T.: Verification and Synthesis of Hybrid Systems. PhD Thesis, INPG (2000)
9. Frehse, G.: PHAVER: Algorithmic Verification of Hybrid Systems past HYTECH. International Journal on Software Tools for Technology Transfer (STTT) 10(3) (June 2008)
10. Girard, A.: Reachability of Uncertain Linear Systems using Zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)
11. Le Guernic, C., Girard, A.: Reachability Analysis of Hybrid Systems Using Support Functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 540–554. Springer, Heidelberg (2009)
12. Girard, A., Le Guernic, C., Maler, O.: Efficient Computation of Reachable Sets of Linear Time-invariant Systems with Inputs. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 257–271. Springer, Heidelberg (2006)
13. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A model checker for hybrid systems. Software Tools for Technology Transfer 1(1-2), 110–122 (1997)

14. Kalman, R.E., Falb, P.L., Arbib, M.A.: Topics in Mathematical System Theory. McGraw-Hill Book Company, New York (1968)
15. Kostoukova, E.K.: State Estimation for dynamic systems via parallelotopes: Optimization and Parallel Computations. Optimization Methods and Software 9, 269–306 (1999)
16. Kvasnica, M., Grieder, P., Baotic, M., Morari, M.: Multi-Parametric Toolbox (MPT). In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 448–462. Springer, Heidelberg (2004)
17. Kurzhanski, A., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 202–214. Springer, Heidelberg (2000)
18. Kurzhanskiy, A., Varaiya, P.: Ellipsoidal Techniques for Reachability Analysis of Discrete-time Linear Systems. IEEE Trans. Automatic Control 52, 26–38 (2007)
19. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Software Tools for Technology Transfert 1(1-2), 134–152 (1997)
20. Lin, M.C., Manocha, D.: Collision and proximity queries. In: Handbook of Discrete and Computational Geometry (2003)
21. Pappas, G., Lafferriere, G., Yovine, S.: A new class of decidable hybrid systems. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 29–31. Springer, Heidelberg (1999)
22. Sankaranarayanan, S., Dang, T., Ivancic, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008)
23. Stursberg, O., Krogh, B.H.: Efficient Representation and Computation of Reachable Sets for Hybrid Systems. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 482–497. Springer, Heidelberg (2003)
24. Varaiya, P.: Reach Set computation using Optimal Control. In: KIT Workshop, Verimag, Grenoble, pp. 377–383 (1998)
25. Yovine, S.: KRONOS: A verification tool for real-time systems. Software Tools for Technology Transfer 1(1-2), 123–133 (1997)

Methods for Knowledge Based Controlling of Distributed Systems

Saddek Bensalem[1], Marius Bozga[1], Susanne Graf[1], Doron Peled[2], and Sophie Quinton[1]

[1] Centre Equation - VERIMAG, 2 Avenue de Vignate, Gieres, France
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Controlling concurrent systems to impose some global invariant is an undecidable problem. One can gain decidability at the expense of reducing concurrency. Even under this flexible design assumption, the synthesis problem remains highly intractable. One practical method for designing controllers is based on checking knowledge properties upon which the processes can make their decisions whether to allow or block transitions. A major deficiency of this synthesis method lies in calculating the knowledge based on the system that we want to control, and not on the resulted system. The original system has less knowledge, and as a result, we may introduce far more synchronization than needed. In this paper we show techniques to reduce this overhead.

1 Introduction

Model checking has provided algorithms for the automatic analysis of systems. Techniques for automating the process of system design, in order to obtain correct-by-construction systems, have been recently studied as well. The synthesis problem from LTL specifications was shown by Pnueli and Rosner [12] to be 2EXPTIME hard for sequential reactive systems and undecidable for concurrent systems. A related problem is to control an already given system in order to force it to satisfy some additional property [14]. For distributed systems, this has also been shown to be undecidable [18,17]. Under the assumption that a system is flexible to the addition of further synchronization, the control problem becomes decidable. A solution based on model checking of knowledge properties was suggested [1,7].

In this paper, we look at the problem of reducing the need for additional synchronization in order to control distributed systems. We identify the main problem of the knowledge approach in using the controlled (source) system to calculate the knowledge. In fact, this is merely an approximation, as the actual knowledge needs to be satisfied by the (target) system after it is being controlled. After the control is applied, there are fewer executions, and fewer reachable states, hence the knowledge cannot decrease.

Our first observation is somewhat surprising: we prove that it is safe to calculate the knowledge based on the source system when considering for the analysis only the executions of the source system that satisfy the desired constraint. This provides a smaller set of executions and reachable states, hence also potentially more knowledge.

A second observation is that once we control a system according to its knowledge properties, we obtain again a system with fewer executions and reachable states: even if in the original system there are states where the system lacks the knowledge to continue, these states may, in fact, already be unreachable. Thus, one needs to make another round of checks on the obtained controlled system.

These two observations can be used in conjunction with other methods for constructing distributed controllers based on knowledge:

- Using knowledge of perfect recall (proposed in [1]).
- Adding coordinations to combine knowledge (proposed in [7]).

We show here that all these techniques are independent of each other, hence can be combined.

2 Preliminaries

The model used in this paper is Petri Nets. The method and algorithms developed here can equally apply to other models, e.g., transition systems and communicating automata.

Definition 1. *A* Petri Net *N is a tuple (P, T, E, s_0) where*

- *P is a finite set of* places*. The set of* states *(markings) is defined as $S = 2^P$.*
- *T is a finite set of* transitions*.*
- *$E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between the places and the transitions.*
- *$s_0 \subseteq 2^P$ is the* initial state *(initial marking).*

For a transition $t \in T$, we define the set of *input places* $^\bullet t$ as $\{p \in P | (p, t) \in E\}$, and *output places* $t^\bullet$ as $\{p \in P | (t, p) \in E\}$.

Definition 2. *A transition t is* enabled *in a state s if $^\bullet t \subseteq s$ and $t^\bullet \cap s = \emptyset$. We denote the fact that t is enabled from s by $s[t\rangle$.*

A state s is in *deadlock* if there is no enabled transition from it.

Definition 3. *A transition t can be* fired *(*executed*) from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s. Then, $s' = (s \setminus {}^\bullet t) \cup t^\bullet$.*

Definition 4. *Two transitions t_1 and t_2 are* independent *if $({}^\bullet t_1 \cup t_1{}^\bullet) \cap ({}^\bullet t_2 \cup t_2{}^\bullet) = \emptyset$. Let $I \subset T \times T$ be the* independence *relation. Two transitions are* dependent *if they are not independent.*

As usual, transitions are represented as lines, places as circles, and the relation E is represented by arrows from transitions to places and from places to transitions. We will use Petri Net N of Figure 1 as a running example. In N, there are places p_1, p_2, ..., p_5 and transitions a, b, c, d. We depict a state by putting full circles, called *tokens*, inside the places of this state. In the example of Figure 1, the depicted initial state s_0 is $\{p_1, p_4\}$. If we fire transition a from this initial state, the token from p_1 will be removed, and a token will be placed in p_2. The transitions enabled in s_0 are a and c. In this example, a and b are independent of c and d.

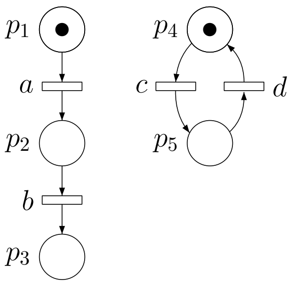


Fig. 1. A Petri Net N with priorities $a \ll \{c, d\} \ll b$

Definition 5. *An* execution *is a maximal (i.e. it cannot be extended) alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri Net, such that for each states s_i in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$.*

We denote the executions of a Petri Net N by $exec(N)$. The prefixes on the executions in a set X are denoted by $pref(X)$. A state is *reachable* in a Petri Net if it appears in at least one of its executions. We denote the reachable states of a Petri Net N by $reach(N)$. The reachable states of our running example N are $\{p_1, p_4\}$, $\{p_1, p_5\}$, $\{p_2, p_4\}$, $\{p_2, p_5\}$, $\{p_3, p_4\}$ and $\{p_3, p_5\}$.

We use places also as state predicates where $s \models p_i$ iff $p_i \in s$. This is extended to Boolean combinations on such predicates in a standard way. For a state s, we denote by φ_s the formula that is a conjunction of the places that are in s and the negated places that are not in s. Thus, φ_s is satisfied by state s and by no other state. For the Petri Net in Figure 1, the initial state s is characterized by $\varphi_s = p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4 \wedge \neg p_5$. For a set of states $Q \subseteq S$, we can write a *characteristic formula* $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or use any equivalent propositional formula. We say that a formula φ is an *invariant* of a Petri Net N if $s \models \varphi$ for each $s \in reach(N)$, i.e., if φ holds in every reachable state.

Definition 6. *A* process *of a Petri Net N is a subset of the transitions $\pi \subseteq T$ satisfying that for each $t_1, t_2 \in \pi$, such that $(t_1, t_2) \in I$, there is no reachable state s in which both t_1 and t_2 are enabled.*

We assume a given set of processes Π that covers all the transitions of the net, i.e., $\bigcup_{\pi \in \Pi} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes.

Definition 7. *The* neighborhood *$ngb(\pi)$ of a process π is the set of places $\bigcup_{t \in \pi} ({}^\bullet t \cup t^\bullet)$.*

We want to enforce global properties on Petri Nets in a distributed fashion. For a Petri Net N, we consider a property of the form $\Psi \subseteq S \times T$. That is, Ψ defines the allowed global states, and, furthermore, which transition may be fired in each allowed state. Note that as a special case Ψ can represent an *invariant* (when the transitions are not constrained).

Definition 8. *Let N be a Petri Net and $\Psi \subseteq S \times T$. We denote (N, Ψ) the pair made of N and the property Ψ that we want to enforce. A transition t of N is* enabled with respect to Ψ *in a state s if $s[t\rangle$ and, furthermore, $(s, t) \in \Psi$. An* execution *of (N, Ψ) is a maximal prefix $s_0t_1s_1t_2s_2t_3\ldots$ of an execution of N such that for each state s_i in the sequence, $(s_i, t_{i+1}) \in \Psi$. We denote the executions of (N, Ψ) by $exec(N, \Psi)$, and the set of states reachable in these executions by $reach(N, \Psi)$. We assume that those sets are nonempty.*

Clearly, $reach(N, \Psi) \subseteq reach(N)$ and $exec(N, \Psi) \subseteq pref(exec(N))$; recall that restricting N according to Ψ may introduce deadlocks.

In particular, we are interested in enforcing priority policies. Indeed, a priority order $\ll$ is a partial order relation among the transitions T of N and thus defines a set Ψ in a straightforward manner. We use priorities as a running example. If Ψ is defined by a priority order $\ll$, then $(s, t) \in \Psi$ if $s[t\rangle$ is enabled in s and has a maximal priority among the transitions enabled in s. That is, there is no other transition r with $s[r\rangle$ such that $t \ll r$. We write $exec(N, \ll)$ and $reach(N, \ll)$ instead of $exec(N, \Psi)$ and $reach(N, \Psi)$, respectively. Note that priority orders do not introduce new deadlocks, and thus we have $exec(N, \ll) \subseteq exec(N)$.

Let us now consider the prioritized Petri Net N of Figure 1. The executions of N, when the priorities are *not* taken into account, include those with finite prefixes (where states are abstracted away) *abcd*, *acbd*, *acdb*, *cadb*. However, when taking the priorities into account, the prioritized executions of N contain only alternations of c and d.

Definition 9. *The* local information *of a process π of a Petri Net N in a state s is $s|_\pi = s \cap nbg(\pi)$.*

That is, the local information of a process π in a given state consists of the restriction of the state to the neighborhood of π. It plays the role of a *local state* of π in s. Our definition of local information is only one among possible definitions that can be used for modeling the part of the state that the system is aware of at any given moment.

Definition 10. *Define an equivalence relation $\equiv_\pi \subseteq reach(N) \times reach(N)$ such that $s \equiv_\pi s'$ when $s|_\pi = s'|_\pi$.*

It is easy to see that the enabledness of a transition depends only on the local information of a process that contains it, i.e., if $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.

We cannot always make a local decision, based on the local information of processes (and sometimes sets of processes) that would guarantee that the global property Ψ is enforced. Indeed, in a prioritized Petri Net $(N, \ll)$, there may exist different states $s, s' \in reach(N)$ such that $s \equiv_\pi s'$, a transition $t \in \pi$ is an enabled transition in s with maximal priority, but in s' this transition is not maximal with respect to the priority order among the enabled transitions.

To reason about properties, we will use predicates. We can easily construct the following formulas, representing state sets, using only propositions representing places of the Petri Net:

– $\varphi_{reach(N)}$: all the reachable states of N. Similarly, $\varphi_{reach(N,\Psi)}$ denotes the reachable states of (N, Ψ).
– $\varphi_{en(t)}$: the states in which transition t is enabled.
– $\varphi_{\Psi(t)}$: the states s in which transition t is enabled and $(s,t) \in \Psi$. Formally: $\varphi_{\Psi(t)} = \varphi_{en(t)} \wedge \bigvee_{(s,t)\in\Psi} \varphi_s$
– φ^{Ψ}_{df}: the deadlock-free reachable states. That is, the states in which at least one transition is enabled w.r.t. Ψ, i.e., in which there is no deadlock in (N, Ψ). Formally: $\varphi^{\Psi}_{df} = \varphi_{reach(N,\Psi)} \wedge \bigvee_{t\in T} \varphi_{\Psi(t)}$.
– $\varphi_{s|_\pi}$: the states in which the local information of process π is $s|_\pi$.

For Ψ representing priority constraints, we denote $\varphi_{\Psi(t)}$ by $\varphi_{max(t)}$: the states in which transition t has a maximal priority among all the enabled transitions of the system. That is, $\varphi_{max(t)} = \varphi_{en(t)} \wedge \bigwedge_{t \ll r} \neg\varphi_{en(r)}$. We can perform model checking in order to calculate these formulas, and store them in a compact way, e.g., using BDDs.

3 Knowledge Based Approach for Distributed Control

In this section, we adapt the support policy introduced in [7] to Petri Nets.

3.1 The Support Policy

The problem we want to solve is the following:

> Given a Petri Net with a constraint (N, Ψ), we want to obtain a Petri Net N' such that *exec*$(N') \subseteq$ *exec*(N, Ψ). In particular, this means that *reach*(N') must not introduce deadlock states that are not already in *reach*(N, Ψ) or be empty. In this case, we say that N' *implements* (N, Ψ).

For simplicity of the transformation, we consider extended Petri Nets [5], where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 11. *An* extended Petri Net *has, in addition to the Petri Net components, for each process $\pi \in \Pi$ a finite set of variables V_π and (1) for each variable $v \in V_\pi$, an initial value v_0 (2) for each transition $t \in T$, an enabling condition en_t and a transformation predicate f_t on the variables $V_t = \cup_{\pi\in proc(t)} V_\pi$, where $proc(t)$ is the set of processes to which t belongs. In order to fire t, en_t must hold in addition to the usual Petri Net enabling condition on the input and output places of t. When t is executed, in addition to the usual changes to the tokens, the variables V_t are updated according to f_t.*

A Petri Net N' *extends* N if N' is an extended Petri Net obtained from N according to Definition 11. The comparison between the original Petri Net N and N' extending it is based only on places and transitions. That is, we ignore (project out) the additional variables.

Lemma 1. *For a Petri Net N' extending N, $exec(N') \subseteq pref(exec(N))$.*

Proof. The extended net N' only strengthens the enabling conditions and gives values to the added variables, thus it can only restrict the executions. However, these restrictions may result in new deadlocks. □

As we saw in the previous section, it is not possible in general to decide, based only on the local information of a process or a set of processes, whether some enabled transition is allowed by Ψ. We may, however, exploit some model checking based analysis of the system to identify the cases where such decision can be made.

Our approach for a local or semi-local decision on firing transitions is based on the knowledge of processes [4]. Basically, the knowledge of a process in a given (global) state is the set of reachable states that are consistent with the local information of that process.

Definition 12. *The process π* knows *a (Boolean) property φ in a state s, denoted $s \models K_\pi\varphi$, exactly when for each s' such that $s \equiv_\pi s'$, we have that $s' \models \varphi$.*

We obtain immediately from the definitions that if $s \models K_\pi\varphi$ and $s \equiv_\pi s'$, then $s' \models K_\pi\varphi$. Furthermore, the process π knows φ in state s exactly when $(\varphi_{reach(N)} \wedge \varphi_{s|_\pi}) \rightarrow \varphi$ is a tautology. Given a Petri Net and a Boolean property φ, one can perform model checking in order to decide whether $s \models K_\pi\varphi$. We have the following monotonicity property:

Theorem 1. *Let N be a Petri Net and N' an extension of N. If $s \models K_\pi\varphi$ in N, then $s \models K_\pi\varphi$ also in N'.*

Proof. The extended Petri Net N' restricts the executions, and possibly the set of reachable states, of N. Each local state $s|_\pi$ is part of fewer global states, and thus the knowledge in $s|_\pi$ can only increase. □

Monotonicity is important to ensure Ψ in N'. The knowledge allowing to enforce Ψ by the imposed transformation is calculated based on N, but is used to control the execution of the transitions of N'. Monotonicity thus ensures the correctness of N'.

We propose a support policy that consists in extending the original Petri Net with a disjunctive decentralized controller [19]. In general, a controller blocks some transitions in order to satisfy a given constraint. This is done by adding a supervisor process [14], which is usually an automaton that runs synchronously with the controlled system. Supervisors are often (finite state) automata observing the controlled system, progressing according to the transitions they observe, and blocking some of the enabled transitions depending on its current state. A *decentralized* controller sets up one supervisor per process. A *decentralized disjunctive* controller allows a transition to be fired if *at least one* of the supervisors supports it.

Our construction of such a disjunctive controller is based on a support table as introduced in [1,7] which indicates in each local state which transitions are supported. We do not formalize the details of the construction here, but the intuition provided here should be sufficient.

> In a state s, a transition t is *supported by a process π containing t* if and only if π knows in s (and thus in all s' which π cannot distinguish from s) about (s, t) respecting Ψ, i.e., $s \models K_\pi \varphi_{\Psi(t)}$; a transition can be fired (is enabled) in a state only if, in addition to its original enabledness condition, at least one of the processes containing it *supports* it.

To implement the support policy, we first create a *support table Δ* as follows: we check for each process π, reachable state $s \in \mathit{reach}(N)$ and transition $t \in \pi$, whether $s \models K_\pi \varphi_{\Psi(t)}$. If it holds, we put in the support table at the entry $s|_\pi$ the transitions t that are responsible for satisfying this property. In fact, as $s \models K_\pi \varphi$ and $s \equiv_\pi s'$ implies that $s' \models K_\pi \varphi$, it is sufficient to check this for a single representative state containing $s|_\pi$ out of each equivalence class of '$\equiv_\pi$'.

We construct for a Petri Net N a support table Δ and use it to control (restrict) the executions of N to satisfy the property Ψ. Each process π in N is equipped with the entries of this table of the form $s|_\pi$ for s a reachable state. Before firing a transition, a process π of N consults the entry $s|_\pi$ that corresponds to its current local information, and supports only the transitions that appear in that entry. This can be represented as an extended Petri Net N^Δ.

The construction of the support table is simple and its size is limited to the number of different local informations of the process and not to the (sometimes exponentially larger) size of the state space.

3.2 Solutions When the Support Policy Fails

Sometimes the knowledge based analysis does not provide an indication for a controller. Consider the prioritized Petri Net $(N, \ll)$ of Figure 1. The right process π_r, upon having a token in p_4, does not support c; the priorities dictate that c can be executed if b is not enabled, since c has a lower priority than b. But this information is not locally available to the right process, which cannot distinguish between the cases where the left process has a token in p_1, p_2 or p_3. To tackle this issue, several suggestions have been made:

1. Use knowledge of perfect recall [11,1]. This means that the knowledge is not based only on the local information, but also on the limited history that each process can observe. Although the history is not finitely bounded, it is enough to calculate the set of states where the rest of the system can reside at each point. A subset construction can be used to supply for each process an automaton that is updated according to the local history. This construction is very expensive: the size of this automaton can be exponential in the number of global states. Although in this way we extend our knowledge (by separating local informations according with different histories), this still does not guarantee that a distributed controller can be found.

2. Combine the knowledge of certain processes together by synchronizing them. The definition of knowledge can be based on equivalence classes of states that share the same local information of several processes. With the combined knowledge, one can achieve more situations where the maximal priority transition is known. However, to use this knowledge at runtime, these sets of processes need to be able to access their joint local information. This means synchronizing them, at the cost of losing concurrency. At the limit, all processes can be combined, and no actual concurrency remains.
3. Instead of the fixed synchronization between processes, one may use temporary synchronization [7]. Processes interact to achieve common knowledge. This does not reduce the concurrency as much as the previous method, but requires some overhead in sending messages to achieve the temporary synchronization.

In the following two sections, we propose two additional techniques, which are orthogonal to the previous ones, to handle the case where the support policy fails.

4 Support Policy Based on the Controlled System

The first technique is based on the following observation:

> Instead of calculating the knowledge with respect to *all* the executions of the original system, we may calculate it based on the executions of the original system that invariantly satisfy $\varPsi$.

The set of global states on these executions are a subset of the reachable states, and, furthermore, for each local information, the set of global states containing it is contained in the corresponding set of the original Petri Net. Thus, our knowledge in each global configuration may not decrease but possibly grows. Still, we need to show that calculating knowledge using this set of executions produces a correct controller.

Theorem 2. *Let N be a Petri Net and $\varPsi$ a property to be enforced. Let $\varDelta$ be the support table calculated for $reach(N, \varPsi)$, and let $N^{\varDelta}$ be the extended Petri Net constructed for $\varDelta$. Then $exec(N^{\varDelta}) \subseteq pref(exec(N, \varPsi))$.*

Proof. When a transition t of $N^{\varDelta}$ is supported in some state s according to the support table $\varDelta$, then for some supporting process $\pi \in \varPi$, $s \models K_\pi \varphi_{\varPsi(t)}$. By definition of the knowledge operator, this implies that $(s, t) \in \varPsi$. Thus, each firing of a transition of $N^{\varDelta}$ preserves $\varPsi$. However, it is possible that at some point, there is not enough knowledge to support any transition. □

The above proof does not guarantee that $N^{\varDelta}$ implements $(N, \varPsi)$: indeed, *reach* (N') may introduce deadlocks which are not in $reach(N, \varPsi)$ because in some states not enough knowledge is available to support transitions.

Let $\varphi_{support(\pi)}$ denote the disjunction of the formulas $\varphi_{s|_\pi}$ such that the entry $s|_\pi$ is nonempty in the support table. A sufficient condition for N^Δ to implement (N, Ψ) is:

$$\varphi_{df}^{\Psi} \rightarrow \bigvee_{\pi \in \Pi} \varphi_{support(\pi)} \tag{1}$$

This condition requires that for each state in *reach*(N, Ψ) that is not in deadlock, at least one transition is supported. When this condition does not hold, we say that the support table is *incomplete*.

For the prioritized Petri Net of Figure 1, the calculation of the knowledge based on the prioritized executions provides a controller, whereas the calculation based on the original (non prioritized) system did not. If we analyze the knowledge based on the constrained executions, then c and d are fired alternately, and p_2 is never reached, hence b is never enabled. In this case, our knowledge in p_4 and in p_5 allows us to execute c or d, respectively, and avoid the deadlock.

Let us look now at a more elaborate example. Consider Petri Net N_1 of Figure 2 with the given priority rules. The separation of transitions of N_1 according to processes is represented using dashed lines.

The example shows three processes π_1 (left), π_2 (in the middle), π_3 (right) that use binary synchronizations and priorities to enforce mutual exclusion for the execution of critical sections $(b_i r_i e_i)_{i\in[1,3]}$. Intuitively, priority rules $b_i \ll \{r_j, e_j\}$

$\forall i \in [1,3],\ b_i \ll \{r_j, e_j\}_{j\neq i}$
$\forall i \in [1,3],\ r_i \ll e_{j\neq i}$
$s_{23} \ll \{b_1, r_1, e_1\}$
$b_2 \ll b_3$

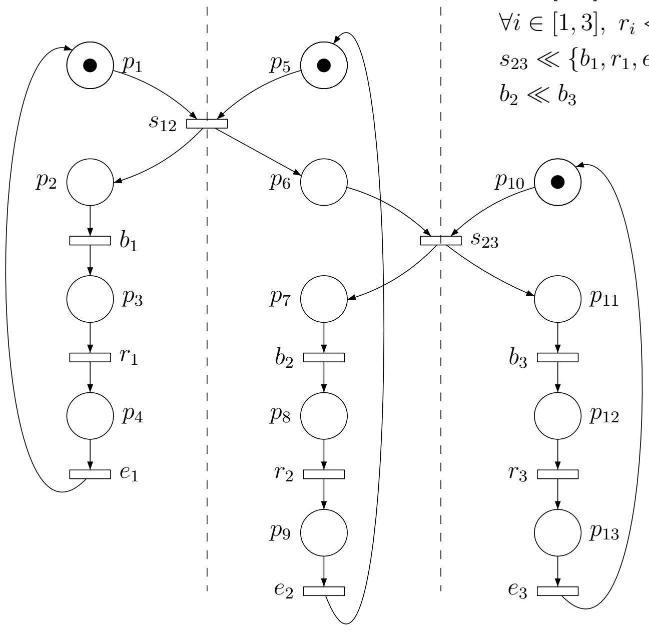


Fig. 2. A Petri Net N_1 with three processes π_1, π_2 and π_3

and $r_i \ll e_j$ give higher priority to transitions close to the end of the critical sections over the others. This enforces the mutual exclusion. Moreover, priority rules $s_{23} \ll \{b_1, r_1, e_1\}$ and $b_2 \ll b_3$ enforce a particular execution order of critical sections: repeatedly π_1 followed by π_3 and then by π_2.

Using the method of [1] described in section 3, no controller is found. Indeed, as all states are reachable, no process has enough knowledge to enter or progress in its critical section.

Now, if we calculate the support table on the prioritized executions, then we are able to construct a controller for N_1. Indeed, in the prioritized executions, there is always at most one process in its critical section. Thus, process π_1 always supports all its transitions as it can only enter the critical section in global states in which the other processes are blocked in front of a synchronization. Process π_3 supports all its transitions except s_{23}. Process π_2 supports transition s_{23} when π_1 is in p_1, transition b_2 when π_3 is in p_{10}, and transitions r_2 and e_2 in all cases.

5 Controllers Based on an Incomplete Support Table

In this section we show that even an incomplete support table Δ for (N, Ψ) may still define a controller N^{Δ} that implements (N, Ψ). The reason is that states that are reachable in the executions of (N, Ψ) may be unreachable when applying the calculated support table. The executions according to the support table may be a subset of the executions of (N, Ψ), and the problematic states may not occur.

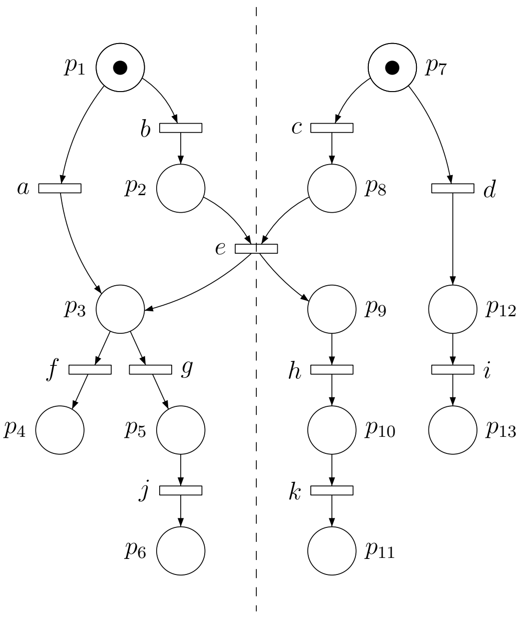
We illustrate this now on an example with priorities. Consider Petri Net N_2 of Figure 3. It represents two processes π_l (left) and π_r (right) with a single joint transition, which means that π_l can observe whether π_r is in one of the places p_8 and p_9. Similarly, π_l can observe whether π_r is in p_2 or in p_3. The table in Figure 3 shows the set of reachable states of N_2, including its termination (deadlock) states, denoted $\blacksquare$. Non-reachable states are in grey.

Suppose that the following set of priority rules must be enforced for the Petri Net N_2: $k \ll j$ and $c \ll b \ll i$.

The support table is calculated based on the knowledge of the original system. Table 1 presents a view of the global states of the Petri Net.

– non-reachable (grey), or
– in termination or deadlock ($\blacksquare$) or
– reachable and non-deadlock.

In the latter case, the cell contains the transitions which are supported in this state by any of the processes (i.e., we have accumulated all the transitions supported by the local states that constitute together the global state). The blanks in this incomplete table represent states in which no process supports any transition. There are two such states, namely $\{p_4, p_{10}\}$ and $\{p_6, p_{10}\}$. The situation in both states is the following: π_l has terminated and π_r could take transition k, but without an additional synchronization, there is no way for π_r to know that it may safely execute k.



$\pi_l = \{a, b, e, f, g, j\}$
$ngb(\pi_l) = \{p_1, \dots, p_6, p_8, p_9\}$

$\pi_r = \{c, d, e, h, i, k\}$
$ngb(\pi_r) = \{p_2, p_3, p_7, \dots, p_{13}\}$

$reach(N_2) =$

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1			▨	▨	▨		
p_2			▨	▨	▨		■
p_3							
p_4		■			■		■
p_5							
p_6		■			■		■

Fig. 3. A Petri Net N_2 with two processes π_l and π_r

Table 1. Support policy for N_2 with priorities $k \ll j$ and $c \ll b \ll i$

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d	▨	▨	▨	▨	a, i	a
p_2	c, d	e	▨	▨	▨	i	■
p_3	f, g, c, d	f, g	f, g, h	f, g, k	f, g	f, g, i	f, g
p_4	d	■	h		■	i	■
p_5	j, d	j	j, h	j	j	j, i	j
p_6	d	■	h		■	i	■

Note that in state $\{p_1, p_7\}$, process π_l supports a and process π_r supports d; it is impossible for π_l to know whether π_r is in p_{12} or not, and therefore b (which has lower priority than i) is not supported by π_l. Similarly, π_r does not support c (which has lower priority than b). While c is supported, e.g., in $\{p_2, p_7\}$, transition b is never supported, hence never fired in N_2^{Δ}, although it is allowed according to the priority rules in some states.

As a consequence, the set of states reachable in N_2^{Δ} may be smaller than $reach(N_2, \ll)$. Indeed, $reach(N_2^{\Delta})$ does not contain any state including the places p_2 together with p_9, p_{10} or p_{11}. This means in particular that the blanks in Table 1 are in fact not reachable, and thus N_2^{Δ} implements $(N_2, \ll)$.

6 Comparison with History-Based Controllers

We show now that the use of perfect recall is independent of the methods proposed in this paper, meaning that in some cases only history is able to provide a controller, while in others it is still relevant to check whether an incomplete table provides a controller.

Consider the Petri Net N_2 of Figure 3, this time with the priorities $g \ll k$ and $f \ll i$. In this case, the set of reachable states is the same, regardless of the use or not of priorities. Consequently, there is no difference between the support policy based on the unrestricted system and the prioritized executions. Moreover, this support policy fails because there are two reachable global states where no process is supporting a transition, appearing as blanks in Table 2: $\{p_3, p_{11}\}$ and $\{p_3, p_{13}\}$. Furthermore, these global states are also reachable in the controlled system, meaning that the heuristics applied in the previous example does not help either.

Nevertheless, this example may be controlled if perfect recall is used. If the left process π_l can remember the path it takes to reach p_3, it can distinguish between reaching p_3 directly after p_1 (by firing a) or respectively by passing through place p_2. Now, the set of reachable states contains enough information for the support policy to succeed.

Table 2. Support policy for N_2 with priorities $g \ll k$ and $f \ll i$ without history

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, b, c, d	a, b	▨	▨	▨	a, b, i	a, b
p_2	c, d	e	▨	▨	▨	i	■
p_3	c, d	f, g	f, g, h	k		i	
p_4	c, d	■	h	k	■	i	■
p_5	j, c, d	j	j, h	j, k	j	j, i	j
p_6	c, d	■	h	k	■	i	■

Our last example illustrates the combined use of perfect recall and an incomplete table to build a controller. Consider again the Petri Net N_2 of Figure 3, now with priorities $g \ll k$, $f \ll \{i, k\}$ and $c \ll b \ll i$. On one hand, building the support table using the prioritized executions does not provide enough knowledge to control the system, and the incomplete support table does not provide a controller. On the other hand, the use of history as shown previously does not help either. Table 3 reflects the incomplete support table constructed using jointly the prioritized executions and perfect recall. Additional information related to perfect recall is presented in the rows and columns of the table only when it is relevant for the support table. We can observe that in $\{p_3 \text{ after } p_2, p_{11}\}$, no transition is supported by any process. However, the system can be controlled according to this table. Indeed, no extra deadlock (blank) is actually reachable within the controlled system for a reason similar to one presented in the example of Section 5. This means that only the combination of several techniques leads to a controller.

Table 3. Support policy for N_2 with $g \ll k$, $f \ll \{i, k\}$, $c \ll b \ll i$ and history

	p_7	p_7 after p_3	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d	▨	▨	▨	▨	▨	a, i	a
p_2	c, d	▨	e	▨	▨	▨	i	■
p_3 after p_1	▨	c, d, g	f, g	▨	▨	▨	g, i	g
p_3 after p_2	▨	▨	▨	f, g, h	k		▨	▨
p_4	▨	c, d	■	h	k	■	i	■
p_5	▨	j, c, d	j	j, h	j, k	j	j, i	j
p_6	▨	c, d	■	h	k	■	i	■

7 Implementation and Experimental Results

In [7], we implemented a prototype for experimenting with knowledge based controlling of distributed systems. We have integrated the two results of this paper. More precisely, the support table is now built directly from the set of reachable states in the prioritized executions. Then, if the table contains empty entries, we check the reachability of the states in which no transition can be supported before adding synchronization.

We present here some results illustrating the improvements thus obtained. Let us go back to the Petri Net of Figure 2. Using the approach of [7], several additional synchronizations were added in order to check maximality of transitions in the critical section. Out of 80 reachable states, 26 are global states in which no process can support an interaction. More precisely, 4 transitions out of 11 always require a synchronization to be fired: these transitions are b_1, r_1, b_3, r_3. As a result, an execution of 10,000 steps contains exactly 3636 ($= 10000 \times 4/11$) synchronizations. Using the prioritized executions to build the support table, we do not need any synchronization to build a controller.

Consider now a simplification of this example with two processes instead of three. In this case, interestingly, the method of [7] would not result in the execution of any additional synchronization. The reason for this is that the states requiring additional synchronizations are exactly the states in which no transition can be supported, meaning that synchronizations are added only when they are necessary. As these states are unreachable in the prioritized executions, no synchronization ever takes place. This emphasizes the fact that both approaches can be combined efficiently.

8 Conclusion

Calculating knowledge properties can be used for constructing a distributed controller that imposes some state property Ψ on a system. A transformation is used to impose a global property Ψ invariantly. To maintain the concurrent nature of the system, the decision on which transitions to support needs to be made locally. Checking whether one can impose such a transformation without adding interaction is undecidable [18,17,7]. Knowledge can be used to help in constructing such a controller. When a process locally knows that executing a transition will satisfy Ψ, then it is safe to support it. By combining the knowledge of different processes [7], if we are allowed to add synchronization, the synthesis becomes decidable, since at the limit, we may obtain a fully synchronized, i.e., a global system. Now, adding extensive synchronizations is undesirable.

We observe here that the knowledge approach for constructing a distributed controller is based on analyzing the original system in order to achieve the invariance of Ψ after the transformation. Thus, the use of knowledge calculated on the original version may be pessimistic in concluding when transitions can be supported. This brings us to two useful observations that can remove the need for some of the additional interactions used to control the system:

1. Although the analysis of the knowledge of the system is done with the original system, it is safe to use only its executions that enforce Ψ. This gives fewer executions and fewer reachable states and enhances the knowledge.
2. Blocking transitions (not supporting them) because of lack of knowledge has a propagating effect that can prevent reaching other states. Thus, even when the result of the knowledge analysis may seem to lack the ability of supporting a way to continue from some states, this may not be the case. Indeed, analyzing the system after imposing the restriction forced by the analysis may result in a system that does not introduce new deadlock: the deadlocks appearing in states where no enabled transition is supported are in fact unreachable.

In this paper, we showed that using these two observations is orthogonal to other tools used to force knowledge based control such as using knowledge of perfect recall and adding temporary interactions or fixed synchronizations between processes. We propose to use these knowledge based techniques as a practical way of synthesizing distributed controllers.

References

1. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. 5643, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
2. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
3. Emerson, E.A., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs using Fixpoints. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980)
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
5. Genrich, H.J., Lautenbauch, K.: System Modeling with High-level Petri Nets. Theoretical Computer Science 13, 109–135 (1981)

6. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 443–466. Springer, Heidelberg (2004)
7. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control Through Model Checking. In: CAV 2010. Springer, Heidelberg (2010)
8. Halpern, J.Y., Zuck, L.D.: A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols. Journal of the ACM 39(3), 449–478 (1992)
9. Hoare, C.A.R.: Communicating Sequential Processes. Communication of the ACM 21, 666–677 (1978)
10. Manna, Z., Pnueli, A.: How to Cook a Temporal Proof System for Your Pet Language. In: POPL 1983, Austin, TX, pp. 141–154 (1983)
11. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140, 115–157 (1980)
12. Pnueli, A., Rosner, R.: Distributed Reactive Systems Are Hard to Synthesize. In: FOCS 1990, St. Louis, USA, vol. II, pp. 746–757. IEEE, Los Alamitos (1990)
13. Queille, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: 5th International Symposium on Programming, pp. 337–350 (1981)
14. Ramadge, P.J., Wonham, W.M.: Supervisory Control of a Class of Discrete-Event Processes. SIAM Journal on Control and Optimization 25(1), 206–230 (1987)
15. Rudie, K., Laurie Ricker, S.: Know Means No: Incorporating Knowledge into Discrete-Event Control Systems. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
16. Rudie, K., Murray Wonham, W.: Think Globally, Act Locally: Decentralized Supervisory Control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
17. Thistle, J.G.: Undecidability in Decentralized Supervision. System and Control Letters 54, 503–509 (2005)
18. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters 90(1), 21–28 (2004)
19. Yoo, T.S., Lafortune, S.: A General Architecture for Decentralized Supervisory Control of Discrete-Event systems. Discrete Event Dynamic Systems, Theory & Applications 12(3), 335–377 (2002)

Composing Reachability Analyses of Hybrid Systems for Safety and Stability*

Sergiy Bogomolov, Corina Mitrohin, and Andreas Podelski

University of Freiburg
Department of Computer Science
Freiburg, Germany
{bogom,mitrohin,podelski}@informatik.uni-freiburg.de

Abstract. We present a method to enhance the power of a given reachability analysis engine for hybrid systems. The method works by a new form of composition of reachability analyses, each on a different relaxation of the input hybrid system. We present preliminary experiments that indicate its practical potential for checking safety and stability.

1 Introduction

A standard technique in programming languages is to improve the precision of the analysis of a program by preceding it with an auxiliary analysis. The auxiliary analysis is in general used to infer the validity of an invariant at a specific program location. For example, an interval analysis is used to infer a lower and an upper bound on the possible values of a program variable at a given program location, which means that an assertion of the form $l \leq x \leq u$ is a valid invariant for the program location. It is hence safe to use this invariant for refining the abstract program used for a subsequent program analysis. A simple way to refine the abstract program is to insert an `assume` statement (e.g., `assume(l<= x <= u)`).

Unfortunately, it is not clear how one can directly transfer this technique from programs to hybrid systems. In a hybrid system, where the value of a variable changes not only through updates in transitions from one location to another but evolves dynamically in one location, an invariant in the form of a state assertion associated with a program location does not seem useful for improving a subsequent analysis. For one thing, an invariant must account for a continuum of states. For another, a state assertion does not seem suitable to describe the interdependence between the possible values of continuous variables x and y whose evolution is prescribed by differential equations.

In essence, the problem is to incorporate the result of the reachability analysis of one abstraction into the reachability analysis of another abstraction of a given

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center Automatic Verification and Analysis of Complex Systems (SFB/TR 14 AVACS). See www.avacs.org for more information.

hybrid system. In this paper, we propose to base the solution of this problem on the notion of *dwell time*. The concept of dwell time has already shown its usefulness for the analysis of hybrid systems (see our discussion of related work further below). It refers to the amount of time which the hybrid system can resp. must spend ('dwell') in a location between an entry and an exit. The mutual interdependence between the possible values of continuous variables x and y can be approximated via bounds on the the dwell time, i.e., bounds on the possible values of x propagate to bounds on the dwell time which again propagate to bounds on the possible values of y. We propose to use an auxiliary 'dwell time variable' d for interfacing a reachability analysis with an auxiliary analysis. We encode the result of the auxiliary analysis by bounds on the possible values of d in reachable states of an (in general coarse) abstraction of the hybrid system. We use these bounds to improve the precision of the reachability analysis of another (in some sense, complementary) abstraction as follows. We add the dwell time variable d as a continuous variable (in addition to other continuous variables, in case the abstraction is expressed by a hybrid system), and we refine the abstraction by constraints on d.

In summary, we present new methods both for inferring and for exploiting dwell time bounds. As a consequence, we obtain an approach where one can incorporate the result of a first, auxiliary reachability analysis into a second reachability analysis in order to refine its abstraction and thus improve its precision. Our approach is generic in that it uses reachability analysis as a black-box method. We present initial experiments that indicate its practical potential for checking safety and stability.

2 Hybrid Systems, Relaxation, Refinement

A *hybrid system* is formally a tuple $\mathcal{H} = (Loc, V, Init, R^{\mathsf{cont}}, R^{\mathsf{disc}}, Inv)$ defining

- the finite set of locations Loc,
- the set of continuous variables $V = \{v_1, \ldots, v_n\}$,
- the initial condition, given by the constraint $Init(\ell)$ for each location ℓ,
- the continuous transition relation, given by the expression $e = R^{\mathsf{cont}}(\ell)(v)$ for each continuous variable v and each location ℓ; the expression e (in the variables $v_1, \ldots, v_n$) is used in the differential equation $\dot{v} = e$ that defines the flow of the continuous variable v in the location ℓ,
- the discrete transition relation, given by a set R^{disc} of transitions; a transition is formally a tuple (ℓ, g, ξ, ℓ') defining
  - the source location ℓ and the target location ℓ',
  - the guard, given by a constraint g,
  - the update, given by a (possibly empty) set ξ of assignments $v := e$ of expressions to continuous variables,
- the invariant, given by the constraint $Inv(\ell)$ for each location ℓ.

An example of a hybrid system is given in Figure 1.

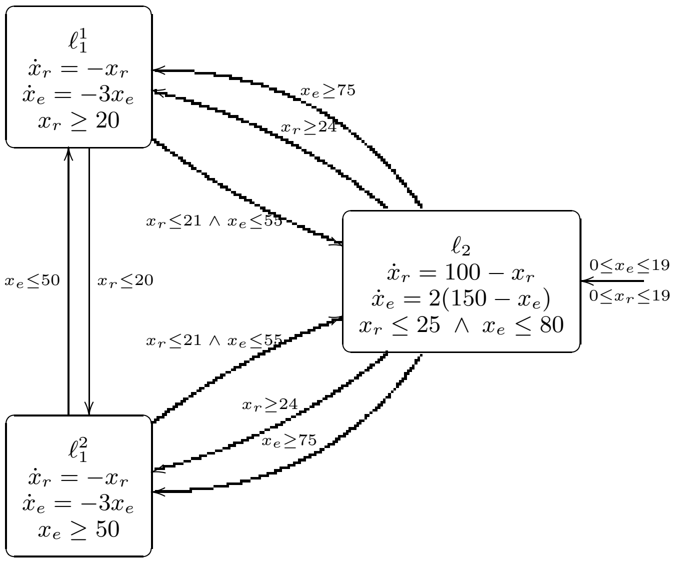


Fig. 1. Example hybrid system, modeling a temperature controller with one internal engine. The temperature of the plant is controlled through a thermostat which turns the engine off and on, depending on the temperature of the room (modeled by the continuous variable x_r) and the temperature of the engine (modeled by the continuous variable x_e). We model the mode 'off' by the two locations ℓ_1^1 and ℓ_1^2, and the mode 'on' by the location ℓ_2.

A *state* of the hybrid system $\mathcal{H}$ is a tuple $(\ell, v_1, \dots, v_n)$ consisting of a location ℓ in *Loc* and values of the continuous variables in V.

The hybrid system can be represented by a labeled graph (as in Figure 1), where the set of nodes is the set of locations *Loc*, and the set of edges is defined by the discrete transition relation, i.e.,

$$E = \{(\ell, \ell') \mid \exists(\ell, g, \xi, \ell') \in R^{\mathsf{disc}}\}.$$

We now give the formal definition of the semantics of a hybrid system, in the form of the set of its runs. We write $\mathcal{T}$ for the set of all continuous time points which are denoted by non-negative real values, i.e., $\mathcal{T} = \mathbb{R}_0^+$. A run ρ assigns to every time point t in $\mathcal{T}$ a location and a valuation of the variables v in V. Formally, a run ρ is a tuple

$$\rho = (\hat{\ell}, \hat{v}_1, \dots, \hat{v}_n)$$

of functions $\hat{\ell} : \mathcal{T} \rightarrow Loc$ (for the current location) and $\hat{v} : \mathcal{T} \rightarrow \mathbb{R}$ (for the current value of the variable v in V), such that there exists an infinite sequence of switching time points,

$$(\tau_i)_{i \in \omega} \in \mathcal{T}^\omega$$

which starts in 0 and is strictly increasing, i.e., $\tau_0 = 0$ and $\tau_i < \tau_{i+1}$, such that the following five conditions hold:

- "non-zenoness"
$$\forall t \in \mathcal{T}\ \exists i :\ t \leq \tau_i \tag{1}$$
- "switching time"
$$\forall i\ \forall t \in [\tau_i, \tau_{i+1}) : \hat{\ell}(t) = \hat{\ell}(\tau_i) \tag{2}$$
- "continuous evolution"
$$\forall v \in V\ \forall i\ \forall t \in [\tau_i, \tau_{i+1}) : \quad \frac{d}{dt}\hat{v}(t) = e[\hat{v}_1, \ldots, \hat{v}_n](t) \tag{3}$$
  where $e = R^{\mathsf{cont}}(\ell)(v)$ with $\ell = \hat{\ell}(\tau_i)$,
- "invariants"
$$\forall i\ \forall t \in [\tau_i, \tau_{i+1}) :\ (\hat{v}_1(t), \ldots, \hat{v}_n(t)) \models \mathit{Inv}(\ell) \tag{4}$$
  where $\ell = \hat{\ell}(\tau_i)$,
- "discrete transition firing"
$$\begin{aligned} &\forall i\ \exists\ (\ell, g, \xi, \ell') \in R^{\mathsf{disc}} : \\ &\qquad \hat{\ell}\ (\tau_i) = \ell \\ &\qquad \hat{\ell}\ (\tau_{i+1}) = \ell' \\ &\qquad \exists\ \sigma : V \to \mathbb{R}\ \forall v \in V : \\ &\qquad\qquad \sigma(v) = \lim_{u \to \tau_{i+1}} \hat{v}(u) \\ &\qquad\qquad \sigma \models g \\ &\qquad\qquad \hat{v}(\tau_{i+1}) = \begin{cases} \sigma(e), & \text{if } v := e \in \xi \\ \sigma(v), & \text{otherwise.} \end{cases} \end{aligned} \tag{5}$$

Condition (1) states that we do not allow Zeno behavior. The time sequence $(\tau_i)_{i\in\omega}$ identifies the time points where location switches may occur, which is expressed in Condition (2). Only at those points discrete transitions may be taken. Condition (3) expresses that the dynamics of the continuous variables obeys their respective differential equations. Condition (4) requires that for each location the valuation of continuous variables satisfies the local invariant while staying in that location. Condition (5) expresses that whenever a discrete transition is taken, variables may be assigned new values, obtained by evaluating the right-hand side of the respective assignment using the previous values of variables. If there is no such assignment, the variable maintains its previous value, which is determined by taking the limit of the trajectory of the variable as t converges to the switching time τ_{i+1}.

Definition 1 (Relaxation, Refinement). *A hybrid system $\mathcal{H}^{\#}$ is a* relaxation *of $\mathcal{H}$, written $\mathcal{H}^{\#} \sqsupseteq \mathcal{H}$, if it has more runs than $\mathcal{H}$, i.e.,*

$$\mathrm{Run}(\mathcal{H}^{\#}) \supseteq \mathrm{Run}(\mathcal{H}).$$

The system $\mathcal{H}$ is called, in turn, a refinement *of $\mathcal{H}^{\#}$.*

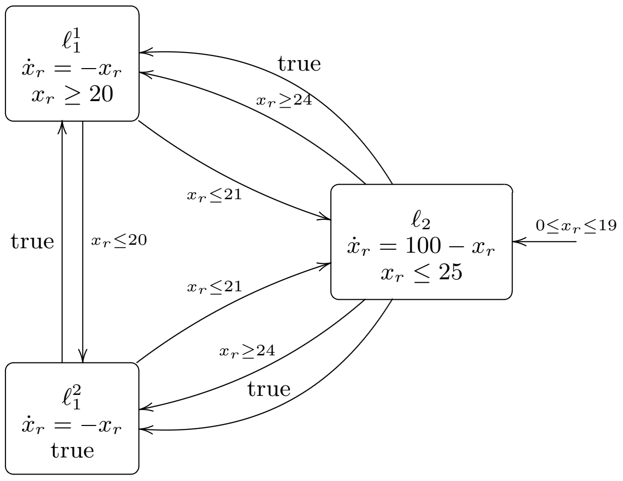


Fig. 2. The relaxation of the heating system from Figure 1 obtained by eliminating the continuous variable x_e

By definition, each (reachable) location ℓ and edge (ℓ, ℓ') of $\mathcal{H}$ belongs also to $\mathcal{H}^{\#}$. There are several ways to construct a relaxation $\mathcal{H}^{\#}$ from $\mathcal{H}$, for example, by projection or by weakening of constraints; see, e.g. [8]. Figure 2 presents the relaxation of the heating system from Figure 1 obtained by eliminating the continuous variable x_e.

3 Dwell Time Bounds

A family of *dwell time bounds* $dtb = (dtb_{\mathsf{low}}, dtb_{\mathsf{high}})$ for a hybrid system $\mathcal{H}$ is given by a constant

$$dtb_{\mathsf{low}}(\ell, \ell') = c_{(\ell,\ell')}$$

for each transition from the location ℓ to the location ℓ', and a constant

$$dtb_{\mathsf{high}}(\ell) = c_{\ell}$$

for each location ℓ. (Here, we index a dwell time guard by a pair of locations, and not by the transition. We assume wlog. that a transition is unique for its pair of source and target locations.)

Definition 2 (Validity of dwell time bounds). *The dwell time bound $dtb_{\mathsf{low}}(\ell, \ell') = c_{(\ell,\ell')}$ for the transition from the location ℓ to the location ℓ' is valid if in every run of $\mathcal{H}$, the time spent between the entry of the location ℓ and its exit towards the location ℓ' is always greater than or equal to $c_{(\ell,\ell')}$.*

The dwell time bound $dtb_{\mathsf{high}}(\ell) = c_{\ell}$ for the location ℓ is valid if in every run of $\mathcal{H}$, the time spent between the entry and the exit of the location ℓ is always smaller than or equal to c_{ℓ}.

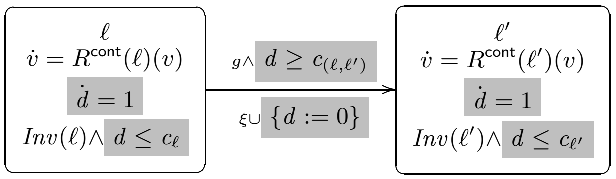

ℓ
$\dot{v} = R^{\mathsf{cont}}(\ell)(v)$
$\dot{d} = 1$
$Inv(\ell) \wedge d \leq c_\ell$
$g \wedge d \geq c_{(\ell,\ell')}$
$\xi \cup \{d := 0\}$
ℓ'
$\dot{v} = R^{\mathsf{cont}}(\ell')(v)$
$\dot{d} = 1$
$Inv(\ell') \wedge d \leq c_{\ell'}$

Fig. 3. $\mathsf{DTR}(\mathcal{H}, dtb)$, the dwell time refinement of $\mathcal{H}$ for a given family of dwell time bounds dtb valid for $\mathcal{H}$, is obtained from $\mathcal{H}$ by a transformation that adds the highlighted items; among those, the dwell time constraints $d \leq c_\ell$ and $d \geq c_{(\ell,\ell')}$

A family of dwell time bounds $dtb = (dtb_{\mathsf{low}}, dtb_{\mathsf{high}})$ *for a given hybrid system* $\mathcal{H}$ *is valid if each of its dwell time bounds are valid.*

That is, the valid dwell time bound $dtb_{\mathsf{low}}(\ell, \ell')$ for the transition from the location ℓ to the location ℓ' is a lower bound for the time spent in the location ℓ, or, equivalently, for the length of the interval $[\tau_i, \tau_{i+1}]$ formed by two consecutive switching time points for a discrete transition into ℓ and a discrete transition from ℓ to ℓ'.

Also, the valid dwell time bound $dtb_{\mathsf{high}}(\ell)$ is an upper bound for the time spent in the location ℓ, or, equivalently, for the length of the interval $[\tau_i, \tau_{i+1}]$ formed by two consecutive switching time points for a discrete transition into ℓ respectively away from ℓ.

4 Dwell Time Refinement

Dwell Time Refinement for $\mathcal{H}$ *(*$\mathsf{DTR}(\mathcal{H}, dtb)$*).* Given a family $dtb = (dtb_{\mathsf{low}}, dtb_{\mathsf{high}})$ of dwell time bounds valid for $\mathcal{H}$, we construct a new hybrid system $\mathsf{DTR}(\mathcal{H}, dtb)$ informally as follows (see also Figure 3).

– We add a continuous variable d.
– In each location, we set the slope of d to 1 (i.e., d evolves like a clock).
– In each discrete transition, the variable d is reset to 0 (i.e., we add the assignment $d := 0$ to update of the transition).
– We add $d \geq dtb_{\mathsf{low}}(\ell, \ell')$ as a conjunct to the guard of the transition from the location ℓ to the location ℓ' (we call this conjunct the *dwell time guard*).
– We add $d \leq dtb_{\mathsf{high}}(\ell)$ as a conjunct to the invariant of the location ℓ (we call this conjunct the *dwell time invariant*).

Formally, we define $\mathsf{DTR}(\mathcal{H}, dtb)$ as the hybrid system

$$\mathsf{DTR}(\mathcal{H}, dtb) = (Loc, V_d, Init_d, \mathsf{R}_d^{\mathsf{cont}}, \mathsf{R}_d^{\mathsf{disc}}, Inv_d)$$

where

– the set of locations Loc is the same as in $\mathcal{H}$,

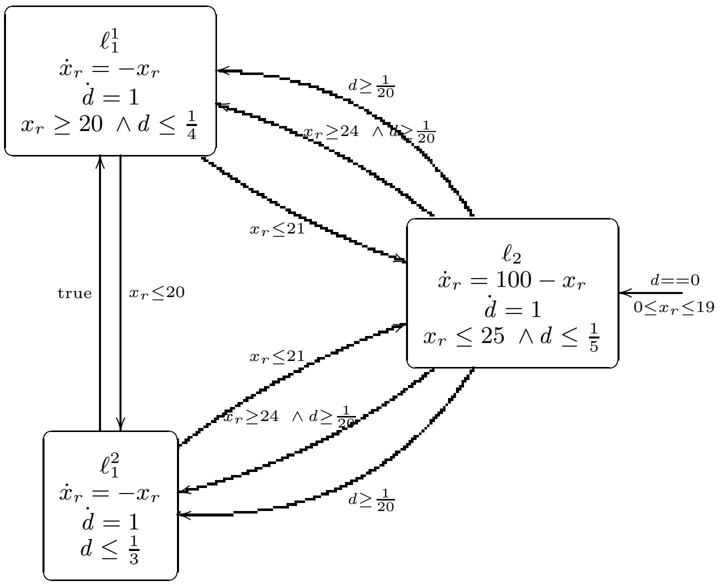


Fig. 4. $\mathsf{DTR}(\mathcal{H}^\#, \mathit{dtb})$, the dwell time refinement of the relaxation $\mathcal{H}^\#$ in Figure 2 for a given family of dwell time bounds dtb valid for $\mathcal{H}$, where the dwell time bounds dtb are the ones inferred automatically by the method given in Section 5

– the set of continuous variables V_d contains the new variable d and all variables of $\mathcal{H}$,
$$V_d = \{d\} \cup V$$
– the initial condition Init_d fixes the value of the new variable d to 0 in each location,
$$\mathit{Init}_d(\ell) \equiv \mathit{Init}(\ell) \wedge (d = 0)$$
– the continuous transition relation R_d^{cont} assigns constant 1 to the derivative of the new variable,
$$\begin{aligned} R_d^{\mathsf{cont}}(\ell)(d) &= 1 \\ R_d^{\mathsf{cont}}(\ell)(v) &= R^{\mathsf{cont}}(\ell)(v) \quad \text{for } v \in V \end{aligned}$$
– the discrete transition relation R_d^{disc} adds the assignment $d := 0$ to the update of each transition, and it adds the dwell time guard $d \geq \mathit{dtb}_{\mathsf{low}}(\ell, \ell')$ as a conjunct to the guard of the transition from ℓ to ℓ',
$$(\ell,\ g \wedge d \geq \mathit{dtb}_{\mathsf{low}}(\ell, \ell')\ ,\ \xi \cup \{d := 0\}\ ,\ \ell') \in R_d^{\mathsf{disc}} \quad \text{if} \quad (\ell, g, \xi, \ell') \in R^{\mathsf{disc}}$$
– the invariant Inv_d adds the dwell time invariant $d \leq \mathit{dtb}_{\mathsf{high}}(\ell)$ as a conjunct to the invariant of the location ℓ,
$$\mathit{Inv}_d(\ell) \equiv \mathit{Inv}(\ell) \wedge d \leq \mathit{dtb}_{\mathsf{high}}(\ell)$$

The definition of the validity of dtb as a family of dwell time bounds for $\mathcal{H}$ is equivalent to saying that $\mathsf{DTR}(\mathcal{H}, dtb)$ has the same set of runs as $\mathcal{H}$. Dwell time refinement becomes interesting when it is applied to a relaxation $\mathcal{H}^\#$ of $\mathcal{H}$.

Theorem 1. *If dtb is a valid family of dwell time bounds for $\mathcal{H}$, and $\mathcal{H}^\#$ is a relaxation of $\mathcal{H}$, then $\mathsf{DTR}(\mathcal{H}^\#, dtb)$ is also a relaxation of $\mathcal{H}$.* □

5 Inferring Dwell Time Bounds

Given a relaxation $\mathcal{H}^\#$ of a hybrid system $\mathcal{H}$, we define the operation $\mathsf{DTI}(\mathcal{H}^\#)$, which infers

$$dtb = \mathsf{DTI}(\mathcal{H}^\#),$$

a family of dwell time bounds which is valid for $\mathcal{H}$. The operation DTI consists of applying a syntactic transformation ST to the hybrid system $\mathcal{H}^\#$ and then applying a reachability analysis to the resulting hybrid system $ST(\mathcal{H}^\#)$, for every location ℓ. The syntactic transformation ST is very simple. Given the location ℓ, we add a dwell time variable d, which is a continuous varible. The dwell time variable is reset to 0 by every transition entering the location ℓ. It evolves like a clock variable (i.e., it has constant slope 1) in the location ℓ and it does not evolve in every other location (i.e., it has the constant slope 0). The syntactic transformation of $\mathcal{H}^\#$ thus consists of adding updates $d := 0$ to every incoming transition and adding the differential equation $\dot{d} = 1$ to the location ℓ and the differential equation $\dot{d} = 0$ to every other location.

Having computed a safe approximation of the set of reachable states of $ST(\mathcal{H}^\#)$, we are interested only in the set

$$\mathsf{Reach}_d(\ell')$$

of values of the continuous variable d of states at the neighbor locations ℓ' of ℓ.

- We define the constant $c_{(\ell,\ell')}$ as the minimum of $\mathsf{Reach}_d(\ell')$.
- We define the constant c_ℓ as the maximum of the union of sets $\mathsf{Reach}_d(\ell')$ for all neighbor locations ℓ'.

By repeating the above computation of the dwell time guards and invariants for all locations ℓ of the hybrid system $\mathcal{H}$ we obtain a valid family of dwell time bounds dtb.

For concreteness, we illustrate the syntactic transformation for the particular relaxation that we use in our experiments (see Section 7). We consider the relaxation $\mathrm{Proj}(\mathcal{H}, \ell)$ of $\mathcal{H}$ by projecting $\mathcal{H}$ to one single location, say ℓ, and its successor locations ℓ'. While abstracting away all predecessors of ℓ, we collect the entry guards and the invariants of the predecessors of ℓ. By applying the syntactic transformation ST to this relaxation, we obtain the hybrid system $\mathrm{Proj}_\mathrm{d}(\mathcal{H}, \ell)$. Informally (see also Figure 5), $\mathrm{Proj}_\mathrm{d}(\mathcal{H}, \ell) = ST(\mathrm{Proj}(\mathcal{H}, \ell))$ consists of only the location ℓ and its *neighbor locations* ℓ', which are the target locations of transitions from ℓ, and an additional clock variable d.

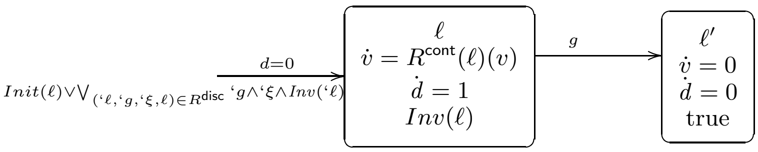

$Init(\ell) \vee \bigvee_{(`\ell,`g,`\xi,\ell)\in R^{\mathsf{disc}}} `g \wedge `\xi \wedge Inv(`\ell)$
$d{=}0$
ℓ
$\dot{v} = R^{\mathsf{cont}}(\ell)(v)$
$\dot{d} = 1$
$Inv(\ell)$
g
ℓ'
$\dot{v} = 0$
$\dot{d} = 0$
true

Fig. 5. Given the hybrid system $\mathcal{H}$ and the location ℓ, the figure presents the hybrid system $\mathrm{Proj}_{\mathrm{d}}(\mathcal{H}, \ell)$ used for the inference of dwell time constraints by reachability analysis. The new initial condition for the location ℓ uses the disjunction of the guards $`g$ of incoming transitions in conjunction with the invariants of their source locations $`\ell$

Formally,

$$\mathrm{Proj}_{\mathrm{d}}(\mathcal{H}, \ell) = (Loc_\ell,\ V \cup \{d\}, Init_\ell, R^{\mathsf{cont}}_\ell, R^{\mathsf{disc}}_\ell, Inv_\ell)$$

with

– the set of locations Loc_ℓ that consists of ℓ and all its successor locations ℓ',
– the set of continuous variables $V \cup \{d\}$, where d is a new clock variable,
– the initial condition for the location ℓ is augmented by the disjunction of the guards $`g$ of incoming transitions in conjunction with the invariants of their source locations $`\ell$ (pronounced "old-ell"), and the update,

$$Init_\ell(\ell) \ \equiv\ d = 0 \wedge (Init(\ell) \vee \bigvee_{(`\ell,`g,`\xi,\ell)\in R^{\mathsf{disc}}} `g \wedge `\xi \wedge Inv(`\ell))$$

– the initial condition for each neighbor location ℓ' is the Boolean constant *false*,
– the continuous transition relation R^{cont}_ℓ is given by

$$\begin{aligned} &R^{\mathsf{cont}}_\ell(\ell)(v) = R^{\mathsf{cont}}(\ell)(v) \text{ if } v \neq d \\ &R^{\mathsf{cont}}_\ell(\ell)(d) = 1 \\ &R^{\mathsf{cont}}_\ell(\ell') = 0 \quad \text{for each neighbor location } \ell' \end{aligned}$$

– the discrete transition relation R^{disc}_ℓ is the discrete transition relation R^{disc} restricted to transitions from ℓ,
– the invariant of the location ℓ is equal to $Inv(\ell)$,
– the invariant in each neighbor location ℓ' is the Boolean constant *true*.

Strictly speaking, $\mathrm{Proj}(\mathcal{H}, \ell)$ is not a relaxation of $\mathcal{H}$ (in the sense fixed in Section 2), since it has not the same set of locations and the same set of edges the set of its runs cannot be a superset of the set of runs $\mathcal{H}$. It is straightforward to extend the definition of $\mathrm{Proj}(\mathcal{H}, \ell)$ such that we obtain a relaxation in the strict sense.

6 Composing Reachability Analyses

We now define the method to sequentially compose reachability analyses. The method incorporates the result of the reachability analysis of one relaxation $\mathcal{H}_1^{\#}$

into the reachability analysis of another relaxation $\mathcal{H}_2^\#$ of a given hybrid system $\mathcal{H}$. We want to note that to get good experimental results it is crucial to choose judicious relaxations $\mathcal{H}_1^\#$ and $\mathcal{H}_2^\#$; several methods to obtain relaxations are mentioned in Section 2 and Section 5.

- The input to the method consists of two hybrid systems $\mathcal{H}_1^\#$ and $\mathcal{H}_2^\#$. The first one, $\mathcal{H}_1^\#$, embodies the auxiliary (in general, rather coarse) relaxation of $\mathcal{H}$. The second one, $\mathcal{H}_2^\#$, embodies another (in some sense, complementary) relaxation of $\mathcal{H}$. (As a consequence of our definition of relaxation, the three hybrid systems $\mathcal{H}$, $\mathcal{H}_1^\#$ and $\mathcal{H}_2^\#$ all have the same set of locations.)
- We infer $dtb_{\mathcal{H}_1^\#} = \mathsf{DTI}(\mathcal{H}_1^\#)$, a family of dwell time bounds valid for $\mathcal{H}$, by applying, repeatedly for each location ℓ, a reachability analysis to the hybrid system that is obtained from $\mathcal{H}_1^\#$ by the simple syntactic transformation described in Section 5.
- We use the family of dwell time bounds $dtb_{\mathcal{H}_1^\#}$ for the dwell time refinement of $\mathcal{H}_2^\#$; we obtain the hybrid system $\mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#})$ from $\mathcal{H}_2^\#$ by the syntactic transformation described in Section 4.
- The output of the method is the result of the reachability analysis applied to the hybrid system $\mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#})$.

The method is sound by Theorem 1, i.e., the output of the method is a safe approximation of the set of reachable states of the original hybrid system $\mathcal{H}$ and the following relation holds.

$$\mathcal{H} \sqsubseteq \mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#}) \sqsubseteq \mathcal{H}_2^\#$$

Completeness is of theoretical interest only; trivially, for every relaxation $\mathcal{H}_1^\#$ of $\mathcal{H}$ there exists a relaxation $\mathcal{H}_2^\#$ of $\mathcal{H}$ such that the hybrid system $\mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#})$ has the same runs as $\mathcal{H}$. Generally, for a given relaxation $\mathcal{H}_2^\#$ of $\mathcal{H}$, it is not possible to find a relaxation $\mathcal{H}_1^\#$ of $\mathcal{H}$ such that the inferred dwell time bounds are precise enough, i.e., such that $\mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#})$ has the same runs as $\mathcal{H}$.

A family of dwell time bounds incarnates a rather conservative approximation. This is because, intuitively, a bound at a location ℓ accounts for every visit of the location by a run of the hybrid system; i.e., it ignores the history of the run. As a tradeoff, one can choose a rather coarse relaxation for $\mathcal{H}_1^\#$ and obtain (practically optimal) dwell time bounds rather efficiently (this is confirmed by our practical experiments, where the cost for the inference of the dwell time bounds is negligible).

7 Case Studies

7.1 Case Study: Stability Verification

For a proof of concept, we have implemented the sequential composition of reachability analyses and automatic computation of dwell time bounds and used it to

Table 1. Case study on three previously unsolved instances of the stability verification problem: a water tank system with a parametrized number of tanks (three and four) and a temperature controller with a parametrized number of internal engines (two). Execution times are in seconds on a Pentium 2.7 GHz, with Linux Debian 2.1.18. The regions in the specification of the stability criterion are varied for the purpose of comparison of execution times; the stability condition becomes weaker with a larger region; for the purpose of specifying correctness, the larger regions are less interesting. In each case, $\mathcal{H}_2^{\#}$ is a relaxation of $\mathcal{H}$ obtained by eliminating all variables that are not used in the specification of the region. $\mathcal{H}_1^{\#}$ is the relaxation of $\mathcal{H}$ described in Section 5. Both, plain reachability and sequential composition of reachability analyses are realized with PHAVer [4], Version 0.38

system	region	$\mathcal{H}_2^{\#}$		$\mathsf{DTR}(\mathcal{H}_2^{\#}, dtb_{\mathcal{H}_1^{\#}})$	
		result	time	result	time
three water tanks	$x_3 \geq 5$	stable	5.03	stable	4.74
three water tanks	$x_3 \geq 6$	stable	897.82	stable	17.73
three water tanks	$x_3 \geq 7$	stable	13518.26	stable	34.53
three water tanks	$x_3 \geq 8$	–	timeout	stable	57.87
three water tanks	$x_3 \geq 9$	–	timeout	stable	83.13
three water tanks	$x_3 \geq 10$	–	timeout	stable	109.79
four water tanks	$x_4 \geq 5$	stable	18.66	stable	6.84
four water tanks	$x_4 \geq 6$	stable	5052.40	stable	38.38
four water tanks	$x_4 \geq 7$	–	timeout	stable	81.28
four water tanks	$x_4 \geq 8$	–	timeout	stable	160.30
four water tanks	$x_4 \geq 9$	–	timeout	stable	248.09
four water tanks	$x_4 \geq 10$	–	timeout	stable	343.15
two engines heater	$19 \leq x_r \leq 25$	???	14.44	stable	16.11
two engines heater	$20 \leq x_r \leq 25$	???	14.46	stable	17.01
two engines heater	$19 \leq x_r \leq 24$	???	25.04	stable	35.97
two engines heater	$20 \leq x_r \leq 24$	???	24.97	stable	37.39

conduct preliminary experiments with the abstraction-based verification method for region stability of hybrid systems [11,12,13]. This method seems an appealing first target because its preprocessing step doubles the number of continuous variables; i.e., variable elimination is the last resort for relatively small parameters in our scalable benchmark systems. As a consequence, a number of previously unsolved instances of the stability verification problem for classical hybrid system benchmarks were available as test cases. Our method proved to be effective on these tests.

A hybrid system is stable with respect to a given region φ if for every run, there exists a point of time such that from then on, the states on the run are always in the region φ (it may go in and out arbitrarily often before this point of time). The verification method transforms the hybrid system into another one where each continuous variable is duplicated. It is the hybrid system on that a reachability analysis is performed, e.g., by PHAVer [4]. There is an additional step on the output of the reachability analysis which does not matter here; the obstacle to scalability lies in the reachability analysis on the system with the doubled number of variables.

For our case study, we took two classical scalable benchmarks, the water tank system with a parametrized number of tanks and the temperature controller with a parametrized number of internal engines. The instances for three and four tanks and three internal engines were previously unsolved. Using sequential composition of reachability analyses with automatically inferred dwell time bounds for the abstraction of these hybrid systems by variable elimination, the verification method could solve these instances. We refer to Table 1 for the execution times (in sec, on a Pentium 2,7 GHz, with Linux Debian 2.1.18).

In order to obtain more benchmarks for comparing executions, we have varied the regions in the specification of the stability property. For sufficiently large regions, the abstraction by variable elimination was not coarse; i.e., the property could be checked. These regions are, however, not interesting as correctness specifications. The stability property for the water tank system expresses that the final tank (represented by x_3 resp. x_4) must stabilize with a minimum fill-up quantity. The last region is tight; i.e., the stability property does not hot hold for $x_3 \geq 11$ or $x_4 \geq 11$. For the two engines heater, the stability property expresses that the room temperature (represented by x_r) must stabilize within a "comfort zone". The last region is tight; i.e., the system does not stabilize for $x_r \geq 21$ or $x_r \leq 23$.

$\mathcal{H}_1^\#$ is the relaxation of $\mathcal{H}$ described in Section 5. The choice of the variable to be eliminated for the abstraction of $\mathcal{H}$ (yielding relaxation $\mathcal{H}_2^\#$) is determined by specification of the correctness property; i.e., we eliminate all variables that are not used for specifying the region (of 'stable states'). For the interesting regions, the abstraction $\mathcal{H}_2^\#$ is too coarse; the verification tool returns the answer

Table 2. Checking of a safety property: an automatic highway with arbiter example. $\mathcal{H}_1^\#$ is a relaxation of $\mathcal{H}$ obtained by ignoring the synchronization between the parallel components of the hybrid system $\mathcal{H}$. The relaxation $\mathcal{H}_2^\#$ is obtained by ignoring the values of the continuous variables of $\mathcal{H}$. Execution times are in seconds on a Pentium 2.7 GHz, with Linux Debian 2.1.18. Both, plain reachability and sequential composition of reachability analyses are realized with PHAVer [4], Version 0.38.

Number of cars	$\mathcal{H}$	DTR($\mathcal{H}_2^\#$, $dtb_{\mathcal{H}_1^\#}$)
8	2.06	4.29
9	2.93	4.92
10	4.76	5.57
11	32.05	6.78
12	58.87	7.81
13	171.64	7.94
14	829.56	9.39
15	5904.12	10.65
16	timeout	12.48
17	timeout	13.20
18	timeout	14.83
19	timeout	15.47
20	timeout	16.93

"stability unknown", while the verification method applied to $\mathsf{DTR}(\mathcal{H}_2^\#, dtb_{\mathcal{H}_1^\#})$ terminates with the answer "system stable" (see also Table 1, where "???" stands for "stability unknown").

7.2 Case Study: Safety Verification

We take the model of a central arbiter for an automated highway from [8]. We will check the safety property that no cars collide.

In this example the model is represented as a linear hybrid automaton. $\mathcal{H}_1^\#$ is obtained by ignoring the synchronization between the parallel components of the hybrid system $\mathcal{H}$. The relaxation $\mathcal{H}_2^\#$ is obtained by ignoring the values of the continuous variables of $\mathcal{H}$. The detailed results are presented in Table 2. We can see that using sequential composition of reachability analyses we can get the results much faster then just using plain reachability analysis with PHAVer.

8 Related Work and Conclusion

Related Work. Dwell time is a natural concept that appears in different uses [2,5,7,9,10,14,16]. Jumping ahead and summarizing the detailed discussion to follow, it seems that our work is the first to investigate the possibility of inferring dwell time bounds via a (black-box) reachability analysis applied to an (in general coarse) auxiliary abstraction, and using them for improving the precision of the reachability analysis of another (in some sense, complementary) abstraction of the hybrid system by a refinement (the refinement being a transformation of hybrid systems).

In [2], the dimension of the model is reduced by projecting out continuous variables and replacing differential equations by differential inclusions and, in case the resulting abstraction is too coarse, adding (manually inferred) information about staying time. The idea of using differential inclusions may be interesting in our setting as well.

In [5], clock variables replace variables with non-linear dynamics, which is possible only if the value of the replaced 'non-linear' variable can be inferred from the value of the replacing clock variable.

In [9,16], the abstract discrete model based on *rectangular cells* is refined by adding (manually inferred) information about the time between the entry and the exit of a rectangular cell. In [14], a hybrid system is transformed into a timed system by replacing each continuous variable with a clock variable. The motivation for such an approach is the possibility to use a model checker for timed automata. It may be interesting to combine the idea of splitting a location according to rectangular cells with our approach, in order to obtain more expressive dwell time constraints.

In [8], a coarse *relaxation abstraction* (which can be done by eliminating continuous variables) is iteratively refined (in a counterexample-guided fashion) by adding back continuous variables. This refinement is orthogonal to, and can potentially be combined with dwell time refinement.

The notion of *average dwell time* in [7,10] refers to a run, in contrast with our notion of dwell time which refers to an individual location. A check of the validity of a given average dwell time (but not its automatic inference) is proposed in [10]). It is not clear how one would relate the two notions. Average dwell time is used to bound the frequency of discrete transitions, which is a condition in a local proof rule for asymptotic stability.

Conclusion and Future Work. In this paper, we have presented new methods both for inferring and for exploiting dwell time bounds. As a consequence, we have obtained an approach where one can incorporate the result of a first, auxiliary reachability analysis into a second reachability analysis in order to refine its abstraction and thus improve its precision. Our approach is generic in that it uses reachability analysis as a black-box method. We have presented initial experiments that indicate its practical potential for checking safety and stability. The experiments used one particular tool, but in principle, the approach can be piggy-packed on other reachability analysis tools (each with its own abstraction method, and with different scopes of applicability), and it will be interesting to explore those options. Also, an interesting perspective is to investigate the Russian doll technique, i.e., the nesting of a sequence of reachability analyses for deriving a sequence of successively stronger dwell time bounds.

References

1. Alur, R., Grosu, R., Hur, Y., Kumar, V., Lee, I.: Modular specification of hybrid systems in CHARON. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 6–19. Springer, Heidelberg (2000)
2. Asarin, E., Dang, T.: Abstraction by projection and application to multi-affine systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 32–47. Springer, Heidelberg (2004)
3. Dang, T.: *d/dt*, `http://www-verimag.imag.fr/~tdang/ddt.html`
4. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
5. Henzinger, T.A., Ho, P.-H.: Algorithmic analysis of nonlinear hybrid systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 225–238. Springer, Heidelberg (1995)
6. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–463. Springer, Heidelberg (1997)
7. Hespanha, J.P., Morse, A.S.: Stability of switched systems with average dwell-time. In: Decision and Control (1999)
8. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007)
9. Maler, O., Batt, G.: Approximating continuous systems by timed automata. In: Fisher, J. (ed.) FMSB 2008. LNCS (LNBI), vol. 5054, pp. 77–89. Springer, Heidelberg (2008)

10. Mitra, S., Liberzon, D., Lynch, N.A.: Verifying average dwell time of hybrid systems. ACM Trans. Embedded Comput. Syst. 8(1) (2008)
11. Podelski, A., Wagner, S.: Model checking of hybrid systems: From reachability towards stability. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 507–521. Springer, Heidelberg (2006)
12. Podelski, A., Wagner, S.: Region stability proofs for hybrid systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 320–335. Springer, Heidelberg (2007)
13. Podelski, A., Wagner, S.: A sound and complete proof rule for region stability of hybrid systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 750–753. Springer, Heidelberg (2007)
14. Puri, A., Varaiya, P.: Verification of hybrid systems using abstractions. In: Hybrid Systems II, pp. 359–369. Springer, Heidelberg (1995)
15. Silva, B.I., Richeson, K., Krogh, B.H., Chutinan, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: ADPM (2000)
16. Stursberg, O., Kowalewski, S., Engell, S.: On the generation of timed discrete approximations for continuous systems. Mathematical and Computer Modeling of Dynamical Systems 6(1), 51–70 (2000)
17. Torrisi, F.D., Bemporad, A.: Hysdel — a tool for generating computational hybrid models for analysis and synthesis problems. IEEE Transactions on Control Systems Technology 12 (2004)

The Complexity of Codiagnosability for Discrete Event and Timed Systems

Franck Cassez[⋆]

National ICT Australia & CNRS
The University of New South Wales
Sydney, Australia

Abstract. In this paper we study the fault codiagnosis problem for discrete event systems given by finite automata (FA) and timed systems given by timed automata (TA). We provide a uniform characterization of codiagnosability for FA and TA which extends the necessary and sufficient condition that characterizes diagnosability. We also settle the complexity of the codiagnosability problems both for FA and TA and show that codiagnosability is PSPACE-complete in both cases. For FA this improves on the previously known bound (EXPTIME) and for TA it is a new result. Finally we address the codiagnosis problem for TA under bounded resources and show it is 2EXPTIME-complete.

1 Introduction

Discrete-event systems [1,2] (DES) can be modelled by finite automata (FA) over an alphabet of *observable* events Σ.

The *fault diagnosis problem* is a typical example of a problem under partial observation. The aim of fault diagnosis is to detect *faulty* sequences of the DES. The assumptions are that the behavior of the DES is known and a model of it is available as a finite automaton over an alphabet $\Sigma \cup \{\tau, f\}$, where Σ is the set of observable events, τ represents the unobservable events, and f is a special unobservable event that corresponds to the faults: this is the original framework introduced by M. Sampath *et al.* [3] and the reader is referred to this paper for a clear and exhaustive introduction to the subject. A *faulty* sequence is a sequence of the DES containing an occurrence of event f. An *observer* which has to detect faults, knows the specification/model of the DES, and it is able to observe sequences of *observable* events. Based on this knowledge, it has to announce whether an observation it makes (in Σ^*) was produced by a faulty sequence (in $(\Sigma \cup \{\tau, f\})^*$) of the DES or not. A *diagnoser* (for a DES) is an observer which observes the sequences of observable events and is able to detect whether a fault event has occurred, although it is not observable. If a diagnoser can detect a fault at most Δ steps[1] after it has occurred, the DES is said to be Δ-diagnosable. It is diagnosable if it is Δ-diagnosable for some $\Delta \in \mathbb{N}$. Checking whether a DES is Δ-diagnosable for

⋆ Author supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.

1 Steps are measured by the number of transitions in the DES.

a given Δ is called the *bounded diagnosability problem*; checking whether a DES is diagnosable is the *diagnosability problem.*

Checking *diagnosability* for a given DES and a fixed set of observable events can be done in polynomial time using the algorithms of [4,5]. If a diagnoser exists there is a finite state one. Nevertheless the size of the diagnoser can be exponential as it involves a determinization step. The extension of this DES framework to timed automata (TA) has been proposed by S. Tripakis in [6], and he proved that the problem of checking diagnosability of a timed automaton is PSPACE-complete. In the timed case however, the diagnoser may be a Turing machine. The problem of checking whether a timed automaton is diagnosable by a diagnoser which is a *deterministic* timed automaton was studied by P. Bouyer *et al.* [7].

Codiagnosability generalizes diagnosability by considering *decentralized architectures*. Such decentralized architectures have been introduced in [8] and later refined in [9,10]. In these architectures, local diagnosers (with their own partial view of the system) can send some information to a coordinator, summarizing their observations. The coordinator then computes a result from the partial results of the local diagnosers. The goal is to obtain a coordinator that can detect the faults in the system. When local diagnosers do not communicate with each other nor with a coordinator (protocol 3 in [8]), the decentralized diagnosis problem is called *codiagnosis* [10,9]. In this case, codiagnosis means that each fault can be detected by at least one local diagnoser. In the paper [10], codiagnosability is considered and an algorithm to check codiagnosability is presented for discrete event systems (FA). An upper bound for the complexity of the algorithm is EXPTIME. In [9], the authors consider a *hierarchical framework* for decentralized diagnosis. In [11] a notion of *robust* codiagnosability is introduced, which can be thought of as a *fault tolerant* (local diagnosers can fail) version of codiagnosability. None of the previous papers has addressed the codiagnosability problems for timed automata. Moreover, the exact complexity of the codiagnosis problems is left unsettled for discrete event systems (FA).

Our Contribution. In this paper, we study the codiagnosability problems for FA and TA. We settle the complexity of the problems for FA (PSPACE-complete), improving on the best known upper bound (EXPTIME). We also address the codiagnosability problems for TA and provide new results: algorithms to check codiagnosability and also codiagnosability under bounded resources. Our contribution is both of theoretical and practical interests. The algorithms we provide are optimal, and can also be implemented using standard model-checking tools like SPIN [12] for FA, or UPPAAL [13] for TA. This means that very expressive languages can be used to specify the systems to codiagnose and very efficient implementations and data structures are readily available.

Organisation of the Paper. Section 2 recalls the definitions of finite automata and timed automata. We also give some results on the Intersection Emptiness Problems (section 2.6) that will be used in the next sections. Section 3 introduces the fault codiagnosis problems we are interested in, and a necessary and sufficient condition that characterizes codiagnosability for FA and TA. Section 4 contains the first main results: optimal algorithms for the codiagnosability problems for FA and TA. Section 5 describes how to synthesize the codiagnosers and the limitations of this technique for TA. Section 6 is devoted to the codiagnosability problem under bounded resources for TA and contains the second main result of the paper.

Omitted proofs can be found in the full version of this paper [14].

2 Preliminaries

Σ denotes a finite alphabet and $\Sigma_\tau = \Sigma \cup \{\tau\}$ where $\tau \notin \Sigma$ is the *unobservable* action. $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$ is the set of boolean values, $\mathbb{N}$ the set of natural numbers, $\mathbb{Z}$ the set of integers and $\mathbb{Q}$ the set of rational numbers. $\mathbb{R}$ is the set of real numbers and $\mathbb{R}_{\geq 0}$ (resp. $\mathbb{R}_{>0}$) is the set of non-negative (resp. positive) real numbers. We denote tuples (or vectors) by $\overline{d} = (d_1, \cdots, d_k)$ and write $\overline{d}[i]$ for d_i.

2.1 Clock Constraints

Let X be a finite set of variables called *clocks*. A *clock valuation* is a mapping $v : X \rightarrow \mathbb{R}_{\geq 0}$. We let $\mathbb{R}^X_{\geq 0}$ be the set of clock valuations over X. We let $\overline{0}_X$ be the *zero* valuation where all the clocks in X are set to 0 (we use $\overline{0}$ when X is clear from the context). Given $\delta \in \mathbb{R}$, $v + \delta$ is the valuation defined by $(v + \delta)(x) = v(x) + \delta$. We let $\mathcal{C}(X)$ be the set of *convex constraints* on X, i.e., the set of conjunctions of constraints of the form $x \bowtie c$ with $c \in \mathbb{Z}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$. Given a constraint $g \in \mathcal{C}(X)$ and a valuation v, we write $v \models g$ if g is satisfied by the valuation v. We also write $[\![g]\!]$ for the set $\{v \mid v \models g\}$. Given a set $R \subseteq X$ and a valuation v of the clocks in X, $v[R]$ is the valuation defined by $v[R](x) = v(x)$ if $x \notin R$ and $v[R](x) = 0$ otherwise.

2.2 Timed Words

The set of finite (resp. infinite) words over Σ is Σ^* (resp. Σ^ω) and we let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. A *language* L is any subset of Σ^∞. A finite (resp. infinite) *timed word* over Σ is a word in $(\mathbb{R}_{\geq 0}.\Sigma)^*.\mathbb{R}_{\geq 0}$ (resp. $(\mathbb{R}_{\geq 0}.\Sigma)^\omega$). *Duration*$(w)$ is the duration of a timed word w which is defined to be the sum of the durations (in $\mathbb{R}_{\geq 0}$) which appear in w; if this sum is infinite, the duration is ∞. Note that the duration of an infinite word can be finite, and such words which contain an infinite number of letters, are called *Zeno* words. We let *Unt*(w) be the *untimed* version of w obtained by erasing all the durations in w. An example of untiming is *Unt*$(0.4\ a\ 1.0\ b\ 2.7\ c) = abc$. In this paper we write timed words as $0.4\ a\ 1.0\ b\ 2.7\ c \cdots$ where the real values are the durations elapsed between two letters: thus c occurs at global time 4.1.

TW$^*(\Sigma)$ is the set of finite timed words over Σ, *TW*$^\omega(\Sigma)$, the set of infinite timed words and *TW*$(\Sigma) =$ *TW*$^*(\Sigma) \cup$ *TW*$^\omega(\Sigma)$. A *timed language* is any subset of *TW*(Σ).

Let $\boldsymbol{\pi}_{\Sigma'}$ be the projection of timed words of *TW*(Σ) over timed words of *TW*(Σ'). When projecting a timed word w on a sub-alphabet $\Sigma' \subseteq \Sigma$, the durations elapsed between two events are set accordingly: for instance for the timed word $0.4\ a\ 1.0\ b\ 2.7\ c$, we have $\boldsymbol{\pi}_{\{a,c\}}(0.4\ a\ 1.0\ b\ 2.7\ c) = 0.4\ a\ 3.7\ c$ (note that projection erases some letters but keep the time elapsed between two letters). Given a timed language L, we let *Unt*$(L) = \{$*Unt*$(w) \mid w \in L\}$. Given $\Sigma' \subseteq \Sigma$, $\boldsymbol{\pi}_{\Sigma'}(L) = \{\boldsymbol{\pi}_{\Sigma'}(w) \mid w \in L\}$.

2.3 Timed Automata

Timed automata are finite automata extended with real-valued clocks to specify timing constraints between occurrences of events. For a detailed presentation of the fundamental results for timed automata, the reader is referred to the seminal paper of R. Alur and D. Dill [15].

Definition 1 (Timed Automaton). *A* Timed Automaton *A is a tuple $(L, l_0, X, \Sigma_\tau, E, Inv, F, R)$ where: L is a finite set of* locations*; l_0 is the* initial location*; X is a finite set of* clocks*; Σ is a finite set of* actions*; $E \subseteq L \times \mathcal{C}(X) \times \Sigma_\tau \times 2^X \times L$ is a finite set of* transitions*; in a transition (ℓ, g, a, r, ℓ'), g is the* guard*, a the* action*, and r the* reset *set; as usual we often write a transition $\ell \xrightarrow{g,a,r} \ell'$; $Inv \in \mathcal{C}(X)^L$ associates with each location an* invariant*; as usual we require the invariants to be conjunctions of constraints of the form $x \preceq c$ with $\preceq \in \{<, \leq\}$; $F \subseteq L$ is the set of* final *locations and $R \subseteq L$ is the set of* repeated *locations.* ■

The size of a TA A is denoted $|A|$ and is the size of the clock constraints i.e., the size of the transition relation E. A *state* of A is a pair $(\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$. A *run* ϱ of A from (ℓ_0, v_0) is a (finite or infinite) sequence of alternating *delay* and *discrete* moves:

$$\varrho = (\ell_0, v_0) \xrightarrow{\delta_0} (\ell_0, v_0 + \delta_0) \xrightarrow{a_0} (\ell_1, v_1) \ \cdots \ \xrightarrow{a_{n-1}} (\ell_n, v_n) \xrightarrow{\delta_n} (\ell_n, v_n + \delta_n) \cdots$$

s.t. for every $i \geq 0$:

- $v_i + \delta \models Inv(\ell_i)$ for $0 \leq \delta \leq \delta_i$;
- there is some transition $(\ell_i, g_i, a_i, r_i, \ell_{i+1}) \in E$ s.t. : (i) $v_i + \delta_i \models g_i$, (ii) $v_{i+1} = (v_i + \delta_i)[r_i]$.

The set of finite (resp. infinite) runs in A from a state s is denoted $Runs^*(s, A)$ (resp. $Runs^\omega(s, A)$). We let $Runs^*(A) = Runs^*(s_0, A)$, $Runs^\omega(A) = Runs^\omega(s_0, A)$ with $s_0 = (l_0, \overline{0})$, and $Runs(A) = Runs^*(A) \cup Runs^\omega(A)$. If ϱ is finite and ends in s_n, we let $last(\varrho) = s_n$. Because of the denseness of the time domain, the unfolding of A as a graph is infinite (uncountable number of states and delay edges). The *trace*, $tr(\varrho)$, of a run ϱ is the timed word $\boldsymbol{\pi}_\Sigma(\delta_0 a_0 \delta_1 a_1 \cdots a_n \delta_n \cdots)$. The duration of the run ϱ is $Duration(\varrho) = Duration(tr(\varrho))$. For $V \subseteq Runs(A)$, we let $Tr(V) = \{tr(\varrho) \mid \varrho \in V\}$, which is the set of traces of the runs in V.

A finite (resp. infinite) timed word w is *accepted* by A if it is the trace of a run of A that ends in an F-location (resp. a run that reaches infinitely often an R-location). $\mathcal{L}^*(A)$ (resp. $\mathcal{L}^\omega(A)$) is the set of traces of finite (resp. infinite) timed words accepted by A, and $\mathcal{L}(A) = \mathcal{L}^*(A) \cup \mathcal{L}^\omega(A)$ is the set of timed words accepted by A.

In the sequel we often omit the sets R and F in TA and this implicitly means $F = L$ and $R = \varnothing$.

A timed automaton A is *deterministic* if there is no τ labelled transition in A, and if, whenever (ℓ, g, a, r, ℓ') and $(\ell, g', a, r', \ell'')$ are transitions of A, $g \wedge g' \equiv$ FALSE. A is *complete* if from each state (ℓ, v), and for each action a, there is a transition (ℓ, g, a, r, ℓ') such that $v \models g$. We note DTA the class of deterministic timed automata.

A finite automaton is a particular TA with $X = \varnothing$. Consequently guards and invariants are vacuously true and time elapsing transitions do not exist. We write $A = (Q,$

$q_0, \Sigma_\tau, E, F, R)$ for a finite automaton. A run is thus a sequence of the form: $\varrho = \ell_0 \xrightarrow{a_0} \ell_1 \cdots\cdots \xrightarrow{a_{n-1}} \ell_n \cdots$ where for each $i \geq 0$, $(\ell_i, a_i, \ell_{i+1}) \in E$. Definitions of traces and languages are the same as for TA. For FA, the duration of a run ϱ is the number of steps (including τ-steps) of ϱ: if ϱ is finite and ends in ℓ_n, *Duration*$(\varrho) = n$ and otherwise *Duration*$(\varrho) = \infty$.

2.4 Region Graph of a Timed Automaton

A *region* of $\mathbb{R}^X_{\geq 0}$ is a conjunction of *atomic* constraints of the form $x \bowtie c$ or $x - y \bowtie c$ with $c \in \mathbb{Z}$, $\bowtie \in \{\leq, <, =, >, \geq\}$ and $x, y \in X$. The *region graph* $RG(A)$ of a TA A is a finite quotient of the infinite graph of A which is time-abstract bisimilar to A [15]. It is a finite automaton on the alphabet $E' = E \cup \{\tau\}$. The states of $RG(A)$ are pairs (ℓ, r) where $\ell \in L$ is a location of A and r is a *region* of $\mathbb{R}^X_{\geq 0}$. More generally, the edges of the graph are tuples (s, t, s') where s, s' are states of $RG(A)$ and $t \in E'$. Genuine unobservable moves of A labelled τ are labelled by tuples of the form $(s, (g, \tau, r), s')$ in $RG(A)$. An edge (g, λ, R) in the region graph corresponds to a discrete transition of A with guard g, action λ and reset set R. A τ move in $RG(A)$ stands for a delay move to the time-successor region. The initial state of $RG(A)$ is $(l_0, \overline{0})$. A final (resp. repeated) state of $RG(A)$ is a state (ℓ, r) with $\ell \in F$ (resp. $\ell \in R$). A fundamental property of the region graph [15] is:

Theorem 1 (R. Alur and D. Dill, [15]). $\mathcal{L}(RG(A)) = Unt(\mathcal{L}(A))$.

The (maximum) size of the region graph is exponential in the number of clocks and in the maximum constant of the automaton A (see [15]): $|RG(A)| = |L| \cdot |X|! \cdot 2^{|X|} \cdot K^{|X|}$ where K is the largest constant used in A.

2.5 Product of Timed Automata

Given a n locations $\ell_1, \cdots, \ell_n$, we write $\overline{\ell}$ for the tuple $(\ell_1, \cdots, \ell_n)$ and let $\overline{\ell}[i] = \ell_i$. Given a letter $a \in \Sigma^1 \cup \cdots \cup \Sigma^n$, we let $I(a) = \{k \mid a \in \Sigma^k\}$.

Definition 2 (Product of TA). *Let* $A_i = (L_i, l^i_0, X_i, \Sigma^i_\tau, E_i, Inv_i)$, $i \in \{1, \cdots, n\}$, *be* n *TA s.t.* $X_i \cap X_j = \varnothing$ *for* $i \neq j$. *The* product *of the* A_i *is the TA* $A = A_1 \times \cdots \times A_n = (L, \overline{l_0}, X, \Sigma_\tau, E, Inv)$ *given by:* $L = L_1 \times \cdots \times L_n$; $\overline{l_0} = (l^1_0, \cdots, l^n_0)$; $\Sigma = \Sigma^1 \cup \cdots \cup \Sigma^n$; $X = X_1 \cup \cdots \cup X_n$; $E \subseteq L \times \mathcal{C}(X) \times \Sigma_\tau \times 2^X \times L$ *and* $(\overline{\ell}, g, a, r, \overline{\ell}') \in E$ *if:*

– *either* $a \in \Sigma \setminus \{\tau\}$, *and*
  1. *for each* $k \in I(a)$, $(\overline{\ell}[k], g_k, a, r_k, \overline{\ell}'[k]) \in E_k$,
  2. $g = \wedge_{k \in I(a)} g_k$ *and* $r = \cup_{k \in I(a)} r_k$;
  3. *for* $k \notin I(a)$, $\overline{\ell}'[k] = \overline{\ell}[k]$;

– *or* $a = \tau$ *and* $\exists j$ *s.t.* $(\overline{\ell}[j], g_j, \tau, r_j, \overline{\ell}'[j]) \in E_j$, $g = g_j$, $r = r_j$ *and for* $k \neq j$, $\overline{\ell}'[k] = \overline{\ell}[k]$.

$Inv(\overline{\ell}) = \wedge^n_{k=1} Inv(\overline{\ell}[k])$. ■

This definition of product also applies to finite automata (no clock constraints).

If the automaton A_i has the set of final locations F_i then the set of final locations for A is $F_1 \times \cdots \times F_n$. For Büchi acceptance, we add a counter c to A which is incremented every time the product automaton A encounters an R_i-location in A_i, following the standard construction for product of Büchi automata. The automaton constructed with the counter c is A^+. The repeated set of states of A^+ is $L_1 \times \cdots \times L_{n-1} \times L_n \times \{n\}$. As the sets of clocks of the A_i's are disjoint[2], the following holds:

Fact 1. $\mathcal{L}^*(A) = \cap_{i=1}^n \mathcal{L}^*(A_i)$ *and* $\mathcal{L}^\omega(A^+) = \cap_{i=1}^n \mathcal{L}^\omega(A_i)$.

2.6 Intersection Emptiness Problem

In this section we give some complexity results for the emptiness problem on products of FA and TA.

First consider the following problem on deterministic finite automata (DFA):

Problem 1 (Intersection Emptiness for DFA)
INPUTS: *n deterministic finite automata $A_i, 1 \leq i \leq n$, over the alphabet Σ.*
PROBLEM: *Check whether $\cap_{i=1}^n \mathcal{L}^*(A_i) \neq \varnothing$.*

The size of the input for Problem 1 is $\sum_{i=1}^n |A_i|$.

Theorem 2 (D. Kozen, [16]). *Problem 1 is PSPACE-complete.*

D. Kozen's Theorem also holds for Büchi languages:

Theorem 3. *Checking whether $\cap_{i=1}^n \mathcal{L}^\omega(A_i) \neq \varnothing$ is PSPACE-complete.*

Problem 1 is PSPACE-hard even if $A_2, \cdots, A_n$ are automata where all the states are accepting and A_1 is the only automaton with a proper set of accepting states (actually one accepting state is enough).

Proposition 1. *Let $A_i, 1 \leq i \leq n$ be n DFA over the alphabet Σ. If for all $A_i, 2 \leq i \leq n$, all states of A_i are accepting, Problem 1 is already PSPACE-hard.*

The next results are counterparts of D. Kozen's results for TA.

Problem 2 (Intersection Emptiness for TA)
INPUTS: *n TA $A_i = (L_i, l_0^i, X_i, \Sigma_\tau^i, E_i, Inv_i, F_i)$, $1 \leq i \leq n$ with $X_k \cap X_j = \varnothing$ for $k \neq j$.*
PROBLEM: *Check whether $\cap_{i=1}^n \mathcal{L}^*(A_i) \neq \varnothing$.*

Theorem 4. *Problem 2 is PSPACE-complete.*

The previous theorem extends to Büchi languages:

Problem 3 (Büchi Intersection Emptiness for TA)
INPUTS: *n TA $A_i = (L_i, l_0^i, X_i, \Sigma_\tau^i, E_i, Inv_i, R_i)$, $1 \leq i \leq n$ with $X_k \cap X_j = \varnothing$ for $k \neq j$.*
PROBLEM: *Check whether $\cap_{i=1}^n \mathcal{L}^\omega(A_i) \neq \varnothing$.*

Theorem 5. *Problem 3 is PSPACE-complete.*

[2] For finite automata, this is is vacuously true.

3 Fault Codiagnosis Problems

We first recall the basics of *fault diagnosis*. The purpose of fault diagnosis [3] is to detect a fault in a system as soon as possible. The assumption is that the model of the system is known, but only a subset Σ_o of the set of events Σ generated by the system are observable. Faults are also unobservable.

Whenever the system generates a timed word $w \in \mathit{TW}^*(\Sigma)$, an external observer can only see $\boldsymbol{\pi}_{\Sigma_o}(w)$. If an observer can detect faults under this partial observation of the outputs of A, it is called a *diagnoser*. We require a diagnoser to detect a fault within a given delay $\Delta \in \mathbb{N}$.

To model timed systems with faults, we use timed automata on the alphabet $\Sigma_{\tau,f} = \Sigma_\tau \cup \{f\}$ where f is the *faulty* (and unobservable) event. We only consider one type of fault, but the results we give are valid for many-types of faults $\{f_1, f_2, \cdots, f_n\}$: indeed solving the many-types diagnosability problem amounts to solving n one-type diagnosability problems [5]. The observable events are given by $\Sigma_o \subseteq \Sigma$ and τ is always unobservable.

The idea of *decentralized* or *distributed* diagnosis was introduced in [8]. It is based on decentralized architectures: local diagnosers and a communication protocol. In these architectures, local diagnosers (with their own partial view of the system) can send to a coordinator some information, using a given communication protocol. The coordinator then computes a result from the partial results of the local diagnosers. The goal is to obtain a coordinator that can detect the faults in the system. When local diagnosers do not communicate with each other nor with a coordinator (protocol 3 in [8]), the decentralized diagnosis problem is called *codiagnosis* [10,9]. In this section we formalize the notion of codiagnosability introduced in [10] in a style similar to [17]. This allows us to obtain a necessary and sufficient condition for codiagnosability of FA but also to extend the definition of codiagnosability to *timed automata*.

In the sequel we assume that the model of the system is a TA $A = (L, l_0, X, \Sigma_{\tau,f}, E, \mathit{Inv})$ and is fixed.

3.1 Faulty Runs

Let $\Delta \in \mathbb{N}$. A run $\varrho = (\ell_0, v_0) \xrightarrow{\delta_0} (\ell_0, v_0 + \delta_0) \xrightarrow{a_0} (\ell_1, v_1) \ \cdots \ \xrightarrow{a_{n-1}} (\ell_n, v_n) \xrightarrow{\delta_n} (\ell_n, v_n + \delta) \ \cdots$ of A is Δ-faulty if: (1) there is an index i s.t. $a_i = f$ and (2) the duration of $\varrho' = (\ell_i, v_i) \xrightarrow{\delta_i} \cdots \xrightarrow{\delta_n} (\ell_n, v_n + \delta_n) \cdots$ is larger or equal to Δ. We let $\mathit{Faulty}_{\geq \Delta}(A)$ be the set of Δ-faulty runs of A. Note that by definition, if $\Delta' \geq \Delta$ then $\mathit{Faulty}_{\geq \Delta'}(A) \subseteq \mathit{Faulty}_{\geq \Delta}(A)$. We let $\mathit{Faulty}(A) = \cup_{\Delta \geq 0} \mathit{Faulty}_{\geq \Delta}(A) = \mathit{Faulty}_{\geq 0}(A)$ be the set of faulty runs of A, and $\mathit{NonFaulty}(A) = \mathit{Runs}(A) \setminus \mathit{Faulty}(A)$ be the set of non-faulty runs of A. Finally, we let $\mathit{Faulty}^{tr}_{\geq \Delta}(A) = \mathit{Tr}(\mathit{Faulty}_{\geq \Delta}(A))$ and $\mathit{NonFaulty}^{tr}(A) = \mathit{Tr}(\mathit{NonFaulty}(A))$ which are the traces[3] of Δ-faulty and non-faulty runs of A.

We also make the assumption that the TA A cannot prevent time from elapsing. For FA, this assumption is that from any state, a discrete transition can be taken. If it is not case, τ loop actions can be added with no impact on the (co)diagnosability status of the

[3] Notice that $\mathit{tr}(\varrho)$ erases τ and f.

system. This is a standard assumption in diagnosability and is required to avoid taking into account these cases that are not interesting in practice.

For discrete event systems (FA), the notion of time is the number of transitions (discrete steps) in the system. A Δ-faulty run is thus a run with a fault action f followed by at least Δ discrete steps (some of them can be τ or even f actions). When we consider codiagnosability problems for discrete event systems, this definition of Δ-faulty runs apply. The other definitions are unchanged.

Remark 1. A timed automaton where discrete actions are separated by one time unit is not equivalent to using a finite automaton when solving a fault diagnosis problem. For instance, a timed automaton can generate the timed words $1.f.1.a$ *and* $1.\tau.1.\tau.1.a$*. In this case, it is* 1*-diagnosable: after reading the timed word* $2.a$ *we announce a fault. If we do not see the* 1*-time unit durations, the timed words* $f.a$ *and* $\tau^2.a$ *give the same observation. And thus it is not diagnosable if we cannot measure time. Using a timed automaton where discrete actions are separated by one time unit gives to the diagnoser the ability to count/measure time and this is not equivalent to the fault diagnosis problem for FA (discrete event systems).*

3.2 Codiagnosers and Codiagnosability Problems

A *codiagnoser* is a tuple of diagnosers, each of which has its own set of observable events Σ_i, and whenever a fault occurs, at least one diagnoser is able to detect it. In the sequel we write $\boldsymbol{\pi}_i$ in place of $\boldsymbol{\pi}_{\Sigma_i}$ for readability reasons. A codiagnoser can be formally defined as follows:

Definition 3 **($(\Delta, \mathcal{E})$-Codiagnoser).** *Let A be a timed automaton over the alphabet $\Sigma_{\tau,f}$, $\Delta \in \mathbb{N}$ and $\mathcal{E} = (\Sigma_i)_{1 \leq i \leq n}$ be a family of subsets of Σ. A $(\Delta, \mathcal{E})$-*codiagnoser *for A is a mapping $\overline{D} = (D_1, \cdots, D_n)$ with $D_i : TW^*(\Sigma_i) \rightarrow \{0, 1\}$ such that:*

- *for each $\varrho \in NonFaulty(A)$, $\sum_{i=1}^{n} \overline{D}[i](\boldsymbol{\pi}_i(tr(\varrho))) = 0$,*
- *for each $\varrho \in Faulty_{\geq \Delta}(A)$, $\sum_{i=1}^{n} \overline{D}[i](\boldsymbol{\pi}_i(tr(\varrho))) \geq 1$.* ■

As for diagnosability, the intuition of this definition is that (i) the codiagnoser will raise an alarm ($\overline{D}$ outputs a value different from 0) when a Δ-faulty run has been identified, and that (ii) it can identify those Δ-faulty runs unambiguously. The codiagnoser is not required to do anything special for Δ'-faulty runs with $\Delta' < \Delta$ (although it is usually required that once it has announced a fault, it does not change its mind and keep outputting 1).

A is $(\Delta, \mathcal{E})$-codiagnosable if there exists a $(\Delta, \mathcal{E})$-codiagnoser for A. A is $\mathcal{E}$-codiagnosable if there is some $\Delta \in \mathbb{N}$ s.t. A is $(\Delta, \mathcal{E})$-codiagnosable.

The standard notions [3] of Δ-diagnosability and Δ-diagnoser are obtained when the family $\mathcal{E}$ is the singleton $\mathcal{E} = \{\Sigma\}$. The fundamental codiagnosability problems for timed automata are the following:

Problem 4 **($(\Delta, \mathcal{E})$-Codiagnosability)**
INPUTS: *A TA $A = (L, l_0, X, \Sigma_{\tau,f}, E, Inv)$, $\Delta \in \mathbb{N}$ and $\mathcal{E} = (\Sigma_i)_{1 \leq i \leq n}$.*
PROBLEM: *Is A $(\Delta, \mathcal{E})$-codiagnosable?*

Problem 5 (Codiagnosability)
INPUTS: *A TA* $A = (L, l_0, X, \Sigma_{\tau,f}, E, \mathit{Inv})$ *and* $\mathcal{E} = (\Sigma_i)_{1 \leq i \leq n}$.
PROBLEM: *Is* A $\mathcal{E}$*-codiagnosable?*

Problem 6 (Optimal delay)
INPUTS: *A TA* $A = (L, l_0, X, \Sigma_{\tau,f}, E, \mathit{Inv})$ *and* $\mathcal{E} = (\Sigma_i)_{1 \leq i \leq n}$.
PROBLEM: *What is the minimum* Δ *s.t.* A *is* $(\Delta, \mathcal{E})$*-codiagnosable?*

The size of the input for Problem 4 is $|A| + \log \Delta + n \cdot |\Sigma|$, and for Problems 5 and 6 it is $|A| + n \cdot |\Sigma|$.

In addition to the previous problems, we will consider the construction of a $(\Delta, \mathcal{E})$-codiagnoser when A is $(\Delta, \mathcal{E})$-codiagnosable in section 5.

3.3 Necessary and Sufficient Condition for Codiagnosability

In this section we generalize the necessary and sufficient condition for diagnosability [6,17] to codiagnosability.

Lemma 1. *A is not $(\Delta, \mathcal{E})$-codiagnosable if and only if $\exists \varrho \in \mathit{Faulty}_{\geq \Delta}(A)$ and*

$$\forall 1 \leq i \leq n, \exists \varrho_i \in \mathit{NonFaulty}(A) \text{ s.t. } \boldsymbol{\pi}_i(\mathit{tr}(\varrho)) = \boldsymbol{\pi}_i(\mathit{tr}(\varrho_i)). \tag{1}$$

Using Lemma 1, we obtain a language based characterisation of codiagnosability extending the one given in [6,17]. Let $\boldsymbol{\pi}_i^{-1}(X) = \{w \in \mathit{TW}^*(\Sigma) \mid \boldsymbol{\pi}_i(w) \in X\}$.

Lemma 2. *A is $(\Delta, \mathcal{E})$-codiagnosable if and only if*

$$\mathit{Faulty}^{tr}_{\geq \Delta}(A) \cap \left(\bigcap_{i=1}^{n} \boldsymbol{\pi}_i^{-1}\big(\boldsymbol{\pi}_i(\mathit{NonFaulty}^{tr}(A))\big) \right) = \varnothing. \tag{2}$$

4 Algorithms for Codiagnosability Problems

4.1 $(\Delta, \mathcal{E})$-Codiagnosability (Problem 4)

Deciding Problem 4 amounts to checking whether equation 2 holds or not. Recall that $A = (L, l_0, X, \Sigma_{\tau,f}, E, \mathit{Inv})$. Let t be a fresh clock not in X. Let $A^f(\Delta) = ((L \times \{0,1\}) \cup \{\mathit{Bad}\}, (l_0, 0), X \cup \{t\}, \Sigma_\tau, E_f, \mathit{Inv}_f)$ with:

- $((\ell, n), g, \lambda, r, (\ell', n)) \in E_f$ if $(\ell, g, \lambda, r, \ell') \in E$, $\lambda \in \Sigma \cup \{\tau\}$;
- $((\ell, 0), g, \tau, r \cup \{t\}, (\ell', 1)) \in E_f$ if $(\ell, g, f, r, \ell') \in E$;
- for $\ell \in L$, $((\ell, 1), t \geq \Delta, \tau, \varnothing, \mathit{Bad}) \in E_f$;
- $\mathit{Inv}_f((\ell, n)) = \mathit{Inv}(\ell)$.

$A^f(\Delta)$ is similar to A but when a fault occurs it switches to a copy of A (encoded by $n = 1$). When sufficient time has elapsed in the copy (more than Δ time units), location *Bad* can be reached. The language accepted by $A^f(\Delta)$ with the set of final states $\{\mathit{Bad}\}$ is thus $\mathcal{L}^*(A^f(\Delta)) = \mathit{Faulty}^{tr}_{\geq \Delta}(A)$. Define $A_i = (L, l_0, X_i, \Sigma_\tau, E_i, \mathit{Inv}_i)$ with:

- $X_i = \{x^i \mid x \in X\}$ (create copies of clocks of A);
- $(\ell, g_i, \lambda, r_i, \ell') \in E_i$ if $(\ell, g, \lambda, r, \ell') \in E$, $\lambda \in \Sigma_i \cup \{\tau\}$ with: g_i is g where the clocks x in X are replaced by their counterparts x^i in X_i; r_i is r with the same renaming;

– $(\ell, g_i, \tau, r_i, \ell') \in E_i$ if $(\ell, g, \lambda, r, \ell') \in E$, $\lambda \in \Sigma \setminus \Sigma_i$
– $Inv_i(\ell) = Inv(\ell)$ with clock renaming (x^i in place of x).

Each A_i accepts only non-faulty traces as the f-transitions are not in A_i. If the set of final locations is L for each A_i, then $\mathcal{L}^*(A_i) = \boldsymbol{\pi}_i(\mathit{NonFaulty}^{tr}(A))$. To accept $\boldsymbol{\pi}_i^{-1}\big(\boldsymbol{\pi}_i(\mathit{NonFaulty}^{tr}(A))$ we add transitions $(\ell, \text{TRUE}, \lambda, \varnothing, \ell)$ for each location ℓ of E_i and for each $\lambda \in \Sigma \setminus \Sigma_i$. Let A_i^* be the automaton on the alphabet Σ constructed this way. By definition of A_i^*, $\mathcal{L}^*(A_i^*) = \boldsymbol{\pi}_i^{-1}\big(\boldsymbol{\pi}_i(\mathit{NonFaulty}^{tr}(A))\big)$.

Define $\mathcal{B} = A^f(\Delta) \times A_1^* \times A_2^* \times \cdots \times A_n^*$ with the set of final locations $F_{\mathcal{B}} = \{\mathit{Bad}\} \times L \times \cdots \times L$. We let $R_{\mathcal{B}} = \varnothing$. Using equation 2 we obtain:

Lemma 3. *A is $(\Delta, \mathcal{E})$-codiagnosable iff $\mathcal{L}^*(\mathcal{B}) = \varnothing$.*

The size of the input for problem 4 is $|A| + \log \Delta + n \cdot |\Sigma|$. The size of $A^f(\Delta)$ is (linear in) the size of A and $\log \Delta$, i.e., $O(|A| + \log \Delta)$. The size of A_i^* is also bounded by the size of A. It follows that $|A^f(\Delta)| + \sum_{i=1}^{n} |A_i^*|$ is bounded by $(n+1)|A|$ and is polynomial in the size of the input of problem 4. We thus have a polynomial reduction from Problem 4 to the intersection emptiness problem for TA. We can now establish the following result:

Theorem 6. *Problem 4 is PSPACE-complete for Timed Automata. It is already PSPACE-hard for Deterministic Finite Automata.*

4.2 $\mathcal{E}$-Codiagnosability (Problem 5)

In this section we show how to solve the $\mathcal{E}$-codiagnosability problem. The algorithm is a generalisation of the procedure for deciding diagnosability of discrete event and timed systems (see [18] for a recent presentation).

For standard fault diagnosis (one diagnoser and $\mathcal{E} = \{\Sigma\}$), A is not diagnosable if there is an infinite faulty run in A the projection of which is the same as the projection of a non-faulty one [18].

The procedure for checking diagnosability of FA and TA slightly differ due to specific features of timed systems. We recall here the algorithms to check diagnosability of FA and TA [18,6] and extend them to codiagnosability.

Codiagnosability for Finite Automata. To check whether a FA A is diagnosable, we build a synchronized product $A^f \times A_1$, s.t. A^f behaves exactly like A but records in its state whether a fault has occurred, and A_1 behaves like A without the faulty runs (transitions labelled f are cut off). This corresponds to $A^f(\Delta)$ defined in section 4.1 without the clock Δ.

A *faulty run* in the product $A^f \times A_1$ is a run for which A^f reaches a faulty state of the form $(q, 1)$. To decide whether A is diagnosable we build an extended version of $A^f \times A_1$ which is a Büchi automaton $\mathcal{B}$ [18]: $\mathcal{B}$ has a boolean variable z which records whether A^f participated in the last transition fired by $A^f \times A_1$. A state of $\mathcal{B}$ is a pair (s, z) where s is a state of $A^f \times A_1$. $\mathcal{B}$ is given by the tuple $((Q \times \{0,1\} \times Q) \times \{0,1\}, ((q_0, 0), q_0, 0), \Sigma_\tau, \longrightarrow_{\mathcal{B}}, \varnothing, R_{\mathcal{B}})$ with:

- $(s,z) \xrightarrow{\lambda}_{\mathcal{B}} (s',z')$ if (i) there exists a transition $t : s \xrightarrow{\lambda} s'$ in $A^f \times A_1$, and (ii) $z' = 1$ if λ is a move of A^f and $z' = 0$ otherwise;
- $R_{\mathcal{B}} = \{(((q,1),q'),1) \mid ((q,1),q') \in A^f \times A_1\}$.

The important part of the previous construction relies on the fact that, for A to be non Σ-diagnosable, A^f should have an infinite faulty run (and take infinitely many transitions) and A_1 a corresponding non-faulty run (note that this one can be finite) giving the same observation. With the previous construction, we have [18]: A is diagnosable iff $\mathcal{L}^\omega(\mathcal{B}) = \varnothing$.

The construction for codiagnosability is an extension of the previous one adding $A_2, \cdots, A_n$ to the product. Let $\mathcal{B}^{co} = A^f \times A_1 \times \cdots \times A_n$ with A_i defined in section 4.1. In $\mathcal{B}^{co}$ we again use the variable z to indicate whether A^f participated in the last move. Define the set of repeated states of $\mathcal{B}^{co}$ by: $R_{\mathcal{B}^{co}} = \{(((q,1),\overline{q}),1) \mid ((q,1),\overline{q}) \in A^f \times A_1 \times \cdots \times A_n\}$. By construction, a state in $R_{\mathcal{B}^{co}}$ is: (1) faulty as it contains a component $(q,1)$ for the state of A^f and (2) A^f participated in the last move as $z = 1$. It follows that:

Lemma 4. *A is $\mathcal{E}$-codiagnosable iff $\mathcal{L}^\omega(\mathcal{B}^{co}) = \varnothing$.*

Theorem 7. *Problem 5 is PSPACE-complete for Deterministic Finite Automata.*

Codiagnosability for Timed Automata. Checking diagnosability for timed automata requires an extra step in the construction of the equivalent of automaton $\mathcal{B}$ defined above: indeed, for TA, a run having infinitely many discrete steps could well be *zeno*, i.e., the duration of such a run can be finite. This extra step in the construction was first presented in [6]. It can be carried out by adding a special timed automaton *Div* with two locations $\{0,1\}$ and synchronizing it with $A^f \times A_1$. If we use $F = \varnothing$ and $R = \{1\}$ for *Div*, any accepted run is *time divergent* and thus cannot be zeno. Let $\mathcal{D} = A^f \times \mathit{Div} \times A_1$ and let $F_{\mathcal{D}} = \varnothing$ and $R_{\mathcal{D}}$ be the set of states where A^f is in a faulty location and *Div* is in location 1. For standard fault diagnosis, the following holds [6,18]: A is diagnosable iff $\mathcal{L}^\omega(\mathcal{D}) = \varnothing$.

The construction to check codiagnosability is obtained by adding $A_2, \cdots, A_n$ in the product. Let $\mathcal{D}^{co} = A^f \times \mathit{Div} \times A_1 \times \cdots \times A_n$.

Lemma 5. *A is $\mathcal{E}$-codiagnosable iff $\mathcal{L}^\omega(\mathcal{D}^{co}) = \varnothing$.*

Theorem 8. *Problem 5 is PSPACE-complete for Timed Automata.*

4.3 Optimal Delay (Problem 6)

Using the results for checking $\mathcal{E}$-codiagnosability and $(\Delta,\mathcal{E})$-codiagnosability, we obtain algorithms for computing the optimal delay.

Lemma 4 reduces codiagnosability of FA to Büchi emptiness on a product automaton. The number of states of the automaton $\mathcal{B}^{co}$ is bounded by $4 \cdot |A|^n$, and the number of faulty states by $2 \cdot |A|^n$. This implies that:

Proposition 2. *Let A be a FA. If A is $\mathcal{E}$-codiagnosable, then A is $(2 \cdot |A|^n, \mathcal{E})$-codiagnosable.*

From Proposition 2, we can conclude that:

Theorem 9. *Problem 6 can be solved in PSPACE for FA.*

For timed automata, a similar reasoning can be done on the region graph of $\mathcal{D}^{co}$. If a TA A is $\mathcal{E}$-codiagnosable, there cannot be any cycle with faulty locations in $RG(\mathcal{D}^{co})$. Otherwise there would be a non-zeno infinite word in $\mathcal{L}(\mathcal{D}^{co})$ and thus an infinite time-diverging faulty run in A, with corresponding non-faulty runs in each A_i, giving the same observation. Let K be the size of $RG(\mathcal{D}^{co})$. If A is $\mathcal{E}$-codiagnosable, then a faulty state in $RG(\mathcal{D}^{co})$ can be followed by at most K states. Otherwise a cycle in the region graph would occur and thus $\mathcal{L}^{\omega}(\mathcal{D}^{co})$ would not be empty. This also implies that all the states (s, r) in $RG(\mathcal{D}^{co})$ that can follow a faulty state must have a *bounded* region. As the amount of time that can elapse in one region is at most 1 time unit[4], the maximum duration of a faulty run in $\mathcal{D}^{co}$ is bounded by K. This implies that:

Proposition 3. *Let A be a TA. If A is $\mathcal{E}$-codiagnosable, then A is $(K, \mathcal{E})$-codiagnosable with $K = |RG(\mathcal{D}^{co})|$.*

The size of the region graph of $\mathcal{D}^{co}$ is bounded by $|L|^{n+1} \cdot ((n+1)|X|+1)! \cdot 2^{(n+1)|X|+1} \cdot M^{(n+1)|X|+1}$. Thus the encoding of constant K has size $O(n \cdot |A|)$.

Theorem 10. *Problem 6 can be solved in PSPACE for Timed Automata.*

5 Synthesis of Codiagnosers

The reader is referred to the extended version of this paper [14] for a detailed presentation of this section. The synthesis of codiagnosers for FA and TA can be carried out by extending the known construction for diagnosers [3].

The construction of a diagnoser for timed automata [6] consists in computing *on-the-fly* the current possible states of the timed automaton A^f after reading a timed word w. This procedure is effective but gives a diagnoser which is a Turing machine. The machine computes a state estimate of A after each observable event, and if it contains only faulty states, it announces a fault.

Obviously the same construction can be carried out for codiagnosis: we define the Turing machines $M_i, 1 \leq i \leq n$ that estimate the state of A. When one M_i's estimate on an input Σ_i-trace w contains only faulty states, we set $D_i(w) = 1$ and 0 otherwise. This tuple of Turing machines is a $(\Delta, \mathcal{E})$-codiagnoser.

Computing the estimates with Turing machines might be too expensive to be implemented at runtime. More efficient and compact codiagnosers might be needed with reasonable computation times. In the next section, we address the problem of codiagnosis for TA under *bounded resources*.

6 Codiagnosis with Deterministic Timed Automata

The fault diagnosis problem using timed automata has been introduced and solved by P. Bouyer *et al.* in [7]. The problem is to determine, given a TA A, whether there exists a *diagnoser* D for A, that can be represented by a deterministic timed automaton.

[4] The constants in the automata are integers.

6.1 Fault Diagnosis with Deterministic Timed Automata

When synthesizing (deterministic) timed automata, an important issue is the amount of *resources* the timed automaton can use: this can be formally defined [19] by the (number of) clocks, Z, that the automaton can use, the maximal constant $\max$, and a *granularity* $\frac{1}{m}$. As an example, a TA of resource $\mu = (\{c,d\}, 2, \frac{1}{3})$ can use two clocks, c and d, and the clocks constraints using the rationals $-2 \leq k/m \leq 2$ where $k \in \mathbb{Z}$ and $m = 3$. A *resource* μ is thus a triple $\mu = (Z, \max, \frac{1}{m})$ where Z is finite set of clocks, $\max \in \mathbb{N}$ and $\frac{1}{m} \in \mathbb{Q}_{>0}$ is the *granularity*. DTA$_\mu$ is the class of DTA of resource μ.

Remark 2. Notice that the number of locations of the DTA in DTA$_\mu$ is not bounded and hence this family has an infinite (yet countable) number of elements.

If a TA A is Δ-diagnosable with a diagnoser that can be represented by a DTA D with resource μ, we say that A is (Δ, D)-diagnosable. P. Bouyer *et al.* in [7] considered the problem of deciding whether there exists a DTA diagnoser with resource μ:

Problem 7 (Δ-DTA-Diagnosability [7])
INPUTS: *A TA* $A = (L, l_0, X, \Sigma_{\tau,f}, E, Inv)$*,* $\Delta \in \mathbb{N}$*, a resource* $\mu = (Z, \max, \frac{1}{m})$.
PROBLEM: *Is there any* $D \in$ *DTA*$_\mu$ *s.t.* A *is* (Δ, D)*-diagnosable ?*

Theorem 11 (P. Bouyer *et al.*, [7]). *Problem 7 is 2EXPTIME-complete.*

The solution to the previous problem is based on the construction of a *two-player safety game*, $G_{A,\Delta,\mu}$. In this game a set of states, *Bad*, must be avoided for A to be Δ-diagnosable. The most permissive winning strategy gives the *set* of all DTA$_\mu$ diagnosers (the most permissive diagnosers) which can diagnose A (or $\varnothing$ is there is none). We refer to the extended version [14], section 6.1 for a detailed presentation of this construction.

6.2 Algorithm for Codiagnosability

In this section we include the alphabet Σ of a DTA in the resource μ and write $\mu = (\Sigma, Z, \max, \frac{1}{m})$.

Problem 8 (Δ-DTA-Codiagnosability)
INPUTS: *A TA* $A = (L, l_0, X, \Sigma_{\tau,f}, E, Inv)$*,* $\Delta \in \mathbb{N}$*, and a family of resources* $\mu_i = (\Sigma_i, Z_i, \max_i, \frac{1}{m_i}), 1 \leq i \leq n$ *with* $\Sigma_i \subseteq \Sigma$.
PROBLEM: *Is there any codiagnoser* $\overline{D} = (D_1, D_2, \cdots, D_n)$ *with* $D_i \in$ *DTA*$_{\mu_i}$ *s.t.* A *is* $(\Delta, \overline{D})$*-codiagnosable ?*

To solve Problem 8, we extend the algorithm given in [7] for DTA-diagnosability. Let G^i be the game G_{A,Δ,μ_i} and *Bad*$_i$ the set of bad states. Given a strategy f_i, we let $f_i(G^i)$ be the outcome[5] of G^i when f_i is played by Player 0. Given $w \in TW^*(\Sigma)$ and a DTA A on Σ, we let *last*(w, A) be the location reached when w is read by A.

[5] $f_i(G^i)$ is a timed transition system.

Table 1. Summary of the Results

	Δ-Codiagnosability	Codiagnosability	Optimal Delay	Synthesis (Bounded Resources)
FA	**PSPACE-C.** PTIME [5,4]	**PSPACE-C.** PTIME [5,4]	**PSPACE** PTIME [5,4]	**EXPTIME** EXPTIME [3]
TA	**PSPACE-C.** PSPACE-C. [6]	**PSPACE-C.** PSPACE-C. [6]	**PSPACE** PSPACE [18]	**2EXPTIME-C.** 2EXPTIME-C. [7]

Lemma 6. *A is $(\Delta, \overline{D})$-codiagnosable iff there is a tuple of strategies $\overline{f}$ s.t.*

$$(1)\ \forall 1 \leq i \leq n, \overline{f}[i] \text{ is state-based on the game } G^i, \text{ and}$$

$$(2)\ \forall w \in Tr(A) \quad \begin{cases} \text{If } S_i = last(\boldsymbol{\pi}_{\Sigma_i}(w), f_i(G^i)),\ 1 \leq i \leq n, \\ \text{then } \exists 1 \leq j \leq n,\ \text{s.t. } S_j \notin Bad_j. \end{cases}$$

Item (2) of Lemma 6 states that there is no word in A for which all the Player 0 in the games G^i are in bad states. The strategies for each Player 0 are not necessarily winning in each G^i, but there is always one Player 0 who has not lost the game G^i. From the previous Lemma, we can obtain the following result:

Theorem 12. *Problem 8 is 2EXPTIME-complete.*

7 Conclusion and Future Work

Table 1 gives an overview of the results described in this paper (bold face) for the codiagnosis problems in comparison with the results for the diagnosis problems (second line, normal face). Our ongoing work is to extend the results on *diagnosis using dynamic observers* [20,17] to the codiagnosis framework.

Ackowledgements. The author would like to thank the anonymous reviewers for their helpful comments.

References

1. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. SIAM Journal of Control and Optimization 25(1), 1202–1218 (1987)
2. Ramadge, P., Wonham, W.: The control of discrete event systems. Proc. of the IEEE 77(1), 81–98 (1989)
3. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of discrete event systems. IEEE Transactions on Automatic Control 40(9) (September 1995)
4. Jiang, S., Huang, Z., Chandra, V., Kumar, R.: A polynomial algorithm for testing diagnosability of discrete event systems. IEEE Transactions on Automatic Control 46(8) (August 2001)
5. Yoo, T.S., Lafortune, S.: Polynomial-time verification of diagnosability of partially-observed discrete-event systems. IEEE Transactions on Automatic Control 47(9), 1491–1495 (2002)

6. Tripakis, S.: Fault diagnosis for timed automata. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 205–224. Springer, Heidelberg (2002)
7. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 219–233. Springer, Heidelberg (2005)
8. Debouk, R., Lafortune, S., Teneketzis, D.: Coordinated decentralized protocols for failure diagnosis of discrete event systems. Discrete Event Dynamic Systems 10(1-2), 33–86 (2000)
9. Wang, Y., Yoo, T.S., Lafortune, S.: Diagnosis of discrete event systems using decentralized architectures. Discrete Event Dynamic Systems 17(2), 233–263 (2007)
10. Qiu, W., Kumar, R.: Decentralized failure diagnosis of discrete event systems. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 36(2), 384–395 (2006)
11. Basilio, J., Lafortune, S.: Robust codiagnosability of discrete event systems. In: Society, I.C. (ed.) Proceedings of the American Control Conference (ACC 2009), pp. 2202–2209 (2009)
12. Holzmann, G.J.: Software model checking with spin. Advances in Computers 65, 78–109 (2005)
13. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
14. Cassez, F.: The complexity of codiagnosability for discrete event and timed systems. Research report, National ICT Australia, 24 pages, document available from arXiv (April 2010), http://arxiv.org/abs/1004.2550
15. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
16. Kozen, D.: Lower bounds for natural proof systems. In: FOCS, pp. 254–266. IEEE, Los Alamitos (1977)
17. Cassez, F., Tripakis, S.: Fault diagnosis with static or dynamic diagnosers. Fundamenta Informaticae 88(4), 497–540 (2008)
18. Cassez, F.: A Note on Fault Diagnosis Algorithms. In: 48th IEEE Conference on Decision and Control and 28th Chinese Control Conference, Shanghai, P.R. China. IEEE Computer Society, Los Alamitos (December 2009)
19. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
20. Cassez, F., Tripakis, S., Altisen, K.: Sensor minimization problems with static or dynamic observers for fault diagnosis. In: 7th Int. Conf. on Application of Concurrency to System Design (ACSD 2007), pp. 90–99. IEEE Computer Society, Los Alamitos (2007)
21. Aceto, L., Laroussinie, F.: Is your model checker on time? on the complexity of model checking for timed modal logics. J. Log. Algebr. Program. 52-53, 7–51 (2002)

On Scenario Synchronization

Duc-Hanh Dang[1], Anh-Hoang Truong[1], and Martin Gogolla[2]

[1] University of Engineering and Technology,
Vietnam National University of Hanoi,
144 Xuan Thuy, Cau Giay, Hanoi, Vietnam
{hanhdd,hoangta}@vnu.edu.vn

[2] Department of Computer Science, University of Bremen,
D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. In software development a system is often viewed by various models at different levels of abstraction. It is very difficult to maintain the consistency between them for both structural and behavioral semantics. This paper focuses on a formal foundation for presenting scenarios and reasoning the synchronization between them. We represent such a synchronization using a transition system, where a state is viewed as a triple graph presenting the connection of current scenarios, and a transition is defined as a triple graph transformation rule. As a result, the conformance property can be represented as a Computational Tree Logic (CTL) formula and checked by model checkers. We define the transition system using our extension of UML activity diagrams together with Triple Graph Grammars (TGGs) incorporating Object Constraint Language (OCL). We illustrate the approach with a case study of the relation between a use case model and a design model. The work is realized using the USE tool.

1 Introduction

In software development a system is viewed by various models at different levels of abstraction. Models are defined in different modeling languages such as UML [1] and DSMLs [2]. It is often very difficult to maintain the consistency between them as well as to explain such a relation for both structural and behavioral semantics.

There are several approaches as introduced in [3,2,4] for behavioral semantics of modeling languages. The behavior semantics can be defined as trace-based, translation-based, denotation-based, and execution-based semantics. Such a semantics can also be obtained by semantics mappings as pointed out in [5,6]. The semantics can be represented by different formal methods such as graph transformation in [7], Z in [5,8] for a full formal description for the Unified Modeling Language (UML), and Alloy in [9] for a semantics of modeling languages. Metamodeling is another approach which allows us to define structural semantics of models. Models from modeling languages like UML must conform to the corresponding metamodels, i.e., their well-formedness needs to be ensured. Constraint languages for metamodels such as the Object Constraint Language (OCL) [10] allow us to express better structural semantics of models. In this context the relation between models can be obtained based on mappings between metamodels. On the mappings, transformation rules are defined for a model transformation. This principle is the core of many transformation tools and languages [11,12] as well as the Object Management Group (OMG) standard for model transformation, Query/View/Transformation (QVT) [13].

This paper aims to describe an integrated view on two modeling languages in order to characterize the semantics relation between them. Models within our approach are viewed as a set of execution scenarios of the system. We develop a formal foundation for presenting scenarios and reasoning the synchronization between scenarios. We represent such a synchronization using a transition system, where a state is viewed as a triple graph presenting the connection of current scenarios, and a transition is defined as a triple graph transformation rule. As a result, the conformance property can be represented as a Computational Tree Logic (CTL) fomula and checked by model checkers. We define the transition system using our extension of UML activity diagrams together with Triple Graph Grammars (TGGs) [14] incorporating Object Constraint Language (OCL) [15].

We illustrate our approach with a case study explaining the relation between a use case model and a design model. Use cases [1,16,17,18] have achieved wide acknowledgement for capturing and structuring software requirements. Our approach not only allows us to check the conformance between use case and design models but also to describe operational semantics of use cases in particular and modeling languages in general. We implement our approach based within the UML-based Specification Environment (USE) tool [19].

The rest of this paper is organized as follows. Section 2 presents preliminaries for our work. Section 3 explains scenarios and the synchronization between them in an informal way. Section 4 focuses on the syntax and semantics aspects of scenarios in order to form a formal foundation for scenario synchronization. The core is a transition system for the synchronization. Section 5 shows the CTL formula for the conformance property and explains our implementation in USE. Section 6 discusses related work. This paper is closed with a summary.

2 Preliminaries

This section presents preliminaries for our work. The definitions explained in this section are adapted from the work in [20]. Models in our work are seen as graphs. They are defined by a corresponding metamodel, which is represented as a type graph.

Definition 1. (Graphs and Graph Morphisms). *A graph $G = (G_V, G_E, s_G, t_G)$ consists of a set G_V of nodes, a set G_E of edges, and two functions $s_G, t_G : G_E \to G_V$, the source and the target function.*

Given graphs G, H a graph morphism $f = (f_V, f_E) : G \to H$ consists of two functions $f_V : G_V \to H_V$ and $f_E : G_E \to H_E$ that preserve the source and the target function, i.e., $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$. Graphs and graph *morphisms define the category* **Graph**. *A graph morphism f is injective if both functions f_V, f_E are injective.*

Definition 2. (Typing). *A tuple $(G, type_G)$ of a graph $G = (V, E, s, t)$ together with a graph morphism $type_G : G \to TG$, where TG is a graph, is called a typed graph. Then, TG is called a type graph. Given typed graphs $G = (G, type_G)$ and $H = (H, type_H)$, a typed graph morphism f is a graph morphism $f : G \to H$, such that $type_H \circ f = type_G$.*

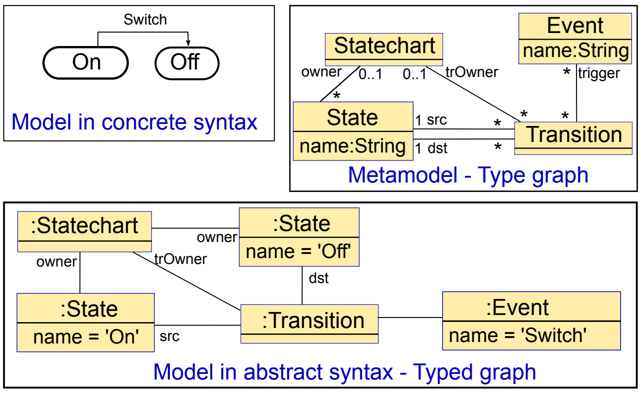


Fig. 1. Statechart as a typed graph conforms to the metamodel as a type graph

Example. Model as graph; Metamodel as type graph: The simplified metamodel which defines the structure of statecharts is represented by a type graph as shown in Fig. 1. Instances of these node types (Statechart, State, Transition, and Event) have to be linked according to the edge types between the node types as well as attributed according to note type attributes.

In order to relate pair of models to each other, we will consider such a combination as a triple graph. Then, a triple graph transformation allows us to build states of the integration.

Definition 3. (Triple Graphs and Triple Graph Morphisms).
Three graphs SG, CG, and TG, called source, connection, and target graph, together with two graph morphisms $s_G : CG \to SG$ and $t_G : CG \to TG$ form a triple graph $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$. G is called empty, if SG, CG, and TG are empty graphs.

A triple graph morphism $m = (s, c, t) : G \to H$ between two triple graphs $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$ and $H = (SH \overset{s_H}{\leftarrow} CH \overset{t_H}{\rightarrow} TH)$ consists of three graph morphisms $s : SG \to SH$, $c : CG \to CH$ and $t : TG \to TH$ such that $s \circ s_G = s_H \circ c$ and $t \circ t_G = t_H \circ c$. It is injective, if morphisms s, c and t are injective. Triple graphs and triple graph morphisms form the category **TripleGraph**.

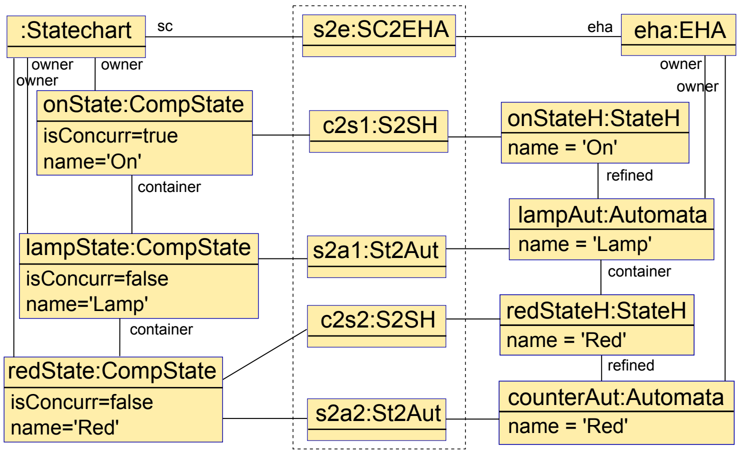


Fig. 2. Triple graph for an integrated SC2EHA model

Example. Triple graph: The graph in Fig. 2 shows a triple graph containing a statechart together with correspondence nodes pointing to the extended hierarchical automata (EHA). References between source and target models denote translation correspondences. For a detailed explanation of the transformation, we refer to the work in [21].

Definition 4. (Triple Graph Transformation Systems)
A triple rule $tr = \mathbf{L} \xrightarrow{tr} \mathbf{R}$ *consists of triple graphs* $\mathbf{L}$ *and* $\mathbf{R}$ *and an injective triple graph morphisms* tr*.*

$$\begin{tikzcd}
L \arrow[d, "tr"'] & = & (SL \arrow[d, "s"'] & CL \arrow[l, "s_L"'] \arrow[r, "t_L"] \arrow[d, "c"'] & TL) \arrow[d, "t"'] \\
R & = & (SR & CR \arrow[l, "s_R"'] \arrow[r, "t_R"] & TR)
\end{tikzcd}$$

Given a triple rule $tr = (s, c, t) : \mathbf{L} \to \mathbf{R}$*, a triple graph* G *and a triple graph morphism* $m = (sm, cm, tm) : \mathbf{L} \to G$*, called triple match* m*, a triple graph transformation step* $G \overset{tr,m}{\Longrightarrow} H$ *from* G *to a triple graph* H *is given by three objects* SH*,* CH *and* TH *in category* **Graph** *with induced morphisms* $s_H : CH \to SH$ *and* $t_H : CH \to TH$*. Morphism* $n = (sn, cn, tn)$ *is called comatch.*

$$\begin{tikzcd}
 & & & SL \arrow[dl, "sm"'] \arrow[dd, dashed] & & CL \arrow[ll] \arrow[rr] \arrow[dl, "cm"'] \arrow[dd, dashed] & & TL \arrow[dl, "tm"'] \arrow[dd] \\
G \arrow[ddd, Rightarrow, "tr"'] & = & (SG \arrow[ddd, "s'"'] & & CG \arrow[ll] \arrow[rr] \arrow[ddd, "c'"'] & & TG) \arrow[ddd, "t'"'] & \\
 & & & SR \arrow[ddl, dashed, "sn"] & & CR \arrow[ll, dashed] \arrow[rr, dashed] \arrow[ddl, dashed, "cn"] & & TR \arrow[ddl, "tn"] \\
 & & & & & & & \\
H & = & (SH & & CH \arrow[ll, "s_H"] \arrow[rr, "t_H"'] & & TH) &
\end{tikzcd}$$

A triple graph transformation system is a structure $TGTS = (S, TR)$ *where* S *is an initial graph and* $TR = \{tr_1, tr_2, ..., tr_n\}$ *is a set of triple rules. Triple graphs in the set* $\{G | S \Rightarrow^* G\}$ *are referred to as reachable states.*

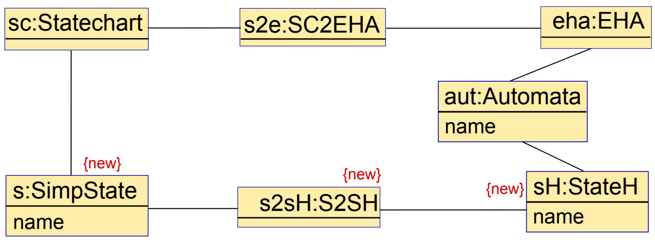


Fig. 3. Triple rule for SC2EHA model transformation

Example. Triple rule: The rule in Fig. 3 is part of a triple graph transformation system that generates statecharts and corresponding EHA models, as introduced in [21]. This rule may create a simple state of a statechart and its corresponding state of the corresponding EHA model at any time.

3 Scenarios and Synchronization

This section explains scenarios and scenario synchronization in an informal way. We focus on activity diagrams and their extensions as a means to present scenarios. Activity diagrams normally allow us to present scenarios at different levels of abstraction, ranging from the very high level such as workflows to lower levels such as execution scenarios of programs. However, they only emphasize the flows in scenarios, and the meaning of actions is unavailable so that the information of scenarios is often incompletely captured by this kind of diagrams. We refine activity diagrams by adding into each action a pair of interrelated object diagrams attached with OCL conditions as pre- and postconditions of the action.

With the extension the semantics of activity diagrams needs to be updated. The key question is how a scenario is defined for each system execution from a specification using extended activity diagrams. Normally, pre- and postconditions for each action incompletely capture the effect of the action, we refer to such activity diagrams as *declarative activity diagrams*. Figure 4 shows an example for declarative activity diagrams. This diagram presents scenarios of the use case "ReturnCar", which describes a fragment of the service of a car rental system. In the diagram, use case snapshots, which include objects, links, and OCL conditions, are denoted by rectangles. Here, we use concepts of the conceptual domain of the system in order to present use case snapshots. System and actor actions, e.g., the actions (1) and (4) are denoted by rounded rectangles. Use case actions, e.g., the action (5) are denoted by the double-line rounded rectangles. A conditional action, e.g., the action (2) is denoted by the dashed-line rounded rectangles. The extension point, e.g., the `Return Late` extension point of the action (4), is denoted by the six-sided polygons.

At the design level, effect of each action is fully reflected by its pre- and postconditions, and scenarios reflecting the system behavior, are completely determined. We refer to the kind of activity diagrams as *operational activity diagrams*.

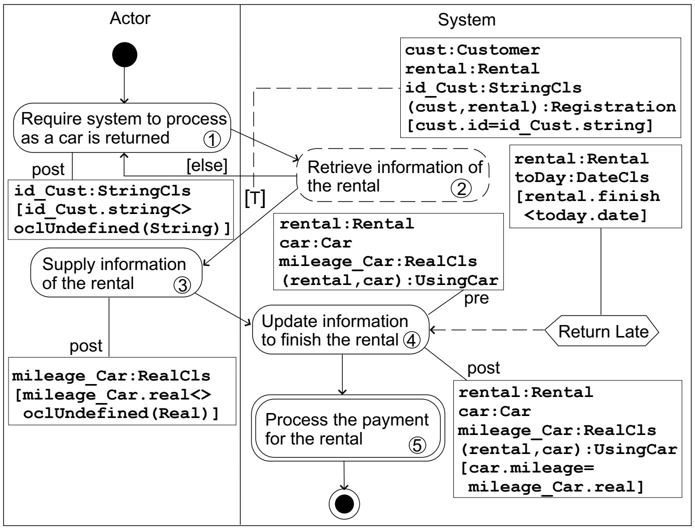

Actor
System
Require system to process as a car is returned ①
post
id_Cust:StringCls [id_Cust.string<> oclUndefined(String)]
[else]
[T]
cust:Customer rental:Rental id_Cust:StringCls (cust,rental):Registration [cust.id=id_Cust.string]
Retrieve information of the rental ②
rental:Rental toDay:DateCls [rental.finish <today.date]
Supply information of the rental ③
rental:Rental car:Car mileage_Car:RealCls (rental,car):UsingCar
pre
post
mileage_Car:RealCls [mileage_Car.real<> oclUndefined(Real)]
Update information to finish the rental ④
Return Late
post
rental:Rental car:Car mileage_Car:RealCls (rental,car):UsingCar [car.mileage= mileage_Car.real]
Process the payment for the rental ⑤

Fig. 4. Scenarios at the use case level of the use case "ReturnCar" presented by declarative activity diagrams

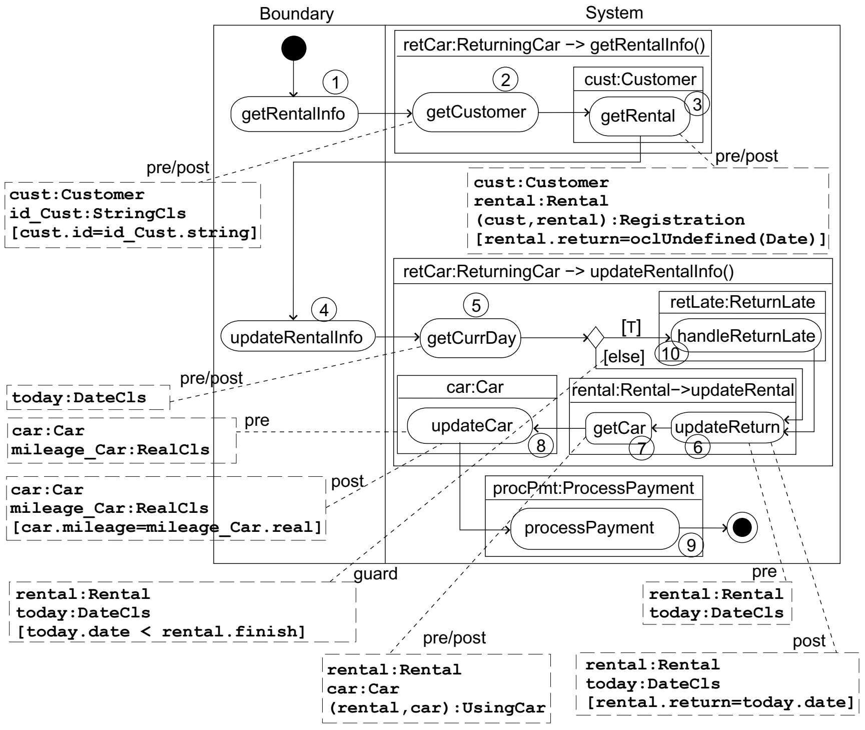

Boundary
System
retCar:ReturningCar -> getRentalInfo()
① getRentalInfo
② getCustomer
cust:Customer
getRental ③
pre/post
cust:Customer id_Cust:StringCls [cust.id=id_Cust.string]
pre/post
cust:Customer rental:Rental (cust,rental):Registration [rental.return=oclUndefined(Date)]
retCar:ReturningCar -> updateRentalInfo()
④ updateRentalInfo
⑤ getCurrDay
[T]
[else]
retLate:ReturnLate
handleReturnLate ⑩
pre/post
today:DateCls
car:Car
rental:Rental->updateRental
pre
car:Car mileage_Car:RealCls
updateCar ⑧
getCar ⑦
updateReturn ⑥
post
car:Car mileage_Car:RealCls [car.mileage=mileage_Car.real]
procPmt:ProcessPayment
processPayment ⑨
guard
rental:Rental today:DateCls [today.date < rental.finish]
pre
rental:Rental today:DateCls
pre/post
rental:Rental car:Car (rental,car):UsingCar
post
rental:Rental today:DateCls [rental.return=today.date]

Fig. 5. Scenarios at the design level of the use case "ReturnCar" presented by operational activity diagrams

Figure 5 shows an example for operational activity diagrams. This diagram captures scenarios at the design level of the use case "ReturnCar", refining by the diagram depicted in Fig. 5. Snapshots at the level are used to specify pre- and postconditions in action contracts and the branch conditions. Actions in a scenario at the design level are organized in a hierarchy by action groups. This hierarchy originates from mappings between a sequence diagram and a corresponding extended activity diagram: The interaction sequence between objects (by messages) is represented by an action sequence. Each message sent to a lifeline in the sequence diagram corresponds to an action or an action group which realizes the object operation invoked by this message. The action group includes actions and may include other action groups. An action group always links to an object node at the corresponding lifetime line.

Scenarios of a declarative activity diagram will depend on scenarios of the operational diagram which refines the activity diagram. We need to clarify scenarios in extended activity diagrams as well as to maintain the conformance between a declarative activity diagram and a corresponding operational activity diagram. Specifically, we need to relate action effects for scenarios at these diagrams to each other. This is based on the refinement of actions in the declarative activity diagram by an action group in the operational activity diagram. A current action at this activity diagram will correspond to a current action at the other activity diagram. In this way a synchronization of scenarios at operational and declarative activity diagrams is formed for each system execution.

4 Scenarios and Synchronization, Formally

First, we focus on the syntax of extended activity diagrams in order to present scenarios. Then, we consider the semantics aspect, where an operational semantics for extended activity diagrams is defined. We aim to build a transition system reflecting the synchronization between scenarios.

4.1 Syntax Aspect

Similar to the work in [22], we also restrict our consideration to well-structured activity diagrams: The building blocks are only sequences, fork-joins, decisions, and loops. We define new meta-concepts in addition to the UML metamodel in order to present extended activity diagrams. Due to the limited space, concepts of the metamodels are only shown in triple rules, which are depicted in the long version of this paper[1]. Here, we refer to them, i.e., metamodels of declarative and operational activity diagram as graphs DG and OG, respectively.

Definition 5. (Declarative Activity Diagrams). *A declarative activity diagram is a graph typed by the graph DG, where DG is a graph corresponding to the metamodel for declarative activity diagrams.*

[1] `http://www.coltech.vnu.edu.vn/~hanhdd/publications/Dang_2010_ATVA.pdf`

Definition 6. (Operational Activity Diagrams). *An operational activity diagram is a graph typed by the graph OG, where OG is a graph corresponding to the metamodel for operational activity diagrams.*

Note that each Action object node in the DG and OG graphs, which represent an action, is attached with SnapshotPattern object nodes, which express pre- and postconditions of the action, respectively. The attribute *snapshot* of a SnapshotPattern node is a graph whose nodes are variables. This graph is typed by the graph CD, which is a graph corresponding to the class diagram of the system (i.e., a system state is a graph typed by CD). For example, Fig. 4 shows SnapshotPatterns as the pre- and postcondition of the action marked by (4).

Well-formedness of extended activity diagrams can be ensured using OCL conditions. For example, it ensures that an activity diagram has exactly one InitialNode and ActivityFinalNode.

4.2 Semantics Aspect - Synchronization by a Transition System

Activity diagrams basically have a Petri-like semantics. However, as discussed in Sect. 3 scenarios from declarative and operational activity diagrams depend with each other. In order to obtain an operational semantics for these extended activity diagrams, we have to define a pair of scenarios in synchronization for each system execution. This section clarifies what a current state of the synchronization is and which transitions are used for it.

State of Scenario Synchronization. In order to form a semantics domain for extended activity diagrams, we define new meta-concepts connecting the metamodels DG and OG to each other. In this way a type graph EG for triple graphs is established. We add the new concept ExecControl into the correspondence part of the triple graph in order to mark current actions of the declarative and operational activity diagrams.

Definition 7. (State of Scenario Synchronization). *Let dG be a declarative activity diagram, and oG be a corresponding operational activity diagram. The state of the scenario synchronization between dG and oG is a triple graph $eG \in EG$ connecting dG and oG to each other:*

- *The current actions of dG and oG are the set of actions $currDG = \{a \in dG | \exists e : ExecControl \cdot (e, a) \in E_{eG}\}$ and $currOG = \{a \in oG | \exists e : ExecControl \cdot (e, a) \in E_{eG}\}$, respectively.*
- *The current snapshot corresponding to the action $a1 \in currDG$ and $a2 \in currOG$ is the snapshot $sp1 : SnapshotPattern | (a1, sp1) \in E_{dG} \wedge \exists e : ExecControl \cdot (e, sp1) \in E_{eG}$ and $sp2 : SnapshotPattern | (a2, sp2) \in E_{oG} \wedge \exists e : ExecControl \cdot (e, sp2) \in E_{eG}$, respectively.*

Example. Figure 6 shows a current state of the synchronization between scenarios shown in Fig. 4 and Fig. 5.

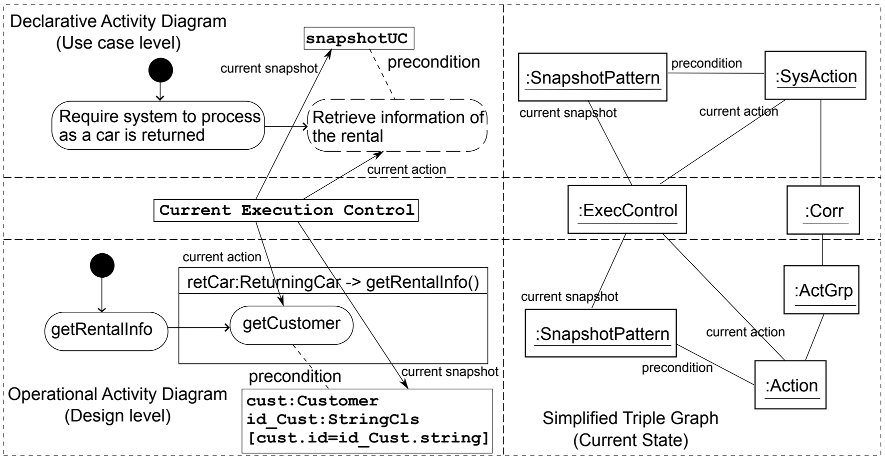


Fig. 6. Current synchronization state of scenarios shown in Fig. 5 and Fig. 4

Transitions. A transition of the system is defined by a graph transformation rule in two cases: (1) The rule is used to transform the current snapshot as the precondition of the current action into the next snapshot as the postcondition; (2) The rule is used to transform the current state (as a triple graph) to the next state by selecting the next current actions. The first case is referred to as snapshot transitions. The transition rules are defined according to the specification of the system. The second case is referred to as action transitions. The transition rules are defined based on the refinement relation between a declarative activity diagram and an operational activity diagram as discussed in Sect. 3. The rules are independent with a concrete system.

Definition 8. (Snapshot Transition). *A snapshot transition is a triple rule which allows us to transform a state eG_1 to the next state eG_2 such that the current actions are unchanged and only the current snapshot of dG or oG is changed from as the precondition snapshot to the postcondition snapshot by a corresponding graph transformation rule. This postcondition snapshot needs to be fulfilled.*

Example. Let us consider the synchronization between scenarios shown in Fig. 4 and Fig. 5. A snapshot transition will transfer from the current state, which refers to the action (4) and its precondition snapshot (as depicted in Fig. 4), to the next state which refers to the postcondition snapshot of this action.

Definition 9. (Action Transition). *An action transition is a triple rule which allows us to transform a state eG_1 to the next state eG_2 such that the current actions and the current snapshots are changed and the current snapshots as the precondition of the current actions in eG_2 are fulfilled.*

Example. An action transition will transfer from the current state, which refers to the action (4) shown in Fig. 4 and the action (8) shown in Fig. 5, to the next state which refers to the action (5) and action (9) of these scenarios.

Definition 10. (Sound and Conformance Property). *Let $TS = (S, \rightarrow, s_0)$ be a transition system, where s_0 is the initial state, i.e., the current actions are the initial actions of the declarative and operational activity diagrams dG and oG; S is a set of reachable states from s_0 by snapshot transitions SR and action transitions AR. The activity diagrams dG and oG are sound and conformed to each other if and only if the following conditions hold:*

1. $\forall s \in S \cdot \exists sys_0, ..., sys_n : CD \cdot (s_i \xrightarrow{r_k \in SR} s_{i+1} \Rightarrow sys_k \xrightarrow{r_k} sys_{k+1}) \wedge \exists e : ExecControl \cdot \forall sp : SnapshotPattern \cdot (e, sp) \in E_s \Rightarrow isValid(sp, sys_n)$, *where $isValid(sp, sys_n)$ indicates the graph sp conforms to the graph sys_n.*
2. $\forall sys : CD \cdot \exists s_1, ..., s_n \in S \cdot (s_i \rightarrow s_{i+1} \wedge isFinalState(s_n) \wedge (s_i \xrightarrow{r_k \in SR} s_{i+1} \Rightarrow sys_k \xrightarrow{r_k} sys_{k+1}) \wedge sys_0 = sys$, *where $isFinalState(s_n)$ indicates s_n is the final state, i.e., the ExecControl object node of this triple graph points to the final nodes of dG and oG.*

In this definition Condition 1 ensures that each snapshot in the snapshot sequence corresponding to the scenario from *dG* and *oG* is valid. Condition 2 ensures that we can always define a pair of scenarios for a system execution starting from a *sys* state.

5 Conformance Property and Tool Support

We aim to obtain an automatic check for the sound and conformance property mentioned above. To utilize model checkers for the goal we need to translate the conditions of Def. 10 into CTL, i.e., the notion of temporal logic most model checkers understand. Now we briefly define the CTL formulas we will use to express our conditions. Note that this is only a subset of CTL.

Definition 11. (CTL Formulas). *Let $TS = (S, \rightarrow, s_0)$ be a transition system by snapshot transitions and action transitions. Let $Comp(s_0)$ be all possible computations starting with the state s_0: $Comp(s) := \{s_0 s_1 s_2 ... | (s_i, s_{i+1}) \in \rightarrow\}$, and let p be some atomic proposition. Then*

$$TS \models \mathbf{AG}(p) \Leftrightarrow \forall s_0 s_1 ... \in Comp(s_0) \forall k \in \mathbb{N} : p \textit{ holds in } s_k$$
$$TS \models \mathbf{AF}(p) \Leftrightarrow \forall s_0 s_1 ... \in Comp(s_0) \exists k \in \mathbb{N} : p \textit{ holds in } s_k$$
$$TS \models \mathbf{EF}(p) \Leftrightarrow \exists s_0 s_1 ... \in Comp(s_0) \exists k \in \mathbb{N} : p \textit{ holds in } s_k$$

We are now ready to formulate our theorem.

Theorem 1. *Let dG and oG be the declarative and operational activity diagrams, respectively. Let $TS = (S, \rightarrow, s_0)$ be a transition system by snapshot transitions and action transitions (S contains exactly those states which are reachable from s_0). dG and oG are sound and conformed to each other if and only if the following CTL formulas hold for TS:*

1. $TS \models \mathbf{AG}(isValidSnapshot)$, *where that isValidSnapshot holds in the state s, denoted as $s \models isValidSnapshot$, means the current SnapshotPattern objects of s are valid.*

2. *$TS \models \mathbf{EF}(AtFinalState)$, where that AtFinalState holds in the state s, denoted as $s \models AtFinalState$, means the ExecControl object node of this triple graph (s) points to the final nodes of dG and oG.*

Proof. We start by pointing out the equivalence of the first condition of Def. 10 and Theor. 1. We have $TS \models \mathbf{AG}(isValidSnapshot) \Leftrightarrow \forall s_0 s_1 ... \in Comp(s_0) \forall k \in \mathbb{N} : s_k \models isValidSnapshot \Leftrightarrow \forall s_0 s_1 ... \in Comp(s_0) \forall k \in \mathbb{N} \cdot \exists sys_0, sys_1, ..., sys_m : CD \cdot (s_i \overset{r_l \in SR}{\longrightarrow} s_{i+1} \Rightarrow sys_l \xrightarrow{r_l} sys_{l+1}) \wedge (s_k \models isValidSnapshot)$. Since $s_k \models isValidSnapshot \Leftrightarrow \exists e : ExecControl \cdot \forall sp : SnapshotPattern \cdot (e, sp) \in E_{s_k} \rightarrow isValid(sp, sys_m)$ this induces the equivalence of the first condition of Def. 10 and Theor. 1.

We will show that Condition 2 of Def. 10 and Theor. 1 is equivalent. We have $TS \models \mathbf{EF}(AtFinalState) \Leftrightarrow \exists s_0 s_1 ... \in Comp(s_0) \exists k \in \mathbb{N} : s_k \models AtFinalState \Leftrightarrow \forall sys_0 : CD \cdot \exists s_0 s_1 ... \in Comp(s_0) \exists k \in \mathbb{N} \exists sys_1, ..., sys_m : CD \cdot (s_i \overset{r_l \in SR}{\longrightarrow} s_{i+1} \Rightarrow sys_l \xrightarrow{r_l} sys_{l+1}) : s_k \models AtFinalState \Leftrightarrow \forall sys_0 : CD \cdot \exists s_0, ..., s_k \in S \cdot (s_i \rightarrow s_{i+1}) \wedge isFinalState(s_k) \wedge (s_i \overset{r_l \in SR}{\longrightarrow} s_{i+1} \Rightarrow sys_l \xrightarrow{r_l} sys_{l+1})$. This induces the equivalence Condition 2 of Def. 10 and Theor. 1. □

Our formal framework has been applied for the running example as depicted in Fig. 4 and Fig. 5: It allows us to check the conformance between use case and design models. With the case study we have defined 10 triple rules for action transitions. Due to the limited space of this paper, they are only shown in the long version of this paper as footnoted in Sect. 4.

We employ the USE tool and its extensions for the implementation. This tool allows us to animate and validate such a scenario synchronization. With USE we can present the declarative and operational activity diagrams as well-formed models since USE supports the specification of metamodels together with OCL conditions. Snapshot transitions and action transitions will be realized as operations in USE. Then, we can carry out transitions of our TS transition system and animate states as object diagrams. Currently, the process is realized in a semi-automatic way. This is suitable for designers to check their design at the early phase of the development process. For an automatic check, we plan to employ the USE feature which supports generating snapshots [23]. Then, CTL formulas can be automatically checked. This point belongs to future work.

6 Related Work

Triple Graph Grammars (TGGs) [14] have been a promising approach for explaining relationships between models, especially, bidirectional transformations. Several tools support model transformation based on TGGs such as MOFLON [12] and AToM3 [24].

Many approaches to model transformation have been introduced. ATL [11] and Kermeta [25] are well-known systems supporting transformation languages. They aim to realize the Query/View/Transformation (QVT) [13] standard for model transformation, which is proposed by the Object Management Group (OMG).

Many researches as surveyed in [26] have been attempted to introduce rigor into use case descriptions. The works in [27,28] propose viewing use cases from the different levels of abstraction. Many works focus on defining a formal semantics of use cases. They are strongly influenced by UML. The formal semantics of use cases in the works is often based on activity diagram or state charts. The works in [29,30] employ the metamodel approach in order to form a conceptual frame for use case modeling. The work in [27] proposes use case charts as an extension of activity diagram in order to define a trace-based semantics of use cases. The works in [31,32,33] propose using state charts to specify use cases. Their aim is to generate test cases from the use case specification.

The works in [22,7] propose using graph transformation to specify use cases, which are seen as activity diagrams. Those works employ the technique analyzing a critical pair of rule sequences in order to check the dependency between use case scenarios. Our work for design scenarios is similar to that work. Unlike them we employ OCL conditions in order to express action contracts.

This paper continues our proposal for the approach to use cases in [34,35]. The core of this approach is viewing use cases as a sequence of use case snapshots and using the integration of TGGs and OCL to define this sequence. The integration of TGGs and OCL is proposed in our previous work in [15,36].

7 Conclusion and Future Work

We have introduced a novel approach to explain the relation of behavioral semantics between models at different levels of abstraction. The heart of it is to analyse scenarios and scenario synchronization. We have developed a theory framework for the aim. This framework is examined with the case study concerning the relation between a use case model and a design model. It brings out a method to check the conformance between use case and design models. This work is implemented using the USE tool.

In future we continue to refine our theory framework so that we can analyse better on scenarios. Exploring triple rules as transitions of the transition system for scenario synchronization is also a focus of our future work. Besides, we will enhance the USE tool in order to obtain more support for the new tasks, especially, for checking CTL formulas.

Acknowledgments

This work was supported by Vietnam's National Foundation for Science and Technology Development (NAFOSTED). We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

1. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. OMG (November 2007)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, 1st edn. Wiley, Chichester (August 2004)
3. Kleppe, A.G.: A Language Description is More than a Metamodel. In: Fourth International Workshop on Software Language Engineering, Nashville, USA, October 1 (2007), http://planet-mde.org/atem2007/
4. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? Computer 37(10), 64–72 (2004)
5. Broy, M., Crane, M., Dingel, J., Hartman, A., Rumpe, B., Selic, B.: 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 318–323. Springer, Heidelberg (2007)
6. Gogolla, M.: (An Example for) Metamodeling Syntax and Semantics of Two Languages, their Transformation, and a Correctness Criterion. In: Bezivin, J., Heckel, R. (eds.) Proc. Dagstuhl Seminar on Language Engineering for Model-Driven Software Development (2004), http://www.dagstuhl.de/04101/
7. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven approach: a static analysis technique based on graph transformation. In: Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, Orlando, Florida, USA, May 19-25. ACM, New York (2002)
8. Evans, A., France, R.B., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 336–348. Springer, Heidelberg (1999)
9. Kelsen, P., Ma, Q.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 690–704. Springer, Heidelberg (2008)
10. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling With Uml, 1st edn. Addison-Wesley Professional, Reading (1998)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
12. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
13. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/07-07-07. OMG (2007)
14. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
15. Dang, D.H., Gogolla, M.: On Integrating OCL and Triple Graph Grammars. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 124–137. Springer, Heidelberg (2009)
16. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Professional, Reading (2004)
17. Cockburn, A.: Writing Effective Use Cases, 1st edn. Addison-Wesley Professional, Reading (2000)

18. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach, 1st edn. Addison-Wesley Professional, USA (June 1992)
19. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. In: Science of Computer Programming (2007)
20. Ehrig, H., Ermel, C., Hermann, F.: On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In: Proceedings of the Third International Workshop on Graph and Model Transformations, pp. 9–16. ACM, New York (2008)
21. Dang, D.H., Gogolla, M.: Precise Model-Driven Transformation Based on Graphs and Metamodels. In: Hung, D.V., Krishnan, P. (eds.) 7th IEEE International Conference on Software Engineering and Formal Methods, Hanoi, Vietnam, November 23-27, pp. 1–10. IEEE Computer Society Press, Los Alamitos (2009)
22. Jurack, S., Lambers, L., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 341–355. Springer, Heidelberg (2008)
23. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and System Modeling 4(4), 386–398 (2005)
24. de Lara, J., Vangheluwe, H.: AToM3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
25. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
26. Hurlbut, R.R.: A Survey of Approaches for Describing and Formalizing Use Cases. Technical Report XPT-TR-97-03, Department of Computer Science, Illinois Institute of Technology, USA (1997)
27. Whittle, J.: Specifying Precise Use Cases with Use Case Charts. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 290–301. Springer, Heidelberg (2006)
28. Regnell, B., Andersson, M., Bergstrand, J.: A Hierarchical Use Case Model with Graphical Representation. In: IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS 1996), Friedrichshafen, Germany, March 11-15, p. 270. IEEE Computer Society, Los Alamitos (1996)
29. Smialek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Complementary Use Case Scenario Representations Based on Domain Vocabularies. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 544–558. Springer, Heidelberg (2007)
30. Durán, A., Bernárdez, B., Genero, M., Piattini, M.: Empirically Driven Use Case Metamodel Evolution. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 1–11. Springer, Heidelberg (2004)
31. Sinha, A., Paradkar, A., Williams, C.: On Generating EFSM Models from Use Cases. In: ICSEW 2007: Proceedings of the 29th International Conference on Software Engineering Workshops, p. 97. IEEE Computer Society, Los Alamitos (2007)
32. Nebut, C., Fleurey, F., Traon, Y.L., Jezequel, J.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering 32(3), 140–155 (2006)

33. Grieskamp, W., Lepper, M., Schulte, W., Tillmann, N.: Testable use cases in the Abstract State Machine Language. In: Proceedings of 2nd Asia-Pacific Conference on Quality Software (APAQS 2001), Hong Kong, China, December 10-11, pp. 167–172. IEEE Computer Society, Los Alamitos (2001)
34. Dang, D.H.: Triple Graph Grammars and OCL for Validating System Behavior. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 481–483. Springer, Heidelberg (2008)
35. Dang, D.H.: Validation of System Behavior Utilizing an Integrated Semantics of Use Case and Design Models. In: Pons, C. (ed.) Proceedings of the Doctoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2007), vol. 262, pp. 1–5 (2007)
36. Gogolla, M., Bttner, F., Dang, D.H.: From Graph Transformation to OCL using USE. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 585–586. Springer, Heidelberg (2008)

Compositional Algorithms for LTL Synthesis[⋆]

Emmanuel Filiot, Nayiong Jin, and Jean-François Raskin

CS, Université Libre de Bruxelles, Belgium

Abstract. In this paper, we provide two compositional algorithms to solve safety games and apply them to provide compositional algorithms for the LTL synthesis problem. We have implemented those new compositional algorithms, and we demonstrate that they are able to handle full LTL specifications that are orders of magnitude larger than the specifications that can be treated by the current state of the art algorithms.

1 Introduction

Context and motivations. The *realizability problem* is best seen as a game between two players [14]. Given an LTL formula ϕ and a partition of its atomic propositions P into I and O, Player 1 starts by giving a subset $o_0 \subseteq O$ of propositions [1], Player 2 responds by giving a subset of propositions $i_0 \subseteq I$, then Player 1 gives o_1 and Player 2 responds by i_1, and so on. This game lasts forever and the outcome of the game is the infinite word $w = (i_0 \cup o_0)(i_1 \cup o_1)(i_2 \cup o_2) \cdots \in (2^P)^\omega$. Player 1 wins if the resulting infinite word w is a model of ϕ. The *synthesis problem* asks to produce a winning strategy for Player 1 when the LTL formula is realizable. The LTL realizability problem is central when reasoning about specifications for reactive systems and has been studied starting from the end of the eighties with the seminal works by Pnueli and Rosner [14], and Abadi, Lamport and Wolper [1]. It has been shown 2EXPTIME-C in [15].[2] Despite their high worst-case computation complexity, we believe that it is possible to solve LTL realizability and synthesis problems in practice. We proceed here along recent research efforts that have brought new algorithmic ideas to attack this important problem.

Contributions. In this paper, we propose two compositional algorithms to solve the LTL realizability and synthesis problems. Those algorithms rely on previous works where the LTL realizability problem for an LTL formula Φ is reduced to the resolution of

⋆ Work supported by the projects: (i) QUASIMODO (FP7- ICT-STREP-214755), Quasimodo: "Quantitative System Properties in Model-Driven-Design of Embedded", `http://www.quasimodo.aau.dk/`, (ii) GASICS (ESF-EUROCORES LogiCCC), Gasics: "Games for Analysis and Synthesis of Interactive Computational Systems", `http://www.ulb.ac.be/di/gasics/`, (iii) Moves: "Fundamental Issues in Modelling, Verification and Evolution of Software", `http://moves.ulb.ac.be`, a PAI program funded by the Federal Belgian Gouvernment, and (iv) ECSPER (ANR-JC09-472677) and SFINCS (ANR-07-SESU-012), two projects supported by the French National Research Agency.

1 Technically, we could have started with Player 2, for modelling reason it is conservative to start with Player 1.

2 Older pioneering works consider the realizability problem but for more expressive and computationally intractable formalisms like MSO, see [19] for pointers.

a safety game $G(\Phi)$ [7] (a similar reduction was proposed independently in [17] and applied to synthesis of distributed controllers). We show here that if the LTL specification has the form $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ i.e., a conjunction of LTL sub-specifications, then $G(\Phi)$ can be constructed and solved compositionally. The compositional algorithms are able to handle formulas that are several pages long while previous non-compositional algorithms were limited to toy examples.

The new algorithms rely on the following nice property of safety games: for any safety game G, there exists a function that maps each position of Player 1 to the set of all actions that are safe to play. We call this function the *master plan* of Player 1 in G. It encompasses all the winning strategies of Player 1. If Λ is the master plan of G then we denote by $G[\Lambda]$ the game G where the behavior of Player 1 is restricted by Λ.

To compute the winning positions of a safety game $G^{12} = G^1 \otimes G^2$ defined as the composition of two sub-games, we compute the master plans for the local components G^1 and G^2 before composition. Let Λ_1 (resp. Λ_2) be the master plan for G^1 (resp. G^2), then the winning positions in G^{12} are the same as the winning positions in $G^1[\Lambda_1] \otimes G^2[\Lambda_2]$. We develop a backward and a forward algorithms that exploit this property.

We have implemented the two compositional algorithms into our prototype Acacia and we provide an empirical evaluation of their performances on classical benchmarks and on a realistic case study taken from the IBM RuleBase tutorial[9]. This implementation is rather used to test the new concepts and to see how they behave for scalability test cases than to provide an advanced and deeply optimized prototype. In particular, our implementation is in Perl (as Lily [10]) and does not use BDDs.

Related works. The first solution [14] to the LTL realizability and synthesis problem was based on Safra's procedure for the determinization of Büchi automata [16].

Following [12], the method proposed in our paper can be coined "Safraless" approach to the realizability and synthesis of LTL as it avoids the determinization (based on the Safra's procedure) of the automaton obtained from the LTL formula. Our approach, as the one proposed in [7], relies on a reduction to safety games.

In [12], Kupferman and Vardi proposed the first Safraless approach that reduces the LTL realizability problem to Büchi games, which has been implemented in the tool Lily [10]. In [13], a compositional approach to LTL realizability and synthesis is proposed. Their algorithm is based on a Safraless approach that transforms the synthesis problem into a Büchi and not a safety game as in our case. There is no notion like the master plan for Büchi games. To the best of our knowledge, their algorithm has not been implemented.

In [3], an algorithm for the realizability problem for a fragment of LTL, known as GR(1), is presented and evaluated on the case study of [9]. The specification into the GR(1) fragment for this case study is not trivial to obtain and so the gain in term of complexity[3] comes with a cost in term of expressing the problem in the fragment. Our approach is different as we want to consider the full LTL logic. In our opinion, it is important to target full LTL as it often allows for writing more declarative and more natural specifications.

In [18], the authors also consider LTL formulas of the form $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$. They propose an algorithm to construct compositionally a parity game from such LTL specifications. Their algorithm uses a variant of Safra's determinization procedure and additionally tries to detect local parity games that are equivalent to safety games

[3] GR(1) has a better worst-case complexity than full LTL.

(because the associated LTL subformula is a safety property). For efficiently solving the entire game, they use BDDs.

In [11], a compositional algorithm is proposed for reasoning about network of components to control under partial observability. The class of properties that they consider is safety properties and not LTL properties. They propose a backward algorithm and no forward algorithm.

The implementation supporting the approaches described in [18] and [3] uses BDDs while our tool Acacia does not. While our algorithms could have been implemented with BDDs, we deliberately decided not to use them for two reasons. First, to fairly compare our Safraless approach with the one proposed in [12] and implemented in Lily, we needed to exclude BDDs as Lily does not use them. Second, several recent works on the efficient implementation of decision problems for automata shown that antichain based algorithms may outperform by several order of magnitude BDD implementations, see [5,6] for more details.

2 Safety Games

In this section, we provide a definition of safety games that is well-suited to support our compositional methods detailed in the following sections. Player 1 will play the role of the system while Player 2 will play the role of the environment. This is why, as the reader will see, our definition of games is asymmetric.

Turn-based games. A *turn-based game* on a finite set of moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$ such that $\mathsf{Moves}_2 \neq \varnothing$ is a tuple $G = (S_1, S_2, \Gamma_1, \Delta_1, \Delta_2)$ where: (i) S_1 is the set of Player 1 positions, S_2 is the set of Player 2 positions, $S_1 \cap S_2 = \varnothing$, we let $S = S_1 \uplus S_2$. (ii) $\Gamma_1 : S_1 \to 2^{\mathsf{Moves}_1}$ is a function that assigns to each position of Player 1 the subset of moves that are available in that position. For Player 2, we assume that all the moves in Moves_2 are available in all the positions $s \in S_2$. (iii) $\Delta_1 : S_1 \times \mathsf{Moves}_1 \to S_2$ is a partial function, defined on pairs (s, m) when Player 1 chooses $m \in \Gamma_1(s)$, that maps (s, m) to the position reached from s. $\Delta_2 : S_2 \times \mathsf{Moves}_2 \to S_1$ is a function that maps (s, m) to the state reached from s when Player 2 chooses m.

We define the partial function Δ as the union of the partial function Δ_1 and the function Δ_2. Unless stated otherwise, we fix for the sequel of this section a turn-based game $G = (S_1, S_2, \Gamma_1, \Delta_1, \Delta_2)$ on moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$.

Given a function $\Lambda : S_1 \to 2^{\mathsf{Moves}_1}$, the *restriction of* G *by* Λ is the game $G[\Lambda] = (S_1, S_2, \widehat{\Gamma_1}, \widehat{\Delta_1}, \Delta_2)$ where for all $s \in S_1$, $\widehat{\Gamma_1}(s) = \Gamma_1(s) \cap \Lambda(s)$ and $\widehat{\Delta_1}$ equals Δ_1 on the domain restricted to the pairs $\{(s, m) \mid s \in S_1 \wedge m \in \widehat{\Gamma_1}(s)\}$ i.e., $G[\Lambda]$ is as G but with the moves of Player 1 restricted by Λ.

Rules of the game. The game on G is played in rounds and generates a finite or an infinite sequence of positions that we call a *play*. In the initial round, the game is in some position, say s_0, and we assume that Player 1 owns that position. Then if $\Gamma_1(s_0)$ is non-empty Player 1 chooses a move $m_0 \in \Gamma_1(s_0)$, and the game evolves to state $s_1 = \Delta_1(s_0, m_0)$, otherwise the game stops. If the game does not stop then the next round starts in s_1. Player 2 chooses a move $m_1 \in \mathsf{Moves}_2$ and the game proceeds to position $s_2 = \Delta_2(s_1, m_1)$. The game proceeds accordingly either for an infinite number of rounds or it stops when a position $s \in S_1$ is reached such that $\Gamma_1(s) = \varnothing$. Player 1 wins if the game does not stop otherwise Player 2 wins (safety winning condition). Our variant of safety games are thus zero-sum games as usual. In particular, the positions $s \in S_1$ such that $\mathit{\Gamma}_1(s) \neq \varnothing$ are the safe positions of Player 1.

Plays and strategies. We now define formally the notions of play, strategy, outcome of a strategy and winning strategies. Given a sequence $\rho = s_0 s_1 \dots s_n \dots \in S^* \cup S^\omega$, we denote by $|\rho|$ its length (which is equal to ω if ρ is infinite). We denote by $\mathsf{first}(\rho)$ the first element of ρ, and if ρ is finite, we denote by $\mathsf{last}(\rho)$ its last element.

A *play* in G is a finite or infinite sequence of positions $\rho = s_0 s_1 \dots s_n \dots \in S^* \cup S^\omega$ such that : (i) if ρ is finite then $\mathsf{last}(\rho) \in S_1$ and $\mathit{\Gamma}_1(\mathsf{last}(\rho)) = \varnothing$; (ii) ρ is consistent with the moves and transitions of G i.e., for all i, $0 \leq i \leq |\rho|$, we have that $s_{i+1} = \mathit{\Delta}(s_i, m)$ for some $m \in \mathit{\Gamma}_1(s_i)$ if $s \in S_1$, or $m \in \mathsf{Moves}_2$ if $s \in S_2$.We denote by $\mathsf{Plays}(G)$ the set of plays in G.

Given a set of finite or infinite sequences $L \subseteq S^* \cup S^\omega$, we write $\mathsf{Pref}_j(L)$, $j \in \{1, 2\}$, for the set of prefixes of sequences in L that end up in a position of Player j. Let $\perp$ be such that $\perp \notin \mathsf{Moves}$. A *strategy for Player 1* in G is a function $\lambda_1 : \mathsf{Pref}_1(\mathsf{Plays}(G)) \rightarrow \mathsf{Moves}_1 \cup \{\perp\}$ which is consistent with the set of available moves i.e., for all $\rho \in \mathsf{Pref}_i(\mathsf{Plays}(G))$, we have that: (i) $\lambda_1(\rho) \in \mathit{\Gamma}_1(\mathsf{last}(\rho)) \cup \{\perp\}$, and (ii) $\lambda_1(\rho) = \perp$ only if $\mathit{\Gamma}_1(\mathsf{last}(\rho)) = \varnothing$. A *strategy for Player 2* in G is a function $\lambda_2 : \mathsf{Pref}_2(\mathsf{Plays}(G)) \rightarrow \mathsf{Moves}_2$. Note that the codomain of a Player 2's strategy never contains $\perp$ as all the moves of Player 2 are allowed at any position, whereas the moves of Player 1 are restricted by $\mathit{\Gamma}_1$.

A play $\rho = s_0 s_1 \dots s_n \dots \in \mathsf{Plays}(G)$ is *compatible* with a strategy λ_j of Player j $(j \in \{1, 2\})$, if for all i, $0 \leq i < |\rho|$, if $s_i \in S_j$ then $s_{i+1} = \mathit{\Delta}_j(s_i, \lambda_j(s_0 s_1 \dots s_i))$.We denote by $\mathsf{outcome}(G, s, \lambda_j)$ the subset of plays in $\mathsf{Plays}(G)$ that are compatible with the strategy λ_j of Player j, and that start in s. We denote by $\mathsf{outcome}(G, \lambda_j)$ the set $\bigcup_{s \in S} \mathsf{outcome}(G, s, \lambda_j)$, and by $\mathsf{outcome}(G, s, \lambda_1, \lambda_2)$ the unique play that is compatible with both λ_1 and λ_2, and starts in s.

The *winning* plays for Player 1 are those that are infinite i.e., $\mathsf{Win}_1(G) = \mathsf{Plays}(G) \cap S^\omega$, or equivalently those that never reach an unsafe position $s \in S_1$ of Player 1 where $\mathit{\Gamma}_1(s) = \varnothing$. A strategy λ_1 is *winning* in G from s_{ini} iff $\mathsf{outcome}(G, s_{\mathsf{ini}}, \lambda_1) \subseteq \mathsf{Win}_1(G)$. A game with such a winning condition in mind is called *safety game*. We denote by $\mathsf{WinPos}_1(G)$ the subset of positions $s \in S$ in G for which there exists λ_1 such that $\mathsf{outcome}(G, s, \lambda_1) \subseteq \mathsf{Win}_1(G)$.

Games with initial position. A *safety game with initial position* is a pair (G, s_{ini}) where $s_{\mathsf{ini}} \in S_1 \cup S_2$ is a position of the game structure G called the *initial position*. The set of plays in (G, s_{ini}) are the plays of G starting in s_{ini}, i.e. $\mathsf{Plays}(G, s_{\mathsf{ini}}) = \mathsf{Plays}(G) \cap s_{\mathsf{ini}} \cdot (S^* \cup S^\omega)$. All the previous notions carry over to games with initial positions.

Solving safety games. The classical fixpoint algorithm to solve safety games relies on iterating the following monotone operator over sets of game positions. Let $X \subseteq S_1 \uplus S_2$:

$$\mathsf{CPre}_1(X) = \{s \in S_1 \mid \exists m \in \mathit{\Gamma}_1(s), \mathit{\Delta}_1(s, m) \in X\} \cup \{s \in S_2 \mid \forall m \in \mathsf{Moves}_2, \mathit{\Delta}_2(s, m) \in X\}$$

i.e., $\mathsf{CPre}_1(X)$ contains all the positions $s \in S_1$ from which Player 1 can force X in one step, and all the positions $s \in S_2$ where Player 2 cannot avoid X in one step. Now, we define the following sequence of subsets of positions:

$$W_0 = \{s \in S_1 \mid \Gamma_1(s) \neq \varnothing\} \cup S_2 \qquad W_i = W_{i-1} \cap \mathsf{CPre}(W_{i-1}) \text{ for all } i \geq 1$$

Denote by $W^\natural$ the fixpoint of this sequence. It is well known that $W^\natural = \mathsf{WinPos}_1(G)$.

Master plan. Let $\Lambda_1 : S_1 \to 2^{\mathsf{Moves}_1}$ be defined as follows: for all $s \in S_1$, $\Lambda_1(s) = \{m \in \Gamma_1(s) \mid \Delta_1(s,m) \in W^\natural\}$ i.e., $\Lambda_1(s)$ contains all the moves that Player 1 can play in s in order to win the safety game. We call Λ_1 the *master plan* of Player 1 and we write it $\mathsf{MP}(G)$. The following lemma states that $\mathsf{MP}(G)$ can be interpreted as a compact representation of all the winning strategies of Player 1 in the game G:

Lemma 1. *For all strategies λ_1 of Player 1 in G, for all $s \in S$, λ_1 is winning in G from s iff λ_1 is a strategy in $(G[\mathsf{MP}(G)], s)$ and $\lambda_1(s) \neq \perp$.*

Now that we have defined and characterized the notion of master plan, we show that we can compute directly the master plan associated with a game using a variant of the CPre operator and sequence W. The variant of CPre considers the effect of some Player 1's move followed by some Player 2's move. Let $\widehat{\mathsf{CPre}} : (S_1 \to 2^{\mathsf{Moves}_1}) \to (S_1 \to 2^{\mathsf{Moves}_1})$ be defined as follows. For all $s \in S_1$, let:

$$\widehat{\mathsf{CPre}}(\Lambda)(s) = \{m \in \Lambda(s) \mid \forall m' \in \mathsf{Moves}_2 : \Lambda(\Delta_2(\Delta_1(s,m), m')) \neq \varnothing\}$$

Consider the following sequence of functions: $\Lambda_0 = \Gamma_1$, and $\Lambda_i = \widehat{\mathsf{CPre}}(\Lambda_{i-1})$, $i \geq 1$. This sequence stabilizes after at most $O(|S|)$ iterations and we denote by $\Lambda^\natural$ the function on which the sequence stabilizes. Clearly, the value on which the sequence stabilizes corresponds exactly to the master plan of G:

Theorem 1. $\Lambda^\natural = \mathsf{MP}(G)$.

3 From LTL Realizability to Safety Games

In this section, after recalling the formal definition of the LTL realizability problem, we recall the essential results of [17,7] where it is shown how to reduce the LTL realizability problem to a safety game problem.

Linear Temporal Logic (*LTL*). The formulas of LTL are defined over a set of atomic propositions P. The syntax is given by: $\phi ::= p \mid \phi \vee \phi \mid \neg\phi \mid \mathcal{X}\phi \mid \phi\mathcal{U}\phi$ with $p \in P$. The notations true, false, $\phi_1 \wedge \phi_2$, $\Diamond\phi$ and $\Box\phi$ are defined as usual. LTL formulas ϕ are interpreted on infinite words $w = \sigma_0\sigma_1\sigma_2 \ldots \in (2^P)^\omega$ via a satisfaction relation $w \models \phi$ inductively defined as follows: (i) $w \models p$ if $p \in \sigma_0$, (ii) $w \models \phi_1 \vee \phi_2$ if $w \models \phi_1$ or $w \models \phi_2$, (iii) $w \models \neg\phi$ if $w \not\models \phi$, (iv) $w \models \mathcal{X}\phi$ if $\sigma_1\sigma_2 \ldots \models \phi$, and (v) $w \models \phi_1\mathcal{U}\phi_2$ if there is $n \geq 0$ such that $\sigma_n\sigma_{n+1} \ldots \models \phi_2$ and for all $0 \leq i < n$, $\sigma_i\sigma_{i+1} \ldots \models \phi_1$.

LTL ***Realizability and Synthesis.*** Let P be a finite set of propositions. Unless otherwise stated, we partition P into I the set of *input signals* controlled by Player 2 (the environment), and O the set of *output signals* controlled by Player 1 (the controller). We let $\Sigma = 2^P$, $\Sigma_I = 2^I$, and $\Sigma_O = 2^O$. The realizability problem is best seen as a game. The players play according to strategies. A strategy for Player 1 is a (total) mapping $\lambda_1 : (\Sigma_O\Sigma_I)^* \to \Sigma_O$ while a strategy for Player 2 is a (total) mapping

$\lambda_2 : \Sigma_O(\Sigma_I\Sigma_O)^* \to \Sigma_I$. The outcome of λ_1 and λ_2 is the word $\mathsf{outcome}(\lambda_1, \lambda_2) = (o_0 \cup i_0)(o_1 \cup i_1)\dots$ such that for all $j \geq 0$, $o_j = \lambda_1(o_0 i_0 \dots o_{j-1} i_{j-1})$ and $i_j = \lambda_2(o_0 i_0 \dots o_{j-1} i_{j-1} o_j)$. In particular, $o_0 = \lambda_1(\epsilon)$ and $i_0 = \lambda_2(o_0)$. Given an LTL formula ϕ, the *realizability problem* is to decide whether there exists a strategy λ_1 of Player 1 such that for all strategies λ_2 of Player 2, $\mathsf{outcome}(\lambda_1, \lambda_2) \models \phi$. If such a strategy exists, we say that the specification ϕ is *realizable*. If an LTL specification is realizable, there exists a finite-state strategy that realizes it [14]. The *synthesis problem* is to find a finite-state strategy that realizes the LTL specification.

Universal CoBüchi automata. LTL formulas are associated with *turn-based automaton* A over Σ_I and Σ_O. A turn-based automaton is a tuple $A = (\Sigma_I, \Sigma_O, Q_I, Q_O, Q_{\mathsf{ini}}, \alpha, \delta_I, \delta_O)$ where Q_I, Q_O are finite sets of input and output states respectively, $Q_{\mathsf{ini}} \subseteq Q_O$ is the set of initial states, $\alpha \subseteq Q_I \cup Q_O$ is the set of final states, and $\delta_I \subseteq Q_I \times \Sigma_I \times Q_O$, $\delta_O \subseteq Q_O \times \Sigma_O \times Q_I$ are the input and output transition relations respectively. Wlog we assume that the automata are *complete*, i.e. for all $t \in \{I, O\}$, all $q \in Q_t$ and all $\sigma \in \Sigma_t$, $\delta_t(q, \sigma) \neq \varnothing$. Turn-based automata A run on words $w = (o_0 \cup i_0)(o_1 \cup i_1) \dots \in \Sigma^\omega$ as follows: a *run* on w is a word $\rho = \rho_0\rho_1 \dots \in (Q_O Q_I)^\omega$ such that $\rho_0 \in Q_{\mathsf{ini}}$ and for all $j \geq 0$, $(\rho_{2j}, o_j, \rho_{2j+1}) \in \delta_O$ and $(\rho_{2j+1}, i_j, \rho_{2j+2}) \in \delta_I$. Let $K \in \mathbb{N}$. We consider the universal co-Büchi (resp. K-co-Büchi) accepting condition, for which a word w is accepted iff any run on w visits finitely many (resp. at most K) accepting states. With the K-co-Büchi interpretation in mind, we say that (A, K) is a universal K-co-Büchi turn-based automaton. We denote by $L_{\mathsf{uc}}(A)$ and $L_{\mathsf{uc},K}(A)$ the languages accepted by A with these two accepting conditions resp. Turn-based automata with universal co-Büchi and K-co-Büchi acceptance conditions are denoted by UCW and $\mathsf{U}K\mathsf{CW}$. As they define set of infinite words, they can be taken as input to the realizability problem.

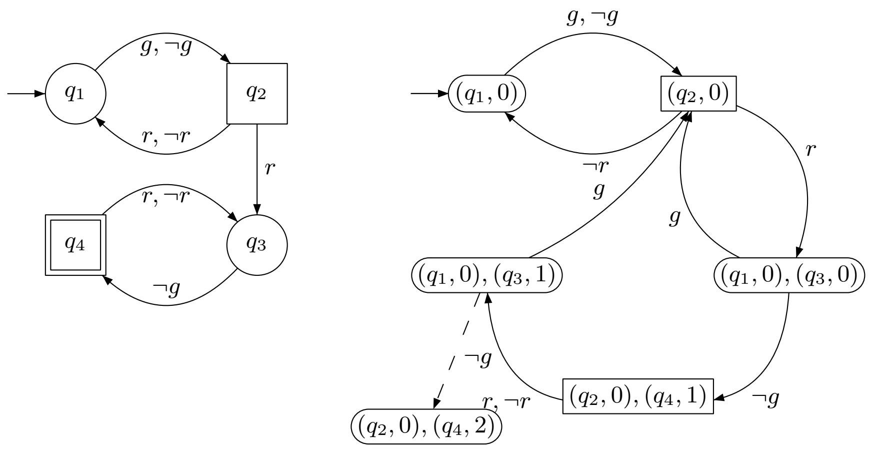
It is known that for any LTL formula one can construct an equivalent UCW A_ϕ (possibly exponentially larger) [20]. Fig. 1(a) represents a UCW equivalent to the formula $\Box(r \to \mathcal{X}(\Diamond g))$, where r is an input signal and g is an output signal. States of Q_O are denoted by circles while states of Q_I are denoted by squares. The transitions on missing letters are going to an additional sink non-accepting state that we do not represent for the sake of readability. If a request r is never granted, then a run will visit the accepting state q_4 infinitely often.

The realizability problem can be reduced from a UCW to a $\mathsf{U}K\mathsf{CW}$ specification:

Theorem 2 ([17,7]). *Let A be a* UCW *over Σ_I, Σ_O with n states and $K = 2n(n^{2n+2} + 1)$. Then A is realizable iff (A, K) is realizable.*

Let us recall the intuition behind the correctness of this result. First, if the specification (A, K) is realizable then clearly the specification A is also realizable as $L_{\mathsf{uc},K}(A) \subseteq L_{\mathsf{uc}}(A)$. Second, if the specification A is realizable then we know that there exists a finite memory strategy λ_1 that realizes it [14]. Any run on any outcome of λ_1 visits accepting states only a number of time equal to K, which is bounded by the size of the strategy. So λ_1 not only realizes the specification A but a stronger specification (A, K).

Reduction to safety game. Clearly $\mathsf{U}K\mathsf{CW}$ specifications are safety properties. The reduction to a safety game relies on the fact that $\mathsf{U}K\mathsf{CW}$ can easily be made deterministic. Given a $\mathsf{U}K\mathsf{CW}$ A, the game $G(A, K)$ is constructed via a subset construction extended with counters for each state q, that count (up to $K + 1$) the maximal number of accepting states which have been visited by runs ending up in q. We set the counter of a state q to -1 when no run on the prefix read so far ends up in q. The set of game positions



(a) UCW (b) Safety game $G(\phi, 1)$

Fig. 1. UCW and safety game for the formula $\phi \equiv \Box(r \rightarrow \mathcal{X}\Diamond g)$

S_1 for Player 1 is therefore the set of functions $F : Q_O$ to $\{-1, \dots, K+1\}$. The set S_2 is similarly defined as the functions $F : Q_I$ to $\{-1, \dots, K+1\}$. The set of moves of both players are the letters they can choose, i.e. $\mathsf{Moves}_1 = \Sigma_O$ and $\mathsf{Moves}_2 = \Sigma_I$. The set of available moves in a position are defined via a successor function succ such that for all $F \in S_i$ and $\sigma \in \mathsf{Moves}_i$,

$$\mathsf{succ}(F, \sigma) = q \mapsto \max\{\min(K+1, F(p) + (q \in \alpha)) \mid q \in \delta(p, \sigma), F(p) \neq -1\}$$

where $\max \varnothing = -1$, and $(q \in \alpha) = 1$ if q is in α, and 0 otherwise. An action $\sigma_1 \in \mathsf{Moves}_1$ is available for Player 1 in a position $F \in S_1$ if the counters of F and $\mathsf{succ}(F, \sigma)$ do not exceed K. More formally, $\sigma \in \Gamma_1(F)$ iff for all $p \in Q_0$ and all $q \in Q_I$, $F(p) \leq K$ and $\mathsf{succ}(F, \sigma)(q) \leq K$. The transition function Δ_1 is defined by $\Delta_1(F, \sigma) = \mathsf{succ}(F, \sigma)$ for all $F \in S_1$ and all $\sigma \in \Gamma_1(s)$. The function Δ_2 is defined by $\Delta_2(F, \sigma) = \mathsf{succ}(F, \sigma)$ for all $F \in S_2$ and all $\sigma \in \mathsf{Moves}_2$. Finally, we start the game in the initial position $F_0 \in S_1$ such that for all $q \in Q_O$, $F(q) = -1$ if q is not initial, and 0 if q is initial but not final, and 1 if q is initial and final.

Associating a safety game with an LTL formula ϕ is done as follows: (1) construct a UCW A_ϕ equivalent to ϕ, (2) construct $G(A_\phi, K)$, denoted as $G(\psi, K)$ in the sequel, where $K = 2n(n^{2n+2} + 1)$ and n is the number of states of A_ϕ.

Incremental algorithm. In practice, for checking the existence of a winning strategy for Player 1 in the safety game, we rely on an incremental approach. For all $K_1, K_2 \cdot 0 \leq K_1 \leq K_2$, if Player 1 can win $G(A, K_1)$, then she can win $G(A, K_2)$. This is because $L_{uc,K_1}(A) \subseteq L_{uc,K_2}(A) \subseteq L_{uc}(A)$. Therefore we can test the existence of strategies for increasing values of K. In all our examples (see Section 6), the smallest K for which Player 1 can win is very small (less than 5).

Example. Fig. 1(b) represents the safety game (for $K = 1$) associated with the formula $\Box(r \rightarrow \mathcal{X}\Diamond g)$. Positions are pairs of states of the UCW with their counter values. Player

1's positions are denoted by circles while Player 2's positions are denoted by squares. The unavailable move of Player 1 from position $(q_2, 0)$ is denoted by a dashed arrow. It goes to a position where a counter exceeds the value K. The master plan of the game corresponds in this case to all the moves attached to plain arrows for Player 1's positions. Indeed Player 1 wins the game iff she never follows the dashed arrow.

Antichain-based symbolic algorithm. In practice, we do not construct the game $G(A, K)$ explicitly, as it may be too large. However it has a nice structure that can be exploited for defining an efficient symbolic implementation for the computation of the sequence of W_i's defined in Section 2. The main idea is to consider an ordering on the positions in $G(A, K)$. Define the relation $\preceq \subseteq \mathcal{F}_I \times \mathcal{F}_I \cup \mathcal{F}_O \times \mathcal{F}_O$ by $F \preceq F'$ iff $\forall q$, $F(q) \leq F'(q)$. It is clear that $\preceq$ is a partial order. Intuitively, if Player 1 can win from F' then she can also win from all $F \preceq F'$, since she has seen less accepting states in F than in F'. The consequence of this observation is that all the sets W_i are downward closed for the relation $\preceq$. It is shown in [7] that consequently all the computations can be done efficiently by manipulating only $\preceq$-maximal elements.

4 Compositional Safety Games

In this section, we define compositional safety games and develop two abstract compositional algorithms to solve such games.

Composition of safety games. We now consider products of safety games. Let G^i, $i \in \{1, \ldots, n\}$, be n safety games $G^i = (S_1^i, S_2^i, \Gamma_1^i, \Delta_1^i, \Delta_2^i)$ defined on the same sets of moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$. Their product, denoted by $\otimes_{i=1}^{i=n} G^i$, is the safety game $G^{\otimes} = (S_1^{\otimes}, S_2^{\otimes}, \Gamma_1^{\otimes}, \Delta_1^{\otimes}, \Delta_2^{\otimes})$[4] defined as follows:

- $S_j^{\otimes} = S_j^1 \times S_j^2 \times \cdots \times S_j^n$, $j = 1, 2$;
- for $s = (s^1, s^2, \ldots, s^n) \in S_1^{\otimes}$, $\Gamma_1^{\otimes}(s) = \Gamma_1^1(s^1) \cap \Gamma_1^2(s^2) \cap \cdots \cap \Gamma_1^n(s^n)$;
- for $j \in \{1, 2\}$ and $s = (s^1, s^2, \ldots, s^n) \in S_j^{\otimes}$, let $m \in \Gamma_1^{\otimes}(s)$ if $j = 1$ or $m \in \mathsf{Moves}_2$ if $j = 2$. Then $\Delta_j^{\otimes}(s) = (t^1, t^2, \ldots, t^n)$, where $t^i = \Delta_j^i(s^i, m)$ for all $i \in \{1, 2, \ldots, n\}$;

Backward compositional reasoning. We now define a backward compositional algorithm to solve the safety game $G^{\otimes}$. The correctness of this algorithm is justified by the following lemmas. For readability, we express the properties for composed games defined from two components. All the properties generalize to any number of components. The first part of the lemma states that to compute the master plan of a composition, we can first reduce each component to its *local* master plan. The second part of the lemma states that the master plan of a component is the master plan of the component where the choices of Player 1 has been restricted by one application of the $\widehat{\mathsf{CPre}}$ operator.

Lemma 2. (a) *Let $G^{12} = G^1 \otimes G^2$, let $\Lambda_1 = \mathsf{MP}(G^1)$ and $\Lambda_2 = \mathsf{MP}(G^2)$ then*

$$\mathsf{MP}(G^{12}) = \mathsf{MP}(G^1[\Lambda_1] \otimes G^2[\Lambda_2])$$

(b) *For any game G,* $\mathsf{MP}(G)$=$\mathsf{MP}(G[\widehat{\mathsf{CPre}}(\Gamma_1)])$.

[4] Clearly, the product operation is associative up to isomorphism.

Let $\Lambda : S_1^1 \times S_1^2 \times \cdots \times S_1^n \rightarrow 2^{\mathsf{Moves}}$, we let $\pi_i(\Lambda)$ the function with domain S_1^i and codomain 2^{Moves_1} such that for all $s \in S_1^i$, $\pi_i(\Lambda)(s)$ is the set of moves allowed by Λ in one tuple $(s^1, s^2, \ldots, s^n)$ such that $s^i = s$. Formally, $\pi_i(\Lambda)(s) = \bigcup\{\Lambda(s^1, s^2, \ldots, s^n) \mid (s^1, s^2, \ldots, s^n) \in S_1^{\otimes},\ s^i = s\}$. Given two functions $\Lambda_1 : S_1 \rightarrow 2^{\mathsf{Moves}_1}$ and $\Lambda_2 : S_1 \rightarrow 2^{\mathsf{Moves}_1}$, we define $\Lambda_1 \cap \Lambda_2$ as the function on domain S_1 such that for all $s \in S_1$: $\Lambda_1 \cap \Lambda_2(s) = \Lambda_1(s) \cap \Lambda_2(s)$. Given two functions $\Lambda_1 : S_1 \rightarrow 2^{\mathsf{Moves}_1}$ and $\Lambda_2 : S_2 \rightarrow 2^{\mathsf{Moves}_1}$, we define $(\Lambda_1 \times \Lambda_2) : S_1 \times S_2 \rightarrow 2^{\mathsf{Moves}_1}$ as $(\Lambda_1 \times \Lambda_2)(s_1, s_2) = \Lambda_1(s_1) \cap \Lambda_2(s_2)$.

Based on Lemma 2, we propose the following compositional algorithm to compute the master plan of a safety game defined as the composition of local safety games. First, compute locally the master plans of the components. Then compose the local master plans and apply one time the $\widehat{\mathsf{CPre}}$ operator to this composition. This application of $\widehat{\mathsf{CPre}}$ compute a new function Λ that contains information about the one-step inconsistencies between local master plans. Project back on the local components the information gained by the function Λ, and iterate. This is formalized in Algorithm 1 whose correctness is asserted by Theorem 3.

Algorithm 1. Backward composition

```
Data: $G^{\otimes} = G^1 \otimes G^2 \otimes \cdots \otimes G^n$
$\Lambda \leftarrow \Gamma_1^{\otimes}$;
repeat
    $\Lambda^i := \mathsf{MP}(G^i[\pi_i(\Lambda)]), 1 \leq i \leq n$;
    $\Lambda := \widehat{\mathsf{CPre}}(\Lambda \cap (\Lambda^1 \times \cdots \times \Lambda^n))$
until $\Lambda$ does not change;
return $\Lambda$
```

Algorithm 2. Forward composition

```
Data: $G^{\otimes} = G^1 \otimes G^2 \otimes \cdots \otimes G^n$
$\Lambda^i := \mathsf{MP}_{\mathsf{Reach}}(G^i, s^i_{\mathsf{ini}}), 1 \leq i \leq n$;
$\Lambda := \mathsf{MP}_{\mathsf{Reach}}(G^1[\Lambda^1] \otimes \cdots \otimes G^n[\Lambda^n],$
        $(s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}}, \ldots, s^n_{\mathsf{ini}}))$
return $\Lambda$
```

Theorem 3. *The value Λ returned by Algorithm 1 is equal to* $\mathsf{MP}(G^{\otimes})$.

Forward compositional reasoning. When solving safety games, we may be interested only in computing winning strategies for a fixed starting position, say s_{ini}. In this case, the value of the master plan is not useful for positions that are not reachable when playing winning strategies from s_{ini}. So, we are interested in computing a master plan only for the winning and *reachable* positions. Given a game G and a state s_{ini}, we denote by $\mathsf{Reach}(G, s_{\mathsf{ini}})$ the subset of positions that are reachable from s_{ini} in G i.e., the states s' such that there exists a finite sequence $s_0 s_1 \ldots s_n$ with $s_0 = s_{\mathsf{ini}}$, $s_n = s'$ and for all i, $0 \leq i < n$, there exists $m \in \Gamma_1(s_i) \cup \mathsf{Moves}_2$ such that $s_{i+1} = \Delta(s_i, m)$. The *master plan of reachable positions* for (G, s_{ini}), denoted by $\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})$ is defined for all $s \in S$ as follows:

$$\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})(s) = \begin{cases} \mathsf{MP}(G)(s) & \text{if } s \in \mathsf{Reach}(G[\Lambda], s_{\mathsf{ini}}) \\ \varnothing & \text{otherwise.} \end{cases}$$

The following lemma shows that for a game defined compositionally, its master plan can also be defined compositionally. For readability we express the lemma only for two components but, as for the previous lemmas, it extends to any number of components:

Lemma 3. *Let* $\Lambda_1 = \mathsf{MP}_{\mathsf{Reach}}(G^1, s^1_{\mathsf{ini}})$ *and* $\Lambda_2 = \mathsf{MP}_{\mathsf{Reach}}(G^2, s^2_{\mathsf{ini}})$.

$$\mathsf{MP}_{\mathsf{Reach}}(G^1 \otimes G^2, (s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}})) = \mathsf{MP}_{\mathsf{Reach}}(G^1[\Lambda_1] \otimes G^2[\Lambda_2], (s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}}))$$

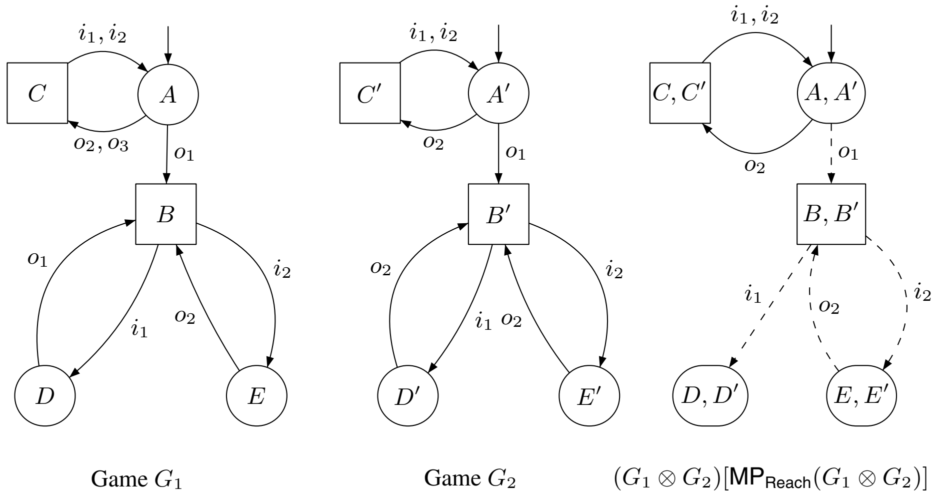


Fig. 2. Two games and their common master plan of reachable states

As composition of safety games is an associative operator, we can use variants of the algorithm above where we first compose some of the components and compute their master plan of reachable positions before doing the global composition.

To efficiently compute the master plan of reachable positions of a game G, we use the OTFUR algorithm of [4]. We say that a position $s \in S_1$ is *unsafe* if $\varGamma_1(s) = \varnothing$, or all its successors are unsafe. A position $s \in S_2$ is unsafe if one of its successors is unsafe. The algorithm explores the state space by starting from the initial state in a forward fashion. When sufficient information is known about the successors of a position s, it back-propagates the unsafe information to s. At the end of the algorithm, the master plan which allows all moves that lead to a safe position is exactly $\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})$. Fig. 2 illustrates the result of the OTFUR algorithms applied on the product of two safety games G_1, G_2 over the possible moves o_1, o_2, o_3 for Player 1 and i_1, i_2 for Player 2. We assume that G_1, G_2 contains only winning actions, i.e. $G_i = G_i[\mathsf{MP}(G_i)]$ for all $i = 1, 2$. The master plan of reachable states for $G_1 \otimes G_2$ corresponds to plain arrows. Dashed arrows are those which have been traversed during the OTFUR algorithm but have been removed due to backpropagation of unsafe information. From node $\langle A, A' \rangle$ the move o_3 is not a common move, therefore o_3 is not available in the product as well. However o_2 is available in both games and leads to C and C' respectively. Similarly, o_1 is available in both games and goes to $\langle B, B' \rangle$. From $\langle B, B' \rangle$ one can reach $\langle D, D' \rangle$ by i_1 but from $\langle D, D' \rangle$ there is no common action. Therefore $\langle D, D' \rangle$ is unsafe. Since one of the successor of $\langle B, B' \rangle$ is unsafe and $\langle B, B' \rangle$ is owned by Player 2, $\langle B, B' \rangle$ is declared to be unsafe as well. All the remaining moves are winning in the $G_1 \otimes G_2$, as they are winning both in G_1 and G_2.

Remark 1. It should be noted that each Λ^i in Alg. 2 can be replaced by the full *master plan* without changing the output of the forward algorithm. Indeed, it is easy to see that $\mathsf{Reach}(G[\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})], s_{\mathsf{ini}}) = \mathsf{Reach}(G[\mathsf{MP}(G)], s_{\mathsf{ini}})$. So, we can mix the backward and forward algorithms. For instance, we can compute locally the master plan of each G^i using the backward algorithm of [7], and then check global realizability using the OTFUR algorithm.

5 Compositional LTL Synthesis and Dropping Assumptions

In this section, we show how to define compositionally the safety game associated with an LTL formula when this formula is given as a conjunction of subformulas i.e., $\psi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$. Assume from now on that we have fixed some $K \in \mathbb{N}$. We first construct for each subformula ϕ_i the corresponding U*K*CW A_{ϕ_i} on the alphabet of ψ[5], and their associated safety games $G(\phi_i, K)$. The game $G(\psi, K)$ for the conjunction ψ is isomorphic to the game $\otimes_{i=1}^{i=n} G(\phi_i, K)$.

To establish this result, we rely on a notion of product at the level of turn-based automata. Let $A_i = (\Sigma_I, \Sigma_O, Q_I^i, Q_O^i, q_0^i, \alpha^i, \delta_I^i, \delta_O^i)$ for $i \in \{1,2\}$ be two turn-based automata, then their product $A_1 \otimes A_2$ is the turn-based automaton defined as $(\Sigma_I, \Sigma_O, Q_I^1 \uplus Q_I^2, Q_O^1 \uplus Q_O^2, Q_{\mathsf{ini}}^1 \uplus Q_{\mathsf{ini}}^2, \alpha_1 \uplus \alpha_2, \delta_I^1 \uplus \delta_I^2, \delta_O^1 \uplus \delta_O^2)$. As we use universal interpretation i.e., we require all runs to respect the accepting condition, it is clear that executing the $A_1 \otimes A_2$ on a word w is equivalent to execute both A_1 and A_2 on this word. So w is accepted by the product iff it is accepted by each of the automata.

Proposition 1. *Let A_1 and A_2 be two* UCW *on the alphabet $\Sigma_1 \uplus \Sigma_2$, and $K \in \mathbb{N}$: (i) $L_{uc}(A_1 \otimes A_2) = L_{uc}(A_1) \cap L_{uc}(A_2)$, (ii) $L_{uc,K}(A_1 \otimes A_2) = L_{uc,K}(A_1) \cap L_{uc,K}(A_2)$*

As the state space and transition relation of $A_1 \otimes A_2$ is the disjunct union of the space spaces and transition relations of A_1 and A_2, the determinization of $A_1 \otimes A_2$ for a fixed $K \in \mathbb{N}$ is equivalent to the synchronized product of the determinizations of A_1 and A_2 for that K, and so we get the following theorem.

Theorem 4. *Let $\psi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, $K \in \mathbb{N}$, $G(\psi, K)$ is isomorphic to $\otimes_{i=1}^{i=n} G(\phi_i, K)$.*

Even if it is natural to write large LTL specifications as conjunctions of subformulas, it is also sometimes convenient to write specifications that are of the form $(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow (\bigwedge_{j=1}^{j=m} \phi_j)$ where ψ_i's formalize a set of assumptions made on the environment (Player 2) and ϕ_j's formalize a set of guarantees that the system (Player 1) must enforce. In this case, we rewrite the formula into the logical equivalent formula $\bigwedge_{j=1}^{j=m}((\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j)$ which is a conjunction of LTL formulas as needed for the compositional construction described above. As logical equivalence is maintained, realizability is maintained as well.

Unfortunately, this formula is larger than the original formula as all the n assumptions are duplicated for all the m guarantees. But, the subformulas $(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j, j \in \{1, \ldots, m\}$ are usually such that to guarantee ϕ_j, Player 1 does not need all the assumptions on the left of the implication. It is thus tempting to remove those assumptions that are *locally* unnecessary in order to get smaller local formulas. In practice, we apply the following rule. Let $\psi_1 \wedge \psi_2 \rightarrow \phi$ be a local formula such that ψ_2 and ϕ do not share common propositions then we replace $\psi_1 \wedge \psi_2 \rightarrow \phi$ by $\psi_1 \rightarrow \phi$. This simplification is correct in the following sense: if the formula obtained after dropping some assumptions in local formulas is realizable then the original formula is also realizable. Further, a Player 1's strategy to win the game defined by the simplified formula is also a Player 1's strategy to win the game defined by the original formula. This is justified

[5] It is necessary to keep the entire alphabet when considering the subformulas to ensure proper definition of the product of games that asks for components defined on the same set of moves.

by the fact that the new formula logically implies the original formula i.e. $\psi_1 \rightarrow \phi$ logically implies $\psi_1 \wedge \psi_2 \rightarrow \phi$. However, this heuristic is not complete because the local master plans may be more restrictive than necessary as we locally forget about global assumptions that exist in the original formula. We illustrate this on two examples.

Let $I = \{\mathsf{req}\}$, $O = \{\mathsf{grant}\}$ and $\phi = (\Box\Diamond\mathsf{req}) \rightarrow \Box\Diamond\mathsf{grant}$. In this formula, the assumption $\Box\Diamond\mathsf{req}$ is not relevant to the guarantee $\Box\Diamond\mathsf{grant}$. Realizing ϕ is thus equivalent to realizing $\Box\Diamond\mathsf{grant}$. However, the set of strategies realizing ϕ is not preserved when dropping the assumption. Indeed, the strategy that outputs a grant after each req realizes ϕ but it does not realize $\Box\Diamond\mathsf{grant}$, as this strategy relies on the behavior of the environment. Thus dropping assumption is weaker than the notion of *open implication* of [8], which requires that the strategies realizing ϕ have to realize $\Box\Diamond\mathsf{grant}$.

As illustrated by the previous example, dropping assumption does not preserve the set of strategies that realize the formula. Therefore, it can be the case that a realizable formula cannot be shown realizable with our compositional algorithm after locally dropping assumptions. In addition, it can be the case that a formula becomes unrealizable after dropping local assumptions. Consider for instance the formula $\phi = \Box\Diamond\mathsf{req} \rightarrow (\Box\Diamond\mathsf{grant} \wedge \Box(\mathcal{X}(\neg\mathsf{grant})\ \mathcal{U}\ \mathsf{req}))$. This formula is realizable, for instance by the strategy which outputs a grant iff the environment signal at the previous tick was a req. Other strategies realize this formula, like those which grant a request every n req signal (n is fixed), but all the strategies that realize ϕ have to exploit the behavior of the environment. Thus there is no strategy realizing the conjunction of $\Box\Diamond\mathsf{grant}$ and ϕ. Consequently, when we decompose ϕ into $\Box\Diamond\mathsf{req} \rightarrow \Box\Diamond\mathsf{grant}$ and $\Box\Diamond\mathsf{req} \rightarrow \Box(\mathcal{X}(\neg\mathsf{grant})\ \mathcal{U}\ \mathsf{req})$, we must keep $\Box\Diamond\mathsf{req}$ in the two formulas.

Nevertheless, in our experiments, the dropping assumption heuristic is very effective and except for one example, it always maintains *compositional realizability*.

Symbolic compositional synthesis with antichains. As mentioned in Sec. 3, we do not construct the games explicitely, but solve them on-the-fly by compactly representing by antichains the set of positions manipulated during the fixpoint computation. In particular, suppose that we are given a conjunction of formulas $\phi_1 \wedge \phi_2$, and some $K \in \mathbb{N}$. For all $i \in \{1, 2\}$, we first solve the subgame $G(\phi_i, K)$ by using the backward fixpoint computation of [7] and get the downward closed set of winning positions (for Player 1), represented by antichains. Some winning positions are owned by Player 1 (resp. Player 2), let this set be $\downarrow W_1$ (resp. $\downarrow W_2$), the downward closure of an antichain W_1 (resp. W_2). Then W_1 and W_2 also provide a compact representation of $\mathsf{MP}(G(\phi_i, K))$. Indeed, let F be a Player 1's position in $G(\phi_i, K)$, then $\mathsf{MP}(G(\phi_i, K))(F)$ is empty if $F \not\in\downarrow W_1$ (the downward closure of W_1), otherwise is the set of moves $\sigma \in \Sigma_O$ such that $\mathsf{succ}(F, \sigma) \in\downarrow W_2$. This symbolic representation is used in practice for the forward and backward compositional algorithms (Algorithms 1 and 2 of Sec. 4).

Moreover, the partial order on game positions can also be exploited by the OTFUR algorithm of Section 4 used in the forward compositional algorithm. Indeed let F be some Player 1's position of some game $G(\phi, k)$. Clearly, F is loosing (for Player 1 the controller) iff all its minimal successors are loosing. We get the dual of this property when F is a position owned by Player 2 (the environment). In this case F is loosing (for the controller) iff one of its maximal successors is loosing. Therefore to decide whether a position is loosing, depending on whether it is a controller or an environment position, we have to visit its minimal or its maximal successors only. In the OFTUR algorithm, this is done by adding to the waiting list only the edges (s', s'') such that s'' is a minimal (or maximal) successor of s'. In the case of a position owned by the controller, we can do even better. Indeed, we can add only one minimal successor in the waiting list at a time. If it turns out that this successor is loosing, we add another minimal successor. Among the minimal successors, the choice is done as follows: we prefer to add an edge (s', s'') such that s'' has already been visited. Indeed, this potentially avoids unnecessary developments of new parts of the game. Note however that this optimization cannot be used to compute the master plan of reachable positions, but only some winning strategy, as some parts of the game may not be explored. In the experiments, we use the backward algorithm to solve the local games and the optimized forward algorithm to solve the global game.

6 Experimental Evaluation

The compositional algorithms have been implemented in our prototype ACACIA [7]. The performances are evaluated on the examples provided with the tool LILY and on a larger specification of a buffer controller inspirated by the IBM rulebase tutorial [9].

Lily's test cases and parametric example. We compare several methods on the realizable examples provided with LILY and on the parametric example of [7]. In those benchmarks, the formulas are of the form $\bigwedge_{i=1}^{i=n} \psi_i \rightarrow \bigwedge_{j=1}^{j=m} \phi_j$ where $\bigwedge_{i=1}^{i=n} \psi_i$ are a set of *assumptions* and $\bigwedge_{j=1}^{j=m} \phi_j$ are a set of *guarantees*. We decompose such formula into several pieces $(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j$, as described in the previous section.

We compare four synthesis methods (Table 1). The first is the monolithic backward method of [7]. The second is the monolithic forward method based on the OTFUR algorithm optimized with antichains. The third method is a compositional method where the local games are solved with the backward algorithm of [7] and the global game with the forward algorithm OTFUR (optimized with antichains). Finally, the last method is the third method where we use the dropping assumption heuristic. For each method, we give the size of the automata (in the case of compositional methods it is the sum of the sizes of every local automata), the time to construct them, the time to check for realizability (*Check Time*), and the total time. The values in bold face are the best total times among all methods.

On small examples, we can see that the benefit of the compositional approach is not big (and in some cases the monolithic approach is even better). However for bigger formulas (demo 3.2 to 3.7), decomposing the formulas decreases the time to construct the automata, and the total realizability time is therefore better.

Now, we evaluate the benefit of dropping assumptions (last group of columns). For those experiments, we only consider the subset of formulas for which this heuristic can be applied. Our dropping heuristic does not work for demo 9 as it becomes unrealizable after the application of dropping assumptions. As we see in the table, the benefit of dropping assumptions is important and is growing with the size of the formulas that are considered. The compositional algorithms outperform the monolithic ones when combined with dropping assumptions. They also show promises for better scalability. This is confirmed by our next benchmark.

A realistic case study. Now, we consider a set of realistic formulas (Table 2). All those formulas are out of reach of the monolithic approach as even the Büchi automaton for the formula cannot be constructed with state of the art tools. The generalized buffer

(GenBuf) originates from the IBM's tutorial for her RuleBase verification tool. The benchmark has also the nice property that it can be scaled up by increasing the number of receivers in the protocol. In this case study, the formulas are of the form $\bigwedge_{i=1}^{i=n} \psi_i \rightarrow \phi_j$ and so they are readily amenable to our compositional algorithms.

In this case study, formulas are large: for example, the sum of the number of states in the UCW of the components is 96 for $\mathsf{gb}(s_2, r_2)$, and 2399 states for $\mathsf{gb}(s_2, r_7)$. Note that the tool Wring cannot handle $\mathsf{gb}(s_2, r_2)$ monolithically.

This case study allows us to illustrate the effect of different strategies for exploiting associativity of the product operation. In particular, we use different ways of parenthesizing the local games. In all those examples, the local games and intermediate combination of local games are solved with the backward compositional algorithm, while the last compositional step (at the top) is done with the forward method. In each strategy we first compute the master plan of each sub-formula. Then the column Flat refers to the strategy that check global realizability directly. The column Binary refers to the strategy that computes global realizability incrementally using the binary tree of sub-formulas. Finally, the column Heuristic refers to the strategy that computes global

Table 1. Performance comparison on Lily's benchmark and parametric example

	Monolothic						Compositional				Compositional + DA			
			BACKWARD (Acacia'09)		FORWARD		FORWARD(global) BACKWARD(local)				FORWARD(global) BACKWARD(local)			
examples	|tbUCW| (states)	tbUCW Time(s)	Check Time(s)	Total time(s)	Check Time(s)	Total time(s)	Σ_i|tbUCW$_i$|	tbUCW Time(s)	Check Time(s)	Total time(s)	Σ_i|tbUCW$_i$|	tbUCW Time(s)	Check Time(s)	Total time(s)
3	20	0.49	0.00	0.49	0.01	0.5	28	0.40	0.01	0.41	17	0.06	0.00	**0.06**
5	26	0.71	0.00	0.71	0.01	0.72	42	0.70	0.02	0.72	34	0.40	0.02	**0.42**
6	37	1.22	0.02	1.24	0.02	1.24	57	1.14	0.03	1.17	45	0.79	0.06	**0.85**
7	22	0.60	0.00	0.60	0.01	0.61	41	0.66	0.02	0.68	33	0.40	0.02	**0.42**
9	13	0.13	0.01	0.14	0.00	**0.13**	31	0.26	0.00	0.26	na	na	na	na
13	7	0.00	0.00	0.00	0.01	0.01	4	0.01	0.00	**0.01**	na	na	na	na
14	14	0.11	0.00	0.11	0.01	**0.12**	27	0.77	0.01	0.78	15	0.03	0.00	0.03
15	16	0.06	0.02	0.08	0.00	**0.06**	22	0.11	0.03	0.14	na	na	na	na
16	21	0.22	0.31	0.53	0.07	**0.29**	45	0.20	0.14	0.34	na	na	na	na
17	17	0.16	0.04	0.20	0.03	**0.19**	23	0.16	0.05	0.21	na	na	na	na
18	22	0.34	0.21	0.55	0.19	**0.53**	45	0.35	0.16	0.51	na	na	na	na
19	18	0.31	0.01	0.32	0.01	0.32	27	0.25	0.03	0.28	27	0.26	0.01	**0.27**
20	105	2.67	0.01	2.68	0.01	2.68	154	2.43	0.03	2.46	101	1.52	0.02	**1.54**
21	27	7.38	0.22	7.60	0.28	7.66	43	1.40	0.52	1.92	44	0.55	0.51	**1.06**
22	45	7.08	0.03	7.11	0.02	7.1	80	10.26	0.05	10.31	49	1.51	0.13	**1.64**
3.2	36	0.94	0.02	0.96	0.00	0.94	40	0.79	0.02	**0.81**	na	na	na	na
3.3	56	1.80	0.15	1.95	0.02	1.82	60	1.21	0.06	**1.27**	na	na	na	na
3.4	84	3.12	1.24	4.36	0.04	3.16	80	1.63	0.10	**1.73**	na	na	na	na
3.5	128	3.52	9.94	13.46	0.12	3.64	100	2.04	0.17	**2.21**	na	na	na	na
3.6	204	10.22	100	110.22	0.46	10.68	120	2.40	0.39	**2.79**	na	na	na	na
3.7	344	26.48	660	686.48	2.35	28.82	140	2.96	1.02	**3.98**	na	na	na	na

Table 2. Performance comparison on a scalability test for the forward methods

					FLAT	BINARY	HEURISTIC	
	k	Σ|tbUCW|	tbUCW Time(s)	UCW_OPT Time(s)	Check Time(s)	Check Time(s)	Check Time(s)	|Moore machine|
gb_s2_r2	2	91	4.83	0.08	0.84	0.99	0.98	54
gb_s2_r3	2	150	8.52	0.17	7.33	36.27	6.99	63
gb_s2_r4	2	265	15.64	0.53	36.88	125.60	24.19	86
gb_s2_r5	2	531	26.48	2.11	154.02	266.36	70.41	107
gb_s2_r6	2	1116	50.70	14.38	889.12	1164.44	335.44	132
gb_s2_r7	2	2399	92.01	148.46	2310.74	timeout	1650.83	149

realizability incrementally using a specific tree of sub-formula defined by the user. The column UCW_OPT refers to the time to optimize the automata with Lily's optimizations (this time was included in the UCW time in Table [1]).

Conclusion. We have provided compositional algorithms for full LTL synthesis. Our algorithm are able to handle formulas that are several pages long (see [2]). We believe that our compositional approach is an essential step to make realizability check more practical. As future works, we plan to improve our tool by considering symbolic data-structures. Currently, the alphabet of signals is handled enumeratively and we believe that substantial gain could be obtain by handling it symbolically. This algorithmic improvement is orthogonal to the ones presented in this paper. It should be noted that the compositional approach we propose is general and can be applied, for example, if some sub-games are not specified using LTL but constructed directly from another specification language. This is important as in practice some modules could be easily specified directly by deterministic automata instead of LTL. Exploring the use of such mixed specification methodology is part of our future works.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. Acacia (2009), http://www.antichains.be/acacia
3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from psl. Electr. Notes Theor. Comput. Sci. 190(4), 3–16 (2007)
4. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
5. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
6. Doyen, L., Raskin, J.F.: Improved algorithms for the automata-based approach to model-checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 451–465. Springer, Heidelberg (2007)

7. Filiot, E., Jin, N., Raskin, J.-F.: An antichain algorithm for LTL realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
8. Greimel, K., Bloem, R., Jobstmann, B., Vardi, M.Y.: Open implication. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 361–372. Springer, Heidelberg (2008)
9. http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/tutorial3/
10. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE, Los Alamitos
11. Kuijper, W., van de Pol, J.: Compositional control synthesis for partially observable systems. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 431–447. Springer, Heidelberg (2009)
12. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–542. IEEE, Los Alamitos (2005)
13. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
14. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM, New York (1989)
15. Rosner, R.: Modular synthesis of reactive systems. Ph.d. dissertation, Weizmann Institute of Science (1992)
16. Safra, S.: On the complexity of ω automata. In: FOCS, pp. 319–327 (1988)
17. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
18. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for ltl games. In: FMCAD, pp. 77–84. IEEE, Los Alamitos (2009)
19. Thomas, W.: Church's problem and a tour through automata theory. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 635–655. Springer, Heidelberg (2008)
20. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1995)

What's Decidable about Sequences?[⋆]

Carlo A. Furia

Chair of Software Engineering, ETH Zurich
caf@inf.ethz.ch
http://se.inf.ethz.ch/people/furia/

Abstract. We present a first-order theory of (finite) sequences with integer elements, Presburger arithmetic, and regularity constraints, which can model significant properties of data structures such as lists and queues. We give a decision procedure for the quantifier-free fragment, based on an encoding into the first-order theory of concatenation; the procedure has PSPACE complexity. The quantifier-free fragment of the theory of sequences can express properties such as sortedness and injectivity, as well as Boolean combinations of periodic and arithmetic facts relating the elements of the sequence and their positions (e.g., "for all even i's, the element at position i has value $i + 3$ or $2i$"). The resulting expressive power is orthogonal to that of the most expressive decidable logics for arrays. Some examples demonstrate that the fragment is also suitable to reason about sequence-manipulating programs within the standard framework of axiomatic semantics.

1 Introduction

Verification is undecidable already for simple programs, but modern programming languages support a variety of sophisticated features that make it all the more complicated. These advanced constructs — such as arrays, pointers, dynamic allocation of resources, and object-oriented abstract data types — are needed because they raise the level of abstraction thus making programmers more productive and programs less defective. In an attempt to keep the pace with the development of programming languages, verification techniques have progressed rapidly over the years.

Further steady progress requires expressive program logics and powerful decision procedures. In response to the evolution of modern programming languages, new decidable program logic fragments and combination techniques for different fragments have mushroomed especially in recent years. Many of the most successful contributions have focused on verifying relatively restricted aspects of a program's behavior, for example by decoupling pointer structure and functional properties in the formal analysis of a dynamic data structure. This narrowing choice, partly deliberate and partly required by the formidable difficulty of the various problems, is effective because different aspects are often sufficiently decoupled that each of them can be analyzed in isolation with the most appropriate, specific technique.

⋆ Work partially supported by Hasler Stiftung, ManCom project, grant #2146.

This paper contributes to the growing repertory of special program logics by exploring the decidability of properties about *sequences*. Sequences of elements of homogeneous type can abstract fundamental features of data structures, such as the content of a dynamically allocated list, a stack, a queue, or an array.

We take a new angle on reasoning about sequences, based on the *theory of concatenation*: a first-order theory where variables are interpreted as words (or sequences) over a finite alphabet and can be composed by concatenating them. Makanin's algorithm for solving word equations [14] implies the decidability of the quantifier-free fragment of the theory of concatenation. Based on this, we introduce a first-order *theory of sequences* $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ whose elements are integers. Section 3.3 presents a decision procedure for the quantifier-free fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$, which encodes the validity problem into the quantifier-free theory of concatenation. The decision procedure is in PSPACE; it is known, however, that Makanin's algorithm is reasonably efficient in practice [1].

The theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ allows concatenating sequences to build new ones, and it includes Presburger arithmetic over elements. The resulting quantifier-free fragment has significant expressiveness and can formalize sophisticated properties such as sortedness, injectivity, and Boolean combinations of arithmetic facts relating elements and their indices in statements such as "for all even i's, the element with index i has value $i + 3$ or $2i$" (see more examples in Section 3.2). It is remarkable that some of these properties are inexpressible in powerful decidable array logics such as those in [4,10,12,11].

On the other hand, $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ forbids explicit indexed access to elements. This restriction, which is required to have a decidable fragment, prevents the explicit modeling of updatable memory operations such as "swap the first element with the element at index i", where i is a scalar program variable. It also differentiates $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ from the theory of arrays and extensions thereof (see Section 5), which can formalize such operations.

In summary, the theory of sequences $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ provides a fresh angle on reasoning "natively" about sequences of integers by means of an abstraction that is orthogonal to most available approaches and can be practically useful (see examples in Section 4). To our knowledge, the approach of the present paper is distinctly new. The absence of prior work on decision procedures for theories of sequences prompted us to compare the expressiveness of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ against that of theories of arrays, which are probably the closest fragments studied. However, the two theories are not meant as direct competitors, as they pertain to partially overlapping, yet largely distinct, domains.

In order to assess the limits of our theory of sequences better, we also prove that several natural extensions of the quantifier-free fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ are undecidable. Finally, we demonstrate reasoning about sequence-manipulating programs with annotations written in the quantifier-free fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$: a couple of examples in Section 4 illustrate the usage of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ formulas with the standard machinery of axiomatic semantics and backward reasoning.

Remark. For space constraints, some details and proofs are deferred to [9].

2 The Theory of Concatenation

In the rest of the paper, we assume familiarity with the standard syntax and terminology of first-order theories (e.g., [3]); in particular, we assume the standard abbreviations and symbols of first-order theories with the following operator precedence: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall$ and $\exists$. $FV(\phi)$ denotes the set of free variables of a formula ϕ. With standard terminology, a formula ϕ is a *sentence* iff it is *closed* iff $FV(\phi) = \emptyset$. A set $\mathcal{Q}$ of strings over $\{\exists, \forall\}$ (usually given in the form of a regular expression) denotes the $\mathcal{Q}$-fragment of a first-order theory: the set of all formulas of the theory in the form $\partial_1 v_1 \partial_2 v_2 \cdots \partial_n v_n \bullet \psi$, where $\partial_1 \partial_2 \cdots \partial_n \in \mathcal{Q}$, $v_1, v_2, \ldots, v_n \in FV(\psi)$, and ψ is quantifier-free. The universal and existential fragments are synonyms for the $\forall^*$- and $\exists^*$-fragment respectively. A fragment is decidable iff the validity problem is decidable for its sentences. It is customary to define the validity and satisfiability problems for a quantifier-free formula ψ as follows: ψ is valid iff the universal closure of ψ is valid, and ψ is satisfiable iff the existential closure of ψ is valid. As a consequence of this definition, the decidability of a quantifier-free fragment whose formulas are closed under negation is tantamount to the decidability of the universal or existential fragments. Correspondingly, in the paper we will allow some freedom in picking the terminology that is most appropriate to the context.

Sequences and concatenation. $\mathbb{Z}$ denotes the set of integer numbers and $\mathbb{N}$ denotes the set of nonnegative integers. Given a set $A = \{a, b, c, \ldots\}$ of constants, a *sequence* over A is any word $v = v(1)v(2)\cdots v(n)$ for some $n \in \mathbb{N}$ where $v(i) \in A$ for all $1 \leq i \leq n$. The symbol ϵ denotes the *empty sequence*, for which $n = 0$. $|v| = n$ denotes the *length* of v. A^* denotes the set of all finite sequences over A including $\epsilon \notin A$.

It is also convenient to introduce the shorthand $v(k_1, k_2)$ with $k_1, k_2 \in \mathbb{Z}$ to describe *subsequences* of a given sequence v. Informally, for positive k_1, k_2, $v(k_1, k_2)$ denotes the subsequence starting at position k_1 and ending at position k_2, both inclusive. For negative or null k_1, k_2, $v(k_1, k_2)$ denotes instead the "tail" subsequence starting from the $|k_1|$-to-last element and ending at the $|k_2|$-to-last element. Finally, for positive k_1 and negative or null k_2, $v(k_1, k_2)$ denotes the subsequence starting at position k_1 and ending at the $|k_2|$-to-last element. Formally, we have the following definition.

$$v(k_1, k_2) \triangleq \begin{cases} v(k_1)v(k_1+1)\cdots v(k_2) & 1 \leq k_1 \leq k_2 \leq |v| \\ v(k_1, |v| + k_2) & k_1 - |v| \leq k_2 < 1 \leq k_1 \\ v(|v| + k_1, |v| + k_2) & 1 - |v| \leq k_1 \leq k_2 < 1 \\ \epsilon & \text{otherwise} \end{cases}$$

For two sequences $v_1, v_2 \in A^*$, $v_1 \star v_2$ denotes their *concatenation*: the sequence $v_1(1)\cdots v_1(|v_1|)v_2(1)\cdots v_2(|v_2|)$. We will drop the concatenation symbol whenever unambiguous. The structure $\langle A^*, \star, \epsilon \rangle$ is also referred to as the *free monoid* with generators in A and neutral element ϵ. The size $|A|$ is the *rank* of the free monoid and it can be finite or infinite.

Decidability in the theory of concatenation. The theory of concatenation is the first-order theory $\mathcal{T}_{\mathsf{cat}}$ with signature

$$\Sigma_{\mathsf{cat}} \quad \triangleq \quad \{\doteq, \circ, \mathcal{R}\}$$

where $\doteq$ is the equality predicate,[1] $\circ$ is the binary concatenation function and $\mathcal{R} \triangleq \{\mathrm{R}_1, \mathrm{R}_2, \ldots\}$ is a set of unary (monadic) predicate symbols called *regularity constraints*. We sometimes write $R_i(x)$ as $x \in R_i$ and $\alpha \not\doteq \beta$ abbreviates $\neg(\alpha \doteq \beta)$.

An interpretation of a formula in the theory of concatenation is a structure $\langle A^*, \star, \epsilon, \underline{\mathcal{R}}, ev\rangle$ where $\langle A^*, \star, \epsilon\rangle$ is a free monoid, $\underline{\mathcal{R}} = \{\underline{\mathrm{R}}_1, \underline{\mathrm{R}}_2, \ldots\}$ is a collection of regular subsets of A^*, and ev is a mapping from variables to values in A^*. The satisfaction relation $\langle A^*, \star, \epsilon, \underline{\mathcal{R}}, ev\rangle \models \phi$ for formulas in $\mathcal{T}_{\mathsf{cat}}$ is defined in a standard fashion with the following assumptions: (1) any variable x takes the value $ev(x) \in A^*$; (2) the concatenation $x \circ y$ of two variables x, y takes the value $ev(x) \star ev(y)$; (3) for each $\mathrm{R}_i \in \mathcal{R}$, the corresponding $\underline{\mathrm{R}}_i \in \underline{\mathcal{R}}$ defines the set of sequences $x \in \underline{\mathrm{R}}_i$ for which $\mathrm{R}_i(x)$ holds (this also subsumes the usage of constants).

The following proposition summarizes some decidability results about fragments of the theory of concatenation; they all are known results, or corollaries of them [9].

Proposition 1

1. *[14,7,17] The universal and existential fragments of the theory of concatenation over free monoids with finite rank are* decidable *in PSPACE.*
2. *The following fragments of the theory of concatenation are* undecidable.
   (a) *[8] The $\forall^*\exists^*$ and $\exists^*\forall^*$ fragments.*
   (b) *[5] The extensions of the existential and universal fragments over the free monoid $\{a, b\}^*$ with: (1) two length functions $|x|_a, |x|_b$ where $|x|_p \triangleq \{y \in p^* \mid y$ has the same number of p's as $x\}$; or (2) the function $Sp(x) \triangleq |x|_a \star |x|_b$.*
3. *[5] The following are* not definable *in the existential or universal fragments.*
   (a) *The set $S^= \triangleq \{a^n b^n \mid n \in \mathbb{N}\}$.*
   (b) *The equal length predicate $Elg(x, y) \triangleq |x| = |y|$.*

It is currently unknown whether the extension of the existential or universal fragment of concatenation with Elg is decidable.

3 A Theory of Sequences

This section introduces a first-order theory of sequences (Section 3.1) with arithmetic, demonstrates it on a few examples (Section 3.2), gives a decision procedure for its universal fragment (Sections 3.3– 3.4), and shows that "natural" larger fragments are undecidable (Section 3.5).

3.1 A Theory of Integer Sequences

We present an arithmetic theory of sequences whose elements are integers. It would be possible to make the theory parametric with respect to the element type. Focusing on integers, however, makes the presentation clearer and more concrete, with minimal loss of generality as one can encode any theory definable in the integer arithmetic fragment.

[1] We use the symbol $\doteq$ to distinguish it from the standard arithmetic equality symbol $=$ used later in the paper.

Syntax. Properties of integers are expressed in Presburger arithmetic with signature:

$$\Sigma_{\mathbb{Z}} \quad \triangleq \quad \{0, 1, +, -, =, <\}$$

Then, our theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ of sequences with integer values has signature:

$$\Sigma_{\mathsf{seq}(\mathbb{Z})} \quad \triangleq \quad \Sigma_{\mathsf{cat}} \cup \Sigma_{\mathbb{Z}}$$

Operator precedence is: $\circ$; $+$ and $-$; $\doteq, =$ and $<$, followed by logic connectives and quantifiers with the previously defined precedence.

We will generally consider formulas in prenex normal form $\mathcal{Q} \bullet \psi$, where $\mathcal{Q}$ is a quantifier prefix and ψ is quantifier-free written in the grammar:

$$\begin{array}{rcl} seq & ::= & var \mid 0 \mid 1 \mid seq \circ seq \mid seq + seq \mid seq - seq \\ fmla & ::= & seq \doteq seq \mid R(seq) \mid seq = seq \mid seq < seq \\ & & \mid \neg fmla \mid fmla \vee fmla \mid fmla \wedge fmla \mid fmla \Rightarrow fmla \end{array}$$

with var ranging over variable names.

Semantics. An interpretation of a sentence of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ is a structure $\langle \mathbb{Z}^*, \star, \epsilon, \underline{\mathcal{R}}, ev \rangle$ with the same meaning as in the theory of concatenation plus the following additional assumptions about arithmetic.[2]

- The interpretation of a sequence $v_1 v_2 \cdots \in \mathbb{Z}^*$ of integers in an integer sub-expression (e.g., in a sum, in an integer equality or inequality) is the first integer in the sequence v_1, with the convention that the interpretation of the empty sequence is 0.[3]
- Conversely, the interpretation of an integer $v \in \mathbb{Z}$ in a sequence sub-expression (e.g., in a concatenation) is the singleton sequence v.
- Addition, subtraction, equality, and less than are interpreted accordingly.

For example, the expression $(1 \circ 0 \circ 0 < 1 + 0 + 1) \wedge (1 \circ 0 = 1 \circ 1)$ evaluates to true because the sequences $1 \circ 0 \circ 0$, $1 \circ 0$, and $1 \circ 1$ are all interpreted as the integer 1.

Shorthands. We introduce several simplifying shorthands.

- A symbol for every constant $k \in \mathbb{Z}$, defined as obvious.
- $\alpha \neq \beta$, $\alpha \leq \beta$, $\alpha \geq \beta$, and $\alpha > \beta$ defined respectively as $\neg(\alpha = \beta)$, $\alpha < \beta \vee \alpha = \beta$, $\neg(\alpha < \beta)$, and $\alpha \geq \beta \wedge \alpha \neq \beta$.
- Shorthands such as $\alpha \leq \beta < \gamma$ or $\beta \in [\alpha, \gamma)$ for $\alpha \leq \beta \wedge \beta < \gamma$.
- Bounded length predicates such as $|x| < k$ for a variable x and a constant $k \in \mathbb{Z}$. $|x| < k$ abbreviates $\mathrm{R}^{<k}(x)$, where $\mathrm{R}^{<k}$ is a regularity constraint that stands for $\{\epsilon\} \cup \bigcup_{0<i<k} \mathbb{Z}^i$.
  The definition of derived expressions such as $k_1 \leq |x| < k_2$ is straightforward.

[2] The presentation of the semantics of the theory is informal and implicit for brevity.

[3] The results of Section 3.5 suggest that interpretations aggregating the values of multiple sequence elements are likely to be undecidable.

- Subsequence functions such as $x(k_1, k_2)$ for a variable x and two constants $k_1, k_2 \in \mathbb{Z}$ with the intended semantics (see Section 2). We define these functions in the theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ by the following rewriting rules, defined on formulas in prenex normal form with quantifier prefix $\mathcal{Q}$:

$$\frac{\mathcal{Q} \ \bullet \ \psi[x(k_1, k_2)]}{\mathcal{Q}\forall u, v, w \ \bullet \ \begin{pmatrix} \kappa_1 \land x \doteq uvw \land |u| = k_1 - 1 \land |v| = k_2 - k_1 + 1 \\ \lor \quad \kappa_2 \land x \doteq uvw \land |u| = k_1 - 1 \land |w| = -k_2 \\ \lor \quad \kappa_3 \land x \doteq uvw \land |v| = -k_1 + k_2 + 1 \land |w| = -k_2 \\ \lor \quad \neg(\kappa_1 \lor \kappa_2 \lor \kappa_3) \land u \doteq v \doteq w \doteq \epsilon \end{pmatrix} \Rightarrow \psi[v]}$$

  where $\kappa_1 \triangleq 1 \leq k_1 \leq k_2 \leq |x|$, $\kappa_2 \triangleq k_1 - |x| \leq k_2 < 1 \leq k_1$, $\kappa_3 \triangleq 1 - |x| \leq k_1 \leq k_2 < 1$.
- $\mathsf{fst}(x)$ and $\mathsf{lst}(x)$ for the first $x(1, 1)$ and last element $x(0, 0)$ of x, respectively.

3.2 Examples

A few examples demonstrate the expressiveness of the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$.

1. **Equality:** sequences u and v are equal.
$$\mathsf{u} \doteq \mathsf{v} \tag{1}$$
2. **Bounded equality:** sequences u and v are equal in the *constant* interval $[l, u]$ for $l, u \in \mathbb{Z}$.
$$\mathsf{u}(l, u) \doteq \mathsf{v}(l, u) \tag{2}$$
3. **Boundedness:** no element in sequence u is greater than value v.
$$\forall h, t \ \bullet \ \mathsf{u} \doteq ht \ \Rightarrow \ t \leq \mathsf{v} \tag{3}$$
4. **Sortedness:** sequence u is sorted (strictly increasing).
$$\forall h, m, t \ \bullet \ \mathsf{u} \doteq hmt \land |m| = 1 \land |t| > 0 \ \Rightarrow \ m < t \tag{4}$$
5. **Injectivity:** u has no repeated elements.
$$\forall h, v_1, m, v_2, t \ \bullet \ \mathsf{u} \doteq hv_1mv_2t \land |v_1| = 1 \land |v_2| = 1 \Rightarrow v_1 \neq v_2 \tag{5}$$
6. **Partitioning:** sequence u is partitioned at *constant* position $k > 0$.
$$\forall h_1, t_1, h_2, t_2 \ \bullet \ \begin{pmatrix} \mathsf{u}(1, k) \doteq h_1t_1 \\ \land \ \mathsf{u}(k + 1, 0) \doteq h_2t_2 \\ \land \ |t_1| > 0 \land |t_2| > 0 \end{pmatrix} \Rightarrow t_1 < t_2 \tag{6}$$
7. **Membership:** *constant* element $k \in \mathbb{Z}$ occurs in sequence u.
$$\mathsf{u} \in (\mathbb{Z}^* k \mathbb{Z}^*) \tag{7}$$
8. **Non-membership:** no element in sequence u has value v.
$$\forall h, t \ \bullet \ \mathsf{u} \doteq ht \land |t| > 0 \ \Rightarrow \ t \neq \mathsf{v} \tag{8}$$

9. **Periodicity:** in non-empty sequence u, elements on even positions have value 0 and elements on odd positions have value 1 (notice that $\mathsf{lst}(h) = 0$ if h is empty).

$$\forall h, t \bullet \mathsf{u} \doteq ht \wedge |t| > 0 \Rightarrow \begin{pmatrix} \mathsf{lst}(h) = 1 \\ \Rightarrow t = 0 \end{pmatrix} \wedge \begin{pmatrix} \mathsf{lst}(h) = 0 \\ \Rightarrow t = 1 \end{pmatrix} \tag{9}$$

10. **Comparison** between indices and values: for every index i, element at position i in the non-empty sequence u has value $i + 3$.

$$\mathsf{fst}(\mathsf{u}) = 1+3 \wedge \forall h, t, v \bullet \mathsf{u} \doteq ht \wedge |h| > 0 \wedge |t| > 0 \wedge \mathsf{lst}(h) = v \Rightarrow \mathsf{fst}(t) = v+1 \tag{10}$$

11. **Disjunction of value constraints:** for every pair of positions $i < j$ in the sequence u, either $\mathsf{u}(i,i) \leq \mathsf{u}(j,j)$ or $\mathsf{u}(i,i) \geq 2\mathsf{u}(j,j)$.

$$\forall h, v_1, m, v_2, t \bullet \mathsf{u} \doteq h v_1 m v_2 t \wedge |v_1| > 0 \wedge |v_2| > 0 \Rightarrow v_1 \leq v_2 \vee v_1 \geq v_2 + v_2 \tag{11}$$

Comparison with theories of arrays. Properties such as strict sortedness (4), periodicity (9), and comparisons between indices and values (10) are inexpressible in the array logic of Bradley et al. [4]. The latter is inexpressible also in the logic of Ghilardi et al. [10] because Presburger arithmetic is restricted to indices. Properties such as (11) are inexpressible both in the SIL array logic of [11] — because quantification on multiple array indices is disallowed — and in the related LIA logic of [12] — because disjunctions of comparisons of array elements are disallowed. Extensions of each of these logics to accommodate the required features would be undecidable.

Conversely, properties such as *permutation*, bounded equality *for an interval specified by indices*, length constraints *for a variable value*, membership *for a variable value*, and the *subsequence* relation, are inexpressible in the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$. Membership and the subsequence relation are expressible in the dual existential fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$, while the other properties seem to entail undecidability of the corresponding $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ fragment (see Section 3.5). Bounded equality, length constraints, and membership, on the other hand, are expressible in all the logics of [4,10,11,12], and [10] outlines a decidable extension which supports the subsequence relation (see Section 5).

3.3 Deciding Properties of Integer Sequences

This section presents a decision procedure $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$ for the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$. The procedure transforms any universal $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ formula into an equi-satisfiable universal formula in the theory of concatenation over the free monoid $\{a, b, c, d\}^*$. The basic idea is to encode integers as sequences over the four symbols $\{a, b, c, d\}$: the sequence $acb^{k_1}a$ encodes a nonnegative integer k_1, while the sequence $adb^{-k_2}a$ encodes a negative integer k_2. Suitable rewrite rules encode all quantifier-free Presburger arithmetic in accordance with this convention. Subsection 3.4 illustrates the correctness and complexity of $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$.

Consider a universal formula of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ in prenex normal form:

$$\forall x_1, \ldots, x_v \bullet \psi \tag{12}$$

where ψ is quantifier-free. Modify (12) by application of the following steps.

1. Introduce fresh variables to normalize formulas into the following form:

$$\begin{array}{lll} fmla & ::= & var \doteq var \mid var \doteq var \circ var \mid R(var) \mid var = 0 \mid var = 1 \\ & & \mid var = var \mid var = var + var \mid var = var - var \mid var < var \\ & & \mid \neg fmla \mid fmla \vee fmla \mid fmla \wedge fmla \mid fmla \Rightarrow fmla \end{array}$$

   Clearly, we can achieve this by applying exhaustively rewrite rules that operate on ψ such as:

$$\frac{\psi[x \circ y]}{e \doteq x \circ y \Rightarrow \psi[e]} \qquad \frac{\psi[x + y]}{f = x + y \Rightarrow \psi[f]}$$

   for fresh variables e, f.

2. For each variable $x_i \in FV(\psi) = \{x_1, \ldots, x_v\}$, introduce the fresh variables h_i, t_i, s_i, m_i (for head, tail, sign, modulus) and rewrite ψ as:

$$\bigwedge_{1 \le i \le v} \left(\begin{pmatrix} x_i \doteq h_i t_i \\ \wedge\, h_i \doteq a s_i m_i a \\ \wedge\, s_i \in \{c, d\} \\ \wedge\, m_i \in b^* \\ \wedge\, t_i \in (acb^*a \cup adb^+a)^* \end{pmatrix} \vee \begin{pmatrix} x_i \doteq \epsilon \\ \wedge\, h_i \doteq a s_i m_i a \\ \wedge\, s_i \doteq c \\ \wedge\, m_i \doteq \epsilon \\ \wedge\, t_i \doteq \epsilon \end{pmatrix} \right) \Rightarrow \psi$$

3. Apply the following rules exhaustively to remove arithmetic equalities:

$$\frac{\psi[x_i = x_j]}{\psi[h_i \doteq h_j]} \qquad \frac{\psi[x_i = 0]}{\psi[h_i \in 0]} \qquad \frac{\psi[x_i = 1]}{\psi[h_i \in 1]}$$

4. Apply the following rule exhaustively to remove differences:

$$\frac{\psi[x_k = x_i - x_j]}{\psi[x_i = x_k + x_j]}$$

5. Apply the following rule exhaustively to remove comparisons:

$$\frac{\psi[x_i < x_j]}{\begin{pmatrix} m_i \doteq m_j \\ \vee\, m_i \doteq m_j p \\ \vee\, m_j \doteq m_i p \end{pmatrix} \Rightarrow \psi \begin{bmatrix} s_i \doteq d \wedge s_j \doteq c \\ \vee \\ s_i \doteq s_j \doteq c \wedge m_j \doteq m_i p \\ \vee \\ s_i \doteq s_j \doteq d \wedge m_i \doteq m_j p \end{bmatrix}}$$

   for fresh $p \in b^+$.

6. Apply the following rule exhaustively to remove sums:

$$\frac{\psi[x_k = x_i + x_j]}{\begin{pmatrix} m_i \doteq m_j \\ \vee\, m_i \doteq m_j p \\ \vee\, m_j \doteq m_i p \end{pmatrix} \Rightarrow \psi \left[\begin{array}{c} s_i \doteq s_j \wedge x_k \doteq a s_i m_i m_j a \\ \vee \\ s_i \not\doteq s_j \wedge m_i \doteq m_j \wedge x_k \doteq aca \\ \vee \\ s_i \not\doteq s_j \wedge m_i \doteq m_j p \wedge x_k \doteq a s_i p a \\ \vee \\ s_i \not\doteq s_j \wedge m_j \doteq m_i p \wedge x_k \doteq a s_j p a \end{array}\right]}$$

for fresh $p \in b^+$.

7. Modify the meaning of regularity constraints as follows: let $\underline{\mathrm{R}}_i$ be defined by a regular expression with constants in $\mathbb{Z}$. Substitute every occurrence of a nonnegative constant $k \in \mathbb{Z}$ by $acb^k a$; every occurrence of a negative constant $k \in \mathbb{Z}$ by $adb^{-k}a$; every occurrence of set $\mathbb{Z}$ by $acb^*a \cup adb^+a$.

The resulting formula is again in form (12) where ψ is now a quantifier-free formula in the theory of concatenation over $\{a, b, c, d\}^*$; its validity is decidable by Proposition 1.

3.4 Correctness and Complexity

Let us sketch the correctness argument for the decision procedure $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$, which shows that the transformed formula is equi-satisfiable with the original one.

The justification for step 1 is straightforward. The following steps introduce a series of substitutions to eliminate arithmetic by reducing it to equations over the theory of concatenation with the unary encoding of integers defined above.

Step 2 requires that any variable x_i is a sequence of the form $(acb^*a \cup adb^+a)^*$ and introduces fresh variables to denote significant parts of the sequence: h_i aliases the first element of the sequence which is further split into its sign s_i (c for nonnegative and d for negative) and its absolute value m_i encoded as a unary string in b^*. The second term of the disjunction deals with the case of x_i being ϵ, which has the same encoding as 0.

The following steps replace elements of the signature of Presburger arithmetic by rewriting them as equations over sequences with the given encoding. Step 3 reduces the arithmetic equality of two sequences of integers to equivalence of the sequences encoding their first elements. Step 4 rewrites equations involving differences with equations involving sums.

Step 5 reduces arithmetic comparisons of two sequences of integers to a case discussion over the sequences h_i, h_j encoding their first elements. Let p be a sequence in b^+ encoding the difference between the absolute values corresponding to h_i and h_j; obviously such a p always exists unless the absolute values are equal. Then, h_i encodes an integer strictly less than h_j iff one of the following holds: (1) h_i is a negative value and h_j is a nonnegative one; (2) both h_i and h_j are nonnegative values and the sequence of b's in h_j is longer than the sequence of b's in h_i; or (3) both h_i and h_j are negative values and the sequence of b's in h_i is longer than the sequence of b's in h_j.

Step 6 reduces the comparison between the value of a sum of two variables and a third variable to an analysis of the three sequences h_i, h_j, h_k encoding the first elements of the three variables. As in step 6, the unary sequence p encodes the difference between the absolute values corresponding to h_i and h_j. Then, h_k encodes the sum of the values encoded by h_i and h_j iff one of the following holds: (1) h_i and h_j have the same sign and h_k contains a sequence of b's which adds up the sequences of b's of h_i and h_j, still with the same sign; (2) h_i and h_j have opposite sign but same absolute value, so h_k must encode 0; (3) h_i and h_j have opposite sign and the absolute value of h_i is greater than the absolute value of h_j, so h_k has the same sign as h_i and the difference of absolute values as its absolute value; or (4) h_i and h_j have opposite sign and the absolute value of h_j is greater than the absolute value of h_i, so h_k has the same sign as h_j and the difference of absolute values as its absolute value.

Finally, step 7 details how to translate the interpretation of the regularity constraints over $\mathbb{Z}$ into the corresponding regularity constraints over $\{a, b, c, d\}$ with the given integer encoding.

It is not difficult to see that all rewriting steps in the decision procedure $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$ increase the size of ψ at most quadratically (this accounts for fresh variables as well). Hence, the PSPACE complexity of the universal fragment of the theory of concatenation (Proposition 1) carries over to $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$.

Theorem 1. *The universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ is decidable in PSPACE with the decision procedure $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$.*

3.5 Undecidable Extensions

Theorem 2. *The following extensions of the $\forall^*$-fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ are undecidable.*

1. *The $\forall^*\exists^*$ and $\exists^*\forall^*$ fragments.*
2. *For any pair of integer constants k_1, k_2, the extension with the two length functions $|x|_{k_1}, |x|_{k_2}$ counting the number of occurrences of k_1 and k_2 in x.*
3. *The extension with an equal length predicate $Elg(x, y) \triangleq |x| = |y|$.*
4. *The extension with a sum function $\sigma(x) \triangleq \sum_{i=1}^{|x|} x(i, i)$.*

Proof. 1. Sentences with one quantifier alternation are undecidable already for the theory of concatenation without arithmetic and over a monoid of finite rank (Proposition 1). Notice that the set of sentences that are expressible both in the $\forall^*\exists^*$ and in the $\exists^*\forall^*$ fragment is decidable [18, Th. 4.4]; however, this set lacks a simple syntactic characterization.

2. Corollary of Proposition 1.
3. We encode the universal theory of $\Pi = \langle \mathbb{N}, 0, 1, +, \pi \rangle$ — where $\pi(x, y) \triangleq x2^y$ — in the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ extended by the Elg predicate; undecidability follows from the undecidability of the existential and universal theories of Π [5, Corollary 5]. All we have to do is showing that $\pi(x, y) = p$ is universally definable in $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ with Elg. To this end, define l_y as a sequence that begins with value y, ends with value 1, and where every element is the successor of the following.

$$\forall h, t \bullet \mathsf{fst}(l_y) = y \land \mathsf{lst}(l_y) = 1 \land l_y \doteq ht \land |h| > 0 \land |t| > 0 \Rightarrow \mathsf{lst}(h) = t + 1$$

As a result l_y is in the form $y, y-1, \ldots, 1$ and hence has length y.[4] Then, $\pi(x, y)$ is universally definable as the sequence p with the same length as l_y, whose last

[4] This technique would allow the definition of the length function $|x|$ and full index arithmetic.

element is x, and where every element is obtained by doubling the value of the element that follows:

$$\forall g, u \bullet Elg(p, l_y) \wedge \mathsf{lst}(p) = x \wedge p \doteq gu \wedge |g| > 0 \wedge |u| > 0 \Rightarrow \mathsf{lst}(g) = u + u$$

Hence p has the form $2^y x, 2^{y-1}x, \ldots, 2^2 x, 2x, x$ which encodes the desired value $x2^y$ in $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$. (Notice that the two universal definitions of l_y and p can be combined into a single universal definition by conjoining the definition of p to the consequent in the definition of l_y).

4. For any sequence x over $\{0, 1\}$ define $Sp(x) = y$ as $y \in 0^*1^* \wedge \sigma(y) = \sigma(x)$. Then, Proposition 1 implies undecidability because this extension of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ can define universal sentences over the free monoid $\{a, b\}^*$ with the function Sp. $\square$

The decidability of the following is instead currently unknown: the extension of the universal fragment with a function $x \oplus 1$ defined as the sequence $x(1) + 1, x(2) + 1, \ldots, x(|x|) + 1$. The fragment allows the definition of the set $S^{=}\{0^n 1^n \mid n \in \mathbb{N}\}$ as the sequences x such that $x \in 0^*1^* \wedge \forall u, v \bullet x \doteq uv \wedge u \in 0^* \wedge v \in 1^* \Rightarrow u \oplus 1 \doteq v$. This is inexpressible in the universal fragment of the theory of concatenation, but the decidability of the resulting fragment is currently unknown (see Proposition 1).

4 Verifying Sequence-Manipulating Programs

This section outlines a couple of examples that use formulas in the theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ to reason about sequence-manipulating programs. An implementation of the decision procedure $\mathcal{D}_{\mathsf{seq}(\mathbb{Z})}$ is needed to tackle more extensive examples; it belongs to future work. The examples are in Eiffel-like pseudo-code [15]; it is not difficult to detail an axiomatic semantics and a backward substitution calculus, using the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$, for the portions of this language employed in the examples.

Mergesort. Consider a straightforward recursive implementation of the Mergesort algorithm; Table 1 (left) shows an annotated version, where $*$ denotes the concatenation operator in the programming language (whose semantics is captured by the corresponding logic operator $\circ$). The annotations specify that the routine produces a sorted array, where predicate $\mathrm{sorted}(\mathsf{u})$ is defined as (cmp. (4)):

$$\mathrm{sorted}(\mathsf{u}) \quad \triangleq \quad \forall h, m, t \bullet \mathsf{u} \doteq hmt \wedge |m| > 0 \wedge |t| > 0 \Rightarrow m \leq t$$

It is impossible to express in $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ another component of the full functional specification: the output is a permutation of the input. This condition is inexpressible in most of the expressive decidable extensions of the theory of arrays that are currently known, such as [4,11] (see also Section 5). Complementary automated verification techniques — using different abstractions such as the multiset [16] — can, however, verify this orthogonal aspect.

We must also abstract away the precise splitting of array a into two halves in line 8. The way in which a is partitioned into l and r is however irrelevant as far as correctness is concerned (it only influences the complexity of the algorithm), hence we can

Table 1. Annotated Mergesort (left) and Array Reversal (right)

```
 1 merge_sort (a: ARRAY): ARRAY
 2 local l, r: ARRAY
 3 do
 4    if |a| ≤ 1 then
 5       { sorted (a) }
 6       Result := a
 7    else
 8       l , r := a[1:|a|/2] , a[|a|/2+1: |a|]
 9       { l ∗ r = a }
10       l , r := merge_sort (l) , merge_sort (r)
11       { sorted (l) ∧ sorted(r) }
12       from Result := ϵ
13       { invariant sorted (Result) ∧ sorted(l) ∧ sorted(r) ∧
14                   lst(Result) ≤ fst(l) ∧ lst(Result) ≤ fst(r) }
15       until |l| = 0 ∨ |r| = 0
16       loop
17          if l. first > r. first then
18             Result := Result ∗ r. first ; r := r. rest
19          else
20             Result := Result ∗ l. first ; l := l. rest
21          end
22       end
23       if |l| > 0 then
24          { |r| = 0 } Result := Result ∗ l
25       else
26          { |l| = 0 } Result := Result ∗ r
27       end
28 { ensure sorted (Result) }
```

```
 1 reverse (a: LIST): LIST
 2 local v: INTEGER ; s: STACK
 3 do
 4    from s := ϵ
 5    { invariant s ○ a = old a }
 6    until a = ϵ
 7    loop
 8       s.push (a. first )
 9       a := a. rest
10    end
11    from Result := ϵ
12    { invariant
13          s ○ Result^R = old a }
14    until s = ϵ
15    loop
16       v := s.top
17       s.pop ; Result.extend (v)
18    end
19 { ensure Result^R = old a}
```

simply over-approximate the instruction on line 8 by a nondeterministic splitting in two continuous non-empty parts.

From the annotated program, we can generate verification conditions by standard backward reasoning. Universal sentences of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ can express the verification conditions, hence the verification process can be automated. Let us see an example on the non-trivial part of the process, namely checking that the formula on lines 13–14 is indeed an inductive invariant. Consider the "then" branch on line 18. Backward substitution of the invariant yields:

$$\begin{gathered} \mathrm{sorted}(\mathbf{Result} * \mathsf{fst}(r)) \ \wedge\ \mathrm{sorted}(l) \ \wedge\ \mathrm{sorted}(r(2,0)) \wedge \\ \mathsf{lst}(\mathbf{Result} * \mathsf{fst}(r)) \le \mathsf{fst}(l) \ \wedge\ \mathsf{lst}(\mathbf{Result} * \mathsf{fst}(r)) \le \mathsf{fst}(r(2,0)) \end{gathered} \tag{13}$$

This condition must be discharged by the corresponding loop invariant hypothesis:

$$\begin{gathered} \mathrm{sorted}(\mathbf{Result}) \ \wedge\ \mathrm{sorted}(l) \ \wedge\ \mathrm{sorted}(r) \wedge \\ \mathsf{lst}(\mathbf{Result}) \le \mathsf{fst}(l) \ \wedge\ \mathsf{lst}(\mathbf{Result}) \le \mathsf{fst}(r) \ \wedge\ |l| \neq 0 \ \wedge\ |r| \neq 0 \ \wedge\ \mathsf{fst}(l) \le \mathsf{fst}(r) \end{gathered} \tag{14}$$

The following sentence (where all free variables are implicitly universally quantified) formalizes, in the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$, the fact that (14) entails (13). Checking its validity means discharging the corresponding verification condition.

$$\begin{array}{ccc} \begin{pmatrix} \mathbf{Result} \doteq h_1 m_1 t_1 \\ \wedge\, |m_1| = 1 \wedge |t_1| > 0 \\ \Rightarrow\ m_1 \le t_1 \end{pmatrix} & \wedge \begin{pmatrix} \mathbf{Result}(0,0) \le l(1,1) \wedge \\ \mathbf{Result}(0,0) \le r(1,1) \wedge \\ |l| \neq 0 \wedge |r| \neq 0 \wedge \mathsf{fst}(l) \le \mathsf{fst}(r) \end{pmatrix} & \\ \wedge \begin{pmatrix} l \doteq h_2 m_2 t_2 \wedge |m_2| = 1 \\ \wedge |t_2| > 0\ \Rightarrow\ m_2 \le t_2 \end{pmatrix} & & \Rightarrow \begin{pmatrix} m \le t \\ \wedge\, m_l \le t_l \\ \wedge\, m_r \le t_r \\ \wedge\, \mathsf{lst}(\mathbf{Result} \circ \mathsf{fst}(r)) \\ \quad \le \mathsf{fst}(l) \\ \wedge\, \mathsf{lst}(\mathbf{Result} \circ \mathsf{fst}(r)) \\ \quad \le \mathsf{fst}(r(2,0)) \end{pmatrix} \\ \wedge \begin{pmatrix} r \doteq h_3 m_3 t_3 \wedge |m_3| = 1 \\ \wedge |t_3| > 0\ \Rightarrow\ m_3 \le t_3 \end{pmatrix} & \wedge \begin{pmatrix} \mathbf{Result} \circ r(1,1) \doteq hmt \\ \wedge\, |m| = 1 \wedge |t| > 0 \wedge\ l \doteq h_l m_l t_l \\ \wedge\, |m_l| = 1 \wedge |t_l| > 0 \\ \wedge\, r(2,0) \doteq h_r m_r t_r \\ \wedge\, |m_r| = 1 \wedge |t_r| > 0 \end{pmatrix} & \end{array}$$

Reversal. In Table 1 (right), a program reverses a sequence of integers, given as a list a, using a stack s. The queries "first" and "rest" respectively return the first element in a list and a copy of the list without its first element, and the command "extend" adds an element to the right of a list; the query "top" and the commands "pop" and "push" for a stack have the usual semantics. In the annotations, s is modeled by a sequence whose first element is the bottom of the stack; the expression **old** a denotes the value of a upon entering the routine.

The superscript $^{\mathsf{R}}$ denotes the reversal of a sequence. We do not know if the extension of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ by a reversal function is decidable. However, the following two simple update axioms are sufficient to handle any program which builds the reverse u^{R} of a sequence u starting from an empty sequence and adding one element at a time:

$$\mathsf{u}^{\mathsf{R}} = \epsilon \Leftrightarrow \mathsf{u} = \epsilon \qquad\qquad |x| = 1 \Rightarrow (\mathsf{u}x)^{\mathsf{R}} = x\mathsf{u}^{\mathsf{R}}$$

Consider, for instance, the verification condition that checks if the invariant of the second loop (lines 11–18) is indeed inductive:

$$s \circ \mathbf{Result}^{\mathsf{R}} = \mathbf{old}\ a \wedge s \neq \epsilon \quad\Rightarrow\quad s(1,-1) \circ (\mathbf{Result} \circ s(0,0))^{\mathsf{R}} = \mathbf{old}\ a$$

After rewriting $(\mathbf{Result} \circ s(0,0))^{\mathsf{R}}$ into $s(0,0) \circ \mathbf{Result}^{\mathsf{R}}$ the implication is straightforward to check for validity.

5 Related Work

A review of the staggering amount of work on decision procedures for theories of complex data types (and integrations thereof) is beyond the scope of this paper; for a partial account see [9,19,13]. In this section, we focus on a few approaches that are most similar to ours and in particular which yield decidable logics that can be compared directly to our theory of sequences (see Section 3.2). The absence of previous work on "direct" approaches to theories of sequences makes the many work on decidable extensions of the theory of arrays the closest to ours in expressive power. As discussed in Section 1, however, our theory of sequences is not meant as a replacement to the theories of arrays, but rather as a complement to them in different domains.

Bradley et al. [4] develop the *array property fragment*, a decidable subset of the $\exists^*\forall^*$ fragment of the theory of arrays. An *array property* is a formula of the form $\exists^*\forall^* \bullet \iota \Rightarrow \nu$, where the universal quantification is restricted to index variables, ι is a guard on index variables with arithmetic (restricted to existentially quantified variables), and ν is a constraint on array values without arithmetic or nested reads, and where no universally quantified index variable is used to select an element that is written to. The array property fragment is decidable with a decision procedure that eliminates universal quantification on index variables by reducing it to conjunctions on a suitable finite set of index values. Extensions of the array property fragment that relax any of the restrictions on the form of array properties are undecidable. Bradley et al. also show how to adapt their theory of arrays to reason about maps.

Ghilardi et al. [10] develop "semantic" techniques to integrate decision procedures into a decidable extension of the theory of arrays. Their $\mathcal{ADP}$ theory merges the quantifier-free extensional theory of arrays with dimension and Presburger arithmetic over indices into a decidable logic. Two extensions of the $\mathcal{ADP}$ theory are still decidable:

one with a unary predicate that determines if an array is *injective* (i.e., it has no repeated elements); and one with a function that returns the *domain* of an array (i.e., the set of indices that correspond to definite values). Ghilardi et al. suggest that these extensions might be the basis for automated reasoning on Separation Logic models. The framework of [10] also supports other decidable extensions, such as the *prefix*, and *sorting* predicates, as well as the *map* combinator also discussed in [6].

De Moura and Bjørner [6] introduce *combinatory array logic*, a decidable extension of the quantifier-free extensional theory of arrays with the *map* and *constant-value* combinators (i.e., array functors). The *constant-value* combinator defines an array with all values equal to a constant; the *map* combinator applies a k-ary function to the elements at position i in k arrays $a_1, \ldots, a_k$. De Moura and Bjørner define a decision procedure for their combinatory array logic, which is implemented in the Z3 SMT solver.

Habermehl et al. introduce powerful logics to reason about arrays with integer values [12,11,2]; unlike most related work, the decidability of their logic relies on automata-theoretic techniques for a special class of counter automata. More precisely, [12] defines the *Logic of Integer Arrays* **LIA**, whose formulas are in the $\exists^*\forall^*$ fragment and allow Presburger arithmetic on existentially quantified variables, difference and modulo constraints on index variables, and difference constraints on array values. Forbidding disjunctions of difference constraints on array values is necessary to ensure decidability. The resulting fragment is quite expressive, and in particular it includes practically useful formulas that are inexpressible in other decidable expressive fragments such as [4]. The companion work [11] introduces the *Single Index Logic* **SIL**, consisting of existentially quantified Boolean combinations of formulas of the form $\forall^* \bullet \iota \Rightarrow \nu$, where the universal quantification is restricted to index variables, ι is a positive Boolean combination of bound and modulo constraints on index variables, and ν is a conjunction of difference constraints on array values. Again, the restrictions on quantifier alternations and Boolean combinations are tight in that relaxing one of them leads to undecidability. The expressiveness of **SIL** is very close to that of **LIA**, and the two logics can be considered two variants of the same basic kernel. The other work [2] shows how to use **SIL** to annotate and reason automatically about array-manipulating programs; the tight correspondence between **SIL** and a class of counter automata allows the automatic generation of loop invariants and hence the automation of the full verification process.

6 Future Work

Future work will investigate the decidability of the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ extended with "weak" predicates or functions that slightly increase its expressiveness (such as that outlined at the end of Section 3.5). We will study to what extent the decision procedure for the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ can be integrated with other decidable logic fragments (and possibly with the dual existential fragment). We will investigate how to automate the generation of inductive invariants for sequence-manipulating programs by leveraging the decidability of the universal fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$. Finally, we will implement the decision procedure, integrate it within a verification environment, and assess its empirical effectiveness on real programs.

Acknowledgements. Thanks to Nadia Polikarpova and Stephan van Staden for comments on a draft of this paper, and to the anonymous reviewers for their remarks.

References

1. Abdulrab, H., Pécuchet, J.P.: Solving word equations. Journal of Symbolic Computation 8(5), 499–521 (1989)
2. Bozga, M., Habermehl, P., Iosif, R., Konecný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009)
3. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
5. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik 34, 337–342 (1988)
6. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Proceedings of 9th Conference on Formal Methods in Computer Aided Design (FMCAD 2009), pp. 45–52 (2009)
7. Diekert, V.: Makanin's algorithm. In: Lothaire, M. (ed.) Algebraic Combinatorics on Words. Cambridge University Press, Cambridge (2002)
8. Durnev, V.G.: Unsolvability of the positive $\forall\exists^3$-theory of a free semi-group. Sibirskiĭ Matematicheskiĭ Zhurnal 36(5), 1067–1080 (1995)
9. Furia, C.A.: What's decidable about sequences? (January 2010), http://arxiv.org/abs/1001.2100
10. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Annals of Mathematics and Artificial Intelligence 50(3-4), 231–254 (2007)
11. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 558–573. Springer, Heidelberg (2008)
12. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 474–489. Springer, Heidelberg (2008)
13. Kuncak, V., Piskac, R., Suter, P., Wies, T.: Building a calculus of data structures. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 26–44. Springer, Heidelberg (2010)
14. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Rossiĭskaya Akademiya Nauk. Matematicheskiĭ Sbornik (Translated in Sbornik Mathematics) 103(2), 147–236 (1977)
15. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
16. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 218–232. Springer, Heidelberg (2008)
17. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. Journal of the ACM 51(3), 483–496 (2004)
18. Seibert, S.: Quantifier hierarchies over word relations. In: Kleine Büning, H., Jäger, G., Börger, E., Richter, M.M. (eds.) CSL 1991. LNCS, vol. 626, pp. 329–352. Springer, Heidelberg (1992)
19. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008), pp. 349–361. ACM, New York (2008)

A Study of the Convergence of Steady State Probabilities in a Closed Fork-Join Network*

Guy Edward Gallasch and Jonathan Billington

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA, 5095, Australia
{guy.gallasch,jonathan.billington}@unisa.edu.au

Abstract. Fork-join structures are important for modelling parallel and distributed systems where coordination and synchronisation occur, but their performance analysis is difficult. The study presented in this paper is motivated by the need to calculate performance measures for computer controlled (agile) manufacturing systems. We consider the manufacture of a class of products made from two components that are created by two parallel production lines. The components are then assembled, requiring synchronisation of the two lines. The products are only created on request from the client. The production lines need to be controlled so that one line does not get ahead of the other by more than a certain amount, N, a parameter of the system. We model this system with a Generalised Stochastic Petri Net, where N is the initial marking of a control place. The customer requests and the two production line rates are modelled by exponential distributions associated with three timed transitions in the model. We use TimeNET to calculate the stationary token distribution of the GSPN for a wide range of the rates as N increases. This reveals that the steady state probabilities converge. We characterise the range of transition rates for which useful convergence occurs and provide a method for obtaining the steady state probabilities to the desired accuracy for arbitrary N.

Keywords: Fork-Join, Generalised Stochastic Petri Nets, Parametric Performance Analysis, Convergence.

1 Introduction

The modelling of parallel and distributed systems where coordination and synchronisation occur requires the use of *fork-join* structures [1]. The performance analysis of systems with synchronisation is difficult because product form solutions do not exist in general. However, due to their importance, fork-join systems have been extensively studied (e.g. [1,2,3,4,5,6,7,8,9,10,11,12]). Many of these papers endeavour to find the mean response time of a job submitted to such systems, which requires that each job is identified.

* Supported by Australian Research Council Discovery Project DP0880928.

The system we wish to analyse is different. We consider the computer control of a class of agile manufacturing systems where goods are assembled from two different components (C1 and C2). On request, the components are developed in parallel in different production lines, perhaps by different companies, and need to be synchronised for assembly. Common examples include the manufacture of tools and kitchen utensils (e.g. potato peelers). The system is driven by customer demand, so that no more goods are produced than those requested, and controlled so that each production line can only be ahead of the other by a certain amount. Further, although the two components are different (e.g. handle and blade), each component is identical, and hence the final product is made by assembling any available C1 component with any available C2 component. Thus components are not identified. This system has some similarity with assemble to order (ATO) systems, where several different products are assembled from any number of components [13]. However, we are concerned with producing components on demand, rather than assembling them on demand. In particular, we are not aware of any work on ATO systems that prevents too many of one component being made ahead of another.

We model this class of system with a Generalised Stochastic Petri Net (GSPN) [14,15]. The GSPN includes a fork-join subnet that represents the two production lines. It also contains an environment which ensures that one line can never be ahead of the other by more than N components, which corresponds to the storage capacity of one of the lines. Our goal is to derive performance measures for this system, such as the average amount one production line is ahead of another. To do this we can in principle use steady state Markov analysis [14,15]. We also want to know how this system behaves as the capacity, N, is varied. Thus we consider N as a positive integer parameter of the system.

This paper investigates the analysis of our GSPN model in an attempt to obtain solutions for its steady state probabilities for arbitrary capacity N. The fork-join subnet comprises two parallel branches where each branch includes an exponential transition with a fixed but arbitrary firing rate. There is a further exponential transition representing requests, that feeds the fork-join. The GSPN is the same as that considered in [16,17], however, their work is only concerned with aggregating the fork-join subnet and hence cannot be used to calculate any performance measures within the fork-join. For example, [16] is concerned with approximating the marking dependent rates of the aggregated transition representing the fork-join.

Due to the complexity of finding exact closed form solutions [18] we use TimeNET [19] to calculate the steady state probabilities for a wide range of transition rates as we vary the capacity N. We demonstrate that the state probabilities converge rapidly so long as the transition rates are not too similar. We characterise the region of the rates for which useful convergence is obtained for approximating the probabilities for $N > N_{cdp}$, where N_{cdp} is the capacity at which the probabilities have converged to a certain number of decimal places, dp. This allows us to obtain results up to a particular accuracy for arbitrary N, within the useful convergence region. By examining trends amongst the state probabilities, we derive a heuristic that allows N_{cdp} to be approximated for a wide range of firing rates and number of decimal places.

The rest of the paper is organised as follows. Section 2 describes the GSPN model. Its related continuous time Markov chain is discussed in Section 3. Section 4 presents our convergence results for the steady state probabilities. Progressions within the steady state probabilities are described in Section 5. Our heuristic for N_{cdp} is described and evaluated in Section 6. Conclusions are drawn in Section 7. Some familiarity with GSPNs and their analysis [14,15] is assumed.

2 Parametric GSPN Model

We consider the parametric GSPN, $GSPN_N$, shown in Fig. 1. $GSPN_N$ includes a fork-join subnet with 2 branches between immediate transitions, t_1 (the fork) and t_2 (the join). The remaining transitions are exponentially distributed timed transitions with their own rates, i.e. λ_i is associated with T_i, $i \in \{0, 1, 2\}$. The left branch of the subnet (P_1, T_1, P_3) represents the production of component C1 (stored in P_3) and the right (P_2, T_2, P_4) the production of C2 (stored in P_4). Place P_0 coordinates the production lines. Initially all places are empty except for P_0 which contains N tokens, representing the smallest capacity of a line. T_0 represents customer requests. Each request results in 1 unit of the raw materials needed to create components C1 and C2 being deposited in places P_1 and P_2 respectively (via the fork). P_1 and P_2 represent stores for the raw material, which is not identified nor related to a particular request. For efficiency, as soon as both components are available they are removed immediately (transition t_2) for assembly, freeing up capacity in the lines. This is modelled by a token being added to P_0 indicating that there is now additional capacity for each component. $GSPN_N$ ensures that the same number of components (C1 and C2) is produced as goods requested, eliminating waste, and that they are produced on demand. It also ensures that no more than N C1 components can be produced in advance of a C2 component and vice versa, providing the desired level of coordination between the production lines.

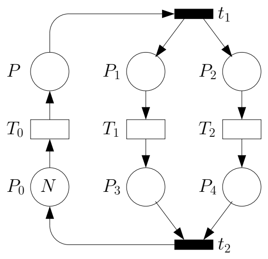


Fig. 1. $GSPN_N$ with a Fork-Join Subnet

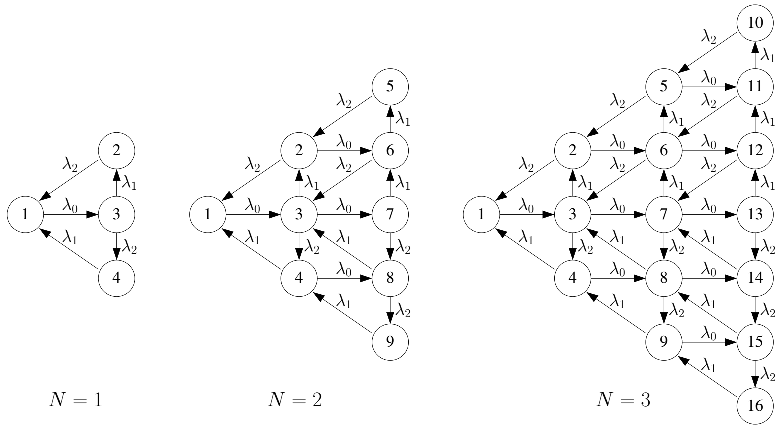


Fig. 2. $CTMC_N$ for $N = 1$, 2 and 3

3 Continuous Time Markov Chain Family

We would like to obtain steady state solutions for the probability of being in a particular marking of the GSPN. Unfortunately, obtaining analytical solutions for the steady state probabilities, for arbitrary N, is a difficult problem [18]. We thus turn to numerical solutions using TimeNET [19], which can calculate the steady state probabilities directly from the GSPN. However, to discuss the results, we firstly provide some insight into the structure of the family of continuous time Markov chains (CTMCs) that are associated with $GPSN_N$, and define some notation for them (see [18] for details).

The family of CTMCs, $CTMC_N$, can be derived from $GSPN_N$ by generating its reduced reachability graph [14,15,18]. We depict $CTMC_N$ for $N = 1, 2, 3$ in Fig. 2. We can observe that the number of states in $CTMC_N$ is $(N + 1)^2$. We number the states from 1 to $(N+1)^2$ in vertical columns, from left to right and top to bottom. The initial marking is mapped to state 1, the top right vertex is state $N^2 + 1$, the bottom right vertex is state $(N + 1)^2$, and the right-most state on the horizontal centre row is numbered $N^2 + N + 1$. Note the symmetry about the horizontal centre row. Finally we denote the steady state probability of being in state i of $CTMC_N$ by $\pi_{(N,i)}, 1 \leq i \leq (N + 1)^2$.

4 Convergence Properties

Using TimeNET, we have performed steady state analysis of $GSPN_N$ for N from 1 to 50 for a wide range of firing rates, which we call *configurations*. Inspection of the results has identified powerful convergence trends that allow the approximation of the steady state probabilities for arbitrary N with excellent accuracy, based on the probabilities for moderately large N.

Because the results depend on ratios of the firing rates, and taking into account the symmetry between λ_1 and λ_2, we fixed λ_1 to 1 while varying λ_0 and λ_2. For

Table 1. Convergence of $\pi_{(N,1)}$ to $\pi_{(N,9)}$ as N increases for $\lambda_0 = 0.2$, $\lambda_1 = 1$ and $\lambda_2 = 4$

	$\pi_{(N,1)}$	$\pi_{(N,2)}$	$\pi_{(N,3)}$	$\pi_{(N,4)}$	$\pi_{(N,5)}$	$\pi_{(N,6)}$	$\pi_{(N,7)}$	$\pi_{(N,8)}$	$\pi_{(N,9)}$
N=1	0.8264	0.0083	0.0331	0.13223140					
N=2	0.7986	0.0078	0.0323	0.12837178	0.0001	0.0006	0.0013	0.0062	0.0247
N=3	0.7934	0.0078	0.0321	0.12755432	0.0001	0.0006	0.0013	0.0062	0.0245
N=4	0.7923	0.0078	0.0321	0.12739070	0.0001	0.0006	0.0013	0.0061	0.0245
N=5	0.7921	0.0078	0.0321	0.12735805	0.0001	0.0006	0.0013	0.0061	0.0245
N=6	0.7921	0.0078	0.0321	0.12735153	0.0001	0.0006	0.0013	0.0061	0.0245
N=7	0.7921	0.0078	0.0321	0.12735022	0.0001	0.0006	0.0013	0.0061	0.0245
N=8	0.7921	0.0078	0.0321	0.12734996	0.0001	0.0006	0.0013	0.0061	0.0245
N=9	0.7921	0.0078	0.0321	0.12734991	0.0001	0.0006	0.0013	0.0061	0.0245
N=10	0.7921	0.0078	0.0321	0.12734990	0.0001	0.0006	0.0013	0.0061	0.0245

N from 1 to 50 we analysed 216 different configurations in which λ_2 varied over 1, 1.5, 2, 3, 4, 5, 8 and 10, and λ_0 over 0.01, 0.02, 0.05, 0.1, 0.125, 0.2, 0.25, $\frac{1}{3}$, 0.5, $\frac{2}{3}$, 0.75, 0.8, 0.9, 1, $\frac{1}{0.9}$, 1.25, $\frac{1}{0.75}$, 1.5, 2, 3, 4, 5, 8, 10, 20, 50 and 100, a total of 10800 TimeNET runs. We chose these values due to convergence being less rapid as the rates approach each other, (i.e. as the ratios approach 1). Because of the dependence on ratios of firing rates, these results can be scaled to an unbounded number of configurations that maintain the ratios: $\frac{\lambda_0}{\lambda_1}$ and $\frac{\lambda_2}{\lambda_1}$.

4.1 Examples of Convergence

Convergence when $\boldsymbol{\lambda_0 < \min(\lambda_1, \lambda_2)}$. Consider the example of $\lambda_0 = 0.2$, $\lambda_1 = 1$ and $\lambda_2 = 4$. Table 1 presents the steady state probabilities of all four states of the CTMC when $N = 1$, all 9 states when $N = 2$, and the first 9 states (i.e. the first 3 columns) for N from 3 to 10. Steady state probabilities increase to the left of the CTMC, due to λ_0 being smaller than both λ_1 and λ_2, and toward states in the lower half of the CTMC (e.g. $\pi_{(N,4)} > \pi_{(N,2)}$) due to $\lambda_2 > \lambda_1$. Table 1 shows that the steady state probabilities for $\pi_{(N,1)}$ to $\pi_{(N,9)}$ converge to four decimal places by $N = 8$. (The same is true of $\pi_{(N,10)}$ to $\pi_{(N,(N+1)^2)}$, not shown in Table 1). For $9 \leq N \leq 50$, we found that all additional states have probabilities of zero (to four decimal places). Hence, for this scenario, the steady state probabilities of states 1 to 81 for arbitrary $N > 8$ can be approximated to four decimal places by $\pi_{(8,1)}$ to $\pi_{(8,81)}$ and by 0 for states 82 to $(N + 1)^2$.

Apart from $\pi_{(N,4)}$, all state probabilities of $CTMC_N$ converge to four decimal places by $N = 5$. The reason is that the value to which $\pi_{(N,4)}$ converges is close to 0.12735. The probabilities given in Table 1 for $\pi_{(N,4)}$ are given to 8 decimal places to illustrate this behaviour, which we call a *rounding anomaly*.

Let us denote by N_{cdp} the minimum value of N by which convergence has occurred to dp decimal places. To validate the above convergence approximation, we took the values of N_{cdp} for $dp = 4$, 5 and 6 decimal places of accuracy ($N_{c4} = N_{c5} = N_{c6} = 8$). We then compared the approximation to the steady state probabilities obtained from TimeNET for $N = 10$ to 100 in increments of 5, and found that the approximation was accurate to the corresponding number of decimal places for each of these cases. Convergence thus provides a powerful result, as it indicates that selection of an appropriate N_{cdp} will provide steady

Table 2. Convergence of $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,N^2+1)}$ as N increases for $\lambda_0 = 5$, $\lambda_1 = 1$ and $\lambda_2 = 4$ (A denotes $(N+1)^2$)

	$\pi_{(N,A)}$	$\pi_{(N,A-1)}$	$\pi_{(N,A-2)}$	$\pi_{(N,A-3)}$	$\pi_{(N,A-4)}$	$\pi_{(N,A-5)}$	$\pi_{(N,A-6)}$	$\pi_{(N,A-7)}$	$\pi_{(N,A-8)}$
N=1	0.6400	0.1600	0.0400						
N=2	0.6063	0.1516	0.0345	0.0121	0.0030				
N=3	0.6012	0.1503	0.0372	0.0083	0.0033	0.0009	0.0002		
N=4	0.6002	0.1501	0.0375	0.0092	0.0021	0.0009	0.0003	0.0001	0.0000
N=5	0.6000	0.1500	0.0375	0.0094	0.0023	0.0005	0.0002	0.0001	0.0000
N=6	0.6000	0.1500	0.0375	0.0094	0.0023	0.0006	0.0001	0.0001	0.0000
N=7	0.6000	0.1500	0.0375	0.0094	0.0023	0.0006	0.0001	0.0000	0.0000
N=8	0.6000	0.1500	0.0375	0.0094	0.0023	0.0006	0.0001	0.0000	0.0000

state probabilities, accurate to the corresponding number of decimal places, for arbitrary $N > N_{cdp}$, where:

1. $\pi_{(N,i)} = \pi_{(N_{cdp},i)}$ for $1 \leq i \leq (N_{cdp}+1)^2$; and
2. $\pi_{(N,i)} = 0$ for $(N_{cdp}+1)^2 < i \leq (N+1)^2$.

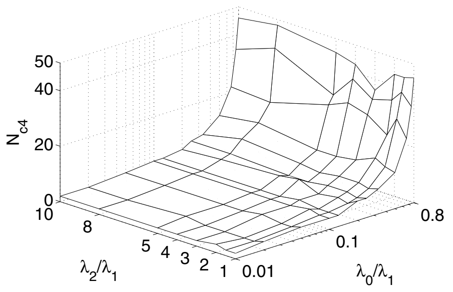
Convergence when $\boldsymbol{\lambda_0 > \min(\lambda_1, \lambda_2)}$. For this scenario, $\lambda_1 = 1$ and $\lambda_2 = 4$ as before, but this time we take $\lambda_0 = 5$.

Table 2 presents the steady state probabilities of the states in the right-most vertical column of $CTMC_N$, from lower edge (state $(N+1)^2$) to upper edge (state N^2+1) as N increases. As expected, probabilities increase toward the lower right vertex of the CTMC. These probabilities (and those not shown in Table 2) converge to four decimal places by $N_{c4} = 7$. It is important to note that in this case the converged probabilities are associated with states that are relative to the lower right vertex (state $(N+1)^2$) as N increases, not the left vertex (state 1) as was the case when $\lambda_0 < \min(\lambda_1, \lambda_2)$.

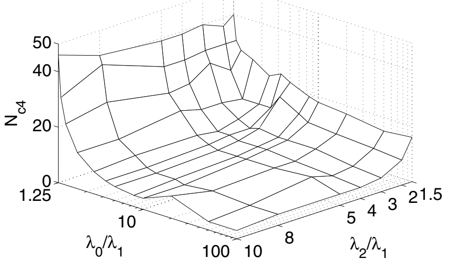
All 'additional' states for $8 \leq N \leq 50$ (for $N = 8$ these are the states along the top edge of the CTMC, given our reference point of the lower right vertex) have probabilities of zero to four decimal places. Hence, it is possible to approximate the steady state probabilities for the system with arbitrary $N > N_{cdp}$ by approximating the probabilities of the lower right states of $CTMC_N$ by the probabilities of the states in $CTMC_{N_{cdp}}$. For this example, with $N_{c4} = 7$, $\pi_{(N,(N+1)^2-14)}$ to $\pi_{(N,(N+1)^2)}$ for $N > 7$ can be approximated by $\pi_{(7,50)}$ to $\pi_{(7,64)}$ (from Table 2), $\pi_{(N,N^2-12)}$ to $\pi_{(N,N^2)}$ can be approximated by $\pi_{(7,37)}$ to $\pi_{(7,49)}$, $\pi_{(N,(N-1)^2-10)}$ to $\pi_{(N,(N-1)^2)}$ can be approximated by $\pi_{(7,26)}$ to $\pi_{(7,36)}$, and so on until $\pi_{(N,(N-6)^2)}$ can be approximated by $\pi_{(7,1)}$. The additional complexity of convergence occuring relative to the lower right vertex gives the following approximation for arbitrary $N > N_{cdp}$:

1. $\pi_{(N,j)} = \pi_{(N_{cdp},i)}$ for $1 \leq i \leq (N_{cdp}+1)^2$ and $j = (N - N_{cdp})^2 + 2(N - N_{cdp})(\lceil(\sqrt{i})\rceil) + i$, where $\lceil x \rceil$ is x rounded up to the nearest integer; and
2. $\pi_{(N,j)} = 0$ for all other j between 1 and $(N+1)^2$,

where the probabilities of $CTMC_{N_{cdp}}$ are translated to the lower right corner of $CTMC_N$, with the remaining probabilities equal to zero.

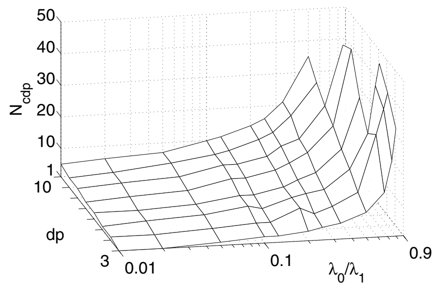


(a) $\lambda_0 < \min(\lambda_1, \lambda_2)$

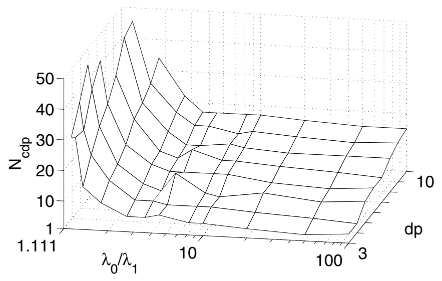


(b) $\lambda_0 > \min(\lambda_1, \lambda_2)$

Fig. 3. N_{c4} values when varying λ_0 and λ_2



(a) $\lambda_0 < \min(\lambda_1, \lambda_2)$



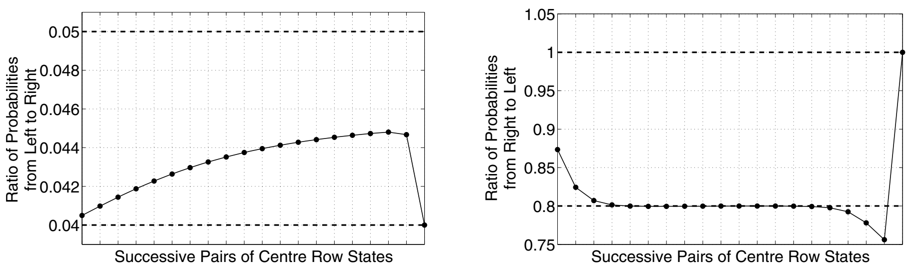
(b) $\lambda_0 > \min(\lambda_1, \lambda_2)$

Fig. 4. N_{cdp} for varying dp from 3 to 10, and varying λ_0, with $\lambda_2 = 4$

4.2 Trends in the Speed of Convergence

The above convergence experiments have been repeated for all configurations and for accuracies of 3 to 10 decimal places. When considering four decimal places of accuracy, this has revealed the values of N_{c4} depicted in Fig. 3. We can see from the graphs that the trend is for N_{c4} to increase increasingly rapidly (i.e. convergence slows increasingly rapidly) as λ_0 approaches $\min(\lambda_1, \lambda_2)$ from both below (Fig. 3 (a)) and above (Fig. 3 (b)). For the cases of $\lambda_0 = 0.9$ and $\lambda_0 = \frac{1}{0.9}$ we found that although many steady state probabilities appear to have converged to four decimal places by $N = 50$, not all had done so. In situations where $\lambda_0 > \min(\lambda_1, \lambda_2)$ we observe that the speed of convergence also slows as λ_2 approaches λ_1.

The graphs in Fig. 4 depict N_{cdp} for $3 \leq dp \leq 10$ and λ_0 from 0.01 to 0.9 (Fig. 4 (a)) and $\frac{1}{0.9}$ to 100 (Fig. 4 (b)), with $\lambda_2 = 4$. A significant trend we observe is that N_{cdp} increases approximately linearly in dp. Again, convergence slows as λ_0 approaches $\min(\lambda_1, \lambda_2)$, as observed for N_{c4} in Fig. 3. The 'roughness' of Figs. 3 and 4 are due to rounding anomalies.



(a) $\lambda_0 = 0.2, \lambda_1 = 1, \lambda_2 = 4$ (b) $\lambda_0 = 5, \lambda_1 = 1, \lambda_2 = 4$

Fig. 5. Ratio of Probabilities of Successive States along the Centre Row for $N = 20$

5 Probability Progressions

As a prelude to finding a method for predicting N_{cdp}, we consider two progressions in the converged steady state probabilities: one along the centre row of states and the other along vertical columns of the CTMC.

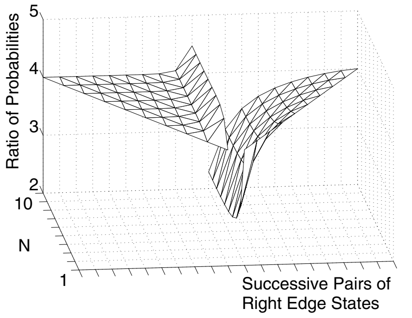
5.1 Progression along the Centre Row of States

In $CTMC_N$ the global balance equation (GBE) [14] relating states $N^2 - N + 1$ and $N^2 + N + 1$ (the two right-most states along the central horizontal row) is $(\lambda_1 + \lambda_2)\pi_{(N,N^2+N+1)} = \lambda_0 \pi_{(N,N^2-N+1)}$ [18]. Hence, the ratio of the probabilities of these two states is given by $\frac{\lambda_0}{\lambda_1+\lambda_2}$.

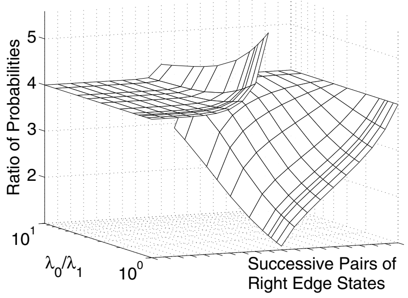
When $\lambda_0 < \min(\lambda_1, \lambda_2)$, the probabilities of successive pairs of states from left to right along the central horizontal row differ by a multiplier of between $\frac{\lambda_0}{\lambda_1+\lambda_2}$ and $\frac{\lambda_0}{\max(\lambda_1,\lambda_2)} = \frac{\lambda_0}{\lambda_2}$ (hence probabilities are decreasing from left to right). $\frac{\lambda_0}{\lambda_2}$ provides an upper bound on the multiplier and hence provides an underapproximation of the change in probability between each pair of states, from left to right. An example is given in Fig. 5 (a) for $\lambda_0 = 0.2$, $\lambda_2 = 4$ and $N = 20$. The horizontal axis represents ratios of probabilities of successive pairs of states along the centre row of the CTMC, from the ratio of $\pi_{(20,1)}$ to $\pi_{(20,3)}$ on the left, to the ratio of $\pi_{(20,381)}$ to $\pi_{(20,421)}$ on the right. Dashed lines indicate the ratios $\frac{\lambda_0}{\lambda_1+\lambda_2}$ (lower bound) and $\frac{\lambda_0}{\lambda_2}$ (upper bound). The ratio is consistently less than $\frac{0.2}{4} = 0.05$.

When $\min(\lambda_1, \lambda_2) < \lambda_0 < \lambda_1 + \lambda_2$, the probabilities still generally decrease from left to right along the central horizontal row but $\frac{\lambda_0}{\lambda_2}$ may no longer provide a lower bound on the change in probability between successive pairs of states. This effect becomes more pronounced as λ_0 approaches $\lambda_1 + \lambda_2$. Hence, for this range of rates, we cannot guarantee that $\frac{\lambda_0}{\lambda_2}$ will provide an underapproximation of the change in probability. However, it still provides a good approximation in all the configurations examined.

When $\lambda_0 \geq \lambda_1 + \lambda_2$, probabilities change from right to left along the central horizontal row with a multiplier that is never greater than $\frac{\lambda_1+\lambda_2}{\lambda_0}$. Hence, $\frac{\lambda_1+\lambda_2}{\lambda_0}$



(a) $\lambda_0 = 5$, N increases from 1 to 10



(b) $N = 10$, λ_0 increases from 1 to 10

Fig. 6. Ratio of Probabilities of successive states from $(N+1)^2$ to N^2+1 for $\frac{\lambda_2}{\lambda_1} = 4$

gives an underapproximation of the change in probability when moving from right to left along the central horizontal row. An example is given in Fig. 5 (b) for $\lambda_0 = 5$, $\lambda_2 = 4$ and $N = 20$. The horizontal axis is as in Fig. 5 (a). Here, the ratio is never greater than $\frac{4+1}{5} = 1$ (upper horizontal dashed line). Note however that $\frac{\lambda_2}{\lambda_0}$ (lower horizontal dashed line) still provides a reasonable approximation in this case.

5.2 Progression from Lower Edge to Upper Edge

When $\lambda_2 > \lambda_1$ the steady state probabilities decrease when moving vertically from the lower edge toward the upper edge. From the global balance equations of the lower and upper vertices of $CTMC_N$ [18] we know that $\pi_{(N,(N+1)^2)}$ is exactly $\frac{\lambda_2}{\lambda_1}$ times larger than $\pi_{(N,(N+1)^2-1)}$, and that $\pi_{(N,N^2+2)}$ is exactly $\frac{\lambda_2}{\lambda_1}$ times larger than $\pi_{(N,N^2+1)}$. We have observed that the ratio of state probabilities is larger than this for pairs of states closer to (but below) the central horizontal line, but is smaller than this for pairs of states closer to (but above) the central horizontal line. An example of this behaviour of the ratios is shown in Fig. 6 (a) for increasing N and in Fig. 6 (b) for increasing λ_0, for $\lambda_1 = 1$ and $\lambda_2 = 4$. These graphs depict the ratio of $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,(N+1)^2-1)}$ on the left, and the ratio of $\pi_{(N,N^2+2)}$ to $\pi_{(N,N^2+1)}$ on the right. When $N = 1$, Fig. 6 (a) shows two ratios, between states 4 and 3, and states 3 and 2, both equal to 4. As N increases, the number of successive pairs of states increases, until $N = 10$ when there are 20 ratios depicted, from the ratio of states 121 to 120 on the left, to the ratio of states 102 to 101 on the right. The ratios of probabilities of pairs of states show that the ratios immediately to either side of state $N^2 + N + 1$ approach constant values as N increases, i.e. as the steady state probabilities converge. In this example, $\frac{\pi_{(10,112)}}{\pi_{(10,111)}} = 4.47$ and $\frac{\pi_{(10,111)}}{\pi_{(10,110)}} = 2.29$. Figure 6 (b) depicts the same 20 ratios for $N = 10$ as λ_0 increases from 1 to 10, and illustrates that the severity of the discrepancy decreases as λ_0 increases.

The change in probability between each successive pair of states from N^2+N+1 to $(N+1)^2$ is hence underapproximated by $\frac{\lambda_2}{\lambda_1}$ and hence $\frac{\pi_{(N,(N+1)^2)}}{\pi_{(N,N^2+N+1)}} < (\frac{\lambda_2}{\lambda_1})^N$. However the same is not true of the states from N^2+1 to N^2+N+1, where the ratio of probabilities of successive pairs of states is smaller than $\frac{\lambda_2}{\lambda_1}$ in all cases except the first pair (N^2+2 to N^2+1). Note however that this overapproximation will become less severe as λ_0 increases (see e.g. Fig. 6 (b)).

A less naïve approximation can be obtained by considering a straight line that always underapproximates the ratios of state probabilities from $\frac{\pi_{(N,N^2+2)}}{\pi_{(N,N^2+1)}}$ to $\frac{\pi_{(N,N^2+N+1)}}{\pi_{(N,N^2+N)}}$. Such an underapproximation is given by $\prod_{x=1}^{N}((\frac{x}{N})(\frac{\lambda_2}{\lambda_1}))$, where the change in probability from state N^2+1 to state N^2+N+1 is given by the straight line from $\frac{\lambda_2}{\lambda_1}$ (ratio between states N^2+2 to N^2+1) to $\frac{1}{N}\frac{\lambda_2}{\lambda_1}$ (the underapproximated ratio between states N^2+N+1 and N^2+N). The idea is that all points along this line will give ratios less than the actual ratios observed when moving from the upper right vertex to the right-most centre line state. The product, $\prod_{x=1}^{N}((\frac{x}{N})(\frac{\lambda_2}{\lambda_1}))$, can be simplified to $(\frac{N!}{N^N})(\frac{\lambda_2}{\lambda_1})^N$, and hence the ratio of $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,N^2+1)}$ can be expressed as $(\frac{N!}{N^N})(\frac{\lambda_2}{\lambda_1})^{2N}$.

6 General Approximation Method Based on Convergence

It is desirable to find a method by which the value of N_{cdp} can be determined for a particular configuration (values of firing rates) and a particular desired accuracy, without having to perform numerous TimeNET experiments to determine N_{cdp} by observation.

The heuristics presented in this section are based on the premise that as N increases, probabilities have converged by the time an increment in N results only in additional probabilities equal to zero (to the desired accuracy) being added to the system. Provided this premise holds, the key is thus to identify which state has the largest of the new probabilities introduced when N is incremented, and to determine the smallest value of N at which this is less than 0.5×10^{-dp}. This value of N should thus overapproximate (or closely approximate) N_{cdp}.

6.1 Heuristic When $\lambda_0 < \min(\lambda_1, \lambda_2)$

When $\lambda_0 < \lambda_1 < \lambda_2$ the probabilities increase to the left of $CTMC_N$ and from top to bottom, hence state 1 has the highest probability and the lower right vertex is the state with the highest probability of the states added each time N is incremented. The probability of being in state $(N+1)^2$ can thus be underapproximated, relative to state 1, by moving left to right along the central horizontal row and then down the right edge to state $(N+1)^2$ as given by

$$\pi_{(N,(N+1)^2)} < \pi_{(N,1)} \left(\frac{\lambda_0}{\lambda_2}\right)^{N-1} \left(\frac{\lambda_2}{\lambda_1}\right)^{N} \left(\frac{\lambda_0}{\lambda_1+\lambda_2}\right)$$

Simplifying, and assuming the worst case, i.e. $\pi_{(N,1)} = 1$, we obtain the following heuristic that can be solved to get the smallest N that satisfies the inequality for a given dp:

$$\left(\frac{\lambda_0}{\lambda_1}\right)^N \left(\frac{\lambda_2}{\lambda_1 + \lambda_2}\right) < 0.5 \times 10^{-dp} \tag{1}$$

6.2 Heuristic When $\min(\lambda_1, \lambda_2) < \lambda_0 < \lambda_1 + \lambda_2$

As discussed in Section 5.1 in this situation probabilities still generally decrease from left to right along the central horizontal row. However, unlike the case when $\lambda_0 < \lambda_1$, it is the lower right vertex with the highest probability (consistent with $\lambda_0 > \min(\lambda_1, \lambda_2)$ and $\lambda_2 > \lambda_1$) and state 1 with the highest of the 'new' probabilities introduced each time N is incremented. Hence, this situation is the reverse of that given in Section 6.1 and the corresponding heuristic is given by

$$\left(\frac{\lambda_1}{\lambda_0}\right)^N \left(\frac{\lambda_1 + \lambda_2}{\lambda_2}\right) < 0.5 \times 10^{-dp} \tag{2}$$

6.3 Heuristic When $\lambda_0 \geq \lambda_1 + \lambda_2$

When $\lambda_0 \geq \lambda_1 + \lambda_2$ probabilities increase left to right along the central horizontal row and increase down toward the lower right vertex, hence state $(N+1)^2$ has the highest probability. Unlike the situation in Section 6.2, we have observed that the largest of the 'new' probabilities introduced each time N is incremented is either $\pi_{(N,1)}$ (left vertex) or $\pi_{(N,N^2+1)}$ (upper right vertex), which seems to be influenced by the relative ratios of $\frac{\lambda_0}{\lambda_2}$ and $\frac{\lambda_2}{\lambda_1}$, although further experimentation is required to ascertain the exact nature of the relationship. Hence, we derive two heuristic expressions based on $\pi_{(N,1)}$ and $\pi_{(N,N^2+1)}$ relative to $\pi_{(N,(N+1)^2)}$ using the progressions from Section 5:

$$\left(\frac{\lambda_1}{\lambda_2}\right)^N \left(\frac{\lambda_1 + \lambda_2}{\lambda_0}\right)^N < 0.5 \times 10^{-dp} \tag{3}$$

and

$$\left(\frac{N^N}{N!}\right) \left(\frac{\lambda_1}{\lambda_2}\right)^{2N} < 0.5 \times 10^{-dp} \tag{4}$$

and take as our approximation of N_{cdp} whichever value of N is largest.

6.4 Results and Discussion

Of the 1728 cases examined (216 configurations, each examined for convergence to 3, 4, 5, 6, 7, 8, 9 and 10 decimal places), convergence was reached by $N = 50$ in 1274 of those cases. The cases where convergence was not reached by $N = 50$ are primarily when $\lambda_0 = \lambda_1 = \lambda_2$, $\lambda_1 = \lambda_2$ for $\lambda_0 > \lambda_1$, and as λ_0 approaches λ_1 from above or below, for increasingly smaller dp.

Results when $\boldsymbol{\lambda_0 < \min(\lambda_1, \lambda_2)}$. Of the 832 cases examined in which $\lambda_0 < \lambda_1$, 688 reached convergence by $N = 50$. Our heuristic gave an N_{cdp} at least as high as the actual N_{cdp} in 620 of those 688 cases (90.1%). Table 3 shows these results for N_{c4}. The majority of the 68 cases where our heuristic failed are due to rounding anomalies: only 36 cases fail when increasing the predicted N_{cdp} values by 1, and only 23 fail when increasing the predicted N_{cdp} values by 2. Rounding anomalies are unavoidable when specifying convergence to a certain number of decimal places. It is likely that the approximate probability will be different from the actual probability in such cases by only $\pm 10^{-dp}$. We thus propose three strategies for reducing the influence of rounding anomalies: consider adding a small number to the predicted value of N_{cdp}; determine N_{cdp} for a slightly higher number of decimal places of accuracy than needed; or take the predicted value of N_{cdp} to give probabilities accurate to $\pm 10^{-dp}$.

Results when $\boldsymbol{\lambda_0 > \min(\lambda_1, \lambda_2)}$. Of the 832 cases examined in which $\lambda_0 > \lambda_1$, 586 reached convergence by $N = 50$. Our heuristic performs more poorly here than when $\lambda_0 < \lambda_1$, as only 424 (72.4%) give N_{cdp} at least as large as the actual value. Two clear trends are evident. One is that the heuristic fails more often for larger λ_0 and larger λ_2. $\lambda_0 \geq 5$ and $\lambda_2 \geq 3$ (56.3% of the convergent cases) account for 137 of the 162 failed cases (84.6%). The other is that the

Table 3. Success of our heuristic in predicting an upper bound for N_{c4}.

$\frac{\lambda_0}{\lambda_1}$	$\frac{\lambda_2}{\lambda_1}$	N_{c4}		$\frac{\lambda_2}{\lambda_1}$	N_{c4}		$\frac{\lambda_2}{\lambda_1}$	N_{c4}		$\frac{\lambda_2}{\lambda_1}$	N_{c4}	
		actual	heuristic		actual	heuristic		actual	heuristic		actual	heuristic
0.01	1	2	3 √	2	3	3 √	4	2	3 √	8	2	3 √
	1.5	2	3 √	3	2	3 √	5	2	3 √	10	2	3 √
0.02	1	3	3 √	2	2	3 √	4	2	3 √	8	2	3 √
	1.5	2	3 √	3	2	3 √	5	3	3 √	10	2	3 √
0.05	1	4	4 √	2	3	4 √	4	3	4 √	8	3	4 √
	1.5	3	4 √	3	4	4 √	5	3	4 √	10	3	4 √
0.1	1	4	5 √	2	5	5 √	4	4	5 √	8	4	5 √
	1.5	4	5 √	3	4	5 √	5	4	5 √	10	4	5 √
0.125	1	4	5 √	2	4	5 √	4	4	5 √	8	4	5 √
	1.5	4	5 √	3	5	5 √	5	4	5 √	10	4	5 √
0.2	1	7	6 ×	2	6	6 √	4	8	7 ×	8	6	7 √
	1.5	5	6 √	3	5	6 √	5	7	7 √	10	5	7 √
0.25	1	7	7 √	2	7	7 √	4	6	7 √	8	6	8 √
	1.5	6	7 √	3	7	7 √	5	6	8 √	10	7	8 √
0.333	1	9	9 √	2	9	9 √	4	8	9 √	8	11	9 ×
	1.5	8	9 √	3	9	9 √	5	9	9 √	10	8	9 √
0.5	1	13	14 √	2	16	14 ×	4	12	14 √	8	15	15 √
	1.5	14	14 √	3	13	14 √	5	12	15 √	10	13	15 √
0.666	1	24	23 ×	2	23	24 √	4	30	24 ×	8	21	25 √
	1.5	30	24 ×	3	28	24 ×	5	27	24 ×	10	22	25 √
0.75	1	35	33 ×	2	35	34 ×	4	38	34 ×	8	40	35 ×
	1.5	39	33 ×	3	33	34 √	5	30	34 √	10	35	35 √
0.8	1	45	42 ×	2	44	43 ×	4	43	44 √	8	48	44 ×
	1.5	44	43 ×	3	36	44 √	5	45	44 ×	10	46	44 ×
0.9	1	≥ 50	88 ?	2	≥ 50	91 ?	4	≥ 50	92 ?	8	≥ 50	93 ?
	1.5	≥ 50	90 ?	3	≥ 50	92 ?	5	≥ 50	93 ?	10	≥ 50	94 ?

Table 4. Convergence of $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,(N+1)^2-8)}$ as N increases for $\lambda_0 = 100$, $\lambda_1 = 1$ and $\lambda_2 = 10$ (A denotes $(N+1)^2$)

	$\pi_{(N,A)}$	$\pi_{(N,A-1)}$	$\pi_{(N,A-2)}$	$\pi_{(N,A-3)}$	$\pi_{(N,A-4)}$	$\pi_{(N,A-5)}$	$\pi_{(N,A-6)}$	$\pi_{(N,A-7)}$	$\pi_{(N,A-8)}$
N=1	0.89206	0.08921	0.00892						
N=2	0.89101	0.08910	0.00883	0.00096	0.00010				
N=3	0.89100	0.08910	0.00891	0.00088	0.00010	0.00001	0.00000		
N=4	0.89100	0.08910	0.00891	0.00089	0.00009	0.00001	0.00000	0.00000	0.00000
N=5	0.89100	0.08910	0.00891	0.00089	0.00009	0.00001	0.00000	0.00000	0.00000
N=6	0.89100	0.08910	0.00891	0.00089	0.00009	0.00001	0.00000	0.00000	0.00000

heuristic fails more often for larger dp. For $dp = 3$ only 9 of the 85 convergent cases fails (10.6%), whereas for $dp = 10$ this has risen to 21 out of 55 (38%).

Closer investigation revealed that some of the failed cases are due to rounding anomalies: increasing the predicted N_{cdp} by 1 or by 2 reduces the number of failed cases from 162 to 97 and 46 respectively. However, we discovered that not all failures are due to rounding anomalies. In those cases, it wasn't the approximation of the progressions that failed in these cases, as they successfully determined an upper bound on the value of N by which an increment in N resulted only in additional probabilities of zero. Rather, it was the premise that all non-zero probabilities have converged by this same value of N that did not hold.

As an example, consider $\lambda_0 = 100, \lambda_1 = 1$ and $\lambda_2 = 10$. Table 4 gives the probabilities of the states along the right edge of the CTMC for $N = 1$ to 4, and the last 9 probabilities (starting from state $(N+1)^2$) for $N = 5$ and 6. Our heuristic predicts that $N_{c5} = 3$, i.e. that by $N = 3$ only zero probabilities will be added when N increments to 4 and beyond. Table 4 shows this to be true for $N = 4$ for the states visible, and this has been confirmed for all other states for $5 \leq N \leq 50$. However, Table 4 shows that both $\pi_{(N,(N+1)^2-3)}$ and $\pi_{(N,(N+1)^2-4)}$ don't converge to 5 decimal places until $N_{c5} = 4$.

7 Conclusions

This paper has presented a summary of the results of an extensive study of the steady state analysis of a parametric Generalised Stochastic Petri Net (GSPN) with a fork-join subnet. The GSPN has relevance to the coordination and synchronisation of computer controlled manufacturing systems which produce two components on demand for assembly into a product. Components 1 and 2 are made at rates λ_1 and λ_2 respectively, with product requests arriving at rate λ_0. The GSPN implements a control which prevents the production of one component exceeding the other by more than N. Using TimeNET, we obtained the steady state probabilities for a wide range of values of the rates, demonstrating convergence as N increases. The nature of this convergence has been reported. Thus approximate results to dp decimal places have been obtained for a GSPN with arbitrarily large N from a GSPN where the steady state probabilities have converged to the same number of decimal places.

We introduced the notation N_{cdp} for the value of N when all probabilities have converged to dp decimal places. For convergence to be useful, N_{cdp} needs to be calculated in a reasonable amount of time. From our experiments with TimeNET, we found that useful convergence occurs for $dp = 3$ when either $\frac{\lambda_0}{\lambda_1}$ is less than about 0.9, or both $\frac{\lambda_0}{\lambda_1}$ is greater than about 1.2 and $\frac{\lambda_2}{\lambda_1}$ is greater than about 1.5. We have also observed that N_{cdp} generally increases linearly with increasing dp.

Analysing the progressions in the converged steady state probabilities enabled the creation of a heuristic to predict the value of N_{cdp} as a function of the transition rates and dp. The heuristic is based on the premise that probabilities have converged to dp decimal places by the time an increment in N results only in the introduction of additional probabilities of zero (i.e. less than 0.5×10^{-dp}). This heuristic was found to work well for a large range of transition rates, hence confirming this premise and allowing the steady state probabilities to be obtained for arbitrary N for a very large range of transition rates in the range of useful convergence. The heuristic fails to produce sufficiently large N_{cdp} in many cases where $\frac{\lambda_0}{\lambda_1} >\approx 5$ and $\frac{\lambda_2}{\lambda_1} >\approx 3$. In these cases, we discovered that the premise did not hold, and that the discrepancy increases with increasing dp.

In the future we would like to improve the heuristic for $\frac{\lambda_0}{\lambda_1} > 5$ and $\frac{\lambda_2}{\lambda_1} > 3$, by re-examining our choice of progressions in the steady state probabilities. Further, a closer study of the progressions may allow us to derive closed form approximations for the steady state probabilities. Finally, many manufacturing processes comprise more than two production lines. To cater for this our model could be extended to any number of branches in the fork-join subnet. This adds a new dimension to the problem. Similar convergence trends observed in such systems have not yet been characterised, which provides us with a significant challenge.

Acknowledgements

We would like to thank the reviewers for their constructive comments.

References

1. Baccelli, F., Massey, W.A., Towsley, D.: Acyclic fork-join queuing networks. Journal of the ACM 36(3), 615–642 (1989)
2. Flatto, L., Hahn, S.: Two Parallel Queues Created by Arrivals with Two Demands I. SIAM Journal of Applied Mathematics 44, 1041–1053 (1984)
3. Flatto, L.: Two Parallel Queues Created by Arrivals with Two Demands II. SIAM Journal of Applied Mathematics 45, 861–878 (1985)
4. Nelson, R., Towsley, D., Tantawi, A.N.: Performance Analysis of Parallel Processing Systems. IEEE Transactions on Software Engineering 14(4), 532–540 (1988)
5. Nelson, R., Tantawi, A.N.: Approximate Analysis of Fork/Join Synchronization in Parallel Queues. IEEE Transactions on Computers 37(6), 739–743 (1988)
6. Kim, C., Agrawala, A.K.: Analysis of the Fork-Join Queue. IEEE Transactions on Computers 38(2), 250–255 (1989)

7. Liu, Y.C., Perros, H.G.: A Decomposition Procedure for the Analysis of a Closed Fork/Join Queueing System. IEEE Transactions on Computers 40(3), 365–370 (1991)
8. Lui, J.C.S., Muntz, R.R., Towsley, D.: Computing Performance Bounds of Fork-Join Parallel Programs under a Multiprocessing Environment. IEEE Transactions on Parallel and Distributed Systems 9(3), 295–311 (1998)
9. Makowski, A., Varma, S.: Interpolation Approximations for Symmetric Fork-Join Queues. Performance Evaluation 20, 145–165 (1994)
10. Varki, E.: Mean value technique for closed fork-join networks. In: SIGMETRICS 1999: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 103–112 (1999)
11. Varki, E.: Response Time Analysis of Parallel Computer and Storage Systems. IEEE Transactions on Parallel and Distributed Systems 12(11), 1146–1161 (2001)
12. Harrison, P., Zertal, S.: Queueing models of RAID systems with maxima of waiting times. Performance Evaluation 64, 664–689 (2007)
13. Song, J.S., Zipkin, P.: Supply chain operations: Assemble-to-order systems. In: Handbook in OR&MS, ch. 11, vol. 11, pp. 561–596. Elsevier, Amsterdam (2003)
14. Bause, F., Kritzinger, P.S.: Stochastic Petri Nets - An Introduction to the Theory, 2nd edn. Vieweg-Verlag (2002)
15. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley and Sons, Chichester (1995)
16. Lilith, N., Billington, J., Freiheit, J.: Approximate Closed-Form Aggregation of a Fork-Join Structure in Generalised Stochastic Petri Nets. In: Proc. 1st Int. Conference on Performance Evaluation Methodologies and Tools, Pisa, Italy. International Conference Proceedings Series, Article 32, vol. 180, 10 p. ACM Press, New York (2006)
17. Doğançay, K.: An Asymptotic Convergence Result for the Aggregation of Closed Fork-Join Generalised Stochastic Petri Nets. In: Proc. IEEE Region 10th Conference, TENCON 2009, Singapore, November 23-26, 6 p. (2009)
18. Billington, J., Gallasch, G.E.: Steady State Markov Analysis of a Controlled Fork-Join Network. Technical Report CSEC-41, Computer Systems Engineering Centre Report Series, University of South Australia (June 2010)
19. University of Ilmenau: TimeNET, `http://www.tu-ilmenau.de/TimeNET`

Lattice-Valued Binary Decision Diagrams*

Gilles Geeraerts, Gabriel Kalyon**, Tristan Le Gall, Nicolas Maquet**, and Jean-Francois Raskin

Université Libre de Bruxelles – Dépt. d'Informatique (méthodes formelles et vérification)
{gigeerae,gkalyon,tlegall,nmaquet,jraskin}@ulb.ac.be

Abstract. This work introduces a new data structure, called Lattice-Valued Binary Decision Diagrams (or LVBDD for short), for the compact representation and manipulation of functions of the form $\theta : 2^{\mathsf{P}} \mapsto \mathcal{L}$, where P is a finite set of Boolean propositions and $\mathcal{L}$ is a finite distributive lattice. Such functions arise naturally in several verification problems. LVBDD are a natural generalisation of multi-terminal ROBDD which exploit the structure of the underlying lattice to achieve more compact representations. We introduce two canonical forms for LVBDD and present algorithms to symbolically compute their conjunction, disjunction and projection. We provide experimental evidence that this new data structure can outperform ROBDD for solving the finite-word LTL satisfiability problem.

1 Introduction

Efficient symbolic data structures are often the cornerstone of efficient implementations of model-checking algorithms [5]. Tools like SMV [6] and NuSMV [8] have been applied with success to industrial-strength verification problems. These tools exploit reduced ordered binary decision diagrams [10] (ROBDD for short) which is the reference data structure that has been designed to compactly encode and manipulate Boolean functions i.e., functions of the form $\theta : 2^P \to \{0, 1\}$. ROBDD are often regarded as the *basic toolbox* of verification, and are, indeed, a *very general symbolic data structure*.

The dramatic success of ROBDD in the field of computer aided verification has prompted researchers to introduce several variants of ROBDD [1, 2] or to extend them to represent other kinds of functions[3, 12]. For instance, MTBDD [12] have been successfully applied to the representation of functions of the form $2^P \mapsto D$, where D is an *arbitrary domain*. However, MTBDD work best when the set D is small, and make no assumptions about the structure of D. In this paper, we introduce a new data structure, called *lattice-valued binary decision diagrams* (or LVBDD for short) to efficiently handle *lattice-valued Boolean functions* (LVBF), i.e. functions of the form $\theta : 2^P \mapsto \mathcal{L}$, where P is a finite set of Boolean propositions and $\mathcal{L}$ is a *finite distributive lattice*.

* Work supported by the projects: (i) Quasimodo: "Quantitative System Properties in Model-Driven-Design of Embedded System", http://www.quasimodo.aau.dk/, (ii) Gasics: "Games for Analysis and Synthesis of Interactive Computational Systems", http://www.ulb.ac.be/di/gasics/, (iii) Moves: "Fundamental Issues in Modelling, Verification and Evolution of Software", http://moves.ulb.ac.be, a PAI program funded by the Federal Belgian Government, (iv) CFV (Federated Center in Verification) funded by the FNRS http://www.ulb.ac.be/di/ssd/cfv/

** These authors are supported by an FNRS-FRIA grant.

$$\begin{tikzcd}
 & \top \arrow[dl] \arrow[d] \arrow[dr] & \\
A\{1,2\} \arrow[d] \arrow[dr] & A\{1,3\} \arrow[dl] \arrow[dr] & A\{2,3\} \arrow[dl] \arrow[d] \\
A\{1\} \arrow[dr] & A\{2\} \arrow[d] & A\{3\} \arrow[dl] \\
 & \bot &
\end{tikzcd}$$

c_1	c_2	c_3	$\theta(v)$
0	0	0	$A_{\{1,2\}}$
0	0	1	$A_{\{1,2\}}$
0	1	0	$\bot$
0	1	1	$A_{\{1\}}$
1	0	0	$A_{\{3\}}$
1	0	1	$A_{\{3\}}$
1	1	0	$A_{\{1,3\}}$
1	1	1	$A_{\{1,3\}}$

Fig. 1. The lattice $\mathcal{L}_A$ and the truth table of an LVBF $\theta : 2^{\{c_1,c_2,c_3\}} \mapsto \mathcal{L}_A$

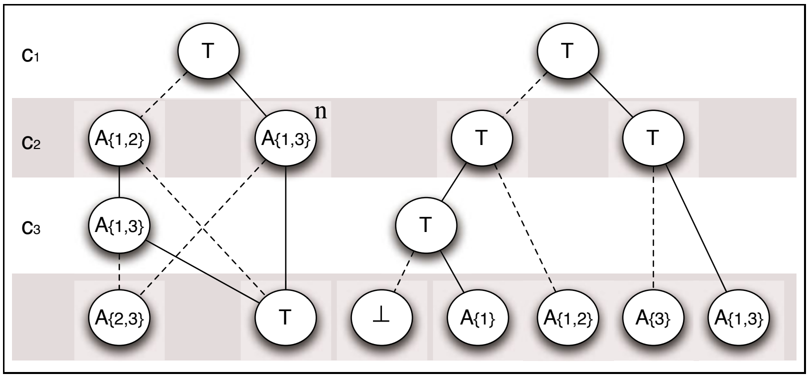


Fig. 2. Two examples of LVBDD resp. in SNF (left) and UNF (right) representing the LVBF θ

In our opinion, such an efficient data structure has potentially many applications, as many examples of algorithms manipulating LVBF can be found in the literature. A first example is the *transition relation of an alternating finite state automaton* (AFA for short). It is well-known that the transition relation of an AFA is an LVBF of the form $2^P \mapsto \mathcal{L}_U$, where $\mathcal{L}_U$ is the *(finite distributive) lattice of upward-closed sets of sets of locations of the automaton*. Exploiting the particular structure of this transition relation is of crucial importance when analyzing AFA, as recently shown in the antichain line of research [13, 23]. This application is the case study we have retained in this paper. A second example, is the field of *multi-valued logics* where formulas have Boolean variables but truth values are taken in an arbitrary finite set, usually a lattice. Such multi-valued formula occur in the field of *multi-valued model checking* [7]. Finally, LVBDD could also be useful in the field of *abstract interpretation* [4], where they would allow to manipulate more efficiently the abstract domains that mix Boolean variables (to encode some control state for instance), and some numerical abstract domain (to encode more fine-grained information about the numerical variables).

Syntactically, LVBDD are an extended form of MTBDD where each node is labeled with a lattice value. The lattice value associated to a path from the root of an LVBDD to a terminal node (which corresponds to a valuation of the Boolean variables) is computed by taking the *greatest lower bound* of the lattice values along the path. Let us provide a simple example; Fig. 1 illustrates a finite distributive lattice $\mathcal{L}_A$, along with an LVBF θ over the propositions c_1, c_2, c_3 and $\mathcal{L}_A$. Intuitively, the lattice $\mathcal{L}_A$ can be seen as the possible truth values of a logic that models the potential disagreement of three observers (for instance, $A_{\{1,2\}}$ represents the situation where the observers 1 and 2 say "true" while observer 3 says "false"). In that context, the function θ represents the "truth statement" of each of the three observers for each valuation. Two different LVBDD that represent θ are illustrated at Fig.2. By following the paths going from the root to a terminal node, one can easily check that both LVBDD represent the function θ.

The LVBDD of Fig. 2 illustrate the fact that several syntactically different LVBDD can represent the same LVBF. In practice, it is often desirable to have *canonical data structures* so we introduce two normal forms for LVBDD, which we respectively call *shared* (SNF) and *unshared* normal form (UNF). The unshared normal form is very similar to MTBDD as it requires that each non-terminal node be labelled with the largest value $\top$ of the lattice. On the other hand, the shared normal form can achieve a much more compact representation of LVBF by exploiting the structure of the lattice to share redundant information along its paths. We show that the SNF has the potential to be *exponentially more compact* than LVBDD in UNF at the price of worst-case-exponential algorithms for the disjunction and intersection. In Section 5, we provide experimental evidence that this exponential behavior is often avoided in practice.

Let us provide some intuition on the shared normal form of LVBDD, using the example of Fig 2. The main idea of the SNF is quite simple: each node is always labeled with the *least possible value with respect to its descendants, and the greatest possible value with respect to its ancestors*, all the while preserving the desired semantics. Let us denote by $\mathsf{paths}(n)$ the set of paths descending from a node n to a terminal node, and let us denote by $\mathsf{value}(\pi)$ the greatest lower bound of the lattice values along a path π. Because of the semantics we have just sketched for LVBDD, one can see that the least possible label of a node n, with respect to its descendants, is exactly $\mathsf{lub}(n) \equiv \bigsqcup \{\mathsf{value}(\pi) \mid \pi \in \mathsf{paths}(n)\}$. In effect, the lattice value $\mathsf{lub}(n)$ *synthesizes the common information* shared by the subgraph rooted by n. This information can be further exploited by the nodes in the subgraph rooted by n in order to have a labeling that contains as few redundancies as possible. More formally, let ℓ be the label of n, and n' be a descendant of n labeled by ℓ'; it is easy to see that we can replace the label of n' by *the largest lattice value* ℓ'' such that $\ell \sqcap \ell'' = \ell'$. We show in Section 2 that this *factorization* operation is well-defined and corresponds to known lattice-theoretic notions. The reader can verify that by applying successive information-sharing and factorization steps repeatedly on the UNF LVBDD on the right of Fig. 2, we obtain the SNF LVBDD on the left. This SNF LVBDD is more compact because the information-sharing and factorization operations have allowed to increase the sharing in the resulting graph.

Related works. While LVBDD represent functions of the form $\theta : 2^{\mathsf{P}} \mapsto \mathcal{L}$, other structures to represent similar but different kinds of functions have been studied in the literature. ROBDD [10], and some variants like ZBDD [1] and *Boolean Expression Diagrams* [2], encode purely Boolean functions $\theta : 2^{\mathsf{P}} \mapsto \{0, 1\}$. MTBDD [12] represent functions of the form $2^{\mathsf{P}} \mapsto D$, but do not exploit the structure of D when it is a lattice. *Edge-Shifted Decision Diagrams* [3] represent functions of the form $\mathcal{L}^{\mathsf{P}} \mapsto \mathcal{L}$ and have been applied to Multi-Valued Model Checking. Finally, *Lattice Automata* [11] represent functions of the form $\Sigma^* \mapsto \mathcal{L}$, where Σ is a finite alphabet.

Structure of the paper. In Section 2, we the define some basic lattice-theoretic notions. In Section 3, we formally define LVBDD and their normal forms. In Section 4, we describe the algorithms to manipulate LVBDD and discuss their worst-case complexity. Finally Section 5 presents experimental evidence that LVBDD-based algorithms can outperform state-of-the-art tools in the context of finite-word LTL satisfiability.

2 Preliminaries

Boolean variables. Let P be a finite set of Boolean propositions. A *valuation* (also called *truth assignment*) $v : \mathsf{P} \mapsto \{0,1\}$ is a function that associates a truth value to each proposition. We denote by 2^{P} the set of all valuations over P. For $v \in 2^{\mathsf{P}}$, $p \in \mathsf{P}$ and $i \in \{0,1\}$, we denote by $v|_{p=i}$ the valuation v' such that $v'(p) = i$ and $v'(p') = v(p')$ for all $p' \in \mathsf{P} \setminus \{p\}$.

Lattices. A finite lattice is a tuple $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$ where L is a finite set of elements, $\sqsubseteq \subseteq L \times L$ is a partial order on L; $\top$ and $\bot$ are two elements from L such that: for all $x \in L : \bot \sqsubseteq x \sqsubseteq \top$; and for all x, y there exists a unique greatest lower bound denoted by $x \sqcap y$ and a unique least upper bound denoted by $x \sqcup y$. A finite lattice $\langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap, \rangle$ is *distributive* (FDL for short) iff, for all x, y, z in L: $(x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z)$ and $(x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$.

Example 1. The lattice of the introduction is $\langle L_A, \sqsubseteq_A, \top = A_{\{1,2,3\}}, \bot = A_{\emptyset}, \sqcup_A, \sqcap_A \rangle$ where $L_A = \{A_I \mid I \subseteq \{1,2,3\}\}$, $A_I \sqsubseteq_A A_K$ iff $I \subseteq K$, $A_I \sqcup_A A_K = A_{I \cup K}$ and $A_I \sqcap_A A_K = A_{I \cap K}$. Another example that will be useful in the sequel is the lattice $\mathcal{L}_{\mathsf{UC}(S)}$ (for a finite set S) of *upward-closed sets of cells of* S. We call a *cell* any finite subset of S. A set of cells U is *upward-closed* iff for any $c \in U$, for any c' s.t. $c \subseteq c'$: $c' \in U$ too. Given a finite set of cells C, we denote by $\uparrow C$ the upward-closure of C, i.e., the set $\{c' \mid \exists c \in C : c \subseteq c'\}$. We denote by $\mathsf{UC}(S)$ the set of all upward-closed sets of cells of S. Then, $\mathcal{L}_{\mathsf{UC}(S)} = \langle \mathsf{UC}(S), \subseteq, \{\emptyset\}, \{S\}, \cup, \cap \rangle$. These two lattices are clearly finite and distributive.

Lattice-valued Boolean functions. Now, let us formalise the notion of *lattice-valued Boolean function*, which is the type of functions that we want to be able to represent and manipulate thanks to LVBDD. For a set of Boolean propositions P and an FDL $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, a *lattice-valued Boolean function* (LVBF for short) over P and $\mathcal{L}$ is a function $\theta : 2^{\mathsf{P}} \mapsto L$. We denote by $\mathsf{LVBF}(\mathsf{P}, \mathcal{L})$ the set of all LVBF over Boolean propositions P and FDL $\mathcal{L}$. We use the shorthands $\theta_1 \sqcap \theta_2$ for $\lambda v \cdot \theta_1(v) \sqcap \theta_2(v)$; $\theta_1 \sqcup \theta_2$ for $\lambda v \cdot \theta_1(v) \sqcup \theta_2(v)$, d for $\lambda v \cdot d$ (with $d \in L$), p for $\lambda v \cdot$ if $v(p) = 1$ then $\top$ else $\bot$ and $\neg p$ for $\lambda v \cdot$ if $v(p) = 0$ then $\top$ else $\bot$ (with $p \in \mathsf{P}$). Given an LVBF θ over P, we denote its *existential quantification* by $\exists \mathsf{P} \cdot \theta \equiv \bigsqcup \{\theta(v) \mid v \in 2^{\mathsf{P}}\}$. For $p \in \mathsf{P}$ and $i \in \{0,1\}$, we define $\theta|_{p=i}$ as the LVBF $\theta' : 2^{\mathsf{P}} \mapsto L$ such that $\theta'(v) = \theta(v|_{p=i})$ for all $v \in 2^{\mathsf{P}}$. Finally, the *dependency set* $I_\theta \subseteq \mathsf{P}$ of an LVBF θ is the set of propositions over which θ depends; formally $I_\theta = \{p \in \mathsf{P} \mid \theta|_{p=0} \neq \theta|_{p=1}\}$.

Example 2. We consider again the lattice $\mathcal{L}_A$ (see Example 1). An example (using the shorthands defined above) of LVBF of the form $2^{\{c_1,c_2,c_3\}} \mapsto \mathcal{L}_A$ is $\theta' \equiv A_{\{1,3\}} \sqcap \big(c_2 \sqcup (\neg c_2 \sqcap A_{\{2,3\}})\big)$. For instance, $\theta'(\langle 1,0,0\rangle) = \theta'(\langle 1,0,1\rangle) = A_{\{1,3\}} \sqcap \big(\bot \sqcup (\top \sqcap A_{\{2,3\}})\big) = A_{\{1,3\}} \sqcap A_{\{2,3\}} = A_{\{3\}}$. Also, $\exists\{c_1,c_2,c_3\} \cdot \theta' = A_{\{1,3\}} \sqcup A_{\{3\}} = A_{\{1,3\}}$.

Recall from the introduction that LVBDD in shared normal form attempt to make the representation of LVBF more compact by *common information-sharing* and *factorization*. Let us formalize these intuitive notions. The idea of *common information-sharing* between lattice values $\ell_1,\ldots,\ell_n$ corresponds to the *least upper bound* $\ell_1 \sqcup \cdots \sqcup \ell_n$ of these values. This can be seen on node n in Fig. 2. All paths containing n correspond to valuations where $c_1 = 1$. In that case, the LVBF θ (see Fig. 1) returns either $A_{\{3\}}$ or $A_{\{1,3\}}$. Hence, n is labelled by the common information $A_{\{3\}} \sqcup A_{\{1,3\}} = A_{\{1,3\}}$. To formalize the idea of *factorization*, we need an additional lattice operator, namely the *relative pseudocomplement* [17]. In an FDL, the *pseudocomplement of x relative to y*, denoted by $x \rightarrow y$, is the *largest* lattice value z s.t. $z \sqcap x = y$. For instance, in $\mathcal{L}_A$, $A_{\{1,3\}} \rightarrow A_{\{3\}} = A_{\{2,3\}}$ (see Fig. 1). Thus, once n has been labelled by $A_{\{1,3\}}$, the label $A_{\{3\}}$ of its left child can be replaced by $A_{\{1,3\}} \rightarrow A_{\{3\}}$, i.e. $A_{\{2,3\}}$.

Relative pseudocomplement. Let $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap\rangle$ be a lattice. For any x, y in L we consider the set $\{z \mid z \sqcap x \sqsubseteq y\}$. If this set has a *unique* maximal element, we call this element the *pseudocomplement of x relative to y* and denote it by $x \rightarrow y$, otherwise $x \rightarrow y$ is undefined. We extend this notion to LVBF $(\mathsf{P}, \mathcal{L})$ as follows. Let $x \in L$; if $x \rightarrow \theta(v)$ is defined for all $v \in 2^{\mathsf{P}}$, then $x \rightarrow \theta$ is defined as $\lambda v \cdot x \rightarrow \theta(v)$. Otherwise, $x \rightarrow \theta$ is undefined. In the case where $\mathcal{L}$ is an FDL, one can easily show that $x \rightarrow y$ is defined for any pair x, y. Moreover, when $x \sqsupseteq y$, $x \rightarrow y$ is the greatest element z such that $z \sqcap x = y$.

Lemma 1. *For all FDL $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap\rangle$, for all $x, y \in L$: $x \rightarrow y$ is defined. Moreover, $y \sqsubseteq x$ implies that (i) $(x \rightarrow y) \sqcap x = y$ and that (ii) for all z such that $z \sqcap x = y$: $z \sqsubseteq (x \rightarrow y)$.*

3 Lattice-Valued Binary Decision Diagrams

In this section, we formally define the *lattice-valued binary decision diagrams* (LVBDD for short) data structure. An LVBDD is a symbolic representation of an LVBF. Syntactically speaking, each LVBDD is a directed, rooted, acyclic graph, whose nodes are labeled by two pieces of information: an index, and a lattice value. LVBDD are thus a strict generalization of *reduced ordered binary decision diagrams* [10] (ROBDD). LVBDD are also closely related to *multi-terminal binary decision diagrams* [12] (MTBDD) but do not generalize them since MTBDD can have arbitrary co-domains.

Definition 1. *Given a set of Boolean propositions $\mathsf{P} = \{p_1,\ldots,p_k\}$ and a finite distributive lattice $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap\rangle$, an* LVBDD *$n$ over* P *and $\mathcal{L}$ is: (i) either a* terminal LVBDD $\langle \mathsf{index}(n), \mathsf{val}(n)\rangle$ *where* $\mathsf{index}(n) = k+1$ *and* $\mathsf{val}(n) \in L$*; or (ii) a* non-terminal LVBDD $\langle \mathsf{index}(n), \mathsf{val}(n), \mathsf{lo}(n), \mathsf{hi}(n)\rangle$*, where* $1 \leq \mathsf{index}(n) \leq k$*,* $\mathsf{val}(n) \in L$ *and* $\mathsf{lo}(n)$ *and* $\mathsf{hi}(n)$ *are (terminal or non-terminal)* LVBDD *such that* $\mathsf{index}(\mathsf{hi}(n)) > \mathsf{index}(n)$ *and* $\mathsf{index}(\mathsf{lo}(n)) > \mathsf{index}(n)$.

In the sequel, we refer to LVBDD also as *"*LVBDD *node"*, or simply *"node"*. For any non-terminal node n, we call $\mathsf{hi}(n)$ (resp. $\mathsf{lo}(n)$) the *high-child* (*low-child*) of n. We denote by $\mathsf{LVBDD}(\mathsf{P}, \mathcal{L})$ the set of all LVBDD over P and $\mathcal{L}$. Finally, the set $\mathsf{nodes}(n)$ of an LVBDD n is defined recursively as follows. If n is terminal, then $\mathsf{nodes}(n) = \{n\}$. Otherwise $\mathsf{nodes}(n) = \{n\} \cup \mathsf{nodes}(\mathsf{lo}(n)) \cup \mathsf{nodes}(\mathsf{hi}(n))$. The *number of nodes* of an LVBDD is denoted by $|n|$.

Semantics of LVBDD. The semantics of LVBDD is an LVBF, as sketched in the introduction. Formally, the semantics is defined by the unary function $[\![\cdot]\!] : \mathsf{LVBDD}(\mathsf{P}, \mathcal{L}) \mapsto \mathsf{LVBF}\,(\mathsf{P}, \mathcal{L})$ such that for any $n \in \mathsf{LVBDD}(\mathsf{P}, \mathcal{L})$, $[\![n]\!] = \mathsf{val}(n)$ if n is terminal, and $[\![n]\!] = \mathsf{val}(n) \sqcap \Big((\neg p_{\mathsf{index}(n)} \sqcap [\![\mathsf{lo}(n)]\!]) \sqcup (p_{\mathsf{index}(n)} \sqcap [\![\mathsf{hi}(n)]\!])\Big)$ otherwise.

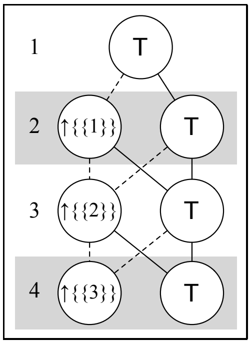
Isomorphisms and reduced LVBDD In order to share common subgraphs in LVBDD, we define a notion of *isomorphism* between LVBDD nodes. Let $n_1, n_2 \in \mathsf{LVBDD}(\mathsf{P}, \mathcal{L})$. We say that n_1 and n_2 are *isomorphic*, denoted by $n_1 \equiv n_2$, iff either (i) n_1 and n_2 are both terminal and $\mathsf{val}(n_1) = \mathsf{val}(n_2)$, or (ii) n_1 and n_2 are both non-terminal and $\mathsf{val}(n_1) = \mathsf{val}(n_2)$, $\mathsf{index}(n_1) = \mathsf{index}(n_2)$, $\mathsf{lo}(n_1) \equiv \mathsf{lo}(n_2)$ and $\mathsf{hi}(n_1) \equiv \mathsf{hi}(n_2)$. An LVBDD n is *reduced* iff (i) for all $n \in \mathsf{nodes}(n)$: either n is terminal or (i) $\mathsf{lo}(n) \neq \mathsf{hi}(n)$ and (ii) for all $n_1, n_2 \in \mathsf{nodes}(n)$: $n_1 \equiv n_2$ implies $n_1 = n_2$.

Normal Forms. It is easy to see that there are LVBF θ for which one can find at least two different *reduced* LVBDD n_1 and n_2 s.t. $[\![n_1]\!] = [\![n_2]\!] = \theta$. For instance, the two LVBDD of Fig. 2 both represent the LVBF θ of Fig. 1. In order to obtain efficient algorithms to manipulate LVBDD, we define *normal forms* that associate to each LVBF a unique LVBDD representing it, up to isomorphism and order of the Boolean propositions. In this work, we define two normal forms for LVBDD: (i) the *unshared normal form* (UNF for short) which is similar to MTBDD, and (ii) the *shared normal form* (SNF for short) in which common lattice values along paths are shared. We associate to each LVBF θ, a unique LVBDD $\mathcal{D}^U(\theta)$ and a unique LVBDD $\mathcal{D}^S(\theta)$ which are respectively the UNF and SNF LVBDD representing θ:

Definition 2 (Unshared normal form). *Let* P *be a set of Boolean propositions and* $\mathcal{L}$ *be an FDL. Then, for all* $\theta \in \mathsf{LVBF}\,(\mathsf{P}, \mathcal{L})$*, the UNF* LVBDD $\mathcal{D}^U(\theta)$ *is the* reduced LVBDD *defined recursively as follows. If* $I_\theta = \emptyset$*, then* $\theta(v) = d$ *for some* $d \in L$*. In this case,* $\mathcal{D}^U(\theta)$ *is the terminal* LVBDD $\langle k+1, d\rangle$*. Otherwise, let* p_i *be the proposition of lowest index in* I_θ*. Then,* $\mathcal{D}^U(\theta)$ *is the non-terminal* LVBDD $\langle i, \top, \mathcal{D}^U(\theta|_{p_i=0}), \mathcal{D}^U(\theta|_{p_i=1})\rangle$.

Definition 3 (Shared normal form). *Let* P *be a set of Boolean propositions and let* $\mathcal{L}$ *be an FDL. Then, for all* $\theta \in \mathsf{LVBF}\,(\mathsf{P}, \mathcal{L})$*, the SNF* LVBDD $\mathcal{D}^S(\theta)$ *is the* reduced LVBDD *defined recursively as follows. If* $I_\theta = \emptyset$*, then* $\theta(v) = d$ *for some* $d \in L$*. In this case,* $\mathcal{D}^S(\theta)$ *is the terminal* LVBDD $\langle k+1, d\rangle$*. Otherwise, let* p_i *be the proposition of lowest index in* I_θ*. Then,* $\mathcal{D}^S(\theta)$ *is the non-terminal* LVBDD $\langle i, \exists \mathsf{P} \cdot \theta, \mathcal{D}^S((\exists \mathsf{P} \cdot \theta) \rightarrow (\theta|_{p_i=0})), \mathcal{D}^S((\exists \mathsf{P} \cdot \theta) \rightarrow (\theta|_{p_i=1}))\rangle$.

It is not difficult to see that for all LVBF θ, Definition 2 and 3 each yield a unique LVBDD $\mathcal{D}^U(\theta)$ and $\mathcal{D}^S(\theta)$. Then, an LVBDD n is in UNF iff $n = \mathcal{D}^U([\![n]\!])$. Similarly, n is in SNF iff $n = \mathcal{D}^S([\![n]\!])$. We denote by $\mathsf{LVBDD}^U(\mathsf{P}, \mathcal{L})$ (resp. $\mathsf{LVBDD}^S(\mathsf{P}, \mathcal{L})$) the set of all LVBDD in UNF (resp. SNF) on set P of Boolean propositions and FDL $\mathcal{L}$.



Example 3. Consider the family of lattice-valued Boolean functions $\theta_i : 2^{\{p_1,\dots p_i\}} \mapsto \mathcal{L}_{\mathsf{UC}(\{1,\dots,i\})}$ for $i \geq 1$, defined as $\theta_i \equiv \sqcap_{1\leq j\leq i}\big(p_j \sqcup \uparrow\{\{j\}\}\big)$. It is easy to see that, for any $i \geq 1$, the SNF LVBDD $\mathcal{D}^S(\theta_i)$ has $2 \times i + 1$ nodes. For instance, $\theta_3 = (p_1 \sqcup \uparrow\{\{1\}\}) \sqcap (p_2 \sqcup \uparrow\{\{2\}\}) \sqcap (p_3 \sqcup \uparrow\{\{3\}\})$ and $\mathcal{D}^S(\theta_3)$ is shown on the left. However, the corresponding MTBDD (or UNF LVBDD) is of exponential size, as for any $v \neq v'$, we have $\theta_i(v) \neq \theta_i(v')$.

4 Algorithms on LVBDD

In this section, we discuss *symbolic* algorithms to manipulate LVBF *via* their LVBDD representation. The proofs of these algorithms have been omitted here, but they can be found in the technical report.

Remark however that we have managed to prove the correctness of the algorithms *for any finite distributive lattice*. This general result can be obtained by exploiting Birkhoff's representation theorem [17], a classical result in lattice theory. Birkhoff's theorem says that any FDL $\mathcal{L}$ is *isomorphic* to a lattice $\hat{\mathcal{L}}$ that has a special structure: its elements are sets of incomparable cells of certain elements of the original lattice $\mathcal{L}$ (the *meet-irreducible elements*). We can thus exploit this structure to prove the algorithms in the special case of $\hat{\mathcal{L}}$ and deduce the correctness of the algorithm on $\mathcal{L}$, thanks to the isomorphism. The interested reader is referred to the technical report for the complete details.

Throughout this section, we assume that we manipulate LVBDD ranging over the set of propositions $\mathsf{P} = \{p_1, \dots, p_k\}$ and the FDL $\mathcal{L} = \langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$. We present algorithms to compute the least upper bound $\sqcup$, the greatest lower bound $\sqcap$, the test for equality, the existential quantification of *all* the Boolean variables and the relative pseudocomplement with a lattice value for LVBDD in SNF. Then, we briefly sketch these operations for UNF LVBDD as they can be easily obtained from the definition, and are very similar to those for MTBDD.

Memory management and memoization. The creation of LVBDD nodes in memory is carried out by function MK: calling $\mathsf{MK}(i, d, \ell, h)$ returns the LVBDD node $\langle i, d, \ell, h \rangle$. As in most BDD packages (see for instance [18]), our implementation exploits caching techniques to ensure that each unique LVBDD is stored only once, even across multiple diagrams. The implementation maintains a global cache that maps each tuple $\langle i, d, \ell, h \rangle$ to a memory address storing the corresponding LVBDD node (if it exists). A call to $\mathsf{MK}(i, d, \ell, h)$ first queries the cache and allocates fresh memory space for $\langle i, d, \ell, h \rangle$ in the case of a cache miss. Thus, MK guarantees that two isomorphic LVBDD always occupy the same memory address, but does not guarantee that any particular normal form is enforced. We assume that cache queries and updates take $\mathcal{O}(1)$ which is what is observed in practice when using a good hash map. We implemented a simple reference counting scheme to automatically free unreferenced nodes from the cache.

We use the standard *memoization* technique in graph traversal algorithms. Each algorithm has access to its own pair of memo! and memo? functions; $\mathsf{memo!}(key, value)$

Algorithm 1. Relative pseudocomplementation for LVBDD in SNF

```
1  begin PseudoCompSNF$(n, d)$
2      if index$(n) = k+1$ then return MK$(k+1, d \rightarrow$ val$(n)$, nil, nil$)$ ;
3      else
4          $d' := (d \rightarrow \mathsf{val}(n)) \sqcap (\mathsf{val}(\mathsf{lo}(n)) \sqcup (\mathsf{val}(\mathsf{hi}(n))))$ ;
5          return MK(index$(n), d'$, lo$(n)$, hi$(n)$) ;
6  end
```

Algorithm 2. Meet of a LVBDD in SNF with a lattice value

```
1  begin ConstMeetSNF$(n, d)$
2      $n' :=$ memo?$(\langle n, d \rangle)$ ;
3      if $n' \neq$ nil then return $n'$ ;
4      else if val$(n) \sqsubseteq d$ then $n' := n$ ;
5      else if val$(n) \sqcap d = \bot$ then $n' :=$ MK$(k+1, \bot$, nil, nil$)$ ;
6      else if index$(n) = k+1$ then $n' :=$ MK$(k+1$, val$(n) \sqcap d$, nil, nil$)$ ;
7      else
8          $\ell :=$ ConstMeetSNF(lo$(n), d)$ ;
9          $h :=$ ConstMeetSNF(hi$(n), d)$ ;
10         if $\ell = h$ then $n' :=$ MK(index$(\ell)$, val$(\ell) \sqcap$ val$(n)$, lo$(\ell)$, hi$(\ell)$) ;
11         else
12             $\ell' :=$ PseudoCompSNF$(\ell, d)$ ; $h' :=$ PseudoCompSNF$(h, d)$ ;
13             $n' :=$ MK(index$(n)$, val$(n) \sqcap d, \ell', h'$) ;
14     memo!$(\langle n, d \rangle, n')$ ;
15     return $n'$ ;
16 end
```

stores a computed value and associates it to a key; memo?(key) returns the previously stored value, or nil if none was found. Both memo! and memo? run in $\mathcal{O}(1)$.

Operations on LVBDD in SNF. The operations on SNF LVBDD and their complexities are summarized in Table 4. The procedure PseudoCompSNF(n, d) (see Algorithm 1) takes an LVBDD in SNF n and a lattice value $d \sqsupseteq$ val(v), and computes the LVBDD n' in SNF such that $[\![n']\!] = d \rightarrow [\![n]\!]$. It runs in $\mathcal{O}(1)$, since it is sufficient to modify the label of the root. The resulting LVBDD is guaranteed to be still in SNF. This procedure will be invoked by the other algorithms to enforce canonicity.

The procedure ConstMeetSNF(n, d) (Algorithm 2), returns the SNF LVBDD representing $[\![n]\!] \sqcap d$, where d is a lattice value. ConstMeetSNF consists in recursively traversing the graph (the two recursive calls at lines 8 and 9 return the new subgraphs ℓ and h), and to call PseudoCompSNF on ℓ and h to enforce canonicity. This procedure runs in $\mathcal{O}(|n|)$, thanks to memoization. The procedure MeetSNF(n_1, n_2) (Algorithm 3) returns the SNF LVBDD representing $[\![n_1]\!] \sqcap [\![n_2]\!]$. The algorithm first performs two recursive calls on n_1 and n_2's respective lo- and hi- sons, which produces LVBDD ℓ and h (not shown on the figure), and computes the value $d =$ val$(n_1) \sqcap$val(n_2) (i.e., $d_1 \sqcap d_2$ on the figure), which will be the label of the root in the result. Then, canonicity of the result is enforced in two steps. First, the conjunction of ℓ and h with d is computed (second step in the figure). Second, the subgraphs returned by the recursive calls are *factorized*

Algorithm 3. Meet of two LVBDD in SNF

```
begin MeetSNF($n_1, n_2$)
    $n'$ := memo?($\langle n_1, n_2 \rangle$) ;
    if $n' \neq$ nil then return $n'$ ;
    else if index($n_1$) $= k+1$ then $n'$ := ConstMeetSNF($n_2$, val($n_1$));
    else if index($n_2$) $= k+1$ then $n'$ := ConstMeetSNF($n_1$, val($n_2$));
    else
        if index($n_1$) = index($n_2$) then
            $\ell$ := MeetSNF(lo($n_1$), lo($n_2$)) ; $h$ := MeetSNF(hi($n_1$), hi($n_2$)) ;
            $d$ := val($n_1$) $\sqcap$ val($n_2$) ;
        else
            if index($n_1$) > index($n_2$) then swap($n_1, n_2$) ;
            $\ell$ := MeetSNF(lo($n_1$), $n_2$) ; $h$ := MeetSNF(hi($n_1$), $n_2$) ;
            $d$ := val($n_1$) ;
        $\ell'$ := ConstMeetSNF($\ell, d$) ; $h'$ := ConstMeetSNF($h, d$) ;
        if $\ell' = h'$ then $n'$ := $\ell'$ ;
        else
            $d'$ := val($\ell'$) $\sqcup$ val($h'$) ;
            $\ell''$ := PseudoCompSNF($\ell', d'$) ; $h''$ := PseudoCompSNF($h', d'$) ;
            $n'$ := MK(index($n_1$), $d', \ell'', h''$) ;
    memo!($\langle n_1, n_2 \rangle, n'$) ; memo!($\langle n_2, n_1 \rangle, n'$) ;
    return $n'$ ;
end
```

w.r.t. the value d, thanks to PseudoCompSNF (third step on the figure). The procedure JoinSNF(n_1, n_2) (Algorithm 4) returns the SNF LVBDD representing $[\![n_1]\!] \sqcup [\![n_2]\!]$.

Both MeetSNF and JoinSNF run in $2^{\mathcal{O}(|n_1|+|n_2|)}$ in the worst case. As an example of worst case for JoinSNF, consider, for any $n \geq 1$, the lattice $\mathcal{L}_{\mathsf{UC}(\{1,\ldots,2n\})}$ and the two LVBF $\theta_n \equiv \sqcap_{1 \leq j \leq n} \big(p_j \sqcup \uparrow\{\{j\}\}\big)$ and $\theta'_n \equiv \sqcap_{1 \leq j \leq n} \big(p_j \sqcup \uparrow\{\{n+j\}\}\big)$. It is easy to check that, for any $n \geq 1$, $|\mathcal{D}^S(\theta_n)| = |\mathcal{D}^S(\theta'_n)| = 2\,n+1$ (see Example 3), but that $\mathcal{D}^S(\theta_n \sqcup \theta'_n)$ has $2^{\mathcal{O}(|n_1|+|n_2|)}$ nodes. A similar example can be built for MeetSNF.

Operations on LVBDD in UNF. For an UNF LVBDD n, $\exists \mathsf{P} : [\![n]\!] = \sqcup_v [\![n]\!](v)$ amounts to $\sqcup_{n' \in N} \mathsf{val}(n')$, where N is the set of terminal nodes of n. This operation is thus in $\mathcal{O}(|n|)$. The computation of the $\sqcap$ and $\sqcup$ operators is done by computing the synchronised product of the two diagrams, similarly to MTBDD [12]. For instance, when n_1 and n_2 are terminal LVBDD in UNF, the UNF LVBDD that represents $[\![n_1]\!] \sqcup [\![n_2]\!]$ is $\langle k+1, \mathsf{val}(n_1) \sqcup \mathsf{val}(n_2) \rangle$. When n_1 and n_2 are non-terminal LVBDD in UNF with the same index, the UNF LVBDD that represents $[\![n_1]\!] \sqcup [\![n_2]\!]$ is obtained by building recursively the LVBDD representing $[\![\mathsf{lo}(n_1)]\!] \sqcup [\![\mathsf{lo}(n_2)]\!]$ and $[\![\mathsf{hi}(n_1)]\!] \sqcup [\![\mathsf{hi}(n_2)]\!]$, and adding a root labelled by $\top$. With memoization, we can achieve polynomial complexity.

5 Empirical Evaluation

In this section, we apply our new data structure to the *satisfiability problem* for the *finite-word linear temporal logic* (LTL for short). In recent works [22, 23], it has been shown

Algorithm 4. Join of two LVBDD in SNF

```
begin ReJoinSNF($n_1, n_2, d_1, d_2$)
    $n' :=$ memo?($\langle n1, n2, d_1, d_2 \rangle$) ;
    if $n' \neq$ nil then return $n'$;
    $d'_1 := d_1 \sqcap$ val($n_1$) ; $d'_2 := d_2 \sqcap$ val($n_2$) ;
    if index($n_1$) $=$ index($n_2$) $= k+1$ then $n' :=$ MK($k+1, d'_1 \sqcup d'_2$, nil, nil) ;
    else
        if index($n_1$) $=$ index($n_2$) then
            $\ell :=$ ReJoinSNF(lo($n_1$), lo($n_2$), $d'_1, d'_2$) ;
            $h :=$ ReJoinSNF(hi($n_1$), hi($n_2$), $d'_1, d'_2$) ;
        else
            if index($n_1$) $>$ index($n_2$) then
                swap($n_1, n_2$) ; swap($d_1, d_2$) ; swap($d'_1, d'_2$) ;
            $\ell :=$ ReJoinSNF(lo($n_1$), $n_2, d'_1, d'_2$) ; $h :=$ ReJoinSNF(hi($n_1$), $n_2, d'_1, d'_2$) ;
        if $\ell = h$ then $n' := \ell$ ;
        else
            $d :=$ val($l$) $\sqcup$ val($h$) ;
            $\ell :=$ PseudoCompSNF($\ell, d$) ; $h :=$ PseudoCompSNF($h, d$) ;
            $n' :=$ MK(index($n_1$), $d, \ell, h$) ;
    memo!($\langle n_1, n_2, d_1, d_2 \rangle, n'$) ; memo!($\langle n_2, n_1, d_2, d_1 \rangle, n'$) ;
    return $n'$ ;
end
begin JoinSNF($n_1, n_2$)
    return ReJoinSNF($n_1, n_2, \top, \top$);
end
```

that algorithms based on both *antichains* and ROBDD, can outperform purely ROBDD-based techniques like the ones implemented in the tools SMV and NuSMV. Our experiments show that an approach based on a combination of antichains and LVBDD in SNF can be even more efficient.

We solve the LTL satisfiability problem by the classical reduction to the language emptiness for *alternating automata* (AFA for short). AFA are a natural generalization of both *non-deterministic* and *universal* automata, as they use both conjunctive and disjunctive constraints to encode the transition relation. Due to lack of space, we do not define AFA formally but we illustrate their semantics on the example AFA of Fig. 3;

Table 1. Time complexity and maximum output size of LVBDD algorithms

Operation	Form	Procedure	Time Complexity	Maximum Size
$[\![n_1]\!] \sqcap [\![n_2]\!]$	UNF	similar to MTBDD	$\mathcal{O}(\lvert n_1 \rvert \, \lvert n_2 \rvert)$	$\lvert n_1 \rvert \, \lvert n_2 \rvert$
$[\![n_1]\!] \sqcup [\![n_2]\!]$	UNF	similar to MTBDD	$\mathcal{O}(\lvert n_1 \rvert \, \lvert n_2 \rvert)$	$\lvert n_1 \rvert \, \lvert n_2 \rvert$
$\exists \mathsf{P} : [\![n]\!]$	UNF	bottom-up propagation	$\mathcal{O}(\lvert n \rvert)$	-
$[\![n_1]\!] \sqcap [\![n_2]\!]$	SNF	MeetSNF(n_1, n_2)	$2^{\mathcal{O}(\lvert n_1 \rvert + \lvert n_2 \rvert)}$	$2^{(\lvert I_{[\![n_1]\!]} \cup I_{[\![n_2]\!]} \rvert + 1)} - 1$
$[\![n_1]\!] \sqcup [\![n_2]\!]$	SNF	JoinSNF(n_1, n_2)	$2^{\mathcal{O}(\lvert n_1 \rvert + \lvert n_2 \rvert)}$	$2^{(\lvert I_{[\![n_1]\!]} \cup I_{[\![n_2]\!]} \rvert + 1)} - 1$
$d \rightarrow [\![n]\!]$	SNF	PseudoCompSNF(n, d)	$\mathcal{O}(1)$	$\lvert n \rvert$
$\exists \mathsf{P} : [\![n]\!]$	SNF	root inspection	$\mathcal{O}(1)$	-

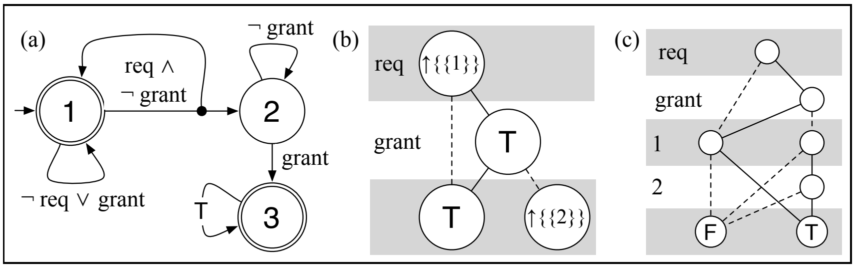


Fig. 3. An AFA (a) with the LVBDD (b) and ROBDD (c) encoding the transitions of location 1

this AFA has three locations $1, 2, 3$, with 1 being the *initial* location and $1, 3$ being the *accepting* locations. A *run* of an AFA is a sequence of sets of locations (called *configurations*); a run is accepting if it ends in a subset of the accepting locations, and is initial if it begins with a configuration that contains the initial location. The language of an AFA is defined as the set of finite words for which the automaton admits an initial accepting run. In the figure, the forked arrows depict the AFA's transitions. Reading a valuation v, the AFA can move from configuration c_1 to c_2 iff for each $\ell \in c_1$ there exists a transition from ℓ labeled by φ such that all target locations are in c_2 and $v \models \varphi$. For example, if the AFA reads the valuation $\mathrm{grant} = 0\ \mathrm{req} = 1$ from $\{1, 3\}$, it must go to $\{1, 2, 3\}$. Alternating automata enjoy the following useful property: the set of successor configurations of any configuration is always *upward-closed* for subset inclusion; this observation is the basis for the antichain-based approach to AFA analysis. We do not recall the framework of antichains here, but it can be found in [14].

The translation from LTL to AFA yields automata which have an alphabet equal to the set of *valuations* of the Boolean propositions of the LTL formula. It is easy to see that, in that case, the transition function of an AFA can be encoded with an LVBF over the set of propositions of the formula and the lattice of upward-closed sets of configurations of the automaton. For example, the LTL formula $\Box(\mathrm{req} \rightarrow \Diamond\mathrm{grant})$ translates to the automaton of Fig. 3 (a), and the LVBF corresponding to the outgoing transitions of location 1 is $\big(\uparrow\{\{1\}\} \sqcap (\neg\mathrm{req} \sqcup \mathrm{grant})\big) \sqcup \big(\uparrow\{\{1, 2\}\} \sqcap \mathrm{req} \sqcap \neg\mathrm{grant}\big)$.

We consider two encodings for the LVBF of AFA transitions. The first encoding uses LVBDD in shared normal form, while the second uses traditional ROBDD. The LVBDD encoding uses one decision variable per proposition of the formula. Each node of these LVBDD is labeled with a lattice value, here an upward-closed set of configurations of the automaton. In this work, we encode these upward-closed sets with ROBDD; other encodings are possible (e.g., covering sharing trees [9]) but we do not discuss them here as this is orthogonal to our work. For the ROBDD encoding of LVBF of AFA transitions, we use one variable per proposition of the LTL formula and location of the automaton. Both encodings are illustrated at Fig. 3 (b) and (c).

We consider three series of parametric scalable LTL formulas: *mutex* formulas, *lift* formulas and *pattern* formulas. The mutex and lift formulas have been used previously as LTL benchmark formulas in [22], and the pattern formulas were used as benchmark formulas by Vardi *et al.* in [15] and previously in [16]. The presentation (given below) of the mutex and lift formulas has been slightly simplified due to space limitations; all the benchmarks in their complete form are available for download at `http://www.antichains.be/atva2010/`.

Table 2. Running times in seconds. ‘n’ is the parameter of the formula, ‘locs’ the number of locations of the AFA, ‘props’ the number of propositions of the LTL formula and ‘iters’ the number of iterations of the fixpoint. Time out set at 1,000 seconds

	n	locs	props	iters	ROBDD	LVBDD	NuSMV + ord	NuSMV
Mutex	40	328	121	3	12	**2**	12	11
	80	648	241	3	74	**12**	31	34
	120	968	361	3	284	**37**	87	180
	160	1288	481	3	t.o.	**79**	206	325
	200	1608	601	3	-	**132**	t.o.	t.o.
Lift	20	98	42	41	1	**1**	1	1
	40	178	82	81	10	**3**	6	10
	60	258	122	121	36	**8**	24	44
	80	338	162	161	83	**17**	53	162
	100	418	202	201	188	**31**	185	581
	120	498	242	241	330	**51**	341	t.o.
	140	578	282	281	t.o.	**81**	t.o.	-
E	100	103	101	2	12	**0.1**	1	1
	200	203	201	2	110	**0.3**	4	4
	300	303	301	2	392	**0.6**	19	19
U	100	102	101	2	**1**	1	2	2
	200	202	201	2	**7**	12	23	19
	300	302	301	2	**39**	44	117	116
R	6	27	8	2	0.2	0.4	**0.1**	**0.1**
	8	35	10	2	20	21	**0.1**	**0.1**
	10	43	12	2	t.o.	t.o.	**0.1**	**0.1**
Q	10	23	12	2	1	1	**0.1**	**0.1**
	15	33	17	2	205	234	**0.1**	**0.1**
	20	43	22	2	t.o.	t.o.	**0.2**	**0.1**
S	200	203	201	2	**0.1**	**0.1**	3	4
	300	303	301	2	**0.1**	**0.3**	6	11
	400	403	401	2	**0.1**	**0.4**	16	25

For each set of formulas, we supply an *initial ordering* of the propositions. Providing a sensible initial ordering is critical to the fair evaluation of BDD-like structures, as these are known to be very sensitive to variable ordering.

The mutex formulas describe the behavior of n concurrent processes involved in a mutual exclusion protocol. The proposition c_i indicates that process i is in its critical section, r_i that it would like to enter the critical section, and d_i that it has completed its execution. The initial ordering on the propositions is $r_1, c_1, d_1, \ldots, r_n, c_n, d_n$. We check that $\mu(n) \wedge \neg(\Diamond r_1 \rightarrow \Diamond d_1)$ is unsatisfiable.

$$\mu(n) \equiv \bigwedge_{i=1}^{n} \big(\Box(c_i \rightarrow \bigwedge_{j\neq i} \neg c_j) \wedge \Box(r_i \rightarrow \Diamond c_i) \wedge \Box(c_i \rightarrow \Diamond\neg c_i) \wedge \neg d_i \wedge \Box((c_i \wedge X\neg c_i) \rightarrow d_i)\big)$$

Table 3. Comparison of ROBDD and LVBDD sizes on the lift example

				LVBDD		ROBDD					
						no reord.		SIFT		WIN2	
n	locs	props	iters	avg.	max.	avg.	max.	avg.	max.	avg.	max.
10	58	22	21	58	284	2,374	8,190	2,374	8,190	2,374	8,190
12	66	26	25	65	300	9,573	32,766	682	32,766	695	32,766
14	74	30	29	75	306	t.o.	t.o.	269	33,812	1,676	131,070
50	218	102	101	201	654	t.o.	t.o.	270	1,925	t.o.	t.o.
100	418	202	201	376	1,304	t.o.	t.o.	469	1,811	t.o.	t.o.

The lift formula describes the behavior of a lift system with n floors. The proposition b_i indicates that the button is lit at floor i, and f_i that the lift is currently at floor i. The initial variable ordering is $b_1, f_1, \ldots, b_n, f_n$. We check that $\lambda(n) \wedge \neg(b_{n-1} \rightarrow (\neg f_{n-1}\, U f_n))$ is unsatisfiable.

$$\lambda(n) \equiv f_1 \wedge \bigwedge_{i=1}^{n} \Big(\Box(b_i \rightarrow (b_i\, U f_i)) \wedge \Box\big(f_i \rightarrow (\neg b_i \wedge \bigwedge_{j \neq i}^{n} \neg f_j \wedge \neg X f_j) \wedge (\bigvee_{i-1 \leq j \leq i+1} XX f_j))\Big)$$

The pattern formulas of [15] are found below, and their initial proposition ordering is set to $p_1, \ldots, p_n$.

$$E(n) = \textstyle\bigwedge_{i=1}^{n} \Diamond p_i \qquad Q(n) = \bigwedge_{i=1}^{n-1} (\Diamond p_i \vee \Box p_{i+1})$$
$$U(n) = (\ldots (p_i\, U p_2)\, U \ldots)\, U p_n \qquad S(n) = \textstyle\bigwedge_{i=1}^{n} \Box p_i$$
$$R(n) = \textstyle\bigwedge_{i=1}^{n} (\Box\Diamond p_i \vee \Diamond\Box p_{i+1})$$

In order to evaluate the practical performances of LVBDD, we have implemented two nearly identical C++ prototypes, which implement a simple antichain-based forward fixpoint computation [22, 23] to solve the satisfiability problem for finite word LTL. These two prototypes differ only in the encoding of the LVBF of the AFA transitions: one uses LVBDD while the other uses the BuDDy [18] implementation of ROBDD with the SIFT reordering method enabled. On the other hand, a recent survey by Vardi and Rozier [15] identifies NuSMV [8] as one of the best tools available to solve this problem[1]. NuSMV implements many optimization such as *conjunctive-clustering* of the transition relation and *dynamic reordering* of the BDD variables, so we believe that NuSMV provides an excellent point of comparison for our purpose. The use of NuSMV for finite-word LTL satisfiabilty is straightforward: we translate the formula into an AFA, which is then encoded into an SMV module with one input variable (`IVAR`) per proposition and one state variable (`VAR`) per location of the automaton. We then ask NuSMV to verify the property "`CTLSPEC AG !accepting`" where `accepting` is a formula which denotes the set of accepting configurations of the automaton. We invoke NuSMV with the "`-AG`" and "`-dynamic`" command-line options which respectively enable a single forward reachability computation and dynamic reordering of BDD variables. We now present two sets of experiments which illustrate the practical efficiency of LVBDD in terms of *running time* and *compactness*.

[1] Vardi et al. considered *infinite-word* LTL, but this applies just as well to finite-word LTL.

Running time comparison. In our first set of experiments, we have compared the respective running times of our prototypes and NuSMV on the benchmarks described above. These results are reported in Table 2, where we highlight the best running times in bold.

It is well-known that ordered decision diagrams in general are very sensitive to the ordering of the variables. In practice, ROBDD packages implement *dynamic variable reordering techniques* to automatically avoid bad variable orderings. However, these techniques are known to be sensitive to the *initial variable ordering*, so a sensible initial ordering is a necessary component to the fair evaluation of ordered decision diagrams. In our experiments, we have two sets of variables which respectively encode the LTL propositions and the AFA locations. We provide an initial sensible ordering for both sets of variables; the LTL propositions are initially ordered as described previously, and the AFA locations are ordered by following a topological sort[2]. Finally, for the ROBDD-based tools, we provide an initial ordering such that the LTL propositions variables *precede* the AFA location variables. In Table 2, the "NuSMV + ord" and "NuSMV" columns respectively contain the running times of NuSMV when provided with our initial ordering, or without any initial ordering.

On most of the examples, the LVBDD-based prototype performs better than NuSMV and the ROBDD prototype. For the *mutex* and *lift* benchmarks, LVBDD seem to scale much better than ROBDD. We have investigated the scalability of ROBDD on these instances with profiling tools, which revealed that a huge proportion of the run time is spent on variable reordering. Disabling dynamic reordering for either the ROBDD-based prototype or NuSMV on these instances made matters even worse, with neither NuSMV nor our ROBDD-based prototype being able to solve them for parameter values beyond 30. These observations shed light on one of the key strengths of LVBDD in the context of LVBF representation. While ROBDD-based encodings must find a *suitable interleaving* of the domain and co-domain variables, which can be very costly, LVBDD avoid this issue altogether, even when co-domain values are encoded using ROBDD.

Finally, the results for the pattern formulas confirm earlier research [23] by showing that the antichain approach (i.e., columns ROBDD, LVBDD) and the fully-symbolic approach (NuSMV in our case) exhibit performance behaviors that are incomparable in general; in the Q benchmark, the antichains grow exponentially in length, while the S benchmark makes the ROBDD reordering-time grow exponentially.

Compactness comparison. In this set of experiments, we compare the compactness of LVBDD and ROBDD when encoding LVBF occurring along the computation of the fixed point that solves the satisfiability for the *lift* formulas. These experiments are reported in Table 3. We report on the largest and average structure sizes encountered along the fixed point. We performed the experiments for ROBDD both with and without dynamic reordering enabled, and for two different reordering techniques provided in the BuDDy package: SIFT and WIN2. The sizes reported for LVBDD is equal to the number of decision nodes of the LVBDD *plus* the number of unique ROBDD nodes that are used to encode the lattice values labelling the LVBDD. This metric is thus an *accurate* representation of the total memory footprint of LVBDD and is *fair* for the comparison with ROBDD. These experiments show that, as expected, LVBDD are more compact than ROBDD in the context of LVBF representation, although ROBDD can achieve sizes that are comparable with LVBDD, but at the price of a potentially very large reordering overhead. This increased compactness explains the better running times of the LVBDD prototype reported in Table 2.

[2] This ordering is sensible because the translation from LTL produces AFA that are *very weak*.

All experiments were performed with a timeout of 1000 seconds on an Intel Core i7 3.2 Ghz CPU with 12 GB of RAM. A preliminary version of our C++ LVBDD library is freely available at `http://www.ulb.ac.be/di/ssd/nmaquet/#research`.

References

1. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: DAC 1993. ACM, New York (1993)
2. Reif Andersen, H., Hulgaard, H.: Boolean Expression Diagrams. In: LICS. IEEE, Los Alamitos (1997)
3. Devereux, B., Chechik, M.: Edge-Shifted Decision Diagrams for Multiple-Valued Logic. In: JMVLSC. Old City Publishing (2003)
4. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977. ACM, New York (1977)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, J.: Symbolic Model Checking: 10^{20} States and Beyond. In: LICS 1990. IEEE, Los Alamitos (1990)
7. Chechik, M., Devereux, B., Easterbrook, S., Lai, A., Petrovykh, V.: Efficient Multiple-Valued Model-Checking Using Lattice Representations. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 441. Springer, Heidelberg (2001)
8. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
9. Delzanno, G., Raskin, J.-.F., Van Begin, L.: Covering sharing trees: a compact data structure for parameterized verification. In: STTT, vol. 5(2-3). Springer, Heidelberg (2003)
10. Bryant, R.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Trans. on Comp. C-35(8) (1986)
11. Kupferman, O., Lustig, Y.: Lattice Automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
12. Fujita, M., McGeer, P.C., Yang, J.C.Y.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Form. Methods Syst. Des. 10(2-3) (1997)
13. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
14. Doyen, L., Raskin, J.F.: Antichain Algorithms for Finite Automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010)
15. Rozier, K., Vardi, M.: LTL Satisfiability Checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
16. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 53–70. Springer, Heidelberg (2006)
17. Birkhoff, G.: Lattice Theory. Colloquim Publications. Am. Math. Soc., Providence (1999)
18. Lind-Nielsen, J.: Buddy: BDD package, `http://www.itu.dk/research/buddy`
19. NuSMV Model-checker, `http://nusmv.irst.itc.it/`
20. SMV Model-checker, `http://www.cs.cmu.edu/~modelcheck/smv.html`
21. Somenzi, F.: BDD package CUDD, `http://vlsi.colorado.edu/~fabio/CUDD/`
22. Ganty, P., Maquet, N., Raskin, J.F.: Fixpoint Guided Abstraction Refinements for Alternating Automata. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 155–164. Springer, Heidelberg (2009)
23. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Antichains: Alternative Algorithms for LTL Satisfiability. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)

A Specification Logic for Exceptions and Beyond

Cristian Gherghina and Cristina David

Department of Computer Science, National University of Singapore

Abstract. Exception handling is an important language feature for building more robust software programs. It is primarily concerned with capturing abnormal events, with the help of catch handlers for supporting recovery actions. In this paper, we advocate for a specification logic that can uniformly handle exceptions, program errors and other kinds of control flows. Our logic treats exceptions as possible outcomes that could be later remedied, while errors are conditions that should be avoided by user programs. This distinction is supported through a uniform mechanism that captures static control flows (such as normal execution) and dynamic control flows (such as exceptions) within a single formalism. Following Stroustrup's definition [15,9], our verification technique could ensure exception safety in terms of four guarantees of increasing quality, namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee.

1 Introduction

Exception handling is considered to be an important but yet controversial feature for many modern programming languages. It is important since software robustness is highly dependent on the presence of good exception handling codes. Exception failures can account for up to 2/3 of system crashes and 50% of system security vulnerabilities [10]. It is controversial since its dynamic semantics is often considered too complex to follow by both programmers and software tools.

Goodenough provides in [5] a possible classification of exceptions according to their usage. More specifically, they can be used:

– to permit dealing with an operation's failure as either domain or range failure. Domain failure occurs when an operation finds that some input assertion is not satisfied, while range failure occurs when the operation finds that its output assertion cannot be satisfied.
– to monitor an operation, e.g. to measure computational progress or to provide additional information and guidance should certain conditions arise.

In the context of Spec#, the authors of [7] use the terms client failures and provider failures for the domain and range failures, respectively. Client failures correspond to parameter validation, whereas provider failures occur when a procedure cannot perform the task it is supposed to. Moreover, provider failures are divided into admissible failures and observed program errors. To support admissible failures, Spec# provides checked exceptions and throws sets. In contrast, an observed program error occurs if the failure is due to an intrinsic error in the program (for e.g. an array bounds error) or a global failure that is not tied to a particular procedure (for e.g. an out-of-memory error).

Such failures are signaled by Spec# with the use of unchecked exceptions, which do not need to be listed in the procedure's throws set. Likewise for Java, its type system has been used to track a class of admissible failures through checked exceptions.

However, omitting the tracking of unchecked exceptions is a serious shortcoming, since it provides a backdoor for some programmers to deal exclusively with unchecked exceptions. This shortcut allows programmers to write code without specifying or catching any exceptions. Although it may seem convenient to the programmer, it sidesteps the intent of the exception handling mechanisms and makes it more difficult for others to use the code. A fundamental contradiction is that, while unchecked exceptions are regarded as program errors that are not meant to be caught by handlers, the runtime system continues to support the handling of both checked and unchecked exceptions.

The aim of the current work is ensure a higher level of exception safety, as defined by Stroustrup [15] and extended by Li and co-authors [9], which takes into consideration both checked and unchecked exceptions. According to Stroustrup, an operation on an object is said to be exception safe if it leaves the object in a valid state when it is terminated by throwing an exception. Based on this definition, exception safety can be classified into four guarantees of increasing quality:

– **No-leak guarantee:** For achieving this level of exception safety, an operation that throws an exception must leave its operands in well-defined states, and must ensure that every resource that has been acquired is released. For example, all memory allocated must be either deallocated or owned by some object whenever an exception is thrown.
– **Basic guarantee:** In addition to the no-leak guarantee, the basic invariants of classes are maintained, regardless of the presence of exceptions.
– **Strong guarantee:** In addition to providing the basic guarantee, the operation either succeeds, or has no effects when an exception occurs.
– **No-throw guarantee:** In addition to providing the basic guarantee, the operation is guaranteed not to throw an exception.

We propose a methodology for program verification that can guarantee higher levels of exception safety. Our approach can deal with all kinds of control flows, including both checked and unchecked exceptions, thus avoiding the unsafe practice of using the latter to circumvent exception handling. Another aspect worth mentioning is that some of the aforementioned exception safety guarantees might be expensive. For instance, strong guarantee might incur the high cost of roll-back operations as it pushes all the recovery mechanism into the callee. However, such recovery may also be performed by the caller, or there might be cases when it is not even required. Consequently, in Section 4 we improve on the definition of strong guarantee for exception safety.

Moreover, verifying a strong guarantee is generally more expensive than the verification of a weaker guarantee, as it is expected to generate more complex proof obligations (details can be found in Section 7). Hence, according to the user's intention, our system can be tuned to enforce different levels of exception safety guarantees.

1.1 Our Contributions

The main contributions of our paper are highlighted below:

- We introduce a specification logic that captures the states for both normal and exceptional executions. Our design is guided by a novel unification of both static control flows (such as break and return), and dynamic control flows (such as exceptions and errors).
- We ensure exception safety in terms of the guarantees introduced in [15], and extended in [9]. Additionally, we improve the strong guarantee for exception safety. To support a tradeoff between precision and cost of verification, our verification system is flexible in enforcing different levels of exception safety.
- We have implemented a prototype verification system with the above features and validated it with a suite of exception-handling examples.

2 Related Works

Exceptions are undoubtedly important parts of programming language systems that should be adequately handled during program verification. Consequently, in recent years, there have been several research works that tackle the problem of exception handling verification and analysis.

A traditional approach is based on type systems. However, it is tedious and possibly imprecise to simply declare the set of exceptions that may escape from each method. A better scheme is for the inference of uncaught exceptions. For example, [6] employs two analyses of different granularity, at the expression and method levels, to collect constraints over the set of exception types handled by each Java program. By solving the constraint system using techniques from [4], they can obtain a fairly precise set of uncaught exceptions. For exceptions that are being ignored by type system, such as unchecked exceptions for Java and C#, there is a total loss of static information for them. Moreover, conditions leading to exceptions are often quite subtle and cannot be conveniently tracked in a type system.

To overcome the above shortcomings, this paper proposes a more expressive specification logic, which complements the type system through selective tracking on types that are linked to control flows. A more limited idea towards verification of exception handling programs is based on model checking [9]. A good thing is that it does not require any specification for the verification of exception reliability (on the absence of uncaught exceptions and redundant handlers), but annotations are required for the verification of the no-leak guarantee. However, this approach does not presently handle higher levels of exception safety, beyond the no-leak guarantee.

A recent approach towards exception handling in a higher-order setting, is taken in [2]. Exceptions are represented as sums and exception handlers are assigned polymorphic, extensible row types. Furthermore, following a translation to an internal language, each exception handler is viewed as an alternative continuation whose domain is the sum of all exceptions that could arise at a given program point. Though CPS can be used to uniformly handle the control flows, it increases the complexity of the verification process as continuations are first class values.

Spec# also has a specification mechanism for exceptions. While its verification system is meant to analyse C# programs, exceptional specifications are currently useable for only the runtime checking module. The current Spec# prototype for static verification would unsoundly approximate each exception as false, denoting an unreachable program state [8].

Another impressive verification system, based on the Java language, is known as the KeY approach [1]. This system makes use of a modal logic, known as dynamic logic. Like Hoare logic, dynamic logic supports the computation of strongest postcondition for program codes. However, the mechanism in which this is done is quite different since program fragments may be embedded within the dynamic logic itself. This approach is more complex since rewriting rules would have to be specified for each programming construct that is allowed in the dynamic logic. For example, to support exception handling, a set of rewriting rules (that are meaning-preserving) would have to be formulated to deal with raise, break and try-catch-finally constructs, in addition to rewriting rules for block and conditionals and their possible combinations. The KeY approach is meant to be a semi-automated verification system that occasionally prompts the user for choice of rewriting rules to apply. In contrast, our approach is meant to be fully-automated verification system, once pre/post specifications have been designed.

3 Source and Specification Languages

As input language for our system, we consider a Java-like language which we call `SrcLang`. Although we make use of the class hierarchy to define a subtyping relation for exception objects, the treatment of the other object-oriented features, such as instance methods and method overriding, is outside the scope of the current paper. Our language permits only static methods and single inheritance. We have also opted for a monomorphically typed language. These simplifications are orthogonal to our goal of providing a logic for specifying exceptions. We present the full syntax for the input source language in Fig 1. Take note that $\overrightarrow{e}$ denotes $e_1, \ldots, e_n$.

$$\begin{array}{lll}
P & ::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{M} & \textit{program} \\
V & ::= \texttt{pred root::}\mathit{pname}\langle\overrightarrow{v}\rangle \equiv \Phi\ \texttt{inv}\ \pi & \textit{pred declaration} \\
D & ::= \mathit{class}\ c_1\ \mathit{extends}\ c_2\{\overrightarrow{t\ f}\} & \textit{data declaration} \\
t & ::= c \mid p & \textit{user or prim. type} \\
M & ::= t\ m\ (\overrightarrow{[ref]\ t\ v})\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}\ \{e\} & \textit{method declaration} \\
w & ::= v \mid v.f & \textit{variable or field} \\
e & ::= v \mid k \mid \texttt{new}\ c \mid v.f \mid m\ (\overrightarrow{v}) \mid \{t\ v;\ e\} \mid w{:=}e \mid e_1;\ e_2 & \\
 & \quad \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \\
 & \quad \mid (t)\ e & \textit{casting} \\
 & \quad \mid \texttt{raise}\ e & \textit{throw exception} \\
 & \quad \mid \texttt{break}\ [L] \mid \texttt{return}\ e & \textit{break and return} \\
 & \quad \mid \texttt{continue}\ [L] & \textit{loop continue} \\
 & \quad \mid L : e & \textit{labelled expression} \\
 & \quad \mid e_1\ \texttt{finally}\ e_2 & \textit{finally} \\
 & \quad \mid \texttt{try}\ e\ \texttt{catch}\ (c_1\ v_1)\ e_1[\texttt{catch}\ (c_i\ v_i)\ e_i]_{\mathtt{i=2}}^{\mathtt{n}} & \textit{multiple catch handlers} \\
 & \quad \mid \texttt{do}\ e_1\ \texttt{while}\ e_2\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po} & \textit{loop}
\end{array}$$

Fig. 1. Source Language : `SrcLang`

Δ ::= $\Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \mid \Delta \wedge \beta$	*composite formulae*
Φ ::= $\bigvee (\exists v^* \cdot \kappa \wedge \pi)$	*formulae*
π ::= $\gamma \wedge \phi \wedge \tau$	*pure constraints*
γ ::= $v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1 \wedge \gamma_2$	*pointer constraints*
κ ::= $\texttt{emp} \mid v{::}a\langle \overrightarrow{v} \rangle \mid \kappa_1 * \kappa_2$	*heap constraints*
ft ::= $c \mid predef\_flow$	*flow types*
β ::= $fv{=}ft \mid fv_1{=}fv_2$	*flow var constraints*
τ ::= $\texttt{flow} = fset \quad fset ::= \texttt{Ex}(ft) \mid ft - \{ft_1, .., ft_n\}$	*current flow*
ϕ ::= $\texttt{true} \mid \texttt{false} \mid b_1{=}b_2 \mid b_1{\leq}b_2 \mid c{<}v \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$	*Presburger constraints*
b ::= $k \mid v \mid k{\times}b \mid b_1{+}b_2 \mid -b \mid max(b_1,b_2) \mid min(b_1,b_2)$	

Fig. 2. Specification Language

Our language allows for functions (and loops) to be decorated with pre and post conditions that are verified by our tool. Unlike SPEC# or ESC/Java, where specifications for exceptions are captured by a special syntax for exceptional postconditions, we aim for a unified logic that is capable of capturing all kinds of control flows. Our specification, as described in Fig 2, is based on separation logic formulas, introduced by John Reynolds [13], given in disjunctive normal form, but has been enhanced with a control flow annotation. Within a formula, each disjunct consists of a subformula κ referred to as heap part and π, a pure part that represents a heap-independent part of the formula.

The heap part describes the heap footprint which is composed of $*$-separated heap nodes. These nodes, written as $v{::}a\langle \overrightarrow{v} \rangle$, are instances of either a user-defined class ($v{::}c\langle \overrightarrow{v} \rangle$) or a user-defined predicate ($v{::}pname\langle \overrightarrow{v} \rangle$), as an abstraction for a data structure. As shown in Fig.1, each declaration of a user-defined predicate consists of a root pointer, a predicate body and a pure formula that is an invariant of all its instances. The pure part does not capture any heap-based data structures as it contains pointer equalities/inequalities (γ), linear arithmetic (ϕ) and a special subformula τ for modelling the control flow.

A special variable `flow` is used to denote the control flow associated with the respective program state, captured as a disjunct. The possible values of control flow are either $Ex(ft)$ to denote an exact control flow type (not including its subclasses), or $ft - \{ft_1, .., ft_n\}$ to denote a control flow from ft but not from subclasses $ft_1, .., ft_n$. The control flow ft is organised as a subtyping tree hierarchy. With this hierarchy, we can be as precise as required for verifying different exception safety guarantees.

The flow type hierarchy incorporates all the possible control flow types ft, both the ones pertaining to user-defined exceptions and the predefined flow types, $predef\_flow$. This will be further elaborated in Sec.5.2.

Δ denotes a composite formula that is enhanced with an extra pure component β to capture the bindings for flow variables, fv, from the catch handlers. Lastly, each variable in our specification logic may be expressed in either primed form (e.g. v') or unprimed form (e.g. v). The former denotes the latest value of the corresponding variable, while the latter denotes the original value of the same variable. When used in the postcondition, they denote state changes that occur for parameters that are being passed by reference.

4 Examples with Higher Exception Safety Guarantees

We next illustrate our new specification mechanism through a few examples. Let us first consider the following class and predicate definitions.

```
class node { int val; node next}

pred root::ll⟨n⟩ ≡ root=null ∧ n=0 ∨
       ∃r·root::node⟨_, r⟩ ∗ r::ll⟨n−1⟩ inv n≥0;
```

Predicate ll defines a linear-linked list of length n. As elaborated earlier, each predicate describes a data structure, which is a collection of objects reachable from a base pointer denoted by root in the predicate definition. The expression after the **inv** keyword captures a pure formula that always holds for the given predicate.

Let us now consider the method list_alloc allocating memory for a linear-linked list to be pointed by x. The precondition requires x to be null, while the postcondition asserts that either no error was raised, i.e. the flow is norm, and the updated x points to a list with n elements, or an out of memory exception was raised, i.e. the flow is out_of_mem_exc, and x remains null.

Method list_alloc calls an auxiliary method list_alloc_helper which recursively allocates nodes in the list. If an out of memory exception is raised, the latter performs a rollback operation, inside a try-catch block, during which it frees all the memory that was acquired up to that point. Take note that, in the following examples, for illustration purposes, we provide an alternative set of library methods with explicit memory deallocation (via method list_dealloc) that coexists with our Java like language.

```
void list_alloc(int n, ref node x)
requires x=null ∧ n≥0
ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
{list_alloc_helper(n, 0, x); }

void list_alloc_helper(int n, int i, ref node x)
requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
{if(n>i){
   try{x = new node(0, x); }
   catch(out_of_mem_exc exc){
       list_dealloc(i, x);
       raise (new out_of_mem_exc()); }
   list_alloc_helper(n, i+1, x); }
}
```

According to [15], method list_alloc is said to ensure a strong guarantee on exception safety, as it either succeeds in allocating all the required memory cells, x′::ll⟨n⟩ ∧ flow=norm, or it has no effect, x′=null ∧ flow=out_of_mem_exc. However, this rollback operation can be expensive if we have been building a long list. Moreover, there can be cases when such rollbacks are not needed. For instance, in the context of the aforementioned method, a caller might actually accept a smaller amount of the allocated memory.

We propose to improve Stroustrup's definition on strong guarantee as follows: An operation is considered to provide a strong guarantee for exception safety if, in addition to providing basic guarantees, it either succeeds, or its effect is precisely known to the caller. Given this new definition, the method list_alloc_helper_prime defined below is also said to satisfy the strong guarantee. Compared to method list_alloc_helper, the newly revised method has an extra pass-by-reference parameter, no_cells, to denote the number of memory cells that were already allocated. Through this output parameter, the caller is duly informed on the length of list that was actually allocated. This method now two possible outcomes :

- it succeeds and all the required memory cells were allocated, $x'{::}ll\langle n\rangle \wedge no\_cells'=n$;
- an out of memory exception is thrown and no_cells captures the number of successfully allocated memory cells. The caller is duly informed on the amount of acquired memory through the no_cells parameter, and could either use it, run a recovery code to deallocate it, or mention its size and location to its own caller.

```
void list_alloc_helper_prime(int n, int i, ref node x, ref int no_cells)
requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
ensures (x′::ll⟨n⟩ ∧ no_cells′=n ∧ flow=norm) ∨
        (x′::ll⟨no_cells′⟩ ∧ flow=out_of_mem_exc);
{if(n>i){
   try{x = new node(0,x); }
   catch(out_of_mem_exc exc){
      no_cells=i;
      raise (new out_of_mem_exc()); }
   list_alloc_helper(n, i+1, x); }
else no_cells=n; }
```

Next, we will illustrate how to make use of our verification mechanism for enforcing the no-throw guarantee from [15]. For this purpose, let us consider the following swap method which exchanges the data fields stored in two disjoint nodes.

```
void swap(node x, node y)
requires x::node⟨v1, q1⟩*y::node⟨v2, q2⟩
ensures x::node⟨v2, q1⟩*y::node⟨v1, q2⟩ ∧ flow=norm;
{  int tmp = x.val;
   x.val = y.val;
   y.val = tmp; }
```

The precondition requires the nodes pointed by x and y, respectively, to be disjoint, while the postcondition captures the fact that the values stored inside the two nodes are swapped. Additionally, the postcondition asserts that the only possible flow at the end of the method is the normal flow, flow=norm, i.e. no exception was raised. This specification meets the definition of no-throw guarantee given in [15]. Any presence of exception, including an exception caused by null pointer dereferencing, would require the postcondition to explicitly capture each such exceptional flow. Conversely, any absence of exceptional flow in our logic is an affirmation for the no-throw guarantee.

5 Verification for Unified Control Flows

In this section, we present the rules for verifying programs with support for a wide range of control flows. To keep the rules simple, we shall use a core imperative language that was originally presented in [3]. First, we briefly present the key features of our core language, before proceeding to show how our verification rules handle control flows that are organised in a tree hierarchy.

5.1 Core Language

For a easier specification of our verification rules, we translate the source language to a small core language [3]. This core language is detailed in Fig 3. The novelty is represented by two unified constructs for handling control flows. The first one is the output construct `fn#v` which can be used to provide an output with a normal flow via `norm#v`, or a thrown exception via `ty(v)#v`. The type of a raised exception object `v` is captured as its control flow.

The second special construct has the form `try` $\mathtt{e}_1$ `catch` ((`ft@fv`) `v`) $\mathtt{e}_2$ intended to evaluate expression $\mathtt{e}_1$ that could be caught by handler $\mathtt{e}_2$ if the detected control flow is a subtype of `ft`. This construct uses two bound variables, namely `fv` to capture the control flow and `v` to capture the output value. It is more general than the try-catch construct used in Java, since it can capture both normal flow and abnormal control flows,

P	$::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{M}$	*program*
D	$::= \mathtt{class}\ c_1\ \mathtt{extends}\ c_2\ \{\overrightarrow{t\ v}\}$	*data declaration*
V	$::= \mathtt{pred}\ \mathtt{root}{::}pname\langle\overrightarrow{v}\rangle \equiv \Phi\ \mathtt{inv}\ \pi$	*pred declaration*
ft	$::= c \mid predef\_flows$	*flow types*
M	$::= t\ m(\overrightarrow{[\mathtt{ref}]\ t\ v})\ \mathtt{requires}\ \Phi_{pr}\ \mathtt{ensures}\ \Phi_{po}\ \{e\}$	*method decl*
e	$::= fn\#x$	*output (flow&value)*
	$\mid v.f$	*field access*
	$\mid w{:=}v$	*assignment*
	$\mid m(\overrightarrow{v})$	*method call*
	$\mid \{t\ v;\ e\}$	*local var block*
	$\mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$	*conditional*
	$\mid \mathtt{try}\ e_1\ \mathtt{catch}\ (ft[@v_1]\ v_2)\ e_2$	*catch handler*
fn	$::= \mathtt{Ex}(ft) \mid fv \mid \mathtt{ty}(v) \mid v.1$	*flow*
t	$::= c \mid p$	*user or prim. type*
x	$::= v \mid k \mid \mathtt{new}\ c \mid (fv, v) \mid v.2$	*basic value*
w	$::= v \mid v.f$	*var/field*

Fig. 3. Core Language : `Core−U`

including `break`, `continue` and procedural `return`. As a simplification, the usual sequential composition $\mathtt{e}_1;\mathtt{e}_2$ is now a syntactic sugar for `try` e_1 `catch` ((`norm`@_)_) e_2, whose captured value is ignored by use of an anonymous variable .

We extend the core language to embed control flows directly as values, by allowing a pair of control flow and its value (fv, v) to be specified. With this notation, we can save each exception and its output value as an embedded pair that could be later re-thrown. Operations `v.1` and `v.2` are used to access the control flow and the value, respectively, from an embedded pair in `v`.

5.2 Control Flow Hierarchy

To support the generalized try-catch construct we provide a unified view on all control flows through the use of a tree hierarchy supporting a subtyping relation $<:$. All control flow types are subtypes of $\top$. Control flows that can be caught by catch handlers are subtypes of `c−flow`, while `abort` denotes control flows that can never be caught. The latter category includes program error, program termination and non-termination. Control flows corresponding to exceptions (both checked and unchecked) are placed in the subtree hierarchy under the `exc` class. Regarding the static (or local) control-flows, they are grouped under `local` which includes `norm` to denote normal flow, `ret` to signal a method return (covering also methods with multi-return options [14]), `brk` to denote the break out of a loop, and `cont` to denote a jump to the beginning of a loop. The use of a tree hierarchy facilitates formal reasoning, since the disjointedness property between any two flow types can be statically determined without ambiguity.

As opposed to other systems which enforce the restriction that the try-catch construct applies only to exceptional flows, our unified view on control flows will generalize the try-catch construct across the entire domain of control flow types. This domain extension permits a much more streamlined verification mechanism.

A graphical representation of the entire flow hierarchy is given in Fig.4. Each arrow $c_2 \rightarrow c_1$ denotes a subtyping relation $c1 <: c2$.

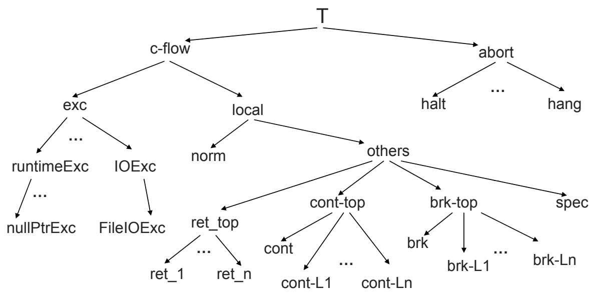


Fig. 4. A Subtype Hierarchy on Control Flows

5.3 Verification Rules

Our verification system requires pre/post conditions to be declared for each method and each loop in the input program. Loops are then transformed to tail-recursive methods where the parameters are passed by reference. With these specifications being given, we can apply modular verification to each method's body using Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$. These are forward verification rules that expect Δ_1 to be given before computing Δ_2. Furthermore, Δ_1 is a captured program state whereby $\texttt{flow} = \top$, while Δ_2 is a disjunctive heap state capturing the entire set of control flows that may escape during the execution of e. For example, we may have $\texttt{x}'{=}\texttt{x}{+}1 \wedge \texttt{flow}{=}\texttt{norm} \vee \texttt{x}'{=}\texttt{x} \wedge \texttt{flow}{=}\texttt{exc}$ to denote a state change for variable x for normal control flow, while an exception leaves the state of x unchanged.

In the remainder of the current section we will focus mainly on the verification of output (with flow&value) and try-catch constructs in Figure 5, since the other rules are largely conventional.

The output construct sets a control flow with a given value. To achieve this, we require each control flow variable to be resolved to an appropriate flow set ($fset$) value,

$$\text{[FV-SPLIT]}$$
$$split(\Delta, c, fv, v) = ((\exists \texttt{flow}\,.[res{\mapsto}v]\Delta \wedge \texttt{flow} <: c \wedge \texttt{flow} = fv), \Delta \wedge \neg(\texttt{flow} <: c))$$

$$\text{[FV-RESOLVE-FV]}\quad \frac{(fv = fset) \in \Delta}{resolve(\Delta, fv) = fset} \qquad \text{[FV-RESOLVE-PAIR-FST]}\quad \frac{\begin{array}{c}(v{=}(fv, \_)) \in \Delta \\ resolve(\Delta, fv) = fset\end{array}}{resolve(\Delta, v.1) = fset} \qquad \text{[FV-RESOLVE-PAIR-SND]}\quad \frac{(v{=}(\_, w)) \in \Delta}{resolve(\Delta, v.2) = w}$$

$$\text{[FV-OUTPUT-NEW]}\quad \frac{\begin{array}{c}resolve(\Delta, ft) = fset \\ \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res}{::}\texttt{c}\langle \ldots \rangle\end{array}}{\vdash \{\Delta\}\, ft\#new\ c\, \{\Delta_1\}} \qquad \text{[FV-RESOLVE-TYPE]}\quad \frac{(type(v) = c) \in \Delta}{resolve(\Delta, \texttt{ty}(v)) = c}$$

$$\text{[FV-OUTPUT-PAIR]}\quad \frac{\begin{array}{c}resolve(\Delta, ft) = fset \\ \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = (fv, v)\end{array}}{\vdash \{\Delta\}\, ft\#(fv, v)\, \{\Delta_1\}} \qquad \text{[FV-RESOLVE-CONST]}\quad \frac{}{resolve(\Delta, \texttt{Ex}(c)) = \texttt{Ex}(c)}$$

$$\text{[FV-TRY-CATCH]}$$
$$\frac{\vdash \{\Delta\}\, e_1\, \{\Delta_1\} \quad (\Delta_2, \Delta_3) = split(\Delta_1, c, fv, v) \quad \vdash \{\Delta_2\}\, e_2\, \{\Delta_4\}}{\vdash \{\Delta\}\, \texttt{try}\ e_1\ \texttt{catch}\ (c@fv\ v)\ e_2\, \{\Delta_3 \vee \exists v, fv \cdot \Delta_4\}}$$

$$\text{[FV-OUTPUT-CONST]}\quad \frac{\begin{array}{c}resolve(\Delta, ft) = fset \\ \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = k\end{array}}{\vdash \{\Delta\}\, ft\#k\, \{\Delta_1\}} \qquad \text{[FV-OUTPUT-VAR]}\quad \frac{\begin{array}{c}resolve(\Delta, ft) = fset \\ \Delta_1{=}\exists \texttt{res}.(\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = v'\end{array}}{\vdash \{\Delta\}\, ft\#v\, \{\Delta_1\}}$$

$$\text{[FV-CALL]}$$
$$\frac{\begin{array}{c}t\ mn(\overrightarrow{(t\ v)})\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}\ \{..\} \in P \\ \rho{=}[u_i'/v_i] \quad \Delta \vdash \rho\Phi_{pr} * \Delta_1 \quad \Delta_2{=}(\Delta_1 * \rho\Phi_{po})\end{array}}{\vdash \{\Delta\}\, mn(\overrightarrow{u})\, \{\Delta_2\}}$$

Fig. 5. Some Verification Rules

before we can set it as the current control flow. We rely on an auxiliary set of rules, called FV−RESOLVE, to obtain the corresponding control flow set from a given program state.

The verification system makes use of the `res` variable in order to store the result of the current operation. Note that for an `output(flow&value)` # construct, the result is the value, which is bound to the `res` variable as seen in the OUTPUT rules.

Regarding the FV−TRY−CATCH rule, we first compute the post-state of expression $\mathtt{e}_1$ as Δ_1. Since Δ_1 may capture a range of control flows, it has to be split into two components, Δ_2 and Δ_3, with the help of the FV−SPLIT rule. The Δ_2 component will model the program states with control flows that can be captured by the catch handler, whereas Δ_3 will model those states with control flows that escape from the catch handler. Moreover, for the case when the control flow is being caught by the handler, the control flow type is bound to fv, and its thrown value is bound to v. These bindings are kept in Δ_2 which is made available as the pre-state for $\mathtt{e}_2$. As these local variables are only valid in the catch handler, we quantify them away in the resulting postcondition.

6 Correctness

In the current section we provide a description of the operational semantics for our calculus. The machine configuration is represented by $\langle e, h, s\rangle$ where e denotes the current program code, h denotes the current heap for mapping addresses to objects, and s denotes the current runtime stack for mapping variables to values. We assume sets *Loc* of locations (positive integer values), *Val* of values (either a constant, a location or a pair of control flow type and value), *Var* of variables (program variables and other meta variables), and *ObjVal* of object values stored in the heap, with $c[f_1{\mapsto}\nu_1, .., f_n{\mapsto}\nu_n]$ denoting an object value of class c where $\nu_1, .., \nu_n$ are current values (from the domain *Val* such that *Loc* $\subset$ *Val*) of the corresponding fields $f_1, .., f_n$. Let $s, h \models \Phi$ denote that stack s and heap h form a model of the constraint Φ, with h, s from the following concrete domains:

$$\begin{array}{lll} h \in \mathit{Heaps} & =_{\mathit{df}} & \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal} \\ s \in \mathit{Stacks} & =_{\mathit{df}} & \mathit{Var} \rightarrow \mathit{Val} \end{array}$$

A complete definition of the model for separation constraints can be found in [11]. For the dynamic semantics to follow through, we have introduced a couple of intermediate constructs. Their syntax is extended from the original expression syntax as shown next, where $l \in \mathit{Loc}$.

$$\begin{array}{ll} e ::= ft\#l & \mathit{flow\ and\ location} \\ \quad\;\; \mid \mathtt{BLK}(\{\overrightarrow{v}\}, e_1) & \mathit{block\ construct} \end{array}$$

For the case of $\mathtt{BLK}(\{\overrightarrow{v}\}, e_1)$, e_1 denotes a residual code of the current block. This new construct is used for handling try-catch constructs, method calls and local blocks. Its main purpose is to provide a lexical scope for local variables that are removed once its expression has been completely evaluated.

6.1 Small-Step Semantics

The small-step dynamic semantics is defined using the transition $\langle e, h, s\rangle \hookrightarrow \langle e_1, h_1, s_1\rangle$, which means that if e is evaluated in stack s, heap h, then e reduces in one step to e_1 and generates the new stack s_1 and new heap h_1. The small-step rules are given in Fig. 6. In the rules for local declaration, method call and try catch, the function $newid()$ returns a fresh identifier, while $[u'/u]$ represents the substitution of u by u'. In order to avoid dynamic binding, every variable whose binding is added to the stack is substituted by a fresh identifier. For instance, in the case of the method call, after performing the renaming of the callee's formal parameters, the mappings for the fresh identifiers are temporarily added to the stack, s. These bindings will be removed after the evaluation of the callee's body.

Regarding those constructs from the input language that may implicitly raise exceptions, such as $v.f$ or `new` c, take note that they are handled by the translation described in [3]. More specifically, our translation rules will make the raised exceptions explicit. For illustration, let us consider the translation of $v.f$:

$$v.f \Rightarrow_T \texttt{if } v{=}\texttt{null then nullPtrExc}\#v \texttt{ else } v.f$$

In the remainder of this section, we will define some notions required for proving the soundness of our verification rules. First, we provide the definition for closed configuration, which ensures that variable access does not get stuck.

Definition 6.1 (Closed Configuration) *A configuration $\langle e, h, s\rangle$ is said to be* closed *if $FV(e) \subseteq dom(s)$ and $addr(s) \cup addr(h) \subseteq dom(h)$.*

Method FV collects the free variables in an expression, while $addr$ returns all addresses from the stack and heap. As their definitions are standard, we omit them from the current work. Next, we define divergent computation for small-step semantics, as follows:

Definition 6.2 (Divergence) *A configuration $\langle e, h, s\rangle$ is said to be* divergent *if its small-step transition $\langle e, h, s\rangle \hookrightarrow^* \langle e', h', s'\rangle$ never terminates with a final expression for e'. We shall represent this divergent computation using $\langle e, h, s\rangle \not\hookrightarrow^*$.*

6.2 Soundness of Verification

The soundness of our verification rules is defined with respect to the small-step dynamic semantics given in Section 6.1. Before stating the soundness theorems, we need to extract the post-state of a heap constraint by:

Definition 6.3 (Poststate) *Given a constraint Δ, $Post(\Delta)$ captures the relation between primed variables of Δ. That is :*

$$\begin{array}{l} Post(\Delta) =_{df} \rho\,(\exists V{\cdot}\Delta), \qquad \textit{where} \\ V = \{v_1, .., v_n\} \textit{ denotes all unprimed program variables in } \Delta \\ \rho = [v_1/v_1', .., v_n/v_n']. \end{array}$$

Theorem 6.1 (Preservation). *Consider a closed configuration $\langle e, h, s\rangle$. If*

$$\vdash \{\Delta\}\ e\ \{\Delta_2\} \quad \textit{and} \quad s, h \models Post(\Delta) \quad \textit{and} \quad \langle e, h, s\rangle \hookrightarrow \langle e_1, h_1, s_1\rangle,$$

then there exists Δ_1, such that $s_1, h_1 \models Post(\Delta_1)$ and $\vdash \{\Delta_1\}\ e_1\ \{\Delta_2\}$.

$$\langle fn\#v,h,s\rangle\hookrightarrow\langle s(fn)\#s(v),h,s\rangle \qquad \langle fn\#(fv,v),h,s\rangle\hookrightarrow\langle s(fn)\#(s(fv),s(v)),h,s\rangle$$

$$\langle fn\#k,h,s\rangle\hookrightarrow\langle s(fn)\#k,h,s\rangle \qquad \langle v_1{:=}v_2,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(),h,s[v_1{\mapsto}s(v_2)]\rangle$$

$$\langle v.f,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(h(s(v)).f),h,s\rangle \qquad \langle v_1.f{:=}v_2,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(),h[s(v_1).f{\mapsto}s(v_2)],s\rangle$$

$$\frac{l=\mathit{fresh}()}{\langle fn\#\texttt{new}\ c,h,s\rangle\hookrightarrow\langle s(fn)\#l,h{+}[l{\mapsto}c(\overrightarrow{\bot})],s\rangle}$$

$$\frac{u{=}newid()}{\langle\{t\ v;e\},h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{u\},e[u/v]),h,s{+}[u{\mapsto}\bot]\rangle}$$

$$\frac{c_1{<:}c\quad u{,}fu{=}newid()}{\langle\texttt{try}\ c_1\#a_1\ \texttt{catch}\ c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{fu,u\},e_2[u/v,fu/fv]),h,s{+}[fu{\mapsto}c_1{,}u{\mapsto}a_1]\rangle}$$

$$\frac{\neg(c_1{<:}c)}{\langle\texttt{try}\ c_1\#a_1\ \texttt{catch}\ c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle c_1\#a_1,h,s\rangle}$$

$$\frac{\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle}{\langle\texttt{try}\ e\ \texttt{catch}\ c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle\texttt{try}\ e_1\ \texttt{catch}\ c@fv\#v\ e_2,h_1,s_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2,h,s\rangle\hookrightarrow\langle e_1,h,s\rangle} \qquad \frac{s(v){=}\texttt{false}}{\langle\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2,h,s\rangle\hookrightarrow\langle e_2,h,s\rangle}$$

$$\frac{t_0\ mn\ (\overrightarrow{t\ u})\ \{e\}\quad \overrightarrow{u'{=}newid()}}{\langle mn(\overrightarrow{v}),h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{\overrightarrow{u}'\},e[\overrightarrow{u'/u}]),h,s{+}[\overrightarrow{u'{\mapsto}s(v)}]\rangle}$$

$$\frac{\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle}{\langle\texttt{BLK}(\{\overrightarrow{v}\},e),h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{\overrightarrow{v}\},e_1),h_1,s_1\rangle}$$

$$\langle\texttt{BLK}(\{\overrightarrow{v}\},c\#a),h,s\rangle\hookrightarrow\langle c\#a,h,s{-}\{\overrightarrow{v}\}\rangle$$

Fig. 6. Small-Step Semantics

Proof: By structural induction on e.

Theorem 6.2 (Progress). *Consider a closed configuration $\langle e,h,s\rangle$. If*

$$\vdash\{\Delta\}\ e\ \{\Delta_1\}\quad \textit{and}\quad s,h\models \mathit{Post}(\Delta),$$

then either e is a value, or there exist s_1,h_1, and e_1, such that $\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle$.

Proof: By structural induction on e.

Theorem 6.3 (Soundness). *Consider a closed configuration $\langle e,h,s\rangle$. Assuming that $\vdash\{\Delta\}\ e\ \{\Delta'\}$ and s,h $\models$ Post(Δ), then either $\langle e,h,s\rangle\hookrightarrow^*\langle v,h',s'\rangle$ terminates with a value v such that $(s'{+}[\texttt{res}{\mapsto}v],h')\models \mathit{Post}(\Delta')$ holds, or it diverges $\langle e,h,s\rangle\not\hookrightarrow^*$.*

Proof Sketch: If the evaluation of e does not diverge, it will terminate in a finite number of steps (say n): $\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle\hookrightarrow\cdots\hookrightarrow\langle e_n,h_n,s_n\rangle$. By Theorem 6.1, there exist $\Delta_1,..,\Delta_n$ such that, $s_i,h_i\models \mathit{Post}(\Delta_i)$, and $\vdash\{\Delta_i\}\ e_i\ \{\Delta'\}$. By Theorem 6.2, The final result e_n must be some value v (or it will make another reduction).

7 Experiments

Our verification system is built using Objective Caml. The proof obligations generated by the verification are discharged by the entailment checking procedure with the help of Omega Calculator [12].

In order to prove the viability of our verification method we tried our prototype implementation against a few examples from SPECjvm2008, a widely used Java benchmark created by SPEC. Due to the focus of our work, we only considered those tests from SPECjvm2008 that are related to exception handling. After annotating the tested methods with pre and post conditions, we were able to successfully verify all the tests. Due to the fact that the KeY approach is semi-automated in the sense that it occasionally prompts the user for choice of rewriting rules, while our approach is fully-automated, we cannot perform a direct comparison of the verification timings.

Among these examples, MyClass is a Java program emphasizing the use of exception handling in the presence of user defined exceptions. Its aim is to detect mishandling of the exception class hierarchy. The main objective of While and ContinueLabel examples is testing abrupt termination in the presence of loops. PayCard is a Java class from a real life Java application that makes heavy use of exceptions while modelling the behaviour of a credit card.

Figure 7 contains the timings obtained when using our system to verify the aforementioned examples.

Programs	LOC	Time (seconds)	Focus
Break (KeY)	20	0.11	`break` handling
MyClass (KeY)	33	0.10	exception hierarchy
While (KeY)	130	2.47	while loops and `break`
ContinueLabel (KeY)	100	0.95	imbricated while loops and `continue`
PayCard (KeY)	70	0.91	general exception handling
SPECjvm2008	190	1.20	general exception handling

Fig. 7. Verification Times

We also verified the examples presented in the paper. Method `list_alloc` was verified in 0.41 seconds, `list_alloc1` in 0.26 seconds and method `swap` in 0.09 seconds. Take note that the verification of `list_alloc1`, which ensures an improved strong exception safety guarantee as according to our approach, is faster by 36% than the verification of `list_alloc` which enforces the original strong guarantee defined in [15].

8 Concluding Remarks

We have presented a new approach to the verification of exception-handling programs based on a specification logic that can uniformly handle exceptions, program errors and other kinds of control flows. The specification logic is currently built on top of the formalism of separation logic, as the latter can give precise description to heap-based data structures. Our main motivation for proposing this new specification logic is to adapt the verification method to help ensure exception safety in terms of the four guarantees of increasing quality introduced in [15] and extended in [9], namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee. During the evaluation process, we found the strong guarantee to be restrictive for some scenarios, as it always forces a recovery mechanism on the callee, should exceptions occur. Hence, we propose to generalise the definition of strong guarantee for exception safety. Our approach has been formalised and implemented in a prototype system, and tested on a suite of exception-handling examples. We hope it would eventually become a useful tool to help programmers build more robust software.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS, vol. 4334. Springer, Heidelberg (2007)
2. Blume, M., Acar, U.A., Chae, W.: Exception handlers as extensible cases. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 273–289. Springer, Heidelberg (2008)
3. David, C., Gherghina, C., Chin, W.N.: Translation and optimization for a core calculus with exceptions. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. ACM Press, New York (2009)
4. Fähndrich, M., Aiken, A.: Program analysis using mixed term and set constraints. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 114–126. Springer, Heidelberg (1997)
5. Goodenough, J.B.: Structured exception handling. In: POPL 1975: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 204–224. ACM, New York (1975)
6. Jo, J.-W., Chang, B.-M., Yi, K., Choe, K.-M.: An uncaught exception analysis for Java. Journal of Systems and Software 72(1), 59–69 (2004)
7. Rustan, K., Leino, M., Schulte, W.: Exception Safety for C#. In: SEFM 2004: Proceedings of the Software Engineering and Formal Methods, Second International Conference, Washington, DC, USA, pp. 218–227. IEEE Computer Society, Los Alamitos (2004)
8. Leino, R.: Personal Communication (January 2009)
9. Li, X., Hoover, H.J., Rudnicki, P.: Towards automatic exception safety verification. In: FM, pp. 396–411 (2006)
10. Maxion, R.A., Olszewski, R.T.: Improving software robustness with dependability cases. In: 28th International Symposium on Fault Tolerant Computing, pp. 346–355 (1998)
11. Nguyen, H.H., David, C., Qin, S.C., Chin, W.N.: Automated Verification of Shape And Size Properties via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
12. Pugh, W.: The Omega Test: A fast practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
13. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE LICS, Copenhagen, Denmark, pp. 55–74 (July 2002)
14. Shivers, O., Fisher, D.: Multi-return function call. In: ICFP 2004: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, pp. 79–89. ACM, New York (2004)
15. Stroustrup, B.: Exception safety: Concepts and techniques. In: Advances in Exception Handling Techniques, pp. 60–76 (2000)

Non-monotonic Refinement of Control Abstraction for Concurrent Programs

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko

Technische Universität München

Abstract. Verification based on abstraction refinement is a successful technique for checking program properties. Conventional abstraction refinement schemes increase precision of the abstraction monotonically, and therefore cannot recover from overly precise refinement decisions. This problem is exacerbated in the context of multi-threaded programs, where keeping track of all control locations in concurrent threads is the inevitably discovered abstraction and is prohibitively expensive. In contrast to the conventional (partition refinement-based) approaches, non-monotonic abstraction refinement schemes rely on re-partitioning and have promising potential for avoiding excess of precision. In this paper, we propose a non-monotonic refinement scheme for the control abstraction (of concurrent programs). Our approach employs a constraint solver to discover re-partitioning at each refinement step. An experimental evaluation of our non-monotonic control abstraction refinement on a collection of multi-threaded verification benchmarks indicates its effectiveness in practice.

1 Introduction

Automatic abstraction [10] is one of the essential components for the construction of software verification tools. The success of verification tools based on abstract domains equipped with widening operators, e.g., ASTREE [5], Clousot [13], and Dagger [17], and software model checkers based on predicate abstraction, e.g., SLAM/SDV [3], Blast [21], Magic [6], F-Soft [23], Terminator [9], and ARMC [30], demonstrates the effectiveness of abstraction in practice. Finding the right abstraction is a difficult task, since a too coarse abstraction may lead to inconclusive verification results and, on the other hand, excess of precision may impose a prohibitive efficiency penalty. In practice, the desired level of details tracked during the abstraction process is determined through a trial-and-error like process that adjusts abstraction at each failed verification attempt. The existing refinement methods can automatically tune precision of various abstraction techniques, including infinite abstract domains equipped with widening operators, e.g., [17, 11], and finitary predicate abstraction domains, e.g., [1,7,22,21,4].

One of the most important properties of the iterative abstraction discovery approaches is called progress of refinement. This property ensures that the verification effort does not get stuck in a loop trying to eliminate the same reason for imprecision over and over again. The majority of existing approaches achieve the progress of refinement by adjusting abstraction *monotonically.* That is, they compute a proper refinement of abstraction at each adjusting step. For example, when using predicate abstraction such monotonic refinement can be easily achieved by adding appropriate predicates to the abstract domain [7,1,22,21,24].

While monotonic refinement approaches are well-studied and widely applied, the monotonicity property can lead to overly precise abstraction and hence unnecessarily slow down verification. In fact, monotonicity is just one possible way to achieve refinement progress and alternative approaches have started to emerge. As an example, consider two sequences of abstraction adjustments shown in Figure 1. The sequence (a) uses a monotonic scheme that creates a properly refined partition at each adjustment step. Assume that after making the first adjustment the verifier recognizes that a re-partitioning following the sequence (b) yields an abstraction that is sufficiently precise to prove the property. While not admissible in monotonic refinement scheme, (b) can be achieved using a non-monotonic refinement scheme, which would lead to more efficient verification that considers two instead of four equivalence classes. The potential of such non-monotonic refinement schemes has been identified in recent verification efforts. In a seminal paper [29], a non-monotonic scheme is used to discover a localization abstraction which is improved based on proofs of unsatisfiability. A second related work describes a method for computing an optimal localization abstraction from a collection of broken traces [19]. These approaches led to effective verification methods able to recover from overly precise abstraction decisions.

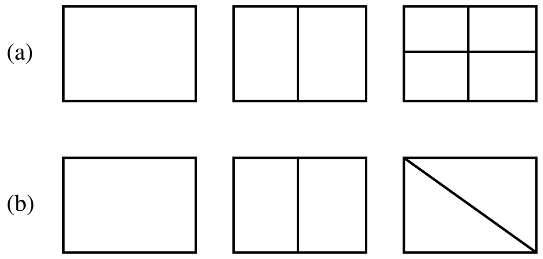


Fig. 1. Abstraction sets of program states using equivalence classes. Boxes denote equivalence classes.

In this paper, we apply a non-monotonic abstraction refinement scheme for the control-flow abstraction of multi-threaded programs. Abstraction of control-flow deals with program counter variables that range over finite sets of control locations of program threads, and is a crucial building block for achieving scalable reasoning about concurrent programs [15,8]. In our experience, monotonic abstraction results in too fine grained partitioning of control locations into equivalence classes and hence is too expensive.

The main component of our scheme is a procedure that takes as input a set of program paths that were the root cause for failed verification attempts so far (including the current evidence for inadequacy of chosen abstraction) and returns a set of predicates that eliminates all these failures. The abstract domain is adjusted by *replacing* the previously used set of predicates by the output of the above procedure (and not adding predicates as in conventional CEGAR-based

```
int t1 = 0, t2 = 0; // ticket variables
bool choosing1 = 0, choosing2 = 0; // boolean flags
int x; //variable to update in critical section


void thr1() {                                 void thr2() {
   int tmp;                                       int tmp;
1  choosing1 = 1;                             11  choosing2 = 1;
2  tmp = t2 + 1;                              12  tmp = t1 + 1;
3  t1 = tmp;                                  13  t2 = tmp;
4  choosing1 = 0;                             14  choosing2 = 0;
5  while (choosing2 != 0);                    15  while (choosing1 != 0);
6  while (t1 >= t2 && t2 != 0);               16  while (t2 >= t1 && t1 != 0);
   // begin: critical section                     // begin: critical section
7  x = 0;                                     17  x = 1;
8  assert(x <= 0);                            18  assert(x >= 1);
   // end: critical section                       // end: critical section
9  t1 = 0;                                    19  t2 = 0;
10 }                                          20 }
```

Fig. 2. An implementation of Lamport's Bakery algorithm

approaches). More specifically, our algorithm crucially relies on a repartitioning step encoded as a SAT problem.

We implemented the two schemes for abstraction refinement and observed on a set of multi-threaded examples that non-monotonic refinement enables an improvement in the verification time ranging from 18% to 52% when compared to a monotonic refinement scheme.

In summary, our contributions are a non-monotonic abstraction refinement algorithm for control-flow abstraction, its implementation and experimental evaluation.

2 Example

In this section, we illustrate our approach on Lamport's Bakery algorithm, which is a classic verification benchmark. We use the complete version of the algorithm [25] with a set of Boolean flags (`choosing1` and `choosing2`) where the reading and the incrementing of the ticket variable is done non-atomically (see lines 2 and 3). We are interested in verifying that the Bakery algorithm achieves mutual exclusion. This safety property is instrumented in Figure 2 using a global variable `x` in the critical section of the two threads. We want to prove that no interleaving of the threads leads to an assertion violation at either line 8 or 18.

To prove the program correct, our algorithm performs a combination of standard abstract reachability computation and non-monotonic abstraction refinement. Abstract states represent sets of concrete program states. If the reachability computation finds an error state to be reachable, we analyze the reason for the failure and update the abstraction, if possible.

For our example, a reason for failure to prove safety is the following interleaving of statements from first and second threads

$$(\underbrace{1,2,3,4,5,}_{\text{thr1}}\underbrace{13,}_{\text{thr2}}\underbrace{6,7,}_{\text{thr1}}\underbrace{17,}_{\text{thr2}}\underbrace{\ 8\ }_{\text{thr1}}),$$

where we identify program statements by the corresponding line numbers. This counterexample is in fact infeasible, and is discovered due to abstraction. Two different reasons make this counterexample infeasible:

- the first statement executed from the second thread cannot be 13,
- the statement 13 cannot be followed in the second thread by the statement 17.

The mismatches between the program locations that lead to the infeasibility of the counterexample are denoted using the following notation: $(11 \not\equiv 13)$, $(14 \not\equiv 17)$. An abstraction function that maps the concrete program locations 11 and 13 to different abstract program locations will be able to avoid this counterexample in subsequent reachability iterations. Similarly, this counterexample can be avoided if the concrete locations 14 and 17 map to different abstract locations $(14 \not\equiv 17)$.

Let us assume that the refinement procedure picks the first possibility. The resulting abstraction function can be represented using the following partition of program locations: $\{11\}, \{13\}, PC_2 \backslash \{11, 13\}, PC_1$. PC_1 and PC_2 represent the sets of all program locations from the first and, respectively, second thread.

A subsequent reachability computation finds another counterexample that can be shown infeasible with the following mismatch relation, $12 \not\equiv 16$. To avoid this counterexample, the partitioning of the locations is updated to $\{11, 12\}, \{13, 16\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. With a third mismatch relation $11 \not\equiv 12$, the program locations are again repartitioned to $\{11, 16\}, \{12, 13\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. Note that these repartitionings are possible only with a non-monotonic abstraction refinement scheme. The standard monotonic refinement would compute a more fine-grained partitioning that is unnecessarily precise and leads to expensive abstract reachability computations. Overall, the verification of the Bakery example using non-monotonic abstraction refinement concludes after 26 seconds and computes a 10-way partitioning of control locations.

The monotonic abstraction refinement concludes after 54 seconds. The mismatch $11 \not\equiv 13$ and the second mismatch $12 \not\equiv 16$ lead immediately to the partitioning $\{11\}, \{12\}, \{13\}, \{16\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. Through all the reachability iterations, the control locations are split into 14 partitions and this large number explains the increased verification time based on monotonic abstraction refinement.

3 Preliminaries

In this section we define programs and computations, and provide a brief description of predicate abstraction-based approach to program verification together with a standard counterexample-guided abstraction refinement procedure.

Programs and computations. We assume an abstract representation of programs by transition systems [27]. A *program* $P = (\Sigma, s_{\mathcal{I}}, \mathcal{T}, s_{\mathcal{E}})$ is given by a set of program *states* Σ, an *initial* state $s_{\mathcal{I}} \in \Sigma$, a set of *transitions* $\mathcal{T}$, and an *error* state $s_{\mathcal{E}} \in \Sigma$. Each transition $\tau \in \mathcal{T}$ has a corresponding *transition relation* $\rho_\tau \subseteq \Sigma \times \Sigma$. The error state $s_{\mathcal{E}}$ is used to represent assertion statements commonly present in programming languages. Each failed assertion leads to $s_{\mathcal{E}}$.

A *computation* of P is a sequence of states $s_1, s_2, \dots$ such that s_1 is the initial state, i.e., $s_1 = s_{\mathcal{I}}$, and there is a transition $\tau \in \mathcal{T}$ between each pair of consecutive states s and s', i.e., $(s, s') \in \rho_\tau$. A state s is *reachable* if it appears in some computation. The program is *safe* if the error state is *not* reachable in any computation.

A *path* is a sequence of transitions. Let $\circ$ be the *relational composition* function for binary relation over states, i.e., for $X, Y \subseteq \Sigma \times \Sigma$ we have $X \circ Y = \{(s, s') \mid \exists s'' \in \Sigma : (s, s'') \in X \wedge (s'', s') \in Y\}$. Then, a path relation ρ_π is a relational composition of transition relations along the path, i.e., for $\pi = \tau_1 \dots \tau_n$ we have $\rho_\pi = \rho_{\tau_1} \circ \dots \circ \rho_{\tau_n}$. A path is *feasible* if its path relation is not empty.

Predicate abstraction. Our goal is to verify whether a given program is safe. To achieve this goal we need to consider all reachable program states and check if the error state appears among them. The set of all reachable states can be computed iteratively using the function *post* : $(\mathcal{T} \times 2^\Sigma) \to 2^\Sigma$ such that

$$\mathit{post}(\tau, S) = \{s' \mid \exists s \in S : (s, s') \in \rho_\tau\} \ .$$

Its least fixed point above $\{s_{\mathcal{I}}\}$ is the set of reachable states, i.e.,

$$s \text{ is reachable if and only if } s \in \mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}(\tau, S), \{s_{\mathcal{I}}\}) \ .$$

The exact computation of the set of reachable states is an undecidable problem, however for the verification purposes a sufficiently close *abstraction* is enough. The framework of abstract interpretation [10] provides a formal foundation for the approximate, yet sound abstraction of reachable states, where abstraction is defined as an *over-approximation*. Given an abstraction function $\alpha : 2^\Sigma \to 2^\Sigma$ such that

$$\forall S \subseteq \Sigma : S \subseteq \alpha(S) \ ,$$

we construct an *abstraction* $\mathit{post}^\#$ of *post* as follows:

$$\mathit{post}^\#(\tau, S) = \alpha(\mathit{post}(\tau, S)) \ .$$

Our abstraction puts together and operates on sets of program states. We call such sets *abstract states* and let $\Sigma^\# = 2^\Sigma$ be the set of all abstract states.

The least fixed point of *post*$^\#$ above the abstraction of the initial state is an over-approximation of the reachable states, i.e.,

$$\mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}^\#(\tau, S), \alpha(\{s_\mathcal{I}\})) \supseteq \mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}(\tau, S), \{s_\mathcal{I}\}) \ .$$

If the error state is not included in the over-approximation then the program is safe, that is, we obtain a sound method for verifying program safety. For completeness of presentation, Appendix A contains an algorithm for abstract fixpoint checking together with the re-construction of counterexamples, which is required by our refinement scheme.

The abstraction function α can be constructed automatically from a given set of basic building blocks, called *predicates*, where a predicate represents a set of program states. Given a set of predicates $\mathcal{P} = \{P_1, \ldots, P_n\}$, where $P_i \subseteq \Sigma$, and a *theorem prover* that can decide validity of subset inclusion between sets of states represented in a logical language, we use an abstraction function $\alpha^\mathcal{P} : 2^\Sigma \to 2^\Sigma$ which returns the strongest conjunction of the predicates implied by S as follows.

$$\alpha^\mathcal{P}(S) = \cap\{P \in \mathcal{P} \mid S \subseteq P\}$$

Abstraction refinement. In order to verify program safety using predicate abstraction, we need to supply a set of predicates. Predicates can be provided manually, collected from the program text by applying heuristics, or derived in a goal-oriented way by using the counterexample-guided abstraction refinement approach [7]. The crux of this approach to predicate discovery lies in leveraging *spurious counterexamples*, which are program paths that expose the coarseness of the abstraction function determined by the currently used set of predicates.

A path $\pi = \tau_1 \ldots \tau_n$ is a spurious counterexample if the abstract reachability computation along the path leads to the error states, i.e.,

$$s_\mathcal{E} \in \mathit{post}^\#(\tau_n, \mathit{post}^\#(\tau_{n-1}, \ldots \mathit{post}^\#(\tau_1, \alpha^\mathcal{P}(\{s_\mathcal{I}\})))\ ,$$

but the actual, not abstracted path does not lead to the error state, i.e., $(s_\mathcal{I}, s_\mathcal{E}) \notin \rho_\pi$. Conventional techniques for analyzing spurious counterexamples use automated reasoning approaches , e.g., proofs [22] and interpolation [21], to extract a set of new predicates that *excludes* the spurious counterexample.

We define an auxiliary predicate *SafeInd* that takes as input a sequence of predicates of length $n+1$ and a sequence of program transitions of length n, where $n \geq 1$, as follows.

$$\begin{aligned} \mathit{SafeInd}(\varphi_0 \ldots \varphi_n, \tau_1 \ldots \tau_n) = \ & s_\mathcal{I} \in \varphi_0 \wedge s_\mathcal{E} \notin \varphi_n \wedge \\ & \qquad \forall i \in 1..n : \textstyle\bigwedge_{i \in 1..n} \mathit{post}(\tau_i, \varphi_{i-1}) \subseteq \varphi_i \end{aligned}$$

Given a spurious counterexample $\tau_1 \ldots \tau_n$, we say that the sequence of predicates $\varphi_0 \ldots \varphi_n$ excludes the counterexample if $\mathit{SafeInd}(\varphi_0 \ldots \varphi_n, \tau_1 \ldots \tau_n)$ holds. We will use *SafeInd* in our non-monotonic refinement scheme.

```
function NonMonRefine
input
   Paths : spurious counterexamples so far
   $\pi$ : current spurious counterexample
begin
   choose $\mathcal{P}$ such that
       $\forall \tau_1 \dots \tau_n \in \{\pi\} \cup \mathsf{Paths}$
           $\exists \varphi_0, \dots, \varphi_n \subseteq \mathcal{P}$ :
               $\mathit{SafeInd}(\varphi_0 \dots \varphi_n, \tau_1 \dots \tau_n)$
   return $\mathcal{P}$
end
procedure NonMonCEGAR
input
   $P$ : program
vars
   $\mathcal{P}$ : abstraction predicates
   Paths : spurious counterexamples so far
begin
   $\mathcal{P} := \emptyset$
   Paths $:= \emptyset$
   repeat
       match FindCounterexample($P, \mathcal{P}$) with
        | Some $\pi$ ->
           if $\rho_\pi = \emptyset$ then
               $\mathcal{P} :=$ NonMonRefine($\mathsf{Paths}, \pi$)
               Paths $:= \{\pi\} \cup \mathsf{Paths}$
           else
               return "Counterexample $\pi$ to program safety"
        | None ->
           return "Program is safe"
end.
```

Fig. 3. Predicate abstraction-based algorithm for checking program safety that is based on the non-monotonic abstraction refinement scheme.

4 Non-monotonic Refinement Scheme

In this section we present an abstraction refinement scheme that adjusts abstraction in a non-monotonic way. By not committing to a monotonic evolution of the abstraction, we can obtain a greater choice of possible refinement steps and hence can reach more favorable efficiency/precision trade-offs.

See Figure 3 for an algorithm NonMonCEGAR that implements a safety verification procedure based on counterexample guided abstraction refinement using the non-monotonic refinement NonMonRefine. The algorithm NonMonCEGAR crucially differs from a conventional CEGAR algorithm by keeping the history of discovered counterexamples, as stored in the variable Paths.

Every time an infeasible counterexample π is found, see lines 4–6, the set of previously discovered counterexamples π together with the current one is passed to NonMonRefine. The function NonMonRefine in our scheme chooses a set of predicates $\mathcal{P}$ that excludes all counterexamples discovered so far. In Section 5 we present an instantiation of NonMonRefine for control-flow abstraction that uses an encoding into SAT to implement lines 1–4 of NonMonRefine.

NonMonCEGAR overwrites the set of abstraction predicates $\mathcal{P}$ using the result of calling NonMonRefine on the current set of counterexamples. At this step, non-monotonicity takes place. Note however that the progress of refinement is guaranteed, as formalized by the theorem below.

Theorem 1 (Progress of refinement in NonMonCEGAR). *The algorithm* NonMonCEGAR *never discovers the same counterexample twice, i.e., for given values of $\mathcal{P}$ and* Paths *we have that if $\pi \in$* Paths *then $\pi \notin$* FindCounterexample$(P, \mathcal{P})$.

5 Non-monotonic Refinement for Control-Flow Abstraction

In this section we present an application of the non-monotonic abstraction refinement scheme to control-flow abstraction for concurrent programs. Our algorithm for the verification of multi-threaded programs [20] relies on the abstraction of control-flow, i.e., over-approximation of set of control locations in which threads can be residing. This abstraction plays a crucial role for enabling scalable reasoning in the multi-threaded setting. Our experiments with a conventional monotonic abstraction refinement procedure for dealing with control-flow abstraction were not satisfactory. The refinement process was creating as many individual abstract values as there are control locations, which subverted the application of abstraction by effectively making the abstraction function to be an identity function. In this section, we only present the non-monotonic control abstraction refinement and refer to [20] for its client algorithm.

We assume a multi-threaded program that consists of N threads whose control locations are given by the set $\mathcal{L}$. For each thread $i \in 1..N$ we use a variable pc_i and its primed version pc_i' to refer to the corresponding program counter value. We consider counterexamples given by sequences of transitions whose transition relations are of the form

$$pc_i = \ell \wedge pc_i' = \ell' \wedge \bigwedge_{j \in 1..N \setminus \{i\}} pc_j = pc_j' \ ,$$

where ℓ and ℓ' are control locations. This transition relation corresponds to a step of the thread i, whereas each other thread $j \in 1..N \setminus \{i\}$ idles and hence does not change its control location. We assume a function $\mathit{from} : \mathcal{T} \to \mathcal{L}$ that given a transition τ returns its start location, which ℓ for the transition relation above.

```
function NONMONCONTROLREFINE
input
  $\mathsf{Paths}$ : spurious counterexamples so far
vars
  $\Phi, \Psi$ : auxiliary constraints
  $m$ : number of partitions
  $B$ : auxiliary propositional variables
  $\mathit{bits}$ : encodes equivalence classes of control locations as bit strings
begin
  $m := 2$
  repeat
    $B := \emptyset$
    for each $\ell \in \mathcal{L}$ do
      $b_1 \dots b_{\lceil \log_2(m) \rceil}$ := fresh propositional variables
      $B := \{b_1, \dots, b_{\lceil \log_2(m) \rceil}\} \cup B$
      $\mathit{bits}(\ell) := b_1 \dots b_{\lceil \log_2(m) \rceil}$
    $\Phi := \mathsf{true}$
    for each $\pi = \tau_1 \dots \tau_n \in \mathsf{Paths}$ do
      $\Psi := \mathsf{false}$
      for each $k \in 0..n$ and $i \in 1..N$ and $j \in 1..n-k+1$ and $\ell \in \mathcal{L}$ such that
        $\mathit{SafeInd}(\underbrace{\mathsf{true} \dots \mathsf{true}}_{k \text{ times}} \underbrace{pc_i = \ell \dots pc_i = \ell}_{j \text{ times}} \underbrace{\mathsf{false} \dots \mathsf{false}}_{n-k-j+1 \text{ times}}, \pi)$
      do
        $\ell' :=$ if $k + j = n + 1$ then $s_{\mathcal{E}}(pc_i)$ else $\mathit{from}(\tau_{k+j})$
        $\Psi := \Psi \vee \mathit{bits}(\ell) \neq \mathit{bits}(\ell')$
      $\Phi := \Psi \wedge \Phi$
    $m := m + 1$
  until exists $\sigma : B \to \{\mathsf{true}, \mathsf{false}\}$ such that $\models \sigma(\Phi)$
  for each $\ell \in \mathcal{L}$ do
    $f_{\equiv}(\ell) := \{\ell' \in \mathcal{L} \mid \sigma(\mathit{bits}(\ell)) = \sigma(\mathit{bits}(\ell'))\}$
  return $f_{\equiv}$
end.
```

Fig. 4. Function NONMONCONTROLREFINE implements an instantiation of the non-monotonic refinement scheme to control-flow abstraction. The application $\sigma(\mathit{bits}(\ell))$ computes a bit string by replacing propositional variables from $\mathit{bits}(\ell)$ by their values as determined in σ.

For example, the counterexample $\pi = \tau_1\tau_2\tau_3\tau_4\tau_5\tau_{13}\tau_6\tau_7\tau_{17}\tau_8$ presented in Section 2 involves the following transition relations:

$$\rho_i = \begin{cases} pc_1 = i \wedge pc_1 = i + 1 \wedge pc_2 = pc_2' \ , & \text{for } i \in \{1, \dots, 8\} \ , \\ pc_1 = pc_1' \wedge pc_2 = i \wedge pc_2' = i + 1 \ , & \text{for } i \in \{13, 17\} \ . \end{cases}$$

The transitions have the following starting locations:

$$\mathit{from}(\tau_i) = i \ , \ \text{for } i \in \{1, \dots, 8, 13, 17\}.$$

Our goal is to compute an equivalence relation $\equiv$ on $\mathcal{L}$ that leads to absence of abstract counterexamples. We represent the equivalence relation by a characteristic function $f_\equiv : \mathcal{L} \to 2^{\mathcal{L}}$ from control locations to equivalence classes. The equivalence classes of this relation are used as predicates defining control abstraction for a thread $i \in 1..N$, i.e.,

$$\alpha(S) = \cup\{f_\equiv(\ell) \mid (S \cap (pc_1 = \ell \cup \cdots \cup pc_N = \ell)) \neq \emptyset\} \ .$$

For example, Section 2 first discovers an equivalence relation that consists of three equivalence classes $\{11\}$, $\{13\}$, and $PC_2 \backslash \{11, 13\}$. This equivalence relation yields a control-flow abstraction that, for example, yields the following result:

$$\alpha(\{pc_2 = 11, pc_2 = 16\}) = \{11\} \cup PC_2 \setminus \{11, 13\} \ .$$

The following observation underlines our algorithm for non-monotonic refinement of control-flow abstraction. Each spurious counterexample, say $\tau_1 \ldots \tau_n$ can be eliminated by keeping track of a certain predicate $pc_i = \ell$, i.e., if $\rho_{\tau_1 \ldots \tau_n} = \emptyset$ then there exists $i \in 1..N$ and $\ell \in \mathcal{L}$ such that for $k \in 1..n$ and $j \in 1..n - k + 1$ holds

$$SafeInd(\underbrace{\mathsf{true} \ldots \mathsf{true}}_{k \text{ times}} \underbrace{pc_i = \ell \ldots pc_i = \ell}_{j \text{ times}} \underbrace{\mathsf{false} \ldots \mathsf{false}}_{n-k-j+1 \text{ times}}, \tau_1 \ldots \tau_n) \ .$$

For our counterexample π shown above, one refinement possibility is given below:

$$SafeInd(\underbrace{pc_2 = 11 \ldots pc_2 = 11}_{6 \text{ times}} \underbrace{\mathsf{false} \ldots \mathsf{false}}_{5 \text{ times}}, \pi) \ .$$

Figure 4 shows an algorithm NonMonControlRefine that computes a characteristic function for adjusting the control-flow abstraction. The algorithm finds an equivalence relation with the minimal number of equivalence classes, which decreases the size of the abstract state space and improves efficiency of the abstract reachability computation. Our implementation relies on a propositional encoding that describes constraints on the characteristic functions. These constraints can be solved efficiency using a state-of-the-art SAT solver.

We illustrate NonMonControlRefine using the counterexample π above, which is taken from Section 2, and assume that the input set Paths contains only the path π. Line 1 in Figure 4 initializes the number of equivalence classes m to 2, which serves as the first candidate. The **repeat** loop (lines 2–17) attempts to find a control abstraction with at most m equivalence classes. If no such abstraction exists then m is incremented and the attempt is repeated. This iteration terminates after at most $|\mathcal{L}|$-many steps, where $|\mathcal{L}|$ is the size of $\mathcal{L}$.

At the first attempt, we start by creating propositional variables that keep track of equivalence classes for control locations, see lines 4–7. For our example, we assume $bits(11) = (b_1 b_2)$ and $bits(13) = (b_3 b_4)$, and hence B contains $\{b_1, b_2, b_3, b_4\}$.

Since Paths contains only one counterexample, namely π, the **for** loop (lines 9–15) is executed only once. This path has two root causes of infeasibility, which leads to two iterations of the inner **for** loop in lines 11–14. At the first one we obtain $pc_2 = 11$, $k = 0$, $j = 6$, and $\ell' = \mathit{from}(\tau_{13}) = 13$. Then, line 14 computes $\Psi = \mathsf{false} \vee \mathit{bits}(11) \neq \mathit{bits}(13)$. This constraint encodes the condition that the control locations 11 and 13 need to be distinguished by the control abstraction, formally, $11 \not\equiv 13$.

The next iteration of the inner **for** loop discovers that for $k = 6$ and $j = 3$ we have

$$\mathit{SafeInd}(\underbrace{\mathsf{true} \ldots \mathsf{true}}_{6 \text{ times}} \underbrace{pc_2 = 14 \ldots pc_2 = 14}_{3 \text{ times}} \mathsf{false}\,\mathsf{false}, \pi) ,$$

and $\ell' = \mathit{from}(\tau_{17}) = 17$. We finish the execution of the inner **for** loop and obtain the final constraint

$$\Phi = \mathit{bits}(11) \neq \mathit{bits}(13) \vee \mathit{bits}(14) \neq \mathit{bits}(17) .$$

The first disjunct in Φ requires that at least one bit of $\mathit{bits}(11)$ is different from the corresponding bit in $\mathit{bits}(13)$. This condition translates to $(b_1 \neq b_3 \vee b_2 \neq b_4)$, which is equivalent to $(b_1 \wedge \neg b_3) \vee (\neg b_1 \wedge b_3) \vee (b_2 \wedge \neg b_4) \vee (\neg b_2 \wedge b_4)$.

The constraint Φ is satisfiable. We consider a solution σ such that $\sigma(\mathit{bits}(11)) = (0\,0)$, $\sigma(\mathit{bits}(13)) = (0\,1)$, $\sigma(\mathit{bits}(14)) = (0\,0)$, and $\sigma(\mathit{bits}(17)) = (0\,0)$. This solution leads to the characteristic function $f_\equiv$ that maps 11, 14, and 17 to the same equivalence class. This equivalence class is different from $f_\equiv(13)$.

At each refinement iteration more and more conjuncts are added to the constraint Φ in line 15. As an additional optimization, we first try to find same number of partitions among program counters as the number of partitions found in the last iteration. If this fails, then we grow the number of partitions one by one. In the worst case, the partition size may grow upto the number of program locations. However, in our experiments the number of control partitions was much lower indicating the benefit of control abstraction.

6 Experiments

We implemented the algorithm NonMonControlRefine in our tool for the verification of multi-threaded programs written in the C language. Since our tool uses both data abstraction and control abstraction, it may be possible that some spurious counterexample can be ruled out by both data abstraction refinement and control abstraction refinement. In this situation, we use an heuristic that prefers data refinement over control refinement. Our tool uses a standard (i.e. monotonic) abstraction refinement scheme for dealing with data variables, and relies on NonMonControlRefine for the discovery of adequate control abstraction. Constraints generated by NonMonControlRefine are resolved using the Z3 solver [12]. Next, we will report our experience with applying NonMonControlRefine to the verification of multi-threaded programs.

We evaluated the non-monotonic refinement scheme in direct comparison with monotonic one and present a summary in Table 1. Our examples include two versions of the Bakery algorithm for mutual exclusion. Bakery [25] is shown

Table 1. Comparison between monotonic and non-monotonic refinement of control abstraction. For each configuration we present (i) the total verification time, its decomposition into time spent on (ii) the abstract reachability computation and (iii) abstraction refinement, together with (iv) the number of equivalence classes $|\equiv|$ that determine the control abstraction.

	Monotonic refinement			Non-monotonic refinement		
Program	Time		$\|\equiv\|$	Time		$\|\equiv\|$
BAKERY-ATOMIC [27]	6.6s	5.7+0.9	8	4.8s	4.1+0.7	7
BAKERY [25]	54s	48.4+5.6	14	26s	23.1+2.9	10
BLUETOOTH [31]	19.5s	16.4+3.1	7	16.4s	11.3+5.1	5
MOZILLA-ORDER-FIXED [26]	2.7s	2.1+0.6	5	1.6s	0.9+0.7	3
TIME-VARYING-MUTEX [14]	9.6s	8.7+0.9	10	7.1s	6.3+0.8	7

in Figure 2, while BAKERY-ATOMIC is its simplified version that increments the ticket variable atomically [27]. BLUETOOTH models the stopping procedure of a Windows NT Bluetooth driver [31], where a worker thread asserts that a boolean flag stopped is not set to false by a second stopper thread. MOZILLA-ORDER-FIXED is the fixed version of a vulnerability from the MOZILLA CVS repository, which was discussed in abbreviated form in [26, Figure 2]. The property to verify is that two operations performed by different threads are executed in the correct order. Lastly, TIME-VARYING-MUTEX illustrates a synchronization idiom found in the Frangipani file system [14], where it is verified if a thread has exclusive access over a disk block.

At a high level, our approach can be viewed as an optimization step with a trade-off. While the non-monotonic refinement keeps the number of equivalence classes $|\equiv|$ smaller, it has to solve a growing set of constraints which may impact on the refinement time. On our set of examples, we observed that the increase in refinement time is acceptable and the coarser abstraction that is discovered leads to a smaller time for abstract reachability computation. Consequently, the time for non-monotonic verification compares favorably to that for verification via monotonic refinement. We found overall time savings ranging from 18% for BLUETOOTH to 52% for BAKERY.

7 Related Work

Our paper builds upon counterexample-based model checking [7,2,22,6], which mostly employs monotonic refinement techniques that consider a single counterexample at a time and are based on weakest preconditions [2] and interpolation [21]. Our non-monotonic scheme eliminates all previously discovered spurious counterexample, which is in contrast to the elimination of all spurious counterexamples of a given length [16].

Previous non-monotonic abstraction refinement approaches focus on data refinement, see e.g. [19,28]. The collection of broken traces in [19] is closely related to our history of counterexamples. While [19] identifies which data variables to keep track by analysing broken traces, our approach first employs a constraint-based reduction, which may be viewed as a generalization. The non-monotonic abstraction refinement using interpolants [28] avoids explicit construction of abstract state transformer that is usually required for program verification. Instead, an interpolation procedure simultaneously adjusts precision for all previously discovered spurious counterexamples. In contrast to [28], our non-monotonic control abstraction imposes additional constraints on the form of the obtained abstraction using constraints.

Monotonicity plays a crucial role for widening operators in abstraction interpretation framework [10] and its automatic refinement [18,11,32,17]. Refinement techniques for widening achieve monotonicity by considering results of abstract reachability tree computation from the previous iterations, see e.g. [18,17]. We are not aware of non-monotonic refinement in this domain.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, p. 158. Springer, Heidelberg (2002)
2. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, p. 103. Springer, Heidelberg (2001)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI (2007)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
6. Chaki, S., Clarke, E.M., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient verification of sequential and concurrent C programs. Formal Methods in System Design 25(2-3), 129–166 (2004)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
8. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. Formal Methods in System Design 34(2), 104–125 (2009)
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
11. Cousot, P., Ganty, P., Raskin, J.-F.: Fixpoint-guided abstraction refinements. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 333–348. Springer, Heidelberg (2007)
12. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

13. Ferrara, P., Logozzo, F., Fähndrich, M.: Safer unsafe code for.NET. In: OOPSLA (2008)
14. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
15. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
16. Glusman, M., Kamhi, G., Mador-Haim, S., Fraer, R., Vardi, M.Y.: Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 176–191. Springer, Heidelberg (2003)
17. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
18. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 474–488. Springer, Heidelberg (2006)
19. Gupta, A., Clarke, E.M.: Reconsidering CEGAR: Learning good abstractions without refinement. In: ICCD, pp. 591–598 (2005)
20. Gupta, A., Popeea, C., Rybalchenko, A.: (2009) (in preparation), http://www.model.in.tum.de/~popeea/research/environment.pdf
21. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
22. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
23. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
24. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
25. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. ACM Commun. 17(8), 453–455 (1974)
26. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS (2008)
27. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer, Heidelberg (1995)
28. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
29. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
30. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL (2007)
31. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI, pp. 14–24 (2004)
32. Ranzato, F., Rossi-Doria, O., Tapparo, F.: A forward-backward abstraction refinement algorithm. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 248–262. Springer, Heidelberg (2008)

A Abstract Fixpoint Checking

In this appendix we briefly revisit abstract fixpoint checking together with the re-construction of counterexample paths. See Figure 5 for the algorithm FIND-COUNTEREXAMPLE. The algorithm takes as input a program and a set of predicates defining the abstraction function. The computation of abstract reachable states is implemented using a queue of abstract states whose successors are yet to be computed. In order to be able to re-construct a counterexample path in case the error state of the program is reached (see line 8), the auxiliary relation Parent keeps track of how each abstract state is reached. The counterexample re-construction is performed in lines 9–13 via a backward traversal.

```
     function FindCounterexample
     input
       $P$ : program
       $\mathcal{P}$ : abstraction predicates
     vars
       Reach : reached abstract states
       Parent : parent relation
       Queue : queue of abstract states
       $n$, $n'$ : abstract states
       $\tau$ : program transition
     begin
1      Parent := $\emptyset$
2      Reach := $\{\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})\}$
3      add $\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})$ to Queue
4      while Queue is not empty do
5          $n$ := take from Queue
6          for each $\tau \in \mathcal{T}$ do
7              $n'$ := $post^{\#}(n, \tau)$
8          if $s_{\mathcal{E}} \in n'$ then
9              $\pi$ := $\tau$
10             while exists $n$ and $\tau$ such that $(n, \tau, n') \in$ Parent do
11                 $n'$ := $n$
12                 $\pi$ := $\tau' \cdot \pi$
13             done
14             return Some $\pi$
15         else if $\neg(\exists m \in$ Reach $: n' \subseteq m)$ then
16             add $n'$ to Queue
17             Reach := $\{n'\} \cup$ Reach
18             Parent := $\{(n, \tau, n')\} \cup$ Parent
19     done
20     return None
     end
```

Fig. 5. Abstract fixpoint checking algorithm combined with a counterexample construction step

An Approach for Class Testing from Class Contracts

Atul Gupta

Indian Institute of Information Technology, Design & Manufacturing,
Jabalpur, MP, INDIA - 482005
atul@iiitdmj.ac.in

Abstract. Adequate testing of a class requires testing valid and invalid sequences of class method interactions. In this paper, we show that *class contracts* can be used to generate effective state based unit tests algorithmically for testing meaningful interactions between methods. Using an abstract state configuration for the object and an initial abstract state, we incrementally search for the methods those can be invoked in the current state and compute resulting abstract states. The same is repeated for each newly generated abstract state till no more new abstract states are generated. This search generates a directed graph from which test sequences can easily be obtained. We applied the proposed approach to test three Java programs seeded with mutation faults and obtained high mutation scores.

Keywords: Class Testing, unit testing, class contracts, abstract state, experiment, mutation.

1 Introduction

The primary intent of class testing is to find discrepancies between the class specifications and the code. Therefore, class specification is the primary reference for the test plan for writing effective unit tests. Testing of methods of a class in isolation is not sufficient for adequate testing of the class and one needs to test valid and invalid sequences of method interactions as well [24,3,23]. However, the methods of a class can be invoked in any order (except the 'constructors') and such sequences can be virtually infinite. Therefore, an important question is how can adequate testing of the method interactions for a given class be performed? This issue is addressed in this paper.

An important constituent of the class specifications is the *class contracts* which specifies the conditions that must hold just prior to the invocation of each method of the class and after its execution [17]. These conditions are *class invariants*, *preconditions* and *postconditions* of the methods. We use Object Constraint Language (OCL) [20] for representing these constraints which include constraints on simple data types as well as complex data types (e.g. a collection). OCL supports a set of query operations (e.g. *Collect operations* for the *Collection* data type) which help to specify complex constraints. These query operations are also useful to construct test case assertions in order to check the outcome of a test case.

Our approach to perform class testing from the *class contracts* is as follows: Using an abstract state configuration of the object and an initial abstract state, we incrementally generate other reachable abstract states by searching for the methods which can be invoked in the current abstract state and compute resulting abstract states. This process complements the identification of test sequences along with formulation of the conditions for checking the test outcomes (test oracles) to construct the unit tests for the class.

We demonstrate the application of the proposed approach by an experiment in which we tested three Java programs for which *class contracts* were given in OCL. For each test-program, we followed the proposed approach to generate unit tests which were later executed on a set program's mutants in order to measure the effectiveness of the proposed approach.

The rest of the paper is organized as follows. We first describe our method to represent the abstract state configuration for an object of a class under test in Section 2. The method to generate object's abstract states and test sequences from its *class contracts* is given in Section 3. In Section 4, we present an experiment evaluating the effectiveness of the proposed class testing approach and discuss other issues relating to the potential application of the work in Section 5. We present a comparison of our work with other related work in Section 6 and summary in Section 7.

2 An Abstract State Configuration for the Class

Typically, an object's state is specified by the concrete values of its variables. But the problem of *state explosion* makes it intractable to analyze such behaviour. This problem can be effectively tackled by the notion of 'abstract states' which facilitates variables to take abstract values (symbolic values) over their input domains. In this situation, the object behaviour can be analyzed by (1) identifying a set of reachable abstract states (2) observing its behaviour by exercising valid and invalid method invocations in each of the reachable states (3) by selecting a set of representational values from the method's input parameter space abstraction. Hence abstraction provides an opportunity to carry out a systematic behavioural analysis which otherwise may not be possible.

Formally, we abstract out class variables (including input parameters of the methods of the class) corresponding to their data types. Specifically,

– Numeric data types like integer, float, and double are mapped to a finite set of disjoined partitions (i.e. ranges) over their valid state space. For example an integer state variable X is mapped to three abstract states $x < 0$, $x = 0$, and $x > 0$.
– Boolean, character, and enumeration data types are mapped to one singleton abstract state for each single value.
– String references S are mapped to either $S = null$ or $S = non - null$.
– Object references Z are mapped either to the abstract state $Z = null$, or to the abstract state $Z.isInstanceOf(C)$ for the objects of each class C which may be referenced by Z.
– Collection references involving a property of the collection are mapped to one singleton abstract state of a *derived variable* (defined later in this section) for each possible value of that property.
– Additional conditions specified on various data types are partitioned accordingly. For example, whether a string S contains a given substring S' is mapped to a

boolean function $contains(S, S')$ which returns $TRUE$ (or $FALSE$) if S contains S' (or S does not contain S'). A *derived variable* is used to store the results of this function.

We handle a complex constraint or a constraint on an object by defining a function which evaluates the given expression and returns a *derived* variable of a primitive data type. A *derived* variable helps to model the complex constraints specified in the *class contracts*. In such cases, we update the *class contracts* to incorporate these *derived* variable(s). A *derived* variable can be defined using the '*def*' construct in OCL. An example for the same is given below.

context Hotel def:
let numberOfSingleRoomAvailable : int =
(self.singleRoom→collect(singleRoom.isAvailable == FALSE)).size()

where *numberOfSingleRoomAvailable* is a *derived* variable which is computed by calling a *Collection* operation $collect(CollectionC)$ (defined in OCL) on the collection $singleRoom$ in class $Hotel$[1] which returns the size of the collection. The derived variable *numberOfSingleRoomAvailable* then can be used in method contracts. For example:

context Hotel:: bookSingleBedRoom(): int
pre: numberOfSingleRoomAvailable > 0

2.1 Identifying Object's State Variables along with Their Abstractions

The state variables set (S_d) for an object of the class under test is obtained by taking a set-union of all the variables (*class* and *derived* variables, excluding the input parameters of the methods) appearing in all the preconditions expressions of the *class contracts*.

The abstract values for a state variable of the object are obtained by processing the *class contracts* for the corresponding class. For each identified state variable, we collect all the individual constraints specified on the state variable in the *class contracts*. This includes all the preconditions, postconditions and the invariant constraints specified on the state variable. Also, we symbolically process all the assignment expressions for the state variable appearing in methods' postconditions to obtain additional (postcondition) constraints on the state variable. We then process these constraints on the state variable and partition its domain into set of mutually exclusive and collectively exhaustive regions by following an approach similar to the one that is used by Kung et al. [15]. For example, the abstract values P_X of a numerical state variable X are computed as given below.

1. Construct an initial set of partitions (intervals) P_X by including all the simple constraints $(C_1, C_2, ..., C_n)$ as obtained above.
2. Select any two constraints C_i and C_j which represent two different intersecting intervals in the domain of X and remove them from P_X.

[1] A class of *HotelManagement* test program that we have used in our experiment presented later in this paper.

3. Form additional intervals $C_i - (C_i \cap C_j)$, $C_j - (C_i \cap C_j)$, and $C_i \cap C_j$.
4. Repeat steps 2 and 3 until no more intersecting partitions remain in P_X.
5. Return P_X.

2.2 The Abstract State Configuration for the Class

The abstract state configuration S of a class is represented by the set of state variables (S_d) along with the domain abstraction defined for each state variable. For an n-state variable configuration, $S_d := \{S1, S2, S3, ..., Sn\}$, where S1 is represented by a set of p-disjoint abstract values, and S2 is represented by a set of q-disjoint abstract values, and so on, the abstract state configuration S for the class is the Cartesian product of the elements of S_d. Formally, it can be represented as

$$\begin{aligned} S \;&:= S1 \times S2 \times .... \times Sn, \\ &:= \{(s_{1i}, s_{2j}, ..., s_{nm}) | s_{1i} \in S1 \wedge s_{2j} \in S2 \wedge ... \wedge s_{nm} \in Sn\}, \text{ where} \\ & S1 := \{s_{11}, s_{12}, ..., s_{1p}\}, \\ & S2 := \{s_{21}, s_{22}, ..., s_{2q}\}, \\ & \vdots \\ & Sn := \{s_{n1}, s_{n2}, ..., s_{nm}\}. \end{aligned}$$

The abstract values of a state variable are mutually exclusive and collectively exhaustive. A specific state k of an object is represented by a specific abstract state configuration $(S_d(k))$, which is an element of set S, i.e.

$$S_d(k) \in S$$

2.3 An Example: Class CoinBox

The class *CoinBox* implements the functionality of a simple vending machine which delivers a drink after receiving two quarters. A customer can withdraw quarters previously inserted by her at any time before the drink is delivered. A specified amount of drink is inserted in the machine when it is empty.

The class specifications in OCL are shown in Figure 1. From the *CoinBox* specifications, and following our approach to select state variables, we get

$$\text{State Variable Set } (S_d) = \{curQtr, allowVend, quantity\}$$

The abstract values for each state variable are obtained from the various constraints specified in the class contracts. The abstraction obtained for each of the three state variables are given as below.

1. $curQtr$ - The set of constraints for this variable is $\{curQtr >= 0$, $curQtr$ = 0, $curQtr == 1$, $curQtr = curQtr@pre + 1\}$. By processing these constraints, we obtain three abstract values for $curQtr$ as $\{= 0, = 1$, and $> 1\}$.
2. $allowVend$ - Being a boolean variable, its abstraction set is $\{TRUE, FALSE\}$.
3. $quantity$ - The set of constraints for this variable is $\{quantity >= 0$, $quantity$ = 0, $quantity$ = $quantity@pre - 1$, $quantity$ = $quantity@pre + m\}$. By processing these constraints symbolically, we obtain two abstract values for $quantity$ as {=0, $> 0\}$.

```
Context CoinBox {
int curQtr, quantity, totalQtrs
boolean allowVend

inv    : int curQtr, quantity, totalQtrs >=0

:: CoinBox( )
post   : self.curQtr  = 0
         self.allowVend = FALSE
         self.quantity =  0
         self.totalQtrs = 0
:: addQtr( ):void // add a quarter in the machine
pre    : self.quantity > 0;
post   : self.curQtr  = curQtr@pre +1
         if (self.curQtr@pre = 1) then
                self.allowVend = TRUE

:: retQtrs( ):void // return quarters back to the user
pre    : self.curQtr > 0;
post   : self.curQtr  = 0
          self.allowVend = FALSE
:: vend( ):void  // deliver a drink
pre    : self.allowVend == TRUE and
         self.quantity > 0;
post   : self.curQtr = 0
         self.allowVend = FALSE
         self.quantity = quantity@pre – 1
         self.totalQtrs = totalQtrs@pre + curQtr@pre
:: addDrink(m: int):void   // add  m unit of drink in
                           // the machine
pre    : self.quantity == 0 and m > 0;
post   : self. quantity = quantity@pre + m
}
```

Fig. 1. Class CoinBox Specifications in OCL

Therefore, the state-space configuration S for the CoinBox example is as follows: $S := \{(s_{1i}, s_{2j}, s_{3k}) | s_{1i} \in curQtr \wedge s_{2j} \in allowVend \wedge s_{nm} \in quantity\}$, where,

$$\begin{aligned} S_d &:= \{curQtr, allowVend, quantity\}, \\ curQtr &:= \{= 0, = 1, > 1\}, \\ allowVend &:= \{TRUE, FALSE\}, \\ quantity &:= \{= 0, > 0\}. \end{aligned}$$

3 Generating Object's Abstract States and Unit Tests

We model the object's state-specific behaviour by a directed graph where a node k represents an abstract state configuration ($S_d(k)$) of the object and an edge (transition) labelled as $m_j(..)$ from abstract state k to another abstract state l represents a valid method invocation for method $m_j(..)$ at state k which changes the state of the object to l. An abstract state is valid (reachable) or not depends on whether a path exists from the initial state of the object (specified by a constructor of the class) to the specified state via a sequence of method invocations. We capture this behaviour in our abstract state generation process where starting at an initial state of the object, we exhaustively search the object state space for all possible method invocations at each of the generated new states.

3.1 Generating Initial State(s)

An initial state configuration $S_d(0)$ is obtained by combining postconditions of a 'constructor' method with the class invariants. Each constraint in the postconditions of the constructor is evaluated to obtain initial abstract values for the corresponding state variables. Remaining state variables are assigned default initial abstract values which are consistent with the class invariants. The generated abstract state corresponds to a specific initial state configuration $S_d(0)$. If an object can be instantiated in multiple ways, as is the case with an overloaded constructor, then for each possible instantiation, an abstract initial state is generated separately.

3.2 Method Invocation in a Given State

A method m_j can be invoked at a given object state k with configuration $S_d(k)$ if its preconditions set $preSet(m_j)$ are 'consistent' with that state. We explore this possibility by comparing the abstract values of the state variables present in the method's preconditions against the current state configuration.

An algorithm $match(S_d(k), preSet(m_j))$ for evaluating a method m_j invocation at a given state configuration $S_d(k)$ is presented in Figure 2. This algorithm takes the current state configuration $(S_d(k))$ and the precondition set $(preSet(m_j))$ for the method m_j, both specified in Disjunctive Normal Form (DNF), and returns the result of the 'match' $(overall - match)$. As stated earlier, a state configuration $S_d(k)$ for a given state k is a conjunction of all the state variables with their abstract values assigned for that state. Each precondition P_i of method m_j $(P_i \in preSet(m_j))$ is matched with the current state configuration $S_d(k)$ to obtain an overall match (outermost loop in the algorithm presented in Figure 2). This is done by evaluating each conjunct C_{ij} of the precondition P_i with the current state configuration $S_d(k)$ separately. This, in turn, requires comparing the two values of a state variable which also appears as a clause variable in conjunct C_{ij}.

```
match ($S_d$(k), preSet($m_j$)) // preSet($m_j$) is the precondition set of method $m_j$

overall-match ← TRUE  // the overall outcome (TRUE or FALSE)
match ← FALSE // to evaluate match for one precondition in preset($m_j$) at a time
for each precondition $P_i \in$ preset($m_j$) && (overall-match == TRUE) do {
    for each conjunct $C_{ij}$ in $P_i$ do { // $P_i$ expressed in DNF and $C_{ij}$ is a conjunct of $P_i$
        result ← TRUE ;        //explore the clauses in $p_i$
        for each clause $c_{ijk}$ in $C_{ij}$ && (result == TRUE) do {
            get the clause variable $X_k$ in clause $c_{ijk}$ ;
            if ($X_k \in S_d$) then { //combine the clause $c_{ijk}$ with state configuration ($S_d$(k))
                bool state_match = $c_{ijk} \wedge S_d(k)$) ;
                if  (state_match == FALSE) then {
                    result ← FALSE ;
                    break;
                }
            }
        }
        if (result)  then match ← TRUE ;
    }
    overall-match ← overall-match $\wedge$ match ;
} return (overall-match)
```

Fig. 2. Algorithm for checking the possibility of a method invocation in a given state

3.3 Obtaining Resulting States

Once an affirmative decision regarding a method m_j invocation in a given state is made, the method's postconditions are combined with the current state to obtain the resulting

```
post-substate-set(postSet(m_j), S_d(k), inputParam(m_j)) // (postSet(m_j) -
Postcondition set of the method m_j and inputParam(m_j) - a set of valid
abstractions for input parameters of method m_j).

for (each post_j ∈ postSet (m_j)) do { // Compute next state abstraction for each state
                                      //variable X
    -  Identify the updated state variable X in post_j
    -  Evaluate the postcondition expression for X according to the data type
       involved
    -  Compute the resulting abstractions of X and Store it in (nextState(X))
                  // nextState(X) – next state abstractions of X
    }
post-substates-set = {nextState(X)   // Initialize post-substates-set with next state
                                     //abstracted values of an updated state variable X
for every other updated state variable Y do {
    -  post-substates-set = post-substates-set × (nextState(Y))
              // '×' stands for the Cartesian-product of the elements of the two set
}
return post-substates-set
```

Fig. 3. Algorithm for obtaining resulting sub-states

states. For this, we determine the changes in the abstract values for the state variables and reflect them in the current state to obtain the resulting states. Each postcondition expression is separately processed to obtain the next state values for the state variables. Identifying the next state abstractions for the state variables of boolean, enumerated, and character (or string) data types is relatively straightforward whereas a numeric variable may require symbolic processing.

Note that a postcondition expression may also contain input parameters of the method. As discussed earlier, we also abstract out method input parameters based on the precondition constraints specified on them. Different valid abstractions on an input parameter (consistent with the precondition constraints) appearing in a postcondition expression of a state variable X may result in additional abstract states for X.

To obtain the resulting states, we compute a set of *sub-states*, each consist of a subset of updated state variables as a result of the method invocation along with their updated abstract values. An algorithm for obtaining resulting sub-states due to method m_j invocation in the current state k is given in Figure 3. The algorithm works in two steps. In the first step, it computes the abstraction information of the updated state variables. This step is undertaken in the first *for-loop* of the algorithm. Subsequently, in the second step, as undertaken in the second *for-loop* of the algorithm presented in Figure 3, it computes a set of changed sub-states by combining the updated abstract values of individual state variables of the sub-states. Finally, we combine the generated 'sub-states' with other state variables (unchanged state variables) to obtain the set of resulting states for the method invocation.

3.4 Example: Class CoinBox Revisited

We demonstrate the generation of abstract states for the CoinBox object. As worked out in the previous section, we identify a state configuration S_d as represented by three state variables and an initial state configuration $S_d(A)$ (assuming A as an initial state) as

$$S_d(A) := (0, FALSE, 0)$$

Only one method $addDrink(m)$ can be called in the current state as for rest of the methods, one or more preconditions are not satisfied. The postcondition of $addDrink$ (m) increment the $quantity$ by a positive amount m and hence the next state value for the $quantity$ is (> 0). This will change the object state to

$$S_d(B) := (0, FALSE, > 0)$$

In state $S_d(B)$, only method $addQtr()$ can be called, which, results in a change of the object state to

$$S_d(C) := (1, FALSE, > 0)$$

At $S_d(C)$, two methods, $addQtr()$ and $retQtrs()$ may be invoked, resulting the next state abstractions as $(2, TRUE, > 0)$ and $(0, FALSE, > 0)$, respectively. (2, $TRUE, > 0)$ is a new state and we mark it as

$$S_d(D) := (2, TRUE, > 0)$$

whereas $(0, FALSE, > 0)$ is the same as $S_d(B)$.

Three methods $addQtr()$, $retQtrs()$ and $vend()$ may be invoked in the newly generated state $S_d(D)$. The invocation of methods $addQtr()$ causes a transition from $S_d(D)$ to itself (a loop). The invocation of method $retQtrs()$ leads to state $S_d(B)$. The abstract value for $quantity$ in state $S_d(D)$ is (> 0). Note that the method $vend()$'s postconditions include $\{quantity$ = $quantity@pre - 1\}$. As described in Section 3.3, the invocation of $vend()$ leads to two different states $S_d(A)$ and $S_d(B)$, corresponding to the two transitions with conditions $[quantity = 1]$ and $[quantity > 1]$, respectively. The complete state graph for the CoinBox class is shown in Figure 4.

3.5 Generating Unit Tests for the Class

Testing a sequence of method invocations includes instantiating the class by means of calling a constructor of that class, followed by a sequence of method invocations to put the object in the desired state, followed by invoking a method of interest, and finally, assessing the resulting state against the expected values (i.e. the test oracles). Generating abstract states incrementally results in formation of a directed graph where a node represents an abstract state and an edge represents a method invocation causing the state transition. A test sequence then be a sequence of method invocations along a *simple* [2] path in the graph. The test sequences so generated are said to follow the Transition-Tree [3] coverage criterion on a state model. The checking of abstract states before and after each method invocation in a test sequence forms the test oracles for a unit test of the class. The test oracles are further strengthened by augmenting them with the method's postconditions.

[2] A *simple* path is a path in the graph from start node to a node that has already appeared in that path.

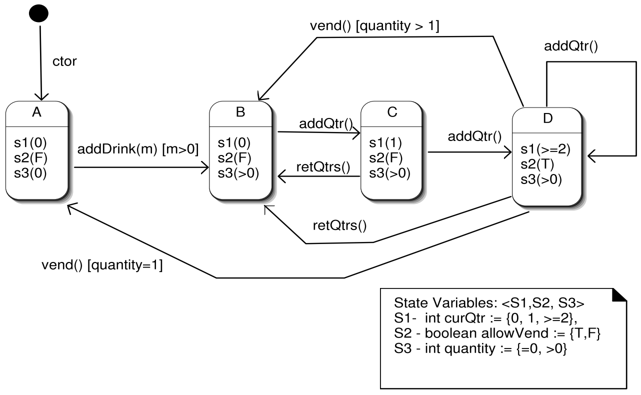


Fig. 4. Generated abstract state model for the CoinBox class

4 Experiment Evaluation

We study the effectiveness of the generated unit tests by testing the classes of three object-oriented programs using mutation analysis. The program units were coded in Java. The class contracts were specified in OCL by the author. For each program, we incrementally generated abstract states for each class exhibiting state-specific behaviour and constructed unit tests as described above.

4.1 Test Programs

A brief summary of the test-programs are given below. Table 1 shows relevant statistics for the three test programs.

Program-1: *LinkedList*
The *LinkedList* class is a class included in Java collection (in our case Java SDK version 1.4.2). For the sake of this experiment, we simplified the code to remove super class dependencies. It consists of 19 methods which include two getter-methods, *size()* and *contains(Object)*. Some methods use the *Collection* interface, into which we deliberately did not seed any faults for the sake of clarity.

Program-2: *HotelManagement*
This program manages a simple room booking system for a hotel providing accommodation for single, double, and function-rooms. The main class ‘Hotel’ facilitates the bookings for the customers into available rooms and ensures that a room is made available for further bookings as soon as it is vacated. It also tracks the movement of presentation equipment between function rooms. The program, as implemented, does not deal with the dates, which means that bookings and other activities are purely run time events.

Program-3: *CruiseControl*
CruiseControl is widely used for state based testing experiments. This program consists of classes mainly composed of small methods exhibiting state-specific behaviour. The invocations of most of the methods are purely governed by the object's states. This program is a graphical simulation of a cruise control system for a vehicle. The graphical interface facilitates turning the engine on and off. When the engine is on, one can accelerate, brake, or put the vehicle in the 'cruise' mode. Specifically, the program uses two threads of control - one that provides manual control of the vehicle, and the other that provides cruise control support for the vehicle.

4.2 Mutation Operators and Faults Seeded

To enable us to measure effectiveness we created multiple mutants of the three programs by manually seeding faults in a systematic way, using a set of applicable mutation operators [14]. The faults were randomly and uniformly distributed in the code under test. The types of the faults and other statistics for the three programs are given in Figure 5 (a), (c), and (e), respectively. The mutation operators used and the kinds of faults inserted by them in this study were:

- Literal Change Operator (LCO) - changing increment to decrement or vice versa, incorrect or missing increment, incorrect or missing state assignment.
- Language Operator Replacement (LOR) - replacing a relational or logical operator by another one of similar type.
- Control Flow Disruption (CFD) - missing or incorrectly placed block-markers, break, continue, or return.
- Statement Swap Operator (SSO) - swapping two statements of the same scope.
- Argument Order Interchange (AOI) - interchanging arguments of similar types in a method-definition or in a method-call.
- Variable Replacement Operator (VRO) - replacing a variable with another of similar type
- Missing Condition Operator (MCO) - missing-out a condition in a composite conditional statement.
- Null Reference Operator (NRO) - causing a null reference.
- Incorrect Initialization Operator (IIO) - Incorrect or missing initialization.

4.3 Generating Unit Tests and Performing Unit Testing

For each program, we generated test sequences by applying the *Transition-Tree* coverage criterion to the object state graph. We applied two strategies for generating unit tests from the object state graph. In the first strategy, we computed the test inputs for the test sequences obtained by applying TT criterion to the state graph. The generated unit tests were coded as JUnit [13] test scripts and subsequently exercised on the set of the test-program's mutants to obtain killed mutants information.

In the next strategy, referred as*Modified Transition-Tree* coverage criterion, we strengthened the TT tests as obtained above by including additional tests for the invalid inputs (those violating the preconditions of corresponding method) and repeated the unit testing.

4.4 Results

For each program, we first highlight the effectiveness of the unit tests obtained using the TT coverage criterion. We then show the improved effectiveness of the unit tests obtained using the $Modified\ TT$ coverage criterion, where the unit tests generated earlier were combined with invalid test inputs based on input data abstraction. We also present code coverage achieved by TT and $Modified\ TT$ tests for each test-program. Subsequently, we address the limitations of the proposed approach from the insights gained by observing the kinds of faults not readily identified.

Program unit #1: LinkedList The analysis of the specification revealed only one state variable ‘size’ with two abstract values, $= 0$, and > 0. We found that unit tests obtained using the TT criterion revealed 82.34% of the faults seeded. When we used $Modified\ TT$ criterion for test generation, the fault detection effectiveness of the unit testing was increased to 90%. The fault data is shown in Figure 5 (a), where each bar in the graph (from bottom to top) demonstrate total number of mutants used of a given type, killed mutants in TT, and additionally killed in $ModifiedTT$ coverage criteria, respectively.

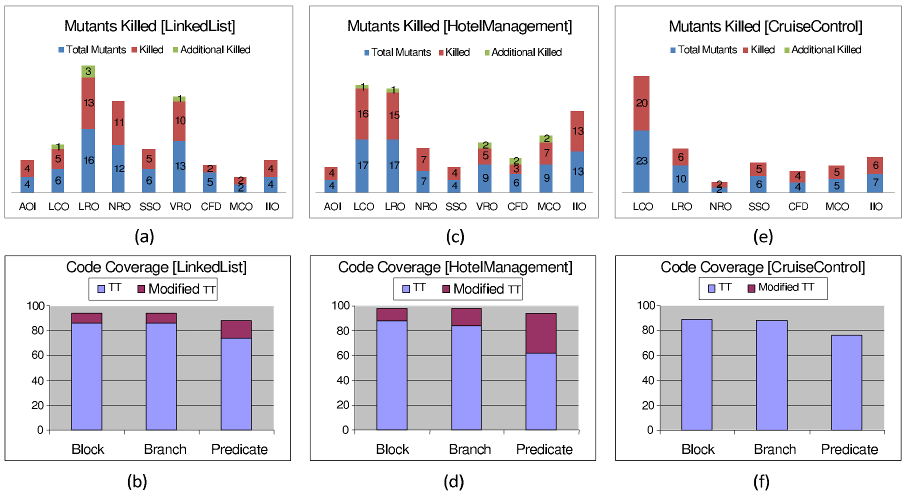


Fig. 5. Faults and Coverage statistics for the three test programs

Figure 5 (b) shows the test code coverage for the TT tests for three code coverage criteria, namely, block, branch, and predicate coverage obtained by using $JavaCode\ Coverage$ [16] tool. Note the increase in code coverage for all the three coverage criteria when we have added more tests according to $Modified\ TT$ criterion.

We tried to gain insight into reasons behind certain faults which were not revealed at all. We found that by inserting an object in the ‘empty’ list, its state changed to ‘non-empty’. There are two methods of the *LinkedList* class, namely, *firstIndexOf(Object o)* and *lastIndexOf(Object o)* which return the index of the object as (=0), as the list

Table 1. Subject Programs Test-statistics

S. No.	Program Name (NCLOC/mutants)	Class Name	# of methods	# of states	# of TT tests	# Faults identified	# of modified TT tests	# of Faults identified
1	LinkedList (380/68)	LinkedList	19	2	184	56	196	61
2	HotelManagement (390/86)	Room	4	2	10	6	12	8
		FunctionRoom	4	2	10	6	11	7
		Hotel	11	27	135	62	154	67
3	CruiseControl (315/57)	CarSimulator	4	6	19	10	19	10
		Controller	8	4	25	26	25	26
		CruiseController	4	2	6	4	6	4
		SpeedControl	4	3	10	8	10	8

contained only one element (at index 0). As both the methods use a *for* loop to search for the desired element in the list, they got iterated for the first time and found the desired object. Two VRO faults and one SSO fault inserted in these methods could only be detected if the list would have an element appearing more than once. This would have differentiated their places in the lists and hence detected the faults. Both the methods had one CFD fault seeded which could only be revealed if the searched element had occupied any other place than the first (in *firstIndexOf(Object o)*) or the last (in *lastIndexOf(Object o)*) place in the list. Another CFD fault in the *remove(Object o)* method remained unidentified for the same reason. Hence, three of the CFD faults seeded in these methods were not revealed.

Program unit #2: HotelManagement. This program unit consists of six classes, three of which exhibit state-specific behaviour. Two of the three classes are in the 'isa' relationship (i.e. one class extends the other) and an 'array' of object-references of the extended class are included in the third class. So we sequentially tested the three classes - first the base class followed by the inherited class and finally the 'container' class.

The tests obtained from TT coverage criteria revealed 86% of the seeded faults. Using $Modified\ TT$ criterion, the fault detection increased to 95%. Figure 5 (d) shows the test code coverage for the TT tests (and modified TT tests) for three code coverage criteria. For the un-revealed faults, we found that all such faults were seeded in one method of the 'Hotel' class, namely, *movePresentationEquipment(Room r)*. As relevant information of this method could not be captured in the state behaviour, some of the seeded faults remained unidentified. Further investigation revealed that stronger contracts for this method could have revealed these faults.

Program unit #3: CruiseControl. In this case, most of the faults were identified by the unit tests obtained using TT coverage criteria. This was due to the fact that the methods of the *CruiseControl* classes were small, singly minded, and without arguments. Therefore, the TT tests were adequate enough to identify most of the errors. However, due to the non-deterministic behaviour of the two threads of control, some regions of the test-program were not reached deterministically by the generated unit tests, and therefore faults present in these regions were missed. Figure 5 (f) shows the test code coverage for the TT tests for three code coverage criteria.

5 Discussions

A general perception is that class specifications are seldom complete and there may be inconsistencies which may limit the correct interpretation of the behaviour of corresponding objects. Chow [5] has demonstrated that one can use finite state models to identify the discrepancies in the specifications. It has been shown that incomplete and/or inconsistent specifications can be synthesized in a step-wise refinement approach leading to more complete and consistent set of class specifications [9].

State based testing may not completely test a given class and similar situations have also been observed in many investigations of state based testing by Briand et al. [1,4,18] and Offutt et al. [21]. They advocated for combining state based testing with other testing approaches like structural [18] and functional testing [1] for improving the effectiveness of unit testing.

6 Related Work

Specifying suitable operational contracts in class specifications is a popular approach [10,17,7,11] for program development which deals with behavioural ambiguities and uncertainties. Operational contracts consist of a set of assertions that must hold immediately before and after the termination of any call to a specified operation. A number of such formalisms support operational contracts which include AsmL [2], Object-Z [19], JML [12], and OCL [20].

Finite state machines represent important form of behaviour modeling and a number of studies generated these models from specifications [5,11]. However, there is a significant problem of state explosion, which inherently limit the use of these FSMs. Many studies tackled the state explosion by restricting the size of the generated FSM [11]. Other difficulties reported for FSM to be used were handling of pointer references, arrays, and non-determinism [15,3].

The problem of state explosion can be tackled by abstraction [24,7,8,22]. Turner et al. [24] combine a group of data values as ‘general substate values’ that are all considered in the same manner. Gao et al. [7] use pre and postcondition expressions to identify the data states for simple data types. Grieskamp et al. [8] use similar consideration for generating finite state machines from by executing the AsmL Specifications, but the generated FSM may be an under-approximation of the true FSM due to missing links between reachable abstract states. Their analysis included methods without any arguments. Paradkar [22] also proposed an approach for generating a FSM based on the abstract state notion where the author has used an informal abstraction procedure for the state generation. More recently, Ciupa et al. [6] describe a tool AutoTest-supported random testing of classes where method contracts are used as test oracles. Our approach generates valid and invalid sequences of method invocations that covers an object state space systematically and test results are compared, besides post conditions, with the object abstract state at that point of time to ensure more comprehensive testing, though a through analysis is required to obtained dependable results.

In our approach, we mechanically compute the abstraction information from the *class contracts*. Our approach to generate object state graph is more comprehensible as it considers methods with input parameters that include primitive and non-primitive data types. The proposed approach has the potential for end-to-end automation of class testing as the test inputs for the test sequences can easily be obtained during the generation of the abstract states of an object. This can greatly improve the efficiency of the testing process.

7 Summary

In this paper, we have proposed an approach to perform unit testing of object-oriented programs from *class contracts*. An object behaviour is modeled as a state graph having abstract states. The state variables for the object and their abstractions are determined from the *class contracts*. The abstract state generation process is then used to obtain unit tests for the class.

We demonstrate that the proposed approach is both practically applicable and effective by performing an experiment which tests the classes of three object-oriented programs using mutation analysis. Our results suggested that most of the seeded faults can be identified by the unit tests so obtained. The approach systematically models the object state-specific behaviour and exhaustively explores its state space to ensure object's correct behaviour in terms of valid and invalid method invocations in reachable states. The approach has considerable potential for automation, which suggests for the proposed approach to be a viable option for unit testing object-oriented programs.

References

1. Antoniol, G., Briand, L.C., Penta, M.D., Labiche, Y.: A case study using the round-trip strategy for state-based class testing. In: ISSRE 2002: Proceedings of the 13th International Symposium on Software Reliability Engineering, Washington, DC, USA, pp. 269–279. IEEE Computer Society, Los Alamitos (2002)
2. AsmL Home Page, http://research.microsoft.com/en-us/projects/asml/ (accessed March 2010)
3. Binder, R.V.: Testing Object-Oriented Systems Models, Patterns, and Tools. Addison-Wesley, Reading (1999)
4. Briand, L.C., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing: A series of experiments. IEEE Trans. Softw. Eng. 30(11), 770–793 (2004)
5. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. 4(3), 178–187 (1978)
6. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: ISSTA 2007: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 84–94. ACM Press, New York (2007)
7. Gao, J.Z., Kung, D.C., Hsia, P., Toyoshima, Y., Chen, C.: Object state testing for object-oriented programs. In: COMPSAC 1995: Proceedings of the 19th International Computer Software and Applications Conference, Washington, DC, USA, pp. 232–238. IEEE Computer Society, Los Alamitos (1995)
8. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 112–122. ACM Press, New York (2002)

9. Gupta, A., Raj, A.: Strengthening method contracts for objects. In: APSEC 2006: Proceedings of the XIII Asia Pacific Software Engineering Conference, Washington, DC, USA, pp. 233–242. IEEE Computer Society, Los Alamitos (2006)
10. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In: OOPSLA/ECOOP 1990: Proceedings of the European Conference on Object-Oriented Programming/Object-Oriented Programming Systems, Languages, and Applications, pp. 169–180. ACM Press, New York (1990)
11. Hoffman, D.M., Strooper, P.A.: ClassBench: A framework for automated class testing. Software Practice and Experience 27(5), 573–597 (1997)
12. JML Homepage, `http://www.eecs.ucf.edu/~leavens/JML/` (accessed March 2010)
13. JUnit Home Page, `http://www.junit.org` (accessed March 2010)
14. Kim, S., Clark, J.A., McDermid, J.A.: The rigorous generation of Java mutation operators using HAZOP. In: Proceedings of ICSSEA, pp. 9–10 (1999)
15. Kung, D., Lu, Y., Venugopalan, N., Hsia, P., Toyoshima, Y., Chen, C., Gao, J.: Object state testing and fault analysis for reliable software systems. In: ISSRE 1996: Proceedings of the The Seventh International Symposium on Software Reliability Engineering, Washington, DC, USA, p. 76. IEEE Computer Society, Los Alamitos (1996)
16. Lingampally, R., Gupta, A., Jalote, P.: A multipurpose code coverage tool for Java. In: HICSS 2007: Proceedings of 40th Annual Hawaii International Conference on System Sciences; Minitrack on Automated Software Testing and Analysis: Techniques, Practices and Tools, pp. 261–270. IEEE Computer Society, Los Alamitos (2007)
17. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
18. Mouchawrab, S., Briand, L.C., Labiche, Y.: Assessing, comparing, and combining statechart-based testing and structural testing: An experiment. In: ESEM 2007: Proceedings of the 1st Symposium on Empirical Software Engineering & Measurements, Washington, DC, USA, pp. 41–50. IEEE Computer Society, Los Alamitos (2007)
19. Object-Z Home Page, `http://www.itee.uq.edu.au/~smith/objectz.html` (accessed March 2010)
20. OCL 2.0 Specifications, `http://www.omg.org/docs/ptc/05-06-06.pdf` (accessed March 2010)
21. Offutt, A.J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Jour. Soft. Test., Veri. and Rel. 13(1), 25–53 (2003)
22. Paradkar, A.: Towards model-based generation of self-priming and self-checking conformance tests for interactive systems. In: SAC 2003: Proceedings of the 2003 ACM Symposium on Applied Computing, pp. 1110–1117. ACM Press, New York (2003)
23. Smith, M.D., Robson, D.J.: Object-oriented programming-the problems of validation. In: ICSM 1990: Proceedings of the Seventh International Conference on Software Maintenance, pp. 272–281 (1990)
24. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: ICSM 1993: Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, pp. 302–310. IEEE Computer Society, Los Alamitos (1993)

Efficient On-the-Fly Emptiness Check for Timed Büchi Automata

Frédéric Herbreteau and B. Srivathsan

LaBRI (Université de Bordeaux - CNRS)

Abstract. The Büchi non-emptiness problem for timed automata concerns deciding if a given automaton has an infinite non-Zeno run satisfying the Büchi accepting condition. The solution to this problem amounts to searching for a cycle in the so-called zone graph of the automaton. Since non-Zenoness cannot be verified directly from the zone graph, additional constructions are required. In this paper, it is shown that in many cases, non-Zenoness can be ascertained without extra constructions. An on-the-fly algorithm for the non-emptiness problem, using an efficient non-Zenoness construction only when required, is proposed. Experiments carried out with a prototype implementation of the algorithm are reported and the results are seen to be promising.

1 Introduction

Timed automata [1] are finite automata extended with clocks. The clock values can be compared with constants and reset to zero on the transitions, while they evolve continuously in the states. The emptiness problem for timed automata is decidable. Consequently, timed automata are used in tools like Uppaal [3] and Kronos [8] for model-checking reachability properties of real-time systems. They have also been successfully used for industrial case studies, e.g. [4].

For the verification of liveness properties, timed automata with Büchi conditions have been considered. The emptiness problem demands if there exists a non-Zeno run that visits an accepting state infinitely often. A run is non-Zeno if the total time elapsed during the run is unbounded. This further requirement makes the problem harder than classical Büchi emptiness.

The solution proposed in [1] relies on a symbolic semantics called the region graph. The emptiness problem for timed Büchi automata is reduced to classical Büchi emptiness on the region graph with an additional time progress requirement. However, the region graph construction is very inefficient for model checking real life systems. Zones are used instead of regions in the aforementioned tools. The zone graph is another symbolic semantics that is coarser than the region graph. Although it is precise enough to preserve reachability properties, it is too abstract to directly infer time progress and hence non-Zenoness.

Two approaches have been proposed in order to ascertain time progress. In [16], the automaton is transformed into a so-called strongly non-Zeno automaton, which has the property that all the runs visiting an accepting state infinitely often are non-Zeno. Hence, non-Zenoness is reduced to Büchi emptiness. However it has been proved in [12] that this construction sometimes results in an exponential blowup.

A careful look at the conditions for Zenoness yields another solution to verify time progress. Two situations can prevent time from diverging on a run. Either the value of a clock is bounded and not reset anymore, or some clock is reset and later checked to zero preventing time elapse in-between. In [12] a guessing zone graph has been introduced to identify clear nodes where time can elapse despite zero-checks. Hence timed Büchi automata emptiness reduces to Büchi emptiness on the guessing zone graph with two infinitary conditions namely clear nodes and accepting nodes. The guessing zone graph is only $|X|+1$ times bigger than the zone graph, where $|X|$ stands for the number of clocks. The straightforward algorithm handling both the above mentioned situations explores the guessing zone graph $\mathcal{O}(|X|)$ number of times.

On a parallel note, the emptiness problem for (untimed) Büchi automata has been extensively studied (see [13] for a survey). The best known algorithms in the case of multiple Büchi conditions are inspired by the Tarjan's SCC algorithm. An on-the-fly algorithm which computes the graph during the search, and terminates as soon as a satisfactory SCC is found, is crucial in the timed settings as the zone graph is often huge. It is known that the Couvreur's algorithm [9] outperforms all on-the-fly SCC-based algorithms.

In this paper, we present an on-the-fly algorithm that solves the emptiness problem for timed Büchi automata. We noticed from several experiments that Büchi emptiness can sometimes be decided directly from the zone graph, i.e. without using any extra construction. This is the way we take here. The Couvreur's algorithm is employed to search for an accepting SCC directly on the zone graph. Then, we face two challenges.

The first challenge is to detect if this SCC has *blocking* clocks, that is clocks that are bounded from above but are not reset in the transitions of the SCC. The SCCs with blocking clocks have to be re-explored by discarding transitions bounding these clocks. Notice that discarding edges may split the SCC into smaller SCCs. The difficult task is to discover the smaller SCCs and explore them *on-the-fly*.

The second challenge is to handle SCCs with *zero-checks*, that is clocks that are tested for value being zero. In this case, the zone graph does not contain enough information to detect time progress. This is where the guessing zone graph construction is required. The involving part of the problem is to introduce this construction on-the-fly. In particular the part of the guessing zone graph that is explored should be restricted only to the transitions from the SCC containing zero-checks.

We propose an algorithm that fulfills both challenges. In the worst case, it runs in time $\mathcal{O}(|ZG|.(|X|+1)^2)$. When the automaton does not have zero checks it runs in time $\mathcal{O}(|ZG|.(|X|+1))$. When the automaton further has no blocking clocks, it runs in time $\mathcal{O}(|ZG|)$. Our algorithm further incorporates an improvement that proves powerful in practice. Indeed, non-Zenoness could be established from situations where the value of some clock is known to increase by 1 infinitely often. Even in the presence of zero-checks, this property avoids exploring the guessing zone graph. Again, this improvement is applied on-the-fly.

Next, we show that all these operations on SCCs can be handled efficiently as the memory needed by our algorithm is comparable to the memory usage of the strongly non-Zeno approach in [16].

Finally, we also report some successful experiments of our algorithm on examples from the literature. All the experiments that we have conducted show that non-Zenoness constructions need not be applied mechanically, hence validating our approach.

Related work. Timed automata with Büchi conditions have been introduced in the seminal paper [1]. The zone approach to model-checking has been introduced in [5] and later used in the context of timed automata [11]. The Büchi emptiness problem using zones has been studied in [6,16,15]. The idea of the strongly non-Zeno approach first appears in [14]. The strongly non-Zeno construction has been implemented in the tool Profounder [16]. Finally, the recent paper [12] introduces the guessing zone graph construction.

Outline. The paper is organized as follows. In Section 2 we define the emptiness problem. Then we present the two approaches that ensure time progress in Section 3. We detail our algorithm and its optimizations in Section 4. Finally, we discuss experiments and future work in Section 5.

2 The Emptiness Problem for Timed Büchi Automata

2.1 Timed Büchi Automata

Let X be a set of clocks, i.e., variables that range over $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. *Clock constraints* are conjunctions of comparisons of variables with integer constants, e.g. $(x \leq 3 \wedge y > 0)$. Let $\Phi(X)$ denote the set of clock constraints over clock variables X.

A *clock valuation* over X is a function $\nu : X \to \mathbb{R}_{\geq 0}$. We denote $\mathbb{R}_{\geq 0}^X$ the set of clock valuations over X, and $\mathbf{0} : X \to \{0\}$ the valuation that associates 0 to every clock in X. We write $\nu \models \phi$ when ν satisfies ϕ, i.e. when every constraint in ϕ holds after replacing every x by $\nu(x)$.

For a valuation ν and $\delta \in \mathbb{R}_{\geq 0}$, let $(\nu + \delta)$ be the valuation such that $(\nu + \delta)(x) = \nu(x) + \delta$ for all $x \in X$. For a set $R \subseteq X$, let $[R]\nu$ be the valuation such that $([R]\nu)(x) = 0$ if $x \in R$ and $([R]\nu)(x) = \nu(x)$ otherwise.

A *Timed Büchi Automaton (TBA)* is a tuple $\mathcal{A} = (Q, q_0, X, T, Acc)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, X is a finite set of clocks, $Acc \subseteq Q$ is a set of accepting states, and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions (q, g, R, q') where g is a *guard*, and R is a *reset* of the transition. Examples of TBA are depicted in Figure 1.

A *configuration* of $\mathcal{A}$ is a pair $(q, \nu) \in Q \times \mathbb{R}_{\geq 0}^X$; with $(q_0, \mathbf{0})$ being the *initial configuration.* A *discrete transition* between configurations $(q, \nu) \xrightarrow{t} (q', \nu')$ for

$t = (q, g, R, q')$ is defined when $\nu \vDash g$ and $\nu' = [R]\nu$. We also have *delay transitions* between configurations: $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta)$ for every q, ν and $\delta \in \mathbb{R}_{\geq 0}$. We write $(q, \nu) \xrightarrow{\delta,t} (q', \nu')$ if $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta) \xrightarrow{t} (q', \nu')$.

A *run* of $\mathcal{A}$ is a finite or infinite sequence of configurations connected by $\xrightarrow{\delta,t}$ transitions, starting from the initial state q_0 and the initial valuation $\nu_0 = \mathbf{0}$:

$$(q_0, \nu_0) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots$$

A run σ *satisfies the Büchi condition* if it visits *accepting configurations* infinitely often, that is configurations with a state from Acc. A *duration* of the run is the accumulated delay: $\sum_{i \geq 0} \delta_i$. A run σ is *Zeno* if its duration is bounded.

Definition 1. *The* Büchi non-emptiness problem *is to decide if $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition.*

The class of TBA we consider is usually known as diagonal-free TBA since clock comparisons like $x - y \leq 1$ are disallowed. Since we are interested in the Büchi non-emptiness problem, we can consider automata without an input alphabet and without invariants since they can be simulated by guards.

The Büchi non-emptiness problem is known to be PSPACE-complete [1].

2.2 The Zone Graph

A zone is a set of valuations defined by a conjunction of two kinds of constraints: comparison of the difference between two clocks with a constant, or comparison of the value of a single clock with a constant. For instance $(x - y \geq 1) \wedge (y < 2)$ is a zone.

The transition relation on valuations can be transposed to zones. Let $\overrightarrow{Z}$ be a zone denoting the result of time elapse from Z, that is, the set of all valuations ν' such that $\nu' = \nu + \delta$ for some $\nu \in Z$ and $\delta \in \mathbb{R}_{\geq 0}$. We have $(q, Z) \xrightarrow{t} (q', Z')$ if Z' is the set of valuations ν' such that $(q, \nu) \xrightarrow{t} (q', \nu')$ for some $\nu \in \overrightarrow{Z}$. It can be checked that Z' is a zone. Difference Bound Matrices (DBMs) can be used for the representation of zones [11]. Transitions are computed efficiently for zones represented by DBMs.

However, the number of reachable symbolic configurations (q, Z) may not be finite [10]. Hence tools like Kronos or Uppaal use an approximation operator *Approx* that reduces the set of zones to a finite set. The *zone graph* of $\mathcal{A}$, denoted $ZG(\mathcal{A})$, has nodes of the form $(q, Approx(Z))$. Observe that $ZG(\mathcal{A})$ only has finitely many nodes. The initial node is (q_0, Z_0) where q_0 is the initial state of $\mathcal{A}$ and Z_0 is the zone where all the clocks are equal to zero. The transitions of the zone graph are $(q, Z) \xrightarrow{t} (q', Approx(Z'))$ instead of $(q, Z) \xrightarrow{t} (q', Z')$.

It remains to define *Approx* in such a way that verification results for $ZG(\mathcal{A})$ entail similar results for $\mathcal{A}$. Let M be the maximal constant in $\mathcal{A}$. M defines the precision of the approximation. We define the *region equivalence* over valuations as $\nu \sim_M \nu'$ iff for every $x, y \in X$:

1. $\nu(x) > M$ iff $\nu'(x) > M$;
2. if $\nu(x) \leq M$, then $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$;
3. if $\nu(x) \leq M$, then $\{\nu(x)\} = 0$ iff $\{\nu'(x)\} = 0$;
4. for every integer $c \in (-M, M)$: $\nu(x) - \nu(y) \leq c$ iff $\nu'(x) - \nu'(y) \leq c$.

The first three conditions ensure that ν and ν' satisfy the same guards. The last one enforces that for every $\delta \in \mathbb{R}_{\geq 0}$ there is $\delta' \in \mathbb{R}_{\geq 0}$, such that valuations $\nu + \delta$ and $\nu' + \delta'$ satisfy the same guards. This definition of region equivalence is introduced in [7]. Notice that it is finer than the equivalence defined in [1] thanks to the last condition which is needed for correctness of Theorem 1 below.

An equivalence class of $\sim_M$ is called a *region*. For a zone Z, we define *Approx* (Z) as the smallest union of regions that is convex and contains Z. Notice that there are finitely many regions, hence finitely many approximated zones.

We call a *path* in the zone graph a finite or infinite sequence of transitions:

$$(q_0, Z_0) \xrightarrow{t_0} (q_1, Z_1) \xrightarrow{t_1} \cdots$$

A path is *initialized* when it starts from the initial node (q_0, Z_0). A run of $\mathcal{A}$ $(q_0, \nu_0) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots$ is an *instance* of a path if $\nu_i \in Z_i$ for every $i \geq 0$. The path is called an *abstraction* of the run.

Approx preserves many verification properties, and among them Büchi accepting paths as stated in the following theorem.

Theorem 1 ([15]). *Every path in $ZG(\mathcal{A})$ is an abstraction of a run of $\mathcal{A}$, and conversely, every run of $\mathcal{A}$ is an instance of a path in $ZG(\mathcal{A})$.*

As a consequence, every run of $\mathcal{A}$ that satisfies the accepting conditions yields a path in $ZG(\mathcal{A})$ that also satisfies the accepting conditions, and conversely. However, it is not possible to determine from a path in $ZG(\mathcal{A})$ if it can be instantiated to a non-Zeno run of $\mathcal{A}$. Hence, we need to preserve more information either in the automaton or in the zone graph.

3 Solutions to the Zenoness Problem

Theorem 1 is a first step towards an algorithm for the Büchi emptiness problem working from the zone graph. It remains to ensure that a given path in $ZG(\mathcal{A})$ has a non-Zeno instance in $\mathcal{A}$. Observe that the zone graph is not pre-stable: a transition $(q, Z) \xrightarrow{t} (q', Z')$ may not have an instance for a valuation $\nu \in Z$. Hence zones are not precise enough to detect time progress.

For example, the state 3 of $\mathcal{A}_1$ in Figure 1 is reachable with zone $Z = (0 \leq x \wedge x \leq y)$ on path from 0 via 2. At first sight, it seems that time can elapse in configuration $(3, Z)$. This is not the case as the transition from 3 to 0 forces x to be equal to 0. Hence, Z has to be separated in two zones: $x = 0 \wedge x \leq y$ that can take the transition from 3 to 0, and $x > 0 \wedge x \leq y$ that is a deadlock node.

We present two approaches from the literature to overcome this problem. Both are used in our algorithm in section 4. They rely on the properties of the

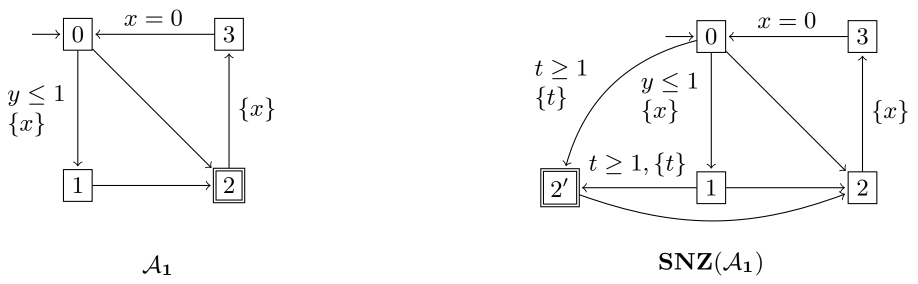


Fig. 1. A Timed Automaton and a Strongly non-Zeno Timed Automaton

transitions $(q, Z) \xrightarrow{g;R} (q', Z')$ in $ZG(\mathcal{A})$. We say that a transition *bounds x from above* if there exists $c \geq 1$ such that $\overrightarrow{Z} \wedge g \Rightarrow (x \leq c)$. Observe that this is equivalent to $\nu''(x) \leq c$ for every valuation ν'' such that both $(q, \nu) \xrightarrow{\delta} (q, \nu'')$ and $\nu'' \models g$. Similarly a transition *bounds x from below* when $\overrightarrow{Z} \wedge g \Rightarrow (x \geq c)$; it *zero checks x* when $\overrightarrow{Z} \wedge g \Rightarrow (x = 0)$. Finally, a transition *resets x* when x belongs to R. Notice that all these properties are easily checked using DBMs for zones and guards.

3.1 Adding One Extra Clock

A common solution to deal with Zeno runs is to transform a TBA into a *strongly non-Zeno automaton*, i.e. such that all runs satisfying the Büchi condition are guaranteed to be non-Zeno.

The construction in [16,2] adds one clock t and duplicates the accepting states in order to ensure that at least one unit of time has passed between two visits to an accepting state. Figure 1 presents $\mathcal{A}_1$ along with its strongly non-Zeno counterpart $SNZ(\mathcal{A}_1)$. The correctness of the approach comes from Lemma 1 below, the proof of which is omitted.

Lemma 1. *If a path in $ZG(\mathcal{A})$ visits infinitely often both a transition that bounds some clock x from below and a transition that resets the same clock x, then all its instances are non-Zeno.*

The converse also holds: if $\mathcal{A}$ has a non-Zeno accepting run, then 1 time unit elapses infinitely often. Hence, the guard $t \geq 1$ is satisfied infinitely often and there exists an accepting run of $SNZ(\mathcal{A})$. By Theorem 1 one can find an accepting path in $ZG(SNZ(\mathcal{A}))$.

Despite being simple, this construction may lead to an exponential blowup. It is shown in [12] that there exists some TBA $\mathcal{A}$ such that the zone graph of $SNZ(\mathcal{A})$ has size $|ZG(\mathcal{A})|.2^{\mathcal{O}(|X|)}$. Our algorithm described in section 4 will use a similar idea as an optimization. However, no new clocks are added and no blowup is inflicted.

3.2 The Guessing Zone Graph

Another solution was introduced in [12] in order to avoid the exponential blowup. Consider a path in $ZG(\mathcal{A})$ that only has Zeno instances. Two situations may occur. Either the path has infinitely many transitions that bound some clock x from above, but only finitely many transitions that reset x. Then, the total time elapsed is bounded. Or, time cannot elapse at all because of zero-checks, as this is the case in state 3 of $\mathcal{A}_1$ in Figure 1.

Zero-checks are handled thanks to a modification of the zone graph. The nodes will now be triples (q, Z, Y) where $Y \subseteq X$ is the set of clocks that can potentially be checked to 0. It means in particular that other clock variables, i.e. those from $X - Y$ are assumed to be bigger than 0. We write $(X - Y) > 0$ for the constraint saying that all the variables in $X - Y$ are not 0.

Definition 2. *Let $\mathcal{A}$ be a TBA over a set of clocks X. The* guessing zone graph *$GZG(\mathcal{A})$ has nodes of the form (q, Z, Y) where (q, Z) is a node in $ZG(\mathcal{A})$ and $Y \subseteq X$. The initial node is (q_0, Z_0, X), with (q_0, Z_0) the initial node of $ZG(\mathcal{A})$. There is a transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ in $GZG(\mathcal{A})$ if there is a transition $(q, Z) \xrightarrow{t} (q', Z')$ in $ZG(\mathcal{A})$ with $t = (q, g, R, q')$, and there are valuations $\nu \in Z$, $\nu' \in Z'$, and δ such that $\nu + \delta \vDash (X - Y) > 0$ and $(q, \nu) \xrightarrow{\delta, t} (q, \nu')$. We also introduce a new auxiliary letter τ, and put transitions $(q, Z, Y) \xrightarrow{\tau} (q, Z, Y')$ for $Y' = \emptyset$ or $Y' = Y$.*

Observe that the definition of transitions reflects the intuition about Y we have described above. Indeed, the additional requirement on the transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ is that it should be realizable when the clocks outside Y are strictly positive; so there should be a valuation satisfying $(X - Y) > 0$ that realizes this transition. This construction entails that from a node $(q, Z, \emptyset)$ every reachable zero-check is preceded by the reset of the variable that is checked, and hence nothing prevents a time elapse in this node. We call such a node *clear*. We call a node (q, Z, Y) *accepting* if it contains an accepting state q. Figure 4 depicts the part of $GZG(\mathcal{A}_1)$ reachable from node $(3, x == 0, \{x, y\})$ with state 1 and all its transitions removed from $\mathcal{A}_1$.

Notice that directly from Definition 2 it follows that a path in $GZG(\mathcal{A})$ determines a path in $ZG(\mathcal{A})$ obtained by removing τ transitions and the third component from nodes.

We say that a path is *blocked* if there is a variable that is bounded from above by infinitely many transitions but reset by only finitely many transitions on the path. Otherwise the path is called *unblocked*. We have the following result.

Theorem 2 ([12]). *A TBA $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition iff there exists an unblocked path in $GZG(\mathcal{A})$ visiting both an accepting node and a clear node infinitely often.*

It is shown in the same paper that the proposed solution does not produce an exponential blowup. This is due to the fact that the zones reachable in $ZG(\mathcal{A})$ *order the clocks*. More precisely, if Z is the zone of a reachable node in $ZG(\mathcal{A})$

then for every two clocks x, y, Z implies that at least one of $x \leq y$, or $y \leq x$ holds [12]. Now, it can be checked that for every node (q, Z, Y) reachable in $GZG(\mathcal{A})$, the set Y respects the order given by Z, that is whenever $y \in Y$ and Z implies $x \leq y$ then $x \in Y$. In the end, a nice polynomial bound is obtained on the size of the guessing zone graph.

Lemma 2 ([12]). *Let $|ZG(\mathcal{A})|$ be the size of the zone graph, and $|X|$ be the number of clocks in $\mathcal{A}$. The number of reachable nodes of $GZG(\mathcal{A})$ is bounded by $|ZG(\mathcal{A})|.(|X| + 1)$.*

4 Solving the Emptiness Problem On-the-Fly

4.1 On-the-Fly Emptiness Check on the Guessing Zone Graph

We propose an on-the-fly algorithm for the Büchi emptiness problem based on Theorem 2. It requires to find an unblocked path in $GZG(\mathcal{A})$ that visits both an accepting node and a clear node infinitely often. To make the presentation of the algorithm simpler, we assume that there is one distinguished Büchi condition that labels exactly all the clear nodes. Hence we search for an unblocked path in $GZG(\mathcal{A})$ that satisfies the Büchi conditions.

Observe that we do not need to exhibit the path; proving its existence is simpler and sufficient[1]. Our problem can thus be stated over the SCCs of $GZG(\mathcal{A})$. A clock is *blocking* in an SCC if it is not reset by any transition in the SCC, but it is bounded from above by one of them. An SCC is *unblocked* when it has no blocking clock. Hence, finding an SCC that is unblocked and that contains an accepting node (for every Büchi condition) yields the required path. Conversely, from such a path, we have a satisfactory SCC.

The algorithm essentially involves three tasks: (1) detecting an accepting SCC, (2) inferring any blocking clocks in the SCC, and (3) re-exploring the SCC with blocking transitions removed. Since the blocking transitions might involve resets of other clocks, the SCCs obtained after removing blocking transitions might in turn contain new blocking clocks. These tasks have to be accomplished on-the-fly. The Couvreur's algorithm [9] finds accepting SCCs in (untimed) Büchi automata on-the-fly. We first adapt it to infer blocking clocks. Next we propose a modification to enable performing the third task. This last contribution is particularly involving. Our algorithm is depicted in Figure 2.

Couvreur's Algorithm. The Couvreur's algorithm, can be viewed as three functions *check_dfs*, *merge_scc* and *close_scc*. The algorithm annotates every node with an integer *dfsnum* and a boolean *opened*. The variable *dfsnum* is assigned based on the order of appearance of the nodes during the depth-first search. The *opened* bit is set to *true* when the node is just opened for exploration by *check_dfs* and is set to false by *close_scc* when the entire SCC of the node

[1] The path is a counter-example to the emptiness of the automaton, hence it can be an expected outcome of the algorithm. As in the untimed case, a short counter-example can be generated in a separate task from the outcome of our algorithm.

has been completely explored. The algorithm uses two stacks *Roots* and *Active*. The *Roots* stack stores the root of each SCC in the current search path. The root is the node of the SCC that was first visited by the DFS. If the roots stack is $s_0 s_1 \dots s_n$, then for $0 \leq i \leq n-1$, s_i is the root of the SCC containing all the nodes with *dfsnum* between $s_i.dfsnum$ and $s_{i+1}.dfsnum$ that have *opened* set to *true* and s_n is the root of the SCC containing all nodes with *dfsnum* greater than $s_n.dfsnum$ which have *opened* set to *true*.

The main function *check_dfs* proceeds by exploring the graph in depth-first search (DFS) order. When a successor t of the current node s is found, $t.dfsnum$ being zero implies t has not been visited yet and $t.opened$ being true implies that t belongs to the same SCC as s. When t belongs to the SCC of s, all the nodes visited in the path from t to s also belong to the SCC of s. These nodes are collected by the function *merge_scc*, which finds the root s_i of t and repeatedly pops the *Roots* stack so that s_i comes to the top, signifying that it is the root of the SCC containing all the nodes visited from t to s. A *maximal* SCC is detected when all the transitions of the current node s have been explored, with s being on the top of the *Roots* stack. The *close_scc* function is now called that sets the *opened* bit of all nodes in the SCC rooted at s to $false$. To identify these nodes, the *Active* stack is used, which stores all the nodes of the partially explored SCCs, in the order of the *dfsnum*.

Detecting Blocking Clocks. Observe that each node s in the *Roots* stack represents an SCC rooted at s. The idea is to augment the *Roots* stack so that along with each node s, the set of clocks (ub) that are bounded above and the set of clocks r that are reset in a transition of the SCC rooted at s, are also stored. The set $ub - r$ gives the set of blocking clocks of the SCC rooted at s. To achieve this, SCCs are stored in the *Roots* stack as tuples $(s, a, ub, r, ub_{in}, r_{in})$ where the extra sets ub_{in} and r_{in} store respectively the clocks that are bounded above and reset in the transition leading to s. The second modification occurs in *merge_scc* which now accumulates these sets while popping the nodes from the *Roots* stack. Figure 3 gives a schematic diagram of the merging procedure.

Re-exploring Blocked SCCs. When a maximal SCC Γ is detected, the set $ub - r$ of the root gives the set of blocking clocks. If $ub - r$ is not empty, Γ needs to be re-explored with the transitions bounding clocks of $blk \cup (ub - r)$ removed. Here blk denotes the blocked clocks when Γ was being explored. Doing this would split Γ into some p smaller maximal SCCs $\Gamma_1, \dots, \Gamma_p$ such that $\Gamma = \bigcup_{i=1}^{i=p} \Gamma_i$. Each Γ_i is reachable from the root of Γ. Thus, $GZG(\mathcal{A})$ would have an accepting run if some Γ_i contains an unblocked accepting cycle. The objective is to identify and explore the sub SCCs $\Gamma_1, \dots, \Gamma_p$ on-the-fly. The function *close_scc* is modified to enable on-the-fly re-exploration with blocked clocks.

Firstly, an additional bit per node called *explore* is needed to identify the nodes that have to be explored again. When Γ is detected and *close_scc* called, the *explore* bit is set to *true* for all the nodes of Γ, if Γ contains blocking clocks. The exploration is started from the root of Γ with the augmented set blk of the blocking clocks. Assume that Γ_i is currently being explored with the set

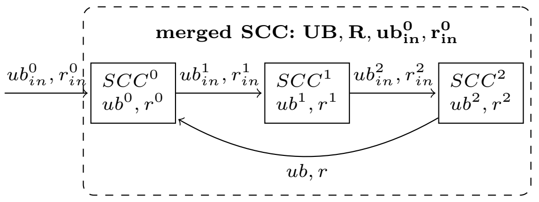
```
function emptiness_check()
count := 0; Roots, Active, Todo := $\emptyset$
check_scc($s_0$,$\emptyset$,$\emptyset$,$\emptyset$)
report $L(\mathcal{A}) = \emptyset$

function check_scc($s$,$ub_{in}$,$r_{in}$,$blk$)
if ($s$.dfsnum = 0)
  count++; $s$.dfsnum := count
$s$.opened := $\top$; $s$.explore := $\bot$
push(Roots, ($s$,$s$.labels,$\emptyset$,$\emptyset$,$ub_{in}$,$r_{in}$))
push(Active, $s$)
for all $s \xrightarrow{g;r} t$ do
  if ($UB(s,g) \cap blk \neq \emptyset$) and ($t \notin$Todo)
    push(Todo, $t$)
  else if ($t$.explore)
    check_scc($t$,$UB(s,g)$,$r$,$blk$)
  else if ($t$.opened)
    merge_scc($UB(s,g)$,$r$,$t$)
if top(Roots) = ($s$,$\cdots$)
  close_scc($blk$)

function merge_scc($ub'_{in}$,$r'_{in}$,$t$)
A:=$\emptyset$; U:=$ub'_{in}$; R:=$r'_{in}$
($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$) := pop(Roots)
while $s$.dfsnum > $t$.dfsnum do
  A:=A$\cup a$; U:=U$\cup ub \cup ub_{in}$
  R:=R$\cup r \cup r_{in}$
  ($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$) := pop(Roots)
A:=A$\cup a$; U:=U$\cup ub$; R:=R$\cup r$
push(Roots, ($s$,A,U,R,$ub_{in}$,$r_{in}$))
if (Acc$\subseteq$A) and (U$\subseteq$R)
  report $L(\mathcal{A}) \neq \emptyset$

function close_scc($blk$)
($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$):=pop(Roots)
repeat
  $u$ := pop(Active)
  $u$.opened := $\bot$
  if (Acc$\subseteq a$) and ($ub \not\subseteq r$)
    $u$.explore := $\top$
until $u = s$
if (Acc$\subseteq a$) and ($ub \not\subseteq r$)
  push(Todo, \$); push(Todo, $s$)
  while top(Todo) $\neq$ \$ do
    $s'$ := pop(Todo)
    if ($s'$.explore)
      check_scc($s'$,$\emptyset$,$\emptyset$,$blk \cup (ub - r)$)
  pop(Todo)
```

Fig. 2. Emptiness Check Algorithm on $GZG(\mathcal{A})$



$$UB = \bigcup\{ub^0, ub^1_{in}, ub^1, ub^2_{in}, ub^2, ub\}$$
$$R = \bigcup\{r^0, r^1_{in}, r^1, r^2_{in}, r^2, r\}$$

Fig. 3. How ub sets and r sets are merged

blk. Each time a transition $s \xrightarrow{g;r} t$ is detected, *check_dfs* checks if it bounds a blocking clock, that is, a clock in blk. This way the discarded transitions are detected on-the-fly. Since t may belong to some $\Gamma_j \neq \Gamma_i$, it is added to the $Todo$ stack for later re-exploration. The $Todo$ stack is the crucial element of the algorithm. The recursion ensures that Γ_i is completely closed before considering a Γ_j on the $Todo$ stack. After Γ_i completes, the node t' on the top of the $Todo$ stack is considered for exploration. However, it is possible that although t' was added due to a discarded transition, it could now belong to another SCC that has been completely closed, through a different enabled transition. Hence t' is explored only if its *explore* bit is true.

Note that it is possible that more blocking clocks are detected on some Γ_i. The re-exploration on Γ_i then updates the $Todo$ stack on-the-fly. However, while popping a node from $Todo$, in order to identify the right set of blocking clocks it has to be explored with, a marker \$ is pushed to the $Todo$ stack whenever a new re-exploration with a new set of blocking clocks is started.

Theorem 3. *The above algorithm is correct and runs in time $\mathcal{O}(|ZG(\mathcal{A})| \cdot |X|^2)$.*

4.2 On-the-Fly Emptiness Check on the Zone Graph

As stated in section 3.2, the guessing zone graph construction detects nodes where time elapse is not prohibited by future zero checks. However, in the absence of zero checks, this construction is not necessary. Recall that $GZG(\mathcal{A})$ has $|X|+1$ times more nodes than $ZG(\mathcal{A})$. Therefore, it is sufficient to consider the guessing zone graph construction only on maximal SCCs with zero checks. We propose an on-the-fly algorithm that checks for an accepting and unblocked SCC on the zone graph $ZG(\mathcal{A})$, detects zero checks and adopts the emptiness check algorithm on the guessing zone graph only when required. The algorithm shown in Figure 2 is run on $ZG(\mathcal{A})$ instead of $GZG(\mathcal{A})$. When a maximal SCC Γ is detected in line 19, it is required to know if Γ contains any zero checks.

Detecting Zero Checks. This is similar to the detection of blocking clocks. Two extra bits zc and zc_{in} are stored along with every tuple $(s, \dots)$ in the $Roots$ stack. zc_{in} tracks zero checks in the transition leading to s and zc remembers if there is a zero check in the SCC rooted at s. When an SCC is detected, the information is merged in *merge_scc* in a way similar to the schematic diagram shown in Figure 3. Thus when Γ is detected, the zc bit of the root reveals the presence of zero checks in Γ. The following lemma says that the algorithm can be terminated if Γ is accepting, unblocked and free from zero-checks.

Lemma 3. *If a reachable SCC in $ZG(\mathcal{A})$ is accepting, unblocked and free from zero-checks, then $\mathcal{A}$ has a non-Zeno accepting run.*

The interesting case occurs when Γ does have zero checks. In this case, we apply the guessing zone graph construction only to the nodes of Γ. If (q^Γ, Z^Γ) is the root of Γ, the guessing zone graph is constructed on-the-fly starting from (q^Γ, Z^Γ, X). Intuitively, we assume that the set of clocks X could potentially be zero at this node and start the exploration. We say that a run ρ of $\mathcal{A}$ *ends* in a maximal SCC Γ of $ZG(\mathcal{A})$ if a suffix of ρ is an instance of a path in Γ. Let $GZG(\mathcal{A})_{|\Gamma}$ be the part of $GZG(\mathcal{A})$ restricted to nodes and transitions that occur in Γ.

Lemma 4. *Let Γ be a reachable maximal SCC in $ZG(\mathcal{A})$, with root (q^Γ, Z^Γ). $\mathcal{A}$ has an accepting non-Zeno run ending in Γ iff $GZG(\mathcal{A})_{|\Gamma}$ has an unblocked accepting SCC reachable from (q^Γ, Z^Γ, X).*

Handling the Zero Checks. The function *close_scc* is modified to identify any zero-checks in the current maximal SCC Γ that was detected. If Γ contains zero checks, then *check_dfs* is called with the node $(q^{\Gamma}, Z^{\Gamma}, X)$ and the current set *blk* of blocking clocks. Note that the *explore* bit is *true* for all the nodes in Γ. Each time a new node $t = (q, Z, Y)$ of $GZG(\mathcal{A})$ is discovered in the for loop $s \xrightarrow{g;r} t$, it is explored only when $(q, Z).explore$ is true. When the exploration of $GZG(\mathcal{A})$ terminates, the *explore* bit of all the nodes of Γ are set to $false$. The *explore* bit is thus responsible for the restriction of the search on $GZG(\mathcal{A})$ only to the nodes that occur in Γ. In particular, this happens on-the-fly.

4.3 An Optimization

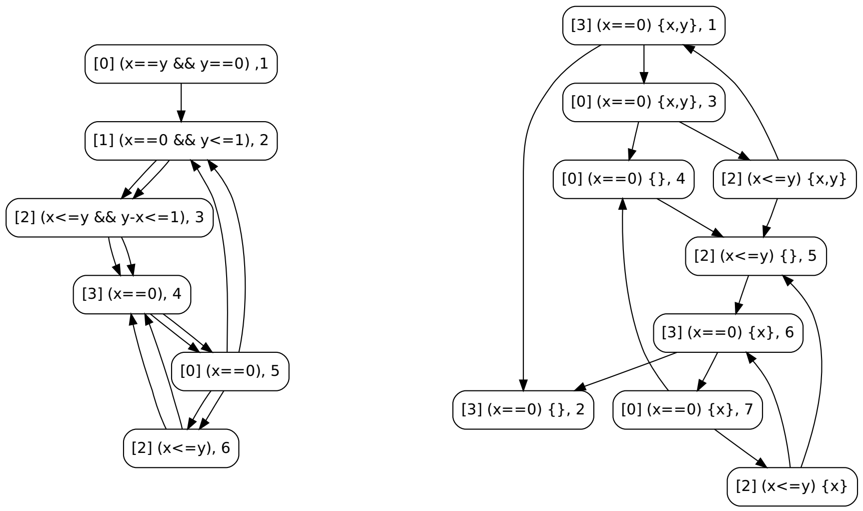
We propose an optimization to the algorithm to enable quicker termination in some cases. We take advantage of clocks in an SCC that are bounded from below, that is of the form $x > c$ or $x \geq c$ with $c \geq 1$. We make use of Lemma 1 which says that if an SCC Γ contains a transition that bounds a clock from below and a transition that resets the same clock, then any path of Γ including both these transitions should have non-Zeno instantiations. This is irrespective of whether Γ contains zero checks. Notice that no new clock is added.

To implement the above optimization, we add clock sets lb and lb_{in} to the *Roots* stack entries. The set lb keeps track of the clocks that are bounded below in the SCC. lb and lb_{in} are handled as explained in Figure 3 for the sets ub and ub_{in}. We also modify the condition for $L(\mathcal{A})$ to be empty. We conclude as soon as an SCC, not necessarily maximal, is accepting and it bounds below and resets the same clock. Observe that it is not useful to keep track of the clocks that are bounded from below while solving zero-checks on $GZG(\mathcal{A})$ as any transition that sets a lower bound in $GZG(\mathcal{A})$ has already been visited in $ZG(\mathcal{A})$.

4.4 The Global Algorithm and Implementation Issues

A run of our algorithm is illustrated in Figure 4. Exploring $ZG(\mathcal{A}_1)$, it computes the maximal SCC rooted at node 2 that is blocked by y. Hence, it re-explores that SCC with set of blocking clocks $blk = \{y\}$. This is seen from the double edges. Notice that the edge from node 5 to node 2 is not crossed as it is discarded by blk. The algorithm now computes the SCC Γ that contains the nodes 4, 5 and 6, that is maximal w.r.t. blk. Γ is unblocked and accepting, but it has zero-checks. Hence the algorithm explores $GZG(\mathcal{A}_1)$ starting from node $(3, x == 0, \{x, y\})$. It does not explore any transition outside of Γ. It eventually finds a non-maximal SCC in $GZG(\mathcal{A}_1)$ that proves $L(\mathcal{A}_1) \neq \emptyset$.

Finally, we discuss memory issues for the implementation of our algorithm. Information about the SCCs in $ZG(\mathcal{A})$ is stored in the *Roots* stack as tuples $(s, a, lb, ub, r, zc, lb_{in}, ub_{in}, r_{in}, zc_{in})$. An amount of $6.|X| + 2$ bits of memory is required to store the membership of each clock to the 6 sets, and the zero-check information. In the worst case, all the nodes in $ZG(\mathcal{A})$ have an entry in the *Roots* stack. Similarly, SCCs in $GZG(\mathcal{A})$ are stored as tuples $(s, a, ub, r, ub_{in}, r_{in})$ in the *Roots* stack. $4.|X|$ bits allow to represent the content of those 4 sets. We



ZG($\mathcal{A}_1$) Part of **GZG**($\mathcal{A}_1$)

Fig. 4. Zone Graphs of $\mathcal{A}_1$

estimate than the overhead due to our algorithm is $4.|X|.(|X| + 1)$ due to the size of $GZG(\mathcal{A})$ w.r.t. the size of $ZG(\mathcal{A})$.

We eventually compare to the strongly non-Zeno construction. Running the Couvreur's algorithm requires to store smaller tuples (s, a). However, the zone in s is represented by a DBM which is an $|X| + 1$-square matrix containing 32 bits integers. Hence, adding one clock increases the memory footprint by $(2.|X|+3).32$ bits. Recall furthermore than $ZG(SNZ(\mathcal{A}))$ can be $2^{\mathcal{O}(|X|)}$ bigger than $ZG(\mathcal{A})$. In the worst case all these nodes are on the *Roots* stack.

5 Experiments and Conclusion

We have implemented our algorithms in a prototype verification tool. The table below presents the results that we obtained on several classical examples. The models are products of Timed Büchi Automata that encode both the processes in the system and the property to verify. The "Zone Graph" column gives the number of nodes in the zone graph. Next, for the "Strongly non-Zeno" construction, we give the size of the resulting zone graph followed by the number of nodes that are visited during verification; similarly for the "Guessing Zone Graph" where the last column corresponds to our fully optimized algorithm.

We have considered three types of properties: reachability properties (mutual exclusion, collision detection for CSMA/CD), liveness properties (access to the resource infinitely often), and bounded response properties (which are reachability properties with real-time requirements). When checking reachability properties we would like counter examples to be feasible by real systems.

This is not the case with Zeno counter examples. Hence Timed Büchi Automata offer a nice framework as they discard Zeno runs.

The strongly non-Zeno construction outperforms the guessing zone graph construction for reachability properties. This is particularly the case for mutual exclusion on the Fischer's protocol and collision detection for the CSMA/CD protocol. For liveness properties, the results are mitigated. On the one hand, the strongly non-Zeno property is once again more efficient for the CSMA/CD protocol. On the other hand the differences are tight in the case of Fischer protocol. The guessing zone graph construction distinguishes itself for bounded response properties. Indeed, the Train-Gate model is an example of exponential blowup for the strongly non-Zeno construction.

We notice that on-the-fly algorithms perform well. Even when the graphs are big, particularly in case when automata are not empty, the algorithms are able to conclude after having explored only a small part of the graph. Our optimized algorithm outperforms the two others on most examples. Particularly, for the CSMA/CD protocol with 5 stations our algorithm needs to visit only 4841 nodes while the two other methods visited 8437 and 21038 nodes. This confirms our initial hypothesis: most of the time, the zone graph contains enough information to ensure time progress.

Our last optimization also proves useful for the FDDI protocol example. One of its processes has zero checks, but since some other clock is bounded from below and reset, it was not necessary to explore the guessing zone graph to conclude non-emptiness.

Models	Zone Graph	Strongly non-Zeno		Guessing Zone Graph		
	size	size	visited	size	visited	visited opt.
Train-Gate2 (mutex)	134	194	194	400	400	134
Train-Gate2 (bound. resp.)	988	227482	352	3840	1137	292
Train-Gate2 (liveness)	100	217	35	298	53	33
Fischer3 (mutex)	1837	3859	3859	7292	7292	1837
Fischer4 (mutex)	46129	96913	96913	229058	229058	46129
Fischer3 (liveness)	1315	4962	52	5222	64	40
Fischer4 (liveness)	33577	147167	223	166778	331	207
FDDI3 (liveness)	508	1305	44	3654	79	42
FDDI5 (liveness)	6006	15030	90	67819	169	88
FDDI3 (bound. resp.)	6252	41746	59	52242	114	60
CSMA/CD4 (collision)	4253	7588	7588	20146	20146	4253
CSMA/CD5 (collision)	45527	80776	80776	260026	260026	45527
CSMA/CD4 (liveness)	3038	9576	1480	14388	3075	832
CSMA/CD5 (liveness)	32751	120166	8437	186744	21038	4841

Conclusion and Future Work. In this paper, we have presented an algorithm for the emptiness problem of Timed Büchi Automata. We claim that our algorithm is on-the-fly as (1) it computes the zone graph during the emptiness check; (2) it does not store the graph, in particular when it splits blocked SCCs; and (3) it stops as soon as an SCC that contains a non-Zeno accepting run is found. Our algorithm is inspired from the Couvreur's algorithm. However, the handling of bounded clocks and the application of the guessing zone graph construction on-the-fly, are two substantial increments. The previous examples are academic case studies; still they show that Büchi properties can be checked as efficiently as reachability properties for Timed Automata, despite the non-Zenoness requirement.

As a future work we plan to extend our algorithm to commonly used syntactic extensions of Timed Automata. For instance Uppaal and Kronos allow to reset clocks to arbitrary values, which is convenient for modeling real life systems. This would require to extend the guessing zone graph construction, and consequently our algorithm. Another interesting question is to precisely characterize the situations where an exponential blowup occurs in the strongly non-Zeno construction. To conclude, it can be seen that the ideas of the prototype could be used to construct a full-fledged tool like Profounder, which implements the strongly non-Zeno construction.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., Madhusudan, P.: Decision problems for timed automata: A survey. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 1–24. Springer, Heidelberg (2004)
3. Behrmann, G., David, A., Larsen, K.G., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST 2006, pp. 125–126 (2006)
4. Bérard, B., Bouyer, B., Petit, A.: Analysing the pgm protocol with UPPAAL. Int. Journal of Production Research 42(14), 2773–2791 (2004)
5. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time petri nets. In: IFIP Congress, pp. 41–46 (1983)
6. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-fly symbolic model checking for real-time systems. In: RTSS 1997, p. 25 (1997)
7. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
8. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a mode-checking tool for real-time systems. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
9. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized büchi automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
10. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
11. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
12. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient emptiness check for timed büchi automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)
13. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
14. Tripakis, S.: Verifying progress in timed systems. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
15. Tripakis, S.: Checking timed büchi emptiness on simulation graphs. ACM Transactions on Computational Logic 10(3) (2009)
16. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed büchi automata emptiness efficiently. Formal Methods in System Design 26(3), 267–292 (2005)

Reachability as Derivability, Finite Countermodels and Verification

Alexei Lisitsa

Department of Computer Science, the University of Liverpool,
Ashton Building, Ashton Street,
Liverpool, L69 7ZF, U.K.
alexei@csc.liv.ac.uk

Abstract. We propose a simple and efficient approach to the verification of parameterized and infinite state system. The approach is based on modeling the reachability relation between parameterized states as deducibility between suitable encodings of the states using formulae of first-order logic. To establish a safety property, namely the non-reachability of unsafe states, a finite model finder is used to generate a finite countermodel, thus providing the witness for non-deducibility. We show that under an appropriate encoding the combination of finite model finding and theorem proving provides us with a decision procedure for the safety of the lossy channel systems. We illustrate the approach by reporting on experiments verifying both alternating bit protocol (specified as a lossy channel system) and a number of parameterized cache coherence protocols specified by extended finite state machines. In these experiments, the finite model finder Mace4 is used.

1 Introduction

In this introductory section we outline the main idea of the proposed method. Let $\mathcal{S} = \langle S, \rightarrow \rangle$ be a transition system comprising the set of states S and transition relation $\rightarrow$. Denote by $\rightarrow^*$ the transitive closure of $\rightarrow$. Consider the encoding $e : s \mapsto \varphi_s$ of states of $\mathcal{S}$ by formulae of first-order predicate logic, satisfying the following property.

The state s' is reachable from s, i.e. $s \rightarrow^* s'$ if and only if $\varphi_{s'}$ is the logical consequence of φ_s, that is $\varphi_s \models \varphi_{s'}$ and $\varphi_s \vdash \varphi_{s'}$.

Here we assume standard definitions of both semantical consequence $\models$ and deducibility $\vdash$ (in a complete deductive system) for first-order predicate logic. Under such assumptions one can translate reachability questions for S to classical questions in logic. Thus, establishing reachability amounts to theorem proving, while deciding non-reachability becomes theorem disproving. It is clear that, due to the undecidability of first-order logic, such an approach can not be universal. However, one may hope that the highly developed automated theorem provers [19] and model finders [2] for first-order logic might be used for automated decision of the (non-)reachability problems.

In this paper we will focus on applications of these ideas to the automated verification of the *safety* properties within *parameterized* and/or *infinite state* systems. The adaption of the idea of "reachability as logical consequence" to the parametric case is not straightforward and appropriate encodings must be crafted carefully for any particular class of problems. In the following sections we will present such encodings for the classes of Lossy Channel Systems and Extended Finite State Machines. Restriction to safety properties, i.e. non-reachability of *unsafe* states, means we will be mainly dealing with automated disproving. To disprove $\varphi_s \models \varphi_{s'}$ it is sufficient to find a countermodel for $\varphi_s \rightarrow \varphi_{s'}$, or, equivalently, the model for $\varphi_s \wedge \neg\varphi_{s'}$. In general, in first-order logic such a model may be inevitably infinite. Furthermore, the set of satisfiable first-order formulae is not recursively enumerable, so one can not hope for complete automation here. As a partial solution we propose using automated *finite* model finders/builders [2] for disproving. Although incomplete in general, this approach has turned out to be complete for subclasses of verification problems and empirically successful for larger classes of practically important parameterized problems.

The results presented in this paper were previously announced in its short version which appeared in [15].

1.1 Preliminaries

We assume that the reader is familiar with the following concepts from logic and algebra which we will use without definitions: first-order predicate logic, first-order models, interpretations of relational, functional and constant symbols, satisfaction $\models$ of a formula in a model, semantical consequence $\models$, deducibility (derivability) $\vdash$ in first-order logic, monoid, homomorphism. We denote the interpretations by square brackets, so, for example, $[f]$ denotes an interpretation of a functional symbol f in a model.

2 Verification of Lossy Channel Systems

An important class of infinite state systems which can be verified by our method is the class of Lossy Channel Systems, which are essentially finite-state automata augmented with unbounded, but lossy, FIFO channels (queues). The messages sent via lossy channels may be lost in transition.

The definitions below are taken from [1].

Definition 1. *A Lossy Channel System $\mathcal{L}$ is a tuple $\langle S, s_0, A, C, M, \delta\rangle$, where S is a finite set of* control states, *$s_0 \in S$ is an* initial control state*, A is a finite set of* actions, *C is a finite set of channels, M is a finite set of messages, δ is a finite set of transitions, each of which is a triple of the form $\langle s_1, op, s_2\rangle$, where $s_1, s_2 \in S$ and op is a label of one of the forms*

- *$c!m$, where $c \in C$ and $m \in M$,*
- *$c?m$, where $c \in C$ and $m \in M$,*
- *a, where $a \in A \cup \{\tau\}$.*

Given a Lossy Channel System $\mathcal{L} = \langle S, s_0, A, C, M, \delta\rangle$, a *global state* of $\mathcal{L}$ is a pair $\langle s, w\rangle$, where $s \in S$ and $w : C \rightarrow M^*$ is a function assigning to each channel a finite sequence of messages (content of the channel). We will also write w_c for $w(c)$ and $w = w_{c_1} \dots w_{c_k}$ for $C = \{c_1, \dots, c_k\}$. We denote the concatenation of two sequences of messages x and y by $x \cdot y$, or simply xy if it does not lead to confusion.

The initial global state γ_0 of $\mathcal{L}$ is a pair $\langle s_0, \epsilon \rangle$ where ϵ assigns an empty sequence to every channel.

The transition relation $\rightarrow$ is a set of triples $\langle \gamma, a, \gamma' \rangle$, where γ and γ' are global states and $a \in A \cup \{\tau\}$. For $\langle \gamma, a, \gamma' \rangle \in\rightarrow$ we write $\gamma \rightarrow^a \gamma'$. The transition relation $\rightarrow$ is now defined as the smallest set satisfying

1. If $\langle s_1, c!m, s_2 \rangle \in \delta$ then $\langle s_1, w \rangle \rightarrow^\tau \langle s_2, w' \rangle$, where w' coincides with w everywhere, except $w'(c) = w(c) \cdot m$, i.e. the message m is appended to the content of the channel c.
2. If $\langle s_1, c?m, s_2 \rangle \in \delta$ and $w(c) = m \cdot l$ then $\langle s_1, w \rangle \rightarrow^\tau \langle s_2, w' \rangle$, where w' coincides with w everywhere except $w'(c) = l$, i.e. the message m is read off the channel c.
3. If $w(c) = x \cdot m \cdot y$, then $\langle s, w \rangle \rightarrow^\tau \langle s, w' \rangle$, where w' coincides with w everywhere except $w'(c) = x \cdot y$.
4. If $\langle s_1, a, s_2 \rangle \in \delta$, then $\langle s_1, w \rangle \rightarrow^a \langle s_2, w \rangle$.

For global states γ and γ', and a sequence $\sigma \in A^*$ the expression $\gamma \Rightarrow^\sigma \gamma'$ denotes the fact that γ' is reachable from γ, that is

$$\gamma = \gamma_1 \rightarrow^{a_1} \gamma_2 \rightarrow^{a_2} \ldots \rightarrow^{a_{n-1}} \gamma_n = \gamma'$$

and σ is the sequence of non-τ actions among $a_1, \ldots, a_{n-1}$. Further, we write $\gamma \rightarrow \gamma'$ to denote $\gamma \rightarrow^a \gamma'$ for some $a \in A \cup \{\tau\}$, and $\gamma \rightarrow^* \gamma'$ to denote $\gamma \Rightarrow^\sigma \gamma'$ for some $\sigma \in A^*$. A *trace* of a lossy channel system L is a sequence $\sigma \in A^*$ such that $\gamma_0 \Rightarrow^\sigma \gamma$ for some γ. The set of all traces of L is denoted by $Traces(L)$.

2.1 Verification of Safety Properties

The general form of the *safety* verification problem for lossy channel systems we address here is as follows.

Given: A lossy channel system $L = \langle S, s_0, A, C, M, \delta \rangle$ and a set $\Sigma \subseteq A^*$
Question: Does $Traces(L) \subseteq \Sigma$ hold?

The set Σ of safe traces represents the expected behaviour, or correctness condition for the system to be verified. From now on we assume that Σ is a *regular language* and is effectively given.

Consider an instance of the above verification problem. Let $M = \langle T, t_0, A, \rho, F \rangle$ be a deterministic finite automaton which accepts the *complement* of Σ. Here T is a finite set of states, $t_0 \in T$ is an initial state, A is the set of actions, $\rho : T \times A \rightarrow T$ is a transition function and F is a set of accepting states.

Definition 2. *Let L and M are as above. The product $L \times M$ is the lossy channel system $\langle S \times T, \langle s_0, t_0 \rangle, A, C, M, \delta' \rangle$, where δ' is a set of triples of the form $\langle \langle s_1, t_1 \rangle, op, \langle s_2, t_2 \rangle \rangle$ where either*

– *op is of the form $c!m$, $c?m$, or τ, and $t_1 = t_2$ and $\langle s_1, op, s_2 \rangle$ is a transition in δ, or*
– *$op \in A$, $\langle s_1, a, s_2 \rangle$ is a transition in δ and $\rho(t_1, a) = t_2$ is a transition in ρ.*

The above question of the verification problem can be equivalently reformulated now as a question of *reachability*:

Is it true that in $L \times M$ no global state of the form $\langle\langle s,t\rangle, w\rangle$ with $t \in F$ is reachable? (1)

So far we closely followed closely the known standard approaches to the verification of lossy channel systems, see e.g. [1]. Now, instead of application of the specialized symbolic reachability algorithm used before, we reduce the above reachability question to a finite model finding task, which is then delegated to the generic finite model finders. This is the topic of the next subsection.

2.2 Verification via Countermodel Finding

Given the reachability question (1), define a translation of the product system $L \times M$ into a set $\Phi_{L\times M}$ of formulae of the first-order predicate logic as follows. The vocabulary of $\Phi_{L\times M}$ consists of

- the set of constant symbols $\mathcal{C} = \{d_s \mid s \in S\} \cup \{d_t \mid t \in T\} \cup \{d_m \mid m \in M\} \cup \{e\}$
- one binary functional symbol $*$
- one predicate symbol R of arity $n+2$, where n is a number of channels in L, i.e. $\mid C \mid$

For a sequence of messages $l \in M^*$ define its term translation $\tilde{l}$ inductively: $\tilde{\Lambda} = e$, $\tilde{m} = d_m$ and $x \tilde{\cdot} y = \tilde{x} * \tilde{y}$. The translation is well-defined modulo the associativity of $*$ which we will specify in the formula.

Given a global state $\gamma = \langle\langle s,t\rangle, w_{c_1} \dots w_{c_n}\rangle$ of $L \times M$ we define its translation to a $n+2$-tuple of terms as $\bar{t}_\gamma = (d_s, d_t, \tilde{w_{c_1}}, \dots, \tilde{w_{c_n}})$. Thus, terms of $\bar{t}_\gamma$ represent the states of L and M together with the content of all channels.

The intended meaning of the atomic formula $R(\bar{t}_\gamma)$ is "the global state γ is reachable."

Let $\Phi_{L\times M}$ be a set of the following formualae, which are all assumed to be *universally closed*. In the formulae $\bar{x}$ denotes a sequence of *different variables* $x_1, \dots, x_n$ and $\bar{x}_{i\mapsto t}$ coincides with $\bar{x}$ everywhere except in ith position where it contains a term t.

$$(x * y) * z = x * (y * z)$$
$$R(d_{s_0}, d_{t_0}, e, \dots, e)$$
$$R(d_{s_1}, d_{t_1}, \bar{x}) \to R(d_{s_2}, d_{t_2}, \bar{x}) \text{ for all } \langle s_1, a, s_2\rangle \in \delta \text{ and } \rho(t_1, a) = t_2$$
$$R(d_{s_1}, y, \bar{x}) \to R(d_{s_2}, y, \bar{x}_{i\mapsto m*x_i}) \text{ for all } \langle s_1, c_i!m, s_2\rangle \in \delta$$
$$R(d_{s_1}, y, \bar{x}_{i\mapsto m*x_i}) \to R(d_{s_2}, y, \bar{x}) \text{ for all } \langle s_1, c_i?m, s_2\rangle \in \delta$$
$$R(y, z, \bar{x}_{i\mapsto (z_1*z_2)*z_3}) \to R(y, z, \bar{x}_{i\mapsto z_1*z_3}) \text{ for all } i = 1, \dots, n$$

Proposition 1 (Adequacy of translation). *If a global state γ is reachable in $L \times M$ then $\Phi_{L\times M} \vdash R(\bar{t_\gamma})$.*

Proof is by induction on the length of the transition sequences in $L \times M$. For the initial global state γ_0 we have $R(\bar{t}_{\gamma_0}) \in \Phi_{L\times M}$, so for the base case of induction we have $\Phi_{L\times M} \vdash R(\bar{t_{\gamma_0}})$. Now, we notice that, if $\gamma \to \gamma'$ is a one step transition of global states in $L \times M$, then $R(\bar{t}_\gamma) \to R(\bar{t}_{\gamma'})$ is a ground instance of one of the formulae in $\Phi_{L\times M}$ (by immediate inspection of the definition of $\Phi_{L\times M}$). Assuming γ is reachable in $L \times M$ and $\Phi_{L\times M} \vdash R(\bar{t}_\gamma)$, then applying *Modus Ponens* we get $\Phi_{L\times M} \vdash R(\bar{t}_{\gamma'})$. That provides the step of induction.

Theorem 1. *For any lossy channel system $L = \langle S, s_0, A, C, M, \delta\rangle$, and deterministic finite automaton $M = \langle T, t_0, A, \rho, F\rangle$, exactly one of following mutually exclusive alternatives holds:*

i) $\Phi_{L\times M} \vdash \vee_{f\in F}\exists x\exists \bar{z} R(x, d_f, \bar{z})$, *or*
ii) there is a finite model $\mathcal{M}$ such that
$\mathcal{M} \models \Phi_{L\times M} \wedge \neg(\vee_{f\in F}\exists x\exists \bar{z} R(x, d_f, \bar{z}))$

Proof. Consider two cases, depending on the answer to the reachability question (1).

Case 1. For some $s \in S, t \in F, w_{c_1}, \dots, w_{c_k} \in M^*$ the global state $\langle\langle s,t\rangle, w_{c_1}, \dots, w_{c_k}\rangle$ of $L \times M$ is reachable. Then by Proposition 1 we have $\Phi_{L\times M} \vdash R(d_s, d_t, \tilde{w_{c_1}}, \dots, \tilde{w_{c_k}})$ and, therefore, $\Phi_{L\times M} \vdash \vee_{f\in F}\exists x\exists \bar{z} R(x, d_f, \bar{z})$. The first alternative holds.

Case 2. For none of the $s \in S, t \in F, w_{c_1}, \dots, w_{c_k} \in M^*$ the global state $\langle\langle s,t\rangle, w_{c_1}, \dots, w_{c_k}\rangle$ of $L \times M$ is reachable. We now construct a finite model $\mathcal{M}$ which satisfies the formula in the second alternative above. The construction crucially uses the *finite* characterization of the set $\hat{V}$ of global states of $L \times M$ from which global states $\langle\langle s,t\rangle, w\rangle$ with $t \in F$ are reachable, given in [1]:

$$\hat{V} = \{\gamma; \exists\gamma'.\gamma' \in V \wedge \gamma' \preceq \gamma\}$$

where V is some *finite* set of global states. Here $\preceq$ denotes a pre-order on the global states defined as follows.

Definition 3. *For $\gamma_1 = \langle\langle s,t\rangle, w_{c_1}, \dots, w_{c_k}\rangle$ and $\gamma_2 = \langle\langle s',t'\rangle, w'_{c_1}, \dots, w'_{c_k}\rangle$ we have $\gamma_1 \preceq \gamma_2$ if and only if $s = s'$, $t = t'$ and for each $i = 1, \dots, k$ w_{c_i} is a (not necessarily contiguous) subword of w'_{c_k}.*

It follows[1] that $\hat{V}$ is a *regular* set. To give more details let $\hat{V} = \cup_{s\in S, t\in T, \gamma\in V}\hat{V}_{s,t,\gamma}$, where $\hat{V}_{s,t,\gamma} = \{\langle s,t,\bar{w}\rangle \in \hat{V} \mid \gamma \preceq \langle s,t,\bar{w}\rangle\}$. Then we have that $\hat{V}_{s,t,\gamma} = \{\langle s,t,w_1,\dots,w_k\rangle \mid w_1 \in L_1^{s,t,\gamma}, \dots, w_k \in L_k^{s,t,\gamma}\}$ and $L_i^{s,t,\gamma} \subseteq M^*$ are all regular sets. Thus, $L_i^{s,t,\gamma}$ is a language of all words which appear in ith channel in the global states $\langle\langle s,t\rangle, \bar{w}\rangle$ greater (in a sense of $\preceq$) than a particular global state $\gamma \in V$.

Notice that, by the assumption of the Case 2, the set of all reachable global states in $L \times M$ is disjoint with $\hat{V}$.

Let $M_i^{s,t,\gamma} = \langle T_i^{s,t,\gamma}, t_{0_i}^{s,t,\gamma}, A, \rho_i^{s,t,\gamma}, F_i^{s,t,\gamma}\rangle$ be a deterministic finite state automaton which recognizes the regular language $L_i^{s,t,\gamma}$.

Now any $m \in M$ induces a transformation

$$\tau_m : \coprod_{s,t,\gamma,i} T_i^{s,t,\gamma} \to \coprod_{s,t,\gamma,i} T_i^{s,t,\gamma}$$

[1] See relevant discussion in [1], pp 15–16, with references to [5,13].

of the disjoint union (coproduct) of all state sets $T_i^{s,t,\gamma}$ as follows. For any $t \in T_i^{s,t,\gamma}$, $\tau_m(t) = \rho_i^{s,t,\gamma}(t,m) \in T_i^{s,t,\gamma}$. Intuitively, τ_m represents the state transitions of all automata after reading a symbol m. All transformations τ_m with $m \in M$ and the operation $\circ$ of composition of transformations generate a monoid $\mathcal{TM} = \langle TM, \circ \rangle$, the transformational monoid of the direct sum of all automata $L_i^{s,t,\gamma}$. Due to the finitness of all $T_i^{s,t,\gamma}$ this monoid is *finite*. Mapping $m \mapsto \tau_m$ is extended to a homomorphism h of the free monoid M^* to $\mathcal{TM}$ in usual way.

Now we return to the construction of the required model $\mathcal{M}$. Define its domain to be a union $S \cup T \cup TM \cup \underline{0}$, where S and T are state sets of the original lossy channel system and correctness checking automaton, respectively; TM is a set of elements of the monoid $\mathcal{TM}$, and $\underline{0}$ is a distinct "dummy" constant.

Define interpretations of the constants:

$[d_s] = s$ for $s \in S$
$[d_t] = t$ for $t \in T$
$[d_m] = \tau_m$ for $m \in M$
$[e] = \underline{e}$, where $\underline{e}$ is an identity element of $\mathcal{TM}$

The interpretation of $*$ is a mapping $[*] : D \times D \to D$, where (in infix notation)

$\tau_1[*]\tau_2 = \tau_1 \circ \tau_2$ for $\tau_1, \tau_2 \in TM$
$x[*]y = \underline{0}$, otherwise

Define an interpretation $[R] \subseteq D^{k+2}$ of the relational symbol R:

$$[R] = \{\langle d_s, d_t, \tau_1, \ldots, \tau_k \rangle \mid$$
$$s \in S, t \in T, \tau_i \in TM, \forall \gamma \in V(\tau_i(t_{0_i}^{s,t,\gamma}) \notin F_i^{s,t,\gamma})\}$$

In other words, R is true on tuples $\langle d_s, d_t, h(\bar{w})) \rangle$ such that no global state $\langle s', t', \bar{w}' \rangle$ with $t' \in F$ is reachable in $L \times M$ from $\langle s, t, \bar{w}) \rangle$. That concludes the definition of finite model $\mathcal{M}$.

By the assumption of Case 2 and definition of $[R]$ we have $\mathcal{M} \models \neg(\bigvee_{f \in F} \exists x \exists \bar{z} R(x, d_f, \bar{z}))$.

To show $\mathcal{M} \models \Phi_{L \times M}$ we consider all clauses in the definition of $\Phi_{L \times M}$ separately:

1. $\mathcal{M} \models (x * y) * z = x * (y * z)$. This holds true by the choice of interpretation of $*$ as a monoid operation $\circ$.
2. $\mathcal{M} \models R(d_{s_o}, d_{t_0}, e, \ldots, e)$. This holds true by the assumption of Case 2 that no bad global state is reachable from the initial state of $L \times M$.
3. $\mathcal{M} \models R(d_{s_1}, d_{t_1}, \bar{x}) \to R(d_{s_2}, d_{t_2}, \bar{x})$ for all $\langle s_1, a, s_2 \rangle \in \delta$ and $\rho(t_1, a) = t_2$. Assume that if $\langle s_1, t_1, \tau_1, \ldots, \tau_k \rangle \in [R]$, then we have $\tau_i = h(w_i)$ for some $w_i \in M^*$, $i = 1 \ldots k$. By the definition of $[R]$, no bad state is reachable from $\langle s_1, t_1, w_1, \ldots w_k \rangle$ in $L \times M$. On the other hand, we have $\langle s_1, t_1, w_1, \ldots w_k \rangle \to^a \langle s_2, t_2, w_1, \ldots w_k \rangle$ in $L \times M$ and, therefore, no bad state is reachable from $\langle s_2, t_2, w_1, \ldots w_k \rangle$ either. It follows that $\langle s_2, t_2, \tau_1, \ldots, \tau_k \rangle \in [R]$.
4. $\mathcal{M} \models R(d_{s_1}, y, \bar{x}) \to R(d_{s_2}, y, \bar{x}_{i \mapsto m * x_i})$ for all $\langle s_1, c_i!m, s_2 \rangle \in \delta$. This case is treated analogously to previous one.

5. $R(d_{s_1}, y, \bar{x}_{i \mapsto m*x_i}) \rightarrow R(d_{s_2}, y, \bar{x})$ for all $\langle s_1, c_i?m, s_2 \rangle \in \delta$. Assume that $\langle s_1, t, \tau_1, \ldots, h(m) \circ \tau_i, \ldots, \tau_k \rangle \in [R]$ then for some $w_1, \ldots, w_k$ we have $\tau_i = h(w_i)$ and no bad state is reachable from $\langle s_1, t, w_1, \ldots, m * w_i, \ldots, w_k \rangle$.
   Since $\langle s_1, t, w_1, \ldots, m*w_i, \ldots, w_k \rangle \rightarrow \langle s_2, t, w_1, \ldots, w_i, \ldots, w_k \rangle$ no bad state is reachable from $\langle s_2, t, w_1, \ldots, w_i, \ldots, w_k \rangle$ either. It follows that $\langle s_2, t, \tau_1, \ldots, \tau_i, \ldots, \tau_k \rangle \in [R]$.
6. $R(y, z, \bar{x}_{i \mapsto (z_1 * z_2) * z_3}) \rightarrow R(y, z, \bar{x}_{i \mapsto z_1 * z_3})$ for all $i = 1, \ldots, n$. This case is treated analogously to previous one.

That concludes the proof of the Theorem 1. □

Note 1. Using finite automata in the model construction is not necessary. Alternatively one may use an algebraic characterization of regular languages as the preimages of subsets of finite monoids under a homomorphism from the free monoid.

Note 2. The above model construction serves only the purpose of proof and it is not efficient in practical use of the method. Instead, we delegate the task of model construction to generic finite model building procedures.

Corollary 1. *Parallel composition of complete theorem proving and complete finite building procedures for first-order predicate logic provides us with a decision procedure for safety properties of lossy channel systems.*

The approach has turned out to be practically efficient. Using translation of the specification from [1] and the finite model finder Mace4 we have verified the Alternative Bit Protocol modeled by an lossy channel system[14]. See Section 4 for experimental results.

3 Verification of Parameterized Cache Coherence Protocols

Another class of the systems to which verification via our finite countermodel finding approach has been applied is that of parameterized cache coherence protocols [7], modeled by Extended Finite State Machines (EFSM)[4] using a *counting abstraction* [7]. The variant of EFSM we consider here is as in [7], namely without input and output signals and with a single location (control state), which is a restriction of general EFSM model introduced in [4]. The states of EFSM are n-dimensional non-negative integer vectors and the transitions are affine transformations with affine pre-conditions. Denote by $\mathbb{Z}^+$ the set of non-negative integers. Let $X = \{x_1, \ldots, x_n\}$ be a finite set of variables. A *state* is a non-negative integer evaluation of all variables from X, that is a mapping $s : X \rightarrow \mathbb{Z}^+$, which we represent by an n-dimensional integer vector $(s(x_1), \ldots s(x_n)) \in {\mathbb{Z}^+}^n$

Definition 4. *An extended finite state machine is a tuple $\langle X, I, T \rangle$ where*

- *$X = \{x_1, \ldots, x_n\}$ is a finite set of variables;*
- *T is a finite set of transitions of the form $G(\bar{x}) \rightarrow T(\bar{x}, \bar{x}')$, where, the guard $G(\bar{x})$ has a form $A \cdot \bar{x} \geq \bar{c}$ and $T(\bar{x}, \bar{x}')$ denotes an affine transformation $\bar{x}' = M \cdot \bar{x} + \bar{d}$;*

here $\bar{c}, \bar{d} \in \mathbb{Z}^{+^n}$ are n-dimensional integer vectors and $A, M \in \mathbb{Z}^{+^n} \times \mathbb{Z}^{+^n}$ are $n \times n$-matrices with unit vectors as columns;
– *$I \subseteq (\mathbb{Z}^+)^n$ is a set of initial states.*

Definition 5. *A run of an extended finite state machine $M = \langle X, I, T\rangle$ is a possibly infinite sequence of states $\bar{c}_0, \dots, \bar{c}_i, \dots$ where $G_i(\bar{c}_i) \to T_i(\bar{c}_i, \bar{c_{i+1}})$ holds true for some transitions $G_i(\bar{x}_i) \to T_i(\bar{x}_i, \bar{x_{i+1}})$ in T for $i \geq 0$. A state $\bar{c}'$ is reachable from $\bar{c}$ in M iff there is a run $\bar{c}, \dots, \bar{c}', \dots$ of M.*

Definition 6. *A set of states $S \subseteq \mathbb{Z}^{+^n}$ is called* upwards closed *if it is of the form $\{(x_1, \dots, x_n) \mid \wedge_{i=1\dots n}(x_i \sim_i c_i)\}$ where $\sim_i$ is either $=$ or $\geq$ and $c_i \in \mathbb{Z}^+$.*

The general form of the parameterized verification problem we consider here is as follows:

Given: An extended finite state machine $M = \langle X, I, T\rangle$ with an upwards closed set $I \subseteq \mathbb{Z}^{+^n}$ of the initial states, and a *finite* family of the upwards closed sets of *unsafe* states $F_1, \dots F_k \in \mathbb{Z}^{+^n}$
Question: Is it true that *none* of the states in $F_1, \dots, F_k$ is reachable from any of the initial states?

Here we assume that I and F_i are constructively given by expressions of the form shown in Definition 6.

As in the case of the verification of lossy channel systems we define a reduction of the above verification problem to the problem of disproving some first-order formula.

Given the verification problem above, we define a translation of an extended finite state machine M into a set Φ_M of formulae of the first-order predicate logic as follows. The vocabulary of Φ_M consists of

constant $\underline{0}$;
unary functional symbol s;
binary functional sysmbol $plus$
n-ary predicate symbol R;

For an integer $l \in \mathbb{Z}^+$ we define its term translation $l \mapsto \tilde{l}$ inductively: $\tilde{0} = \underline{0}$, $\widetilde{k+1} = s(\tilde{k})$. For the state $\gamma = (l_1, \dots, l_n)$ define its translation as $\tilde{\gamma} = (\tilde{l_1}, \dots, \tilde{l_n})$. For example, the state $(2, 3, 0)$ is translated to $(s(s(\underline{0})), s(s(s(\underline{0}))), \underline{0})$. Effectively given upwards closed sets of states are translated into n-tuples of non-ground in general terms. For example, the upward-closed set of states $\{(x, 1, y) \mid x \geq 1, y \geq 2\}$ is translated into a tuple $(s(x), s(\underline{0}), s(s(y)))$. Formally, for an upward closed set of states $S = \{(x_1, \dots, x_n) \mid \wedge_{i=1\dots n}(x_i \sim_i c_i)\}$, its term translation $\tilde{S}$ is defined as a n-tuple of terms $(t_1, \dots, t_n)$ where

$t_i = s^{(c_i)}(\underline{0})$ if $\sim_i$ is $=$, or
$t_i = s^{(c_i)}(x_i)$ if $\sim_i$ is $\geq$.

Further, we define a translation of guarded transitions. A guard of the form $A \cdot \bar{x} \geq \bar{c}$ (for the matrices with unit columns) can be represented as a conjunction $\bigwedge_i Cond_i$ of conditions of the form $\Sigma_{j \in J} x_j \geq c$, where $J \subseteq \{1, \dots, n\}$ and $c \in \mathbb{Z}^+$. The translation of the expression $\Sigma_{j \in J} x_j$, is defined inductively as follows.

$tr(\Sigma_{j\in J}x_j) = plus(x_i, tr(\Sigma_{j\in J-\{i\}}x_j))$ for $i \in J$, and
$tr(\Sigma_{j\in\{i\}}x_j) = x_i$

The translation is extended further to the conditions: $tr(\Sigma_{j\in J}x_j \geq c) = \exists z(tr(\Sigma_{j\in J}x_j) = s^{(c)}(z))$, where z is a fresh variable; and to the guards: $tr(G(\bar{x})) = \bigwedge tr(Cond_i)$ for $G(\bar{x}) = \bigwedge Cond_i$.

The transitions $T(\bar{x}, \bar{x}')$ allowed in the definition of the EFSM above can be represented as the set of updates of individual variables. For $i = 1, \ldots, n$ we have either

$x_i' = c$ for $c \in \mathbb{Z}^+$, or;
$x_i' = \Sigma_{j\in J}x_j + c$, where $J \subseteq \{1, \ldots, n\}$ and $c \in \mathbb{Z}$ (notice that c may be a negative here).

For each of the variables update define its term translation as a *pair* of terms $\langle\tau, \tau'\rangle$, where, τ and τ' represent, intuitively, the values of the corresponding variable before and after transition:

$tr(x_i' = c) = \langle x_i, s^{(c)}(\underline{0})\rangle$
$tr(x_i' = \Sigma_{j\in J}x_j + c) = \langle x_i, s^{(c)}(tr(\Sigma_{j\in J}x_j))\rangle$ if $c \in \mathbb{Z}^+$;
$tr(x_i' = \Sigma_{j\in J}x_j + c) = \langle s^{(-c)}(x_i), tr(\Sigma_{j\in J}x_j)\rangle$ if c is negative;

As before, the predicate R is used to specify the reachable global states.

Now we are ready to present the set of formulae Φ_M describing the reachability for an extended finite state machine $M = \langle X, I, T\rangle$. Let Φ_M be the following set of formulae which we assume to be universally closed:

$R(\tilde{I})$;
$(tr(G(\bar{x})) \wedge R(\tau_1, \ldots, \tau_n)) \to R(\tau_1', \ldots, \tau_n')$ for every $G(\bar{x}) \to T(\bar{x}, \bar{x}')$ in T with $\langle\tau_i, \tau_i'\rangle, i = 1, \ldots, n$ being terms translations of individual variable updates in $T(\bar{x}, \bar{x}')$;
$(plus(0, y) = y) \wedge (plus(i(x), y) = i(plus(x, y)))$.

The adequacy of the translation is given by the following

Proposition 2. *If the above verification problem has a negative answer, that is there is an initial state $s \in I$ and an unsafe state $f \in \cup_{i=1,\ldots,k}F_i$ such that f is reachable from s in M then $\Phi_M \vdash \vee_i \exists \bar{x}_i \tilde{F}_i(\bar{x}_i)$*

Proof is by straightforward induction on the length of transition sequences. □

It follows that finding any model for $\phi_M \wedge \neg(\vee_i \exists \bar{x}_i \tilde{F}_i(\bar{x}_i))$ provides a positive answer for the above verification problem. Unlike the case of the verification of lossy channel systems, we do not claim completeness of the method, so, there is no guarantee that there exists a finite model even if the protocol is safe. We conjecture, though, that *relative completeness* of the proposed method with respect to the methods of [7] holds. The method has turned out to be practically efficent. Using the finite model finder Mace4[17] we have verified [14] in this way all the parameterized cache coherence protocols from [6,7], including Synapse N+1, MESI, MOESI, Firefly, Berkeley, Illinois, Xerox PARC Dragon and Futurebus+, and MSI protocol from [8]. The results of experiments are presented in Section 4.

4 Experimental Results

In the experiments we used the finite model finder Mace4[17] within the package Prover9-Mace4, Version 0.5, December 2007. It is not the latest available version, but it provides with convenient GUI for both the theorem prover and the finite model finder. The system configuration used in the experiments: Microsoft Windows XP Professional, Version 2002, Intel(R) Core(TM)2 Duo CPU, T7100 @ 1.8Ghz 1.79Ghz, 1.00 GB of RAM. The time measurements are done by Mace4 itself, upon completion of the model search it communicates the CPU time used. The table below lists the parameterized/infinite state protocols together with the references and shows the time it took Mace4 to find a countermodel and verify a safety property. The time shown is an average of 10 attempts.

Protocol	Reference	Time
ABP*	[1]	0.93s
MSI	[8]	0.01s
MESI	[7]	0.03s
MOESI	[7]	0.05s
Firefly	[7]	0.03s
Synapse N+1	[7]	0.01s
Illinois	[7]	0.03s
Berkeley	[7]	0.03s
Dragon	[7]	0.05s
Futurebus+	[6]	1.14s

* ABP = Alternating Bit Protocol

In all cases the conjunctions of *all* safety properties specified for the protocols in the corresponding references are verified.

5 Discussion

The classes of verification tasks we tackle in this paper are well-known and have been previously considered in the literature. In [1] the decidability of safety verification for lossy channel systems has been established and an efficient algorithm, based on backward symbolic reachability was proposed. The verification of of parameterized cache coherence protocols has attracted much attention, too - this is practically important class of non-trivial protocols, amenable to automated verification at the behaviour level [18,9,7,10,11,16]. The verification method we propose in this paper differs from previously known ones by championing a pure *reductionist* approach: given a verification task, reduce it to a question in first-order classical logic, which then can be tackled by existing automated procedures. Notice, that in [10,11] the verification of parameterized cache coherence protocols was reduced to the theorem proving in fragments of first-order temporal logic, which in practical terms has proved to be much less efficient than the approach we advocate here. In a sense, the results we present here support that theorem *disproving* in classical logic can be more efficient for verification (at least, for particular classes of protocols) than theorem proving in stronger temporal logics.

We focused in this paper on the verification of safety properties. If the safety property does not hold, the finite model finders can not help, instead, in our "reachability as deducibility" approach the automated theorem provers can be used to establish the violation of the property. Indeed, during the numerious experiments we observed that for incorrect protocols, the proofs produced by Prover9 [17] provide enough information to extract the execution traces violating the properties. The systematic study of relevant questions is yet to be done.

The idea that computation can be modeled faithfully by derivation in some logic has a long tradition in formal approaches to security, starting as early as in work on BAN logic [3]. Very recently, in [12], the verification method based on modeling the execution of security protocols by derivation in first-order logic and finding countermodels was proposed. This method uses the same idea as we present here, but applied to a different class of protocols. A further difference is that in [12] a specialized model building procedure is proposed, while we insist that reduction-based approach using generic finite model building can already be practically efficient and sufficient.

The reductionist approach presented in this paper has also a potential advantage being a *modular*. Any progress in finite model finding procedures can be incorporated immediately in the verification procedure. Perhaps, the most interesting question to address in future work is whether the use of *infinite* model finders [2], that is procedures searching for finite presentation of infinite models, will increase the verifying power of the method.

Acknowledgements. The author is grateful to Michael Fisher for many helpful suggestions on this paper.

References

1. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Information and Computation 127(2), 91–101 (1996)
2. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building. Applied Logic Series, vol. 31. Kluwer, Dordrecht (2004)
3. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Transactions on Computer Systems 8, 18–36 (1990)
4. Cheng, K.-T., Krishnakumar, A.S.: Automatic generation of cunstional vectors using extended finite state machine model. ACM Transactions on Design Automation of Electronic Systems 1(1), 57–79 (1996)
5. Courcelle, B.: On constructing obstruction sets of words. Bulletin of the EATCS (44), 178–185 (1991)
6. Delzanno, G.: Verification of consistency protocols via infinite-state symbolic model checking, a case study. In: Proc. of FORTE/PSTV, pp. 171–188 (2000)
7. Delzanno, G.: Constraint-based Verification of Parametrized Cache Coherence Protocols. Formal Methods in System Design 23(3), 257–301 (2003)
8. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
9. Esparza, J., Finkel, A., Mayr, R.: On the Verification of Broadcast Protocols. In: Proc. 14th IEEE Symp. Logic in Computer Science (LICS), pp. 352–359. IEEE CS Press, Los Alamitos (1999)

10. Fisher, M., Lisitsa, A.: Deductive Verification of Cache Coherence Protocols. In: Proceedings of the 3rd Workshop on Automated Verification of Critical Systems, AVoCS 2003, Southampton, UK, April 2-3, pp. 177–186 (2003), Technical Report DSSE-TR-2003-2
11. Fisher, M., Konev, B., Lisitsa, A.: Practical Infinite-state Verification with Temporal Reasoning. In: Verification of Infinite State Systems and Security. NATO Security through Science Series: Information and Communication, vol. 1. IOS Press, Amsterdam (January 2006)
12. Guttman, J.: Security Theorems via Model Theory, arXiv:0911.2036
13. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. 2, 326–336 (1952)
14. Lisitsa, A.: Verification via Countermodel Finding (2009), http://www.csc.liv.ac.uk/~alexei/countermodel
15. Lisitsa, A.: Reachability as deducibility, finite countermodels and verification. In: Proceedings of AVOCS 2009, pp. 241–243 (2009)
16. Lisitsa, A., Nemytykh, A.: Reachability Analisys in Verification via Supercompilation. In: Proceedings of the Satellite Workshops of DTL 2007, Part 2, vol. 45, pp. 53–67. TUCS General Publication (June 2007)
17. McCune, W.: Prover9 and Mace4, http://www.cs.unm.edu/~mccune/mace4/
18. Pong, F., Dubois, M.: Verification techniques for cache coherence protocols. ACM Computing Surveys 29(1), 82–126 (1997)
19. Robinson, A., Voronkov, A. (eds.): Handbook of Automated Reasoning, vol. I, II. Elsevier/MIT Press (2001)
20. Roychoudhury, A., Ramakrishnan, I.V.: Inductively Verifying Invariant Properties of Parameterized Systems. Automated Software Engineering 11, 101–139 (2004)

LTL Can Be More Succinct

Kamal Lodaya and A V Sreejith

The Institute of Mathematical Sciences
Chennai 600113, India

Abstract. It is well known that modelchecking and satisfiability of Linear Temporal Logic (LTL) are PSPACE-complete. Wolper showed that with grammar operators, this result can be extended to increase the expressiveness of the logic to all regular languages. Other ways of extending the expressiveness of LTL using modular and group modalities have been explored by Baziramwabo, McKenzie and Thérien, which are expressively complete for regular languages recognized by solvable monoids and for all regular languages, respectively. In all the papers mentioned, the numeric constants used in the modalities are in unary notation. We show that in some cases (such as the modular and symmetric group modalities and for threshold counting) we can use numeric constants in binary notation, and still maintain the PSPACE upper bound. Adding modulo counting to LTL[F] (with just the unary future modality) already makes the logic PSPACE-hard. We also consider a restricted logic which allows only the modulo counting of length from the beginning of the word. Its satisfiability is Σ_3^P-complete.

1 Introduction

In this theoretical paper, we consider the extension of LTL to count the number of times a proposition holds modulo n. (More generally, in a recursive syntax, we can count formulas which themselves can have counting subformulas.)

There are many such extensions: Wolper used operators based on right-linear grammars [21], Emerson and Clarke developed the μ-calculus [2]. Henriksen and Thiagarajan's dynamic LTL [8] is an extension based on ideas from process logic [6]. Harel and Sherman had used operators based on automata for PDL [7]. Another extension has propositional quantification [4], but its model checking complexity is nonelementary [22]. More recently we have PSL/Sugar, and Vardi narrates [19] how regular expressions proved to be more successful than finite automata as far as designers in industry were concerned. Baziramwabo et al [1] explicitly have countably many MOD_n^k operators for their logic LTL+MOD. In work concurrent with ours, Laroussine, Meyer and Petonnet have introduced threshold counting [11].

Wolper's grammars, Harel and Sherman's automata, Henriksen and Thiagarajan's regular expressions, all use in effect a unary notation to express n. Hence stating properties using a large n is cumbersome. Consider a model describing properties of a circuit (which works very fast) interleaved with events which take place at regular intervals of time, which can be thought of as happening over very long stretches of the model.

Our first main theorem is that the PSPACE upper bound holds even when we use binary notation to represent the counting, and this can be carried all the way to a logic LTL+SYM, derived from Baziramwabo et al [1], which generalizes LTL+MOD to computation in the symmetric groups S_n. Thus we improve on the model checking procedure developed by Serre for LTL+MOD [15], which gives an EXPSPACE upper bound for formulas in binary notation. Unlike Serre, we do not use alternating automata but ordinary NFA and the standard "formula automaton" construction in our decision procedure.

The word "succinct" in the title of our paper is used in this simple programming sense of being able to use exponentially succinct notation. There are more sophisticated ways in which succinctness appears in temporal logics, which we do not address. A complexity theorist might say that we improve the known complexity of our logic from pseudo-polynomial space to polynomial space.

We have next a technical result showing that the logic LTL[F]+MOD is already PSPACE-hard. Since LTL[F] is NP-complete, this shows that modulo counting is powerful.

So we look to weakening the modulo counting. This is done by only allowing the modulo counting of lengths (rather than the number of times a formula holds). We show that the satisfiability problem of this logic, which we call LTL[F]+LEN, is exactly at Σ_3^P, the third level of the polynomial-time hierarchy, again irrespective of whether we use unary or binary notation.

We do not know if our work will make any impact on verification [2,16,20], since practitioners already know that a binary counter is an inexpensive addition to a modelchecking procedure. We think the finer analysis is of some theoretical interest.

2 Counting and Group Extensions of LTL

2.1 Modulo Counting

We begin by extending the LTL syntax with threshold and modulo counting, and specialization of the latter to length counting. Generalization to computation in an arbitrary symmetric group following Baziramwabo, McKenzie and Thérien [1] is described in the next subsection.

$$\begin{aligned}
&\delta ::= \#\alpha \mid \delta_1 + \delta_2 \mid \delta_1 - \delta_2 \mid c\delta,\ c \in \mathbb{N} \\
&\phi ::= \delta \sim t \mid \delta \equiv r(\mod q),\ q, r, t \in \mathbb{N},\ q \geq 2,\ \sim \in \{<, =, >, \neq, \leq, \geq\} \\
&\alpha ::= p \in \mathit{Prop} \mid \phi \mid \neg\, \alpha \mid \alpha\ \vee\ \beta \mid \mathsf{X}\alpha \mid \alpha\ \mathsf{U}\ \beta
\end{aligned}$$

As usual $\mathsf{F}\alpha$ abbreviates $\mathit{true}\mathsf{U}\alpha$ and $\mathsf{G}\alpha$ is $\neg\mathsf{F}\neg\alpha$. We will use the "length" ℓ to abbreviate $\#\mathit{true}$.

We denote by LTL+MOD the logic whose syntax we defined above. LTL[F]+MOD is a restriction where the U modality is not allowed and threshold counting $\delta \sim t$ is not allowed. (Since we will give a lower bound result, we keep the logic weak and only allow modulo counting.) We also use notation such as LTL[F]+MOD(q) when the counting is restricted to the modulo divisor q. The constants c, q, r, t above are given in binary. Our lower bounds continue to hold even when they are given in unary.

By further restricting the subterm $\#\alpha$ in the δ terms to be ℓ only, we get the logic LTL[F]+LEN which can only count lengths rather than occurrences of propositions or formulae. (We could similarly define LTL[X,U]+LEN.)

We denote by PROP+LEN the logic obtained by removing even the F modality from the syntax of LTL[F]+LEN, so we have propositional logic (interpreted over a word) with some length counting operations.

The semantics for LTL is given by a finite state sequence (or word) M over the alphabet $\wp(Prop)$. Our results also hold for the usual semantics over infinite words, but some of the examples are more sensible with finite words, so we will stick to that in the paper and point out how the arguments need to be changed for infinite words.

$M, i \models p$ iff $p \in M(i)$
$M, i \models \mathsf{X}\alpha$ iff $M, i+1 \models \alpha$
$M, i \models \alpha \ \mathsf{U}\ \beta$ iff for some $m \geq i : M, m \models \beta$
and for all $i \leq l < m : M, l \models \alpha$

For the counting terms, the interpretation of $\#\alpha$ at the index i in the word M is given by the cardinality of the set $\{1 \leq l \leq i \mid M, l \models \alpha\}$. The arithmetic operations in the syntax of δ are then well defined. Other definitions follow, for example:

$M, i \models \delta \equiv r(\mod q)$ iff the cardinality associated with δ at i in M leaves a remainder r when divided by q.

Even length words can be expressed in LTL[F]+LEN by $\mathsf{FG}(\ell \equiv 0(\mod 2))$. On the other hand an even number of occurrences of the holding of a proposition p requires an LTL[F]+MOD formula: $\mathsf{FG}(\#p \equiv 0(\mod 2))$.

The satisfiability problem for a formula α checks if a word model satisfying it exists, and the model checking problem for a rooted transition system (or Kripke structure) $K = (S, \rightarrow, L, s_0)$ and a formula α checks whether all runs of the transition system are models of α.

Variants: We count from the beginning of the word upto and including the present point where the formula is being evaluated. Supposing we needed the number of occurrences of the formula α from the present, before we hit β, to be divisible by q. We could write this using a disjunction of q possibilities, where the present count of $\alpha \equiv i(\mod q)$ and the count at β is also congruent to $i(\mod q)$.

We are assured by Baziramwabo et al [1] that LTL[X,U]+MOD is expressively complete for the logic FO+MOD of Straubing, Thérien and Thomas [18], so we stick to their simple syntax. In the appendix, we adapt an argument of Straubing [17] to show that the corresponding logic LTL[X,U]+LEN is expressively complete for a logic FO[Reg] also defined by Straubing. Thus the counting extensions we have introduced are related to others defined in a different context.

Laroussinie, Meyer and Petonnet [11] introduce counting in the future by indexed modalities, such as $\alpha \mathsf{U}_{\delta=t}\beta$, which counts t occurrences of δ from the present, maintaining the invariant α, until a future occurrence of β. This is equivalent to an LTL formula which is exponential in the size of the given formula, since t is written in binary, but expressively within first order logic FO.

2.2 Group Extension

Now we follow Baziramwabo, McKenzie and Thérien [1] to generalize the modulo counting to a kind of computation in symmetric groups. Our syntax above is extended to allow

$$\phi ::= \#_G(\alpha_1, \ldots, \alpha_k) = h,\ h \in G$$

For the semantics, let us define $\Gamma(M, l) = g_j$ if $M, l \models \neg\alpha_1 \wedge \ldots \neg\alpha_{j-1} \wedge \alpha_j$ for $1 \leq j \leq k$. Also define $\Gamma(M, l) = 1$ (the identity element) if none of the formulae $\alpha_1, \ldots, \alpha_k$ hold at position l. Then:

$$M, i \models \#_G(\alpha_1, \ldots, \alpha_k) = h \text{ iff } (\Pi_{l=0}^{i} \Gamma(M, l)) = h$$

This generalizes the modulo counting we were doing earlier, which can be thought of as working with cyclic groups.

The groups G used in the formulae are symmetric groups specified by their generators. This extension is called LTL+SYM.

For instance, we could specify the symmetric group S_5 (shown in Figure 1) using a syntax such as

```
group S5(5) generators (2 3 4 5 1), (2 1 3 4 5)
```

which specifies a permutation group named S5 with two generators defined as permutations of the elements $(1, 2, 3, 4, 5)$ mapping these elements to the values shown. In general we define a group named G with permutations over the set $\{1, \ldots, n\}, n \geq 2$ and generators $g_1, \ldots, g_k$. Any group can be embedded in a symmetric group [9], but while using symmetric groups the group operations are implicit.

Notice that h in the syntax above is a group element, not necessarily a generator of the group. As with modulo counting, we can have a more succinct syntax by representing h using binary notation. Using the generators is also a succinct way of representing groups (see below for a standard argument). For instance, the symmetric group S_n has $n!$ elements, but can be generated by 2 generators(as shown in example) each generator being a permutation on n elements. The analogue while doing modulo counting is to use binary notation to specify the numbers r and q.

Proposition 1. *Any group has a generating set of logarithmic size.*

Proof. Let G be a group. For an $H \subseteq G$, we denote by $\langle H \rangle$ the group generated by the elements H. Take an element $g_0 \in G$. Let $H_0 = \{g_0\}$. If $\langle H_0 \rangle \neq G$, take g_1 from $G \backslash \langle H_0 \rangle$, and call $H_0 \cup \{g_1\}$ as H_1. Continue doing this until you find an

H_k such that $\langle H_k \rangle = G$. We prove that $\forall i \leq k : \ |\langle H_{i+1} \rangle| \geq 2 \times |\langle H_i \rangle|$. Observe that since $g_{i+1} \notin \langle H_i \rangle$, it implies $g_{i+1} \langle H_i \rangle \cap \langle H_i \rangle = \phi$. Also $|g_{i+1}.\langle H_i \rangle| = |\langle H_i \rangle|$. But $g_{i+1}.\langle H_i \rangle \cup \langle H_i \rangle \subseteq \langle H_{i+1} \rangle$. Therefore $|\langle H_{i+1} \rangle| \geq 2 \times |\langle H_i \rangle|$. Hence $\langle H_{log|G|} \rangle = G$. □

The picture below shows the symmetric group S_n (for $n = 5$) as the transition structure of an automaton. The language accepted can be defined by the formula $F\ (\mathsf{X} false \wedge \#_{S5}(a, b) = (12 \ldots n))$ where the specification of $\mathsf{S5}$ with generators was shown earlier.

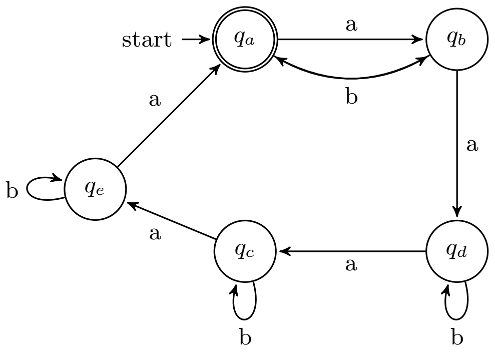


Fig. 1. An automaton representing the symmetric group S_5

3 Succinctness Comes Easy

Our first main theorem shows that the upper bound for LTL satisfiability can be extended to include the modulo and group counting computations, even when specified in binary.

Theorem 1. *If an $LTL[\mathsf{X}, \mathsf{U}]$+SYM formula α_0 is satisfiable then there exists a satisfying model of size exponential in α_0 (even using binary notation for the formula).*

Proof. The Fischer-Ladner closure of a formula α_0 [5] is constructed as usual, where we add the following clauses:

1. The closure of $\#\alpha \equiv r(\mod q)$ includes α and also has $\#\alpha \equiv s(\mod q)$ for every s from 0 to $q - 1$. (Notice that only one of these can be true at a state.)
2. The closure of $\#\alpha \sim t$ includes α, $\#\alpha > t$ and also has $\#\alpha = c$ for every $c \leq t$.
3. The closure of $\#_G(\alpha_1, \ldots, \alpha_k) = h$ includes $\alpha_1, \ldots, \alpha_k$ and also contains the formulae $\#_G(\alpha_1, \ldots, \alpha_k) = h'$ for every element h' of the group. (Only one of these can be true at a state.)

Observe that with binary notation, the closure of a formula α_0 can be exponential in the size of α_0, unlike the usual linear size for LTL, since the constants r, q and h are written in binary notation. A state of the tableau or formula automaton which we will construct is a maximal consistent subset of formulae from the closure of α_0. However, only one of the potentially exponentially many formulae of the form $\#\alpha \equiv s(\mod q)$, $0 \leq s < q$; or of the form $\#\alpha = c$, $0 \leq c \leq t$, and $\#\alpha > t$; or of the form $\#_G(\alpha_1, \ldots, \alpha_k) = h$, $h \in G$; can consistently hold. So a state is also exponential in the size of α_0. Here is a formal argument, using induction on structure of α, that the set of states of the formula automaton M_α is $2^{O(|\alpha|)}$. We denote by $|q|$ and $|G|$ for the input size (binary notation) and by S_α the number of states in M_α.

1. $\alpha = p \in P$. This is trivial.
2. $\alpha = \beta \vee \gamma$. $S_\alpha = S_\beta \times S_\gamma \leq 2^{O(|\beta|+|\gamma|)}$ (By IH)
3. $\alpha = \neg\beta$. This is just change of final states in M_β.
4. $\alpha = \beta \mathsf{U} \gamma$. $S_\alpha = S_\beta \times S_\gamma \leq 2^{O(|\beta|+|\gamma|)}$
5. $\alpha = \#\beta \equiv r(\mod q)$. Since any atom can have only one formula of this kind, $S_\alpha = S_\beta \times q \leq 2^{O(|\beta|+|q|)}$
6. $\alpha = \#\beta \sim t$. Since any atom can have only one formula of this kind, $S_\alpha = S_\beta \times (t+1) \leq 2^{O(|\beta|+|t|)}$
7. $\alpha = \#_G(\alpha_1, \ldots, \alpha_k) = h$. Again any atom can have only one formula of this kind, $S_\alpha = S_{\alpha_1} \times \cdots \times S_{\alpha_k} \times card(G) \leq 2^{O(|\alpha_1|+\cdots+|\alpha_k|+|G|)}$. □

Corollary 1. *$LTL[\mathsf{X}, \mathsf{U}]+SYM$ satisfiability is in* PSPACE*(using binary notation for the syntax).*

Proof. Since the formula automaton has exponentially many states, each state as well as the transition relation can be represented in polynomial space. By using moduli in binary and group generators, a state can be updated along a transition relation in polynomial time. Now we can guess and verify an accepting path in PSPACE. □

Corollary 2. *The complexity of the model checking problem of $LTL[\mathsf{X},\mathsf{U}]+SYM$ is* NLOGSPACE *in the size of the model and* PSPACE *in the size of the formula.*

Proof. Let α_0 be a formula in $LTL[\mathsf{X}, \mathsf{U}]+SYM$ and K a Kripke structure. Theorem 1 shows that for a formula $\neg\alpha_0$ there is an exponential size formula automaton $M_{\neg\alpha_0}$. Verifying $K \models \alpha_0$ is equivalent to checking whether the intersection of the languages corresponding to K and $M_{\neg\alpha_0}$ is nonempty. This can be done by a nondeterministic algorithm which uses space logarithmic in the size of both the models. Since $M_{\neg\alpha_0}$ is exponentially larger than α_0 we get the upper bounds in the statement of the theorem, using Savitch's theorem. The lower bounds are already known for LTL [16]. □

We note that these arguments are not affected by whether we consider finite or infinite word models.

3.1 But Modulo Counting Is Hard

Next we consider the logic LTL[F]+MOD. It can express properties which can be expressed by LTL but not by LTL[F], for example $\mathsf{G}(p \iff \ell \equiv 1(\mod 2))$ expresses alternating occurrences of p and $\neg p$. Our next result shows that the satisfiability problem for LTL[F]+MOD, even with unary notation, is PSPACE-hard.

Theorem 2. *The satisfiability problem for LTL[F]+MOD(2) is* PSPACE-*hard, even with the modulo formulae restricted to counting propositions.*

Proof. Since the satisfiability problem for LTL[X,F] is PSPACE-hard [16], it is sufficient to give a polynomial-sized translation of the modality $\mathsf{X}\alpha$ using counting modulo 2. This is done by introducing two new propositions p^E_α and p^O_α for each such formula, and enforcing the constraints below. Let *EvenPos* abbreviate $\ell \equiv 0 \mod 2$ and *OddPos* abbreviate $\ell \equiv 1 \mod 2$.

$$\mathsf{G}(\alpha \iff ((EvenPos \supset p^E_\alpha) \wedge (OddPos \supset p^O_\alpha)))$$

$$\mathsf{G}((EvenPos \supset \#p^E_\alpha \equiv 0 \mod 2) \wedge (OddPos \supset \#p^O_\alpha \equiv 0 \mod 2))$$

Consider p^E_α. Its count has to be an even number at every even position. Since the count increases by one if even positions satisfy α, it has to increase by one at the preceding odd position. So at an *odd* position, $\mathsf{X}\alpha$ holds precisely when the count of p^E_α is odd. Symmetrically, at an *even* position, $\mathsf{X}\alpha$ holds precisely when the count of p^O_α is odd. So we can replace an occurrence of $\mathsf{X}\alpha$ by the formula

$$(EvenPos \supset \#p^O_\alpha \equiv 1 \mod 2) \wedge (OddPos \supset \#p^E_\alpha \equiv 1 \mod 2).$$

Since α is used only once in the translation, this gives a blowup of the occurrence of $\mathsf{X}\alpha$ by a constant factor. With one such translation for every X modality, the reduction is linear.

No threshold counting formulas $\ell \sim t$ are used in this reduction, as required in the definition of the syntax. □

4 Length Modulo Counting

We now consider the weaker counting formulae $\ell \equiv r(\mod q)$, where ℓ abbreviates $\#true$. So we can only count lengths rather than propositions, which was something we needed in the PSPACE-hardness proof in the previous section.

Note that the language of alternating propositions p and $\neg p$ is in LTL[F]+LEN. It is known [16,3,12] that a satisfiable formula in LTL[F] has a polynomial sized model. Unfortunately LTL[F]+LEN does not satisfy a polynomial model property. Let p_i be distinct primes (in unary notation) in the following formula:

$$\mathsf{F}((\ell \equiv 0(\mod p_1)) \wedge (\ell \equiv 0(\mod p_2)) \wedge \cdots \wedge (\ell \equiv 0(\mod p_n))).$$

Any model which satisfies this formula will be of length at least the product of the primes, which is $\geq 2^n$. We show that the satisfiability problem of LTL[F]+LEN is in Σ^P_3, the third level of the polynomial-time hierarchy.

We give a couple of technical lemmas concerning the logic PROP+LEN which will be crucial to our arguments later.

Lemma 1. *Let α be a PROP+LEN formula. Then the following are equivalent.*

1. $(\forall w, |w| = n \implies \exists k \leq n : w, k \models \alpha)$
2. $(\exists k \leq n, \forall w : |w| = n \implies w, k \models \alpha)$

Proof. $(2 \implies 1)$: This is trivial.
$(1 \implies 2)$: Assume that the hypothesis is true but the claim is false. Let $S = \{w \mid |w| = n\}$. Pick a $w \in S$. By the hypothesis $\exists i \leq n : (w, i) \vDash \alpha$ and we can assume that there exists some $w' \in S$ such that $(w', i) \nvDash \alpha$. If this is not true then we have a witness i, such that $\forall w \in S : (w, i) \vDash \alpha$. Let u_i be the state at the i^{th} location of w'. Replace the i^{th} state in w by u_i without changing any other state in w. Call this new word w''. Now $(w'', i) \nvDash \alpha$. Again by the hypothesis, $\exists j \leq n : (w'', j) \vDash \alpha$. By the same argument given above, $\exists w''' : (w''', j) \nvDash \alpha$. We can replace the j^{th} state of w'' by the the j^{th} state from w''' which makes the resultant word not satisfy α at the j^{th} location. We can continue doing the above procedure. Since n is finite after some finite occurence of the above procedure, we will get a word v such that $\forall k \leq n : (v, k) \nvDash \alpha$. But this implies the hypothesis is wrong and hence a contradiction. □

Our next result is the following. Given a PROP+LEN formula α and two numbers m, n in binary, the problem *BlockSAT* is to check whether there exists a model M of size $m + n$ such that $M, m \models \mathsf{G}\alpha$.

Lemma 2. *BlockSAT can be checked in Π_2^P.*

Proof. The algorithm takes as input a PROP+LEN formula α, along with two numbers m, n in binary. Observe that since n is in binary we cannot guess the entire model. The algorithm needs to check whether there exists a model w satisfying α at all points between m and $m + n$, in other words, whether $\exists w : \forall k : m \leq k \leq m + n, |w| = n \wedge w, k \models \alpha$. Take the complement of this statement, which is $\forall w, |w| = n \implies \exists k : m \leq k \leq m + 1, w, k \models \neg\alpha$. By the previous Lemma 1 we can check this condition by a Σ_2^P machine. Hence *BlockSAT* can be verified by a Π_2^P machine. □

4.1 Succinct Length Modulo Counting Can Be Easier

We show that satisfiability of LTL[F]+LEN can be checked in Σ_3^P, showing that this restriction does buy us something.

Before proceeding into an algorithm, we need to introduce a few definitions. Let α be a formula over a set of propositions P, $SubF(\alpha)$ its set of future subformulae, $prd(\alpha)$ the product over all elements of the set $\{n \mid \delta \equiv r \mod n$ is a subformula of $\alpha\}$.

Let M be a model. We define *witness index* in M for α as $\{max\{j \mid M, j \models \mathsf{F}\beta\} \mid F\beta \in SubF(\alpha)$ and $\exists i : M, i \models \beta\}$. A state at a witness index is called a *witness state.* We say $F\beta$ is witnessed at i if $i = max\{j \mid M, j \models \mathsf{F}\beta\}$. Call all states other than witness states of M as *pad states* of M for α.

We define a model M to be *normal* for α if between any two witness states of M (for α) there are at most $prd(\alpha)$ number of pad states. We claim that if α is satisfiable then it is satisfiable in a normal model.

A normal model of α will be of size $\leq |SubF(\alpha)| \times prd(\alpha)$, which is of size exponential in α. So guessing the normal model is too expensive, but we can guess the witness states (the indices and propositions true at these states), which are polynomial, verify whether the F requirements are satisfied there, and verify if there are enough pad states to fill the gap between the witness states. We will argue that we can use a Π_2 oracle to verify the latter part. The proof is given below.

Theorem 3. *Modelchecking and satisfiability of LTL*[F]*+LEN can be checked in* Σ_3^P *(with binary notation).*

Proof. First of all we observe that modelchecking $M \models \alpha$ reduces to the satisfiability of a formula $\phi_M \supset \alpha$ using a standard construction (for example, see [13]).

Now let α be satisfiable. We guess the following and use it to verify whether there exists a normal word satisfying these guesses.

1. Guess k indices (positions), $u_1 < u_2 < ... < u_k$, where $k \leq |SubF(\alpha)|$ and $\forall i, u_i \leq prd(\alpha)$.
2. Guess the propositions true in the states at these k indices.
3. Guess the propositions true at the start state (if already not guessed).
4. For each of the k indices guess the set of $\mathsf{F}\beta \in SubF(\alpha)$ which are witnessed there. Let the conjunction of all formulae witnessed at u_j be called β_j. (Certain future formulae need not be true in any state in the word.)

We need to verify that there exists a word model M which is normal for α and which satisfies the guesses. Observe that the positions $1, u_1 + 1, ..., u_{k-1} + 1$ in M should all satisfy certain G requirements (the model starts from index 1). If we have guessed that a future formula $\mathsf{F}\beta_0$ is not satisfied in the model, then the entire word should also satisfy its negation $\mathsf{G}\neg\beta_0$. Similary at state $u_i + 1$, $\mathsf{G} \wedge_{j=0}^{i} \neg\beta_j$ should be true.

To verify that all the F, G requirements are satisfied at the witness states (the u_i indices we guessed), we start verifying from the last state u_k. All modalities can be stripped away and verified against the propositions true at this state and the location of the state. To verify $\mathsf{F}\beta_i$ at an intermediate state, we know that only those beyond the current index have been verified in future witness states. We reduce the verification of the rest to that of a pure PROP+LEN formula by making passes from the innermost subformulae outward, which can be done in polynomial time. A more formal description of this algorithm would need to keep track of the formulae satisfied and not satisfied in the future at every witness state.

To verify that the pad states between two witness states satisfy the current set of $\mathsf{G}\beta$ requirements, we need to check that the pad states should satisfy their conjunction $\bigwedge \beta$. Stripping modalities which have been verified, this is a pure PROP+LEN formula γ. What we now need to verify is that at position $u_i + 1$, we want a word of length $u_{i+1} - u_i - 1$ which satisfies $\mathsf{G}\gamma$. From Lemma 2, we see that this is the *BlockSAT* property, checkable in Π_2^P. The algorithm we have described is an NP procedure which uses a Π_2^P oracle and hence is in Σ_3^P. □

This algorithm needs to be somewhat modified when considering satisfiability for infinite word models. First of all, we observe that we can restrict ourselves to considering "lasso" models where we have a finite prefix followed by an infinite loop, and for convenience in dealing with modulo counts, we can take the length of the loop body to be a multiple of $prd(\alpha)$. The procedure described above essentially works for the prefix part of the model, but we have to devise a further procedure which handles the requirements in the loop part of the model. Since the key to this procedure is the verification of *BlockSAT*, which remains unchanged, the extended procedure for satisfiability over infinite word models can also be carried out in Σ_3^P and Theorem 3 continues to hold.

4.2 Satisfiability of Length Modulo Counting Is Hard

In this section we show that the satisfiability problem for $LTL[\mathsf{F}]$+LEN is Σ_3^P-hard, even if we use unary notation and finite word models. We denote by $\beta[\phi/p]$ the formula got by replacing all occurences of the proposition p by ϕ.

Let QBF_3 be the set of all quantified boolean formulae which starts with an existential block of quantifiers followed by a universal block of quantifiers which are then followed by an existential block of quantifiers. Checking whether a QBF_3 formula is true is Σ_3^P-complete. We reduce from evaluation of QBF_3 formulae to satisfiability of our logic.

Theorem 4. *Satisfiability for $LTL[\mathsf{F}]$+LEN is hard for Σ_3^P, even if unary notation is used for the syntax.*

Proof. Let us take a formula β with three levels of alternation and which starts with an existential block.

$$\beta = \exists x_1, ..., x_k \forall y_1, ..., y_l \exists z_1, ..., z_m B(x_1, ..., x_k, y_1, ..., y_l, z_1, ..., z_m)$$

We now give a satisifability-preserving $LTL[\mathsf{F}]$+LEN formula $\widehat{\beta}$ (which can have constants in unary notation) such that β in Σ_3^P-SAT iff $\exists w, (w,1) \vDash \widehat{\beta}$.

Take the first l prime numbers $p_1, ..., p_l$. Replace the y_js by $\ell \equiv 0(\mod p_j)$. Let the resultant formula be called α. We give the formula $\widehat{\beta}$ below. It is a formula over the x and z propositions.

$$\hat{\beta} = \mathsf{G}(\ B[\ell \equiv 0(\mod p_j)/y_j]) \wedge \mathsf{F}(\wedge_{j=1}^{l} \ell \equiv 0(\mod p_j)) \wedge \bigwedge_{i=1}^{k} (\mathsf{G}x_i \vee \mathsf{G}\neg x_i)$$

Thanks to the prime number thorem we do not have to search too far (By the prime number theorem, asymptotically there are l primes within $l \log l$ and hence finding them can be done in polynomial time.) for the primes, and primality testing can be done in polynomial time.

Suppose the quantified boolean formula β is satisfiable. Then there is an assignment v to the x_is which makes the Π_2 subformula ($\forall\exists$ part) true. Consider the formula $\gamma = \beta[v(x_i)/x_i]$. We can represent an assignment to the y_js by an l length bit vector. There are 2^l different bit vectors possible. For each bit vector s we can obtain the formula γ_s, by substituting the y_js with the values from s. But since β is satisfiable, each of the γ_ss are satisfiable. Hence for all these formulae there is a satisfying assignment $Z^s : [m] \to \{0,1\}$ to the variables z_r, for $r = 1, m$.

We are going to construct a word model M which will satisfy $\widehat{\beta}$. Take its length to be $n \geq \Pi_{j=1}^{l} p_j$ so that the future requirement is satisfied (2^{nd} formula). In every state of the word, let the proposition x_i take the value $v(x_i)$. Now we define at state t the valuation of $z_r, r = 1, m$, as follows. Let s be the bitstring represented by ($t \mod p_1 = 0, t \mod p_2 = 0, ..., t \mod p_l = 0$). Set the evaluation of z_r in the t^{th} state of M to be $Z^s(r)$.

Once we do this for all $t \leq n$, we find that $M, t \vDash \beta[\ell \equiv 0(\mod p_j)/y_j][v(x_i)/x_i]$. And because $n \geq \Pi_{j=1}^{l} p_j$ we have $M, 1 \vDash \widehat{\beta}$. We have thus shown that there exists a word model satisfying $\widehat{\beta}$.

For the converse, suppose there is a word model M of length n which satisfies $\widehat{\beta}$. Then $n \geq \Pi_{j=1}^{l} p_j$. Set a valuation v for the x's as $v(x_i) = true$ iff $M, 1 \vDash x_i$. We have to now show that the formula $\gamma = \beta[v(x_i)/x_i]$ is satisfiable for all 2^l assignments to the y_js. That is, for all 2^l assignments to the y_j's there is an assignment to the z_rs which make γ true. Suppose s is a bitstring of length l representing an arbitrary assignment to the y_j's. Take a $t \leq n$, such that s equals the bitstring ($t\ mod\ p_1 = 0, t\ mod\ p_2 = 0, ..., t\ mod\ p_l = 0$). Such a t exists because n is long enough. Let $Z^s(r)$ be the valuation of the z_r in the t^{th} state of M. This assignment to z_r makes the formula α true when the y_j's are assigned according to s. Hence β is satisfiable. $\square$

5 Discussion

We observed in this paper that when LTL is extended with threshold and modulo counting, it does not matter if the specification of the thresholds and moduli is in succinct notation. More generally this holds for computation within a finite symmetric group. This seems to have escaped the notice of verification researchers until now.

Are there other families of automata, where a "standard" enumeration of their states and transitions can be represented in logarithmic notation, and for which the PSPACE bound will continue to hold? We also ask how far these ideas can be extended for pushdown systems.

A patent weakness is that LTL+SYM specifications are far from perspicuous, but we look to demonstrate an idea, and it will take examples from practice to provide useful patterns for the more expressive logic using specification of group properties.

Acknowledgment. We would like to thank N.R. Aravind for suggesting a simplification of the proof of Theorem 4.

References

1. Baziramwabo, A., McKenzie, P., Thérien, D.: Modular temporal logic. In: Proc. 14th LICS, p. 344. IEEE, Los Alamitos (1999)
2. Emerson, E.A., Clarke Jr., E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comp. Program. 2, 241–266 (1982)
3. Etessami, K., Vardi, M.Y., Wilke, T.: First-order logic with two variables and unary temporal logic. Inf. Comput. 179(2), 279–295 (2002)
4. Fine, K.: Propositional quantifiers in modal logic. Theoria 36, 336–346 (1970)
5. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comp. Syst. Sci. 18(2), 194–211 (1979)
6. Harel, D., Kozen, D.C., Parikh, R.J.: Process logic: expressiveness, decidability, completeness. J. Comp. Syst. Sci. 25, 144–170 (1982)
7. Harel, D., Sherman, R.: Dynamic logic of flowcharts. Inf. Contr. 64(1-3), 119–135 (1985)
8. Henriksen, J.G., Thiagarajan, P.S.: Dynamic linear time temporal logic. Ann. Pure Appl. Logic 96(1-3), 187–207 (1999)
9. Herstein, I.N.: Topics in Algebra. Blaisdell (1964)
10. Kamp, J.A.W.: Tense logic and the theory of linear order. PhD thesis, University of California, Los Angeles (1968)
11. Laroussinie, F., Meyer, A., Petonnet, E.: Counting LTL. In: Proc. Time (to appear 2010)
12. Ono, H., Nakamura, A.: On the size of refutation kripke models for some linear modal and tense logics. Studia Logica: An International Journal for Symbolic Logic 39(4), 325–333 (1980)
13. Schnoebelen, P.: The complexity of temporal logic model checking. In: Proc. 4th Adv. Modal Log., Toulouse, pp. 393–436. King's College (2003)
14. Schützenberger, M.-P.: On finite monoids having only trivial subgroups. Inf. Contr. 8, 190–194 (1965)
15. Serre, O.: Vectorial languages and linear temporal logic. Theoret. Comp. Sci. 310(1-3), 79–116 (2004)
16. Sistla, A.P., Clarke Jr., E.M.: The complexity of propositional linear temporal logics. J. ACM 32(3), 733–749 (1985)
17. Straubing, H.: Finite Automata, Formal Logic, and Circuit Complexity. Birkhäuser, Basel (1994)
18. Straubing, H., Thérien, D., Thomas, W.: Regular languages defined with generalized quantifiers. Inf. Comput. 118(3), 289–301 (1995)
19. Vardi, M.Y.: From philosophical to industrial logics. In: Ramanujam, R., Sarukkai, S. (eds.) Logic and Its Applications. LNCS (LNAI), vol. 5378, pp. 89–115. Springer, Heidelberg (2009)
20. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. 115(1), 1–37 (1994)
21. Wolper, P.: Temporal logic can be more expressive. Inf. Contr. 56(1-2), 72–99 (1983)
22. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proc. 24th Found. Comp. Sci., Tucson, pp. 185–194. IEEE, Los Alamitos (1983)

A Expressiveness of LTL[X,U]+LEN

In this appendix, we show that the logic LTL[X,U]+LEN is as expressive as first order logic with regular numerical predicates $FO[Reg]$. This is standard first order logic on word models, with binary predicate symbols for order and equality $x < y$ and $x = y$, and unary predicate symbols $Q_a(x)$ and $x \equiv r(\mod q)$, for every letter a in the alphabet and for $r, q \in \mathbb{N}, q \geq 2$. For more details on this logic, see Straubing's book [17].

First, a lemma. Its converse also holds but we do not need it.

Lemma 3 (Straubing). *Let $L \subseteq A^*$ be a regular language. If $L \in FO[Reg]$ then L is recognized by a morphism η_L to a monoid M, such that $\forall t > 0$, every semigroup contained in $\eta_L(A^t)$ is aperiodic.*

Theorem 5. *A property of words is expressible in FO[Reg] iff it is expressible in LTL[X,U]+LEN.*

Proof. There is a standard translation from an LTL[X,U]+LEN formula, which is essentially the definition of the semantics of the modalities of LTL[X,U]+LEN using an FO[Reg] formula. To prove the other direction, we use the lemma above and the same proof strategy as in Straubing's book [17]. Using the lemma, given a morphism $\eta : A^* \to M$ and a language $L = \eta^{-1}(X)$ such that $X \subseteq M$ and $\forall t > 0$, every semigroup contained in $\eta_L(A^t)$ is aperiodic, we have to show that L can be expressed by an LTL[X,U]+LEN formula.

Consider the following sequence which contains finitely many distinct sets.

$$\eta(A), \eta(A^2), ...,$$

and hence $\exists k, r > 0 : \forall p \geq k, \eta(A^p) = \eta(A^{p+r})$ and hence for a p which is a multiple of r, we have $\eta(A^p) = \eta((A^p)^+) = S$ is a semigroup of M. From the property of η, S is aperiodic. Let $B = A^p$ and let us define $\beta : B^* \to S^1$ by setting $\forall b \in B^* : \beta(b) = \eta(b)$. Now

$$L = \bigcup_{0 \leq |w| < p} wL_w$$

where

$$L_w = \{u \in (A^p)^* : wu \in L\}.$$

Assume that each of the L_w can be expressed by an LTL[X,U]+LEN formula ϕ_w. Let $\phi_w[k]$ be a formula which accepts words whose length is shifted by k. This is inductively defined and the only nontrivial clause is $(\ell \equiv i(\mod p))[k] = \ell \equiv i + k(\mod p)$.

If $w = a_1 a_2 ... a_k$, wL_w can be defined by the following formula.

$$a_1 \wedge \bigwedge_{i=1}^{k-1} \mathsf{X}^i a_{i+1} \wedge \mathsf{X}^k \phi_w[k]$$

Taking some finite union over such languages we will be able to express L by an LTL[X,U]+LEN formula.

It remains to show how we can obtain the formula for each language L_w. Consider a word $v \in B^*$. It belongs to L_w iff

$$\beta(v) \in \{m \in S^1 : m.\eta(w) \in X\}$$

Thus L_w considered as a subset of B^* is recognized by an aperiodic monoid. By the results of Schützenberger [14] and Kamp [10] we know that any language accepted by a homomorphism to an aperiodic monoid can be expressed by an LTL formula and hence L_w can be expressed by an LTL formula ψ over the alphabet B. We give an inductive construction τ from an LTL formula over B^* to an LTL[X,U]+LEN formula over A^* as follows. Let $b = a_1 ... a_p \in B^*$. Then

$$\tau(b) = (\ell \equiv 0 \mod p) \wedge a_1 \wedge \bigwedge_{i=1}^{p-1} \mathsf{X}^i a_{i+1}$$

$$\tau(\neg\alpha) = \neg\tau(\alpha)$$

$$\tau(\alpha_1 \wedge \alpha_2) = \tau(\alpha_1) \wedge \tau(\alpha_2)$$

$$\tau(\mathsf{X}\alpha) = (\ell \equiv 0 \mod p) \wedge \mathsf{X}^{p+1}\tau(\alpha)$$

$$\tau(\alpha_1 \mathsf{U} \alpha_2) = ((\ell \equiv 0 \mod p) \implies \tau(\alpha_1))\mathsf{U}\tau(\ell \equiv 0 \mod p \wedge \alpha_2)$$

Thus $\tau(\psi)$ defines L_w. $\square$

Automatic Generation of History-Based Access Control from Information Flow Specification

Yoshiaki Takata[1] and Hiroyuki Seki[2]

[1] Kochi University of Technology, Tosayamada, Kochi 782-8502, Japan
takata.yoshiaki@kochi-tech.ac.jp
[2] Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan
seki@is.naist.jp

Abstract. This paper proposes a method for automatically inserting check statements for access control into a given recursive program according to a given security specification. A history-based access control (HBAC) is assumed as the access control model. A security specification is given in terms of information flow. We say that a program π satisfies a specification Γ if π is type-safe when we consider each security class in Γ as a type. We first define the problem as the one to insert check statements into a given program π to obtain a program π' that is type-safe for a given specification Γ. This type system is sound in the sense that if a program π is type-safe for a specification Γ, then π has noninterference property for Γ. Next, the problem is shown to be co-NP-hard and we propose an algorithm for solving the problem. The paper also reports experimental results based on our implemented system and shows that the proposed method can work within reasonable time.

1 Introduction

A language-based access control is a promising approach to preventing untrusted modules from accessing confidential data. Stack inspection provided by the Java virtual machine (JVM) and the common language runtime (CLR) is a typical successful example. In a language-based access control, a statement for runtime permission check such as `checkPermission` of JVM (abbreviated as check statement) is placed just before a statement accessing confidential data. Permissions to be checked at each check statement are usually set manually; however, an inappropriate setting of permissions causes either security flaw or unnecessary abortion of execution. Therefore a systematic method for generating appropriate check statements is desired.

This paper assumes a (shallow) history-based access control (HBAC) [1,9] as an access control model and proposes a method for automatically inserting check statements into a given recursive program according to a given security specification. In this paper, a security specification is given in terms of information flow [6]: a specification is an assignment of a security class (e.g. top_secret, confidential, and unclassified) to each input and output variable (or channel) of a program. The set of security classes is assumed to be a finite semilattice. Since one of the main purposes of access control (especially, mandatory access control) is to prevent undesirable information leak by deciding which user (or process) can have access to which resource, it is natural to give a security specification using the concept of information flow.

We say that a program π satisfies a specification Γ if π is type-safe when we consider each security class in Γ as a type. This type system is sound in the sense that if a program π is type-safe for a specification Γ, then π has noninterference property for Γ (Theorem 1). Noninterference property is a widely-used semantic (but undecidable in general) criterion for the confidentiality. Intuitively, a program π has noninterference property for a specification Γ if the content of a variable v in π is not affected by the content of any variable in π whose security class specified by Γ is higher than v.

Next, we define the problem as follows: for a given program π including zero or more check statements with permissions to be checked unspecified and a specification Γ, specify permissions to be checked at each check statements in π so that the resultant program is type-safe for Γ. This definition does not lose generality since check statements are usually placed just before access statements and it can be easily done automatically. Then, the problem is shown to be co-NP-hard (Theorem 2) and we propose an algorithm for solving the problem using a model checking method for pushdown systems (PDS). The idea of the proposed method is simple. If we find an execution trace that violates a specification by analyzing the PDS derived from an HBAC program, then we add appropriate permissions to be checked at a check statement nearest to the undesirable access to remove this execution trace. However, adding new permissions may introduce a new violation of the specification (known as a covert channel). This covert channel can be avoided by carefully designed fixpoint operation given in Section 4.2. The paper also reports experimental results based on our implemented system and shows that the proposed method can generate check statements within reasonable time.

Related Work. Static analysis has been widely studied for programs with stack inspection (or stack-based access control, SBAC) [4,8,11,12,15] and for HBAC [21]. Pottier et al. [17] and Besson et al. [5] proposed type systems such that type safety implies no violation against SBAC. Information flow analysis has a long history stemming from [6] and has been extensively studied for recursive programs using type systems [10,13,14,20]. Information flow analysis has been extended to SBAC [3] and HBAC [2]. The latter showed interesting phenomena that check statements themselves may cause implicit information flow. The work of [16] regarded dynamic permissions as a security class, considered that check statements represent a security specification, and proposed a dynamic control mechanism of information flow. To the authors' knowledge, however, this paper is the first one to deal with the problem of automatic generation of access control statements from a specification of information flow.

The rest of this paper is organized as follows. In Section 2, the syntax and operational semantics of an HBAC program is defined. In Section 3, we define a security specification as well as the notion of type-safety by deriving a pushdown system (PDS) from a given program π and a specification Γ. This PDS in effect constitutes a type system for π under Γ. Also it is shown that type-safety implies noninterference property. In Section 4, the problem is defined and an algorithm for solving the problem by PDS model checking is given, followed by the experimental results. Due to space limitation, formal proofs are omitted. Consult [18] for the proofs.

2 Input Program

2.1 Syntax

A program consists of a set *Func* of functions, a set *In* of input channels, a set *Out* of output channels, and a set *Prm* of permissions. The body of each function is an element of *cseq* defined by the following BNF specification.

$$\begin{aligned}
\mathit{cseq} &::= \mathit{cmd} \mid \mathit{cmd}\,\texttt{;}\,\mathit{cseq} \\
\mathit{cmd} &::= \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{cseq}\ \texttt{fi} \\
&\quad\mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{cseq}\ \texttt{else}\ \mathit{cseq}\ \texttt{fi} \\
&\quad\mid \mathit{out}\ \texttt{:=}\ x \quad\mid\quad x\ \texttt{:=}\ \mathit{in} \quad\mid\quad x\ \texttt{:=}\ \mathit{exp} \\
&\quad\mid x\ \texttt{:=}\ \mathit{func}\texttt{(}\mathit{exp}\texttt{,}\dots\texttt{,}\mathit{exp}\texttt{)} \\
&\quad\mid \texttt{check[}P\texttt{]} \\
\mathit{exp} &::= c \mid x \mid \theta\texttt{(}\mathit{exp}\texttt{,}\dots\texttt{,}\mathit{exp}\texttt{)}
\end{aligned}$$

In the above BNF specification, *out*, *in*, x, c, P, and θ represent an output channel, an input channel, a variable, a constant, a subset of permissions, and a built-in operator, respectively. The return value of a function f is stored in a special variable named ret_f.

A program interacts with its environment only through the input and output channels; the starting function (main function) has no arguments or return value.

2.2 Access Control Mechanism

HBAC is proposed to resolve the weakness of the stack inspection such that it ignores the execution history of functions of which execution is finished [1,2,21]. (See [1] for the design principles of HBAC.)

A subset of permissions is assigned to each function f before runtime. We call the subset the *static permission set* of f, denoted as $SP(f)$. For example, if the set *Prm* of permissions is $\{r, w, d\}$, which are permissions to read, write, and delete a file, respectively, and a function f has a right to read and write the file, then $SP(f) = \{r, w\}$. The runtime system controls another subset of permissions called the *current permission set*. When a program starts, the current permission set is initialized as the static permission set of the starting function. The current permission set is updated when a function is called. The updated current permission set is the intersection of the old current permission set and the static permission set of the callee function; that is, every permission not assigned to the callee function is removed from the current permission set.

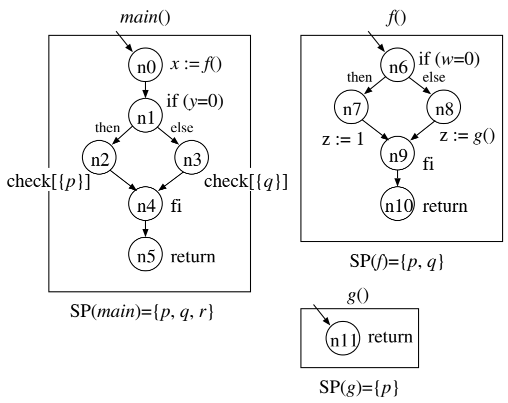


Fig. 1. Program π_0

When the control reaches a check statement `check[`P`]`, the execution is continued if the current permission set includes all permissions in P, and the execution is aborted otherwise.

Hereafter we assume that a program is represented as a control flow graph (such as Figure 1) in which conditional branches are well-nested.

Example 1. Consider a sample program π_0 in Figure 1. The transition of the call stack and the current permission set of π_0 is depicted in Figure 2.

When function g is called at program point n_8 in function f, permission q is removed from the current permission set. After that, when the control reaches n_3 in function *main*, the execution is aborted because q is not in the current permission set. Abortion at a check statement does not occur if function g is never called or the control never reaches n_3.

Since the current permission set is uniquely determined by the set of functions that have invoked so far, we assume that $\mathit{Prm} = \mathit{Func}$ and $\mathit{SP}(f) = \mathit{Prm} \setminus \{f\}$ for each $f \in \mathit{Func}$ without loss of generality[1]. That is, f remains in the current permission set if and only if f has never been invoked. For readability, we write p_f instead of f when it represents an element of Prm.

2.3 Operational Semantics

For a program π, we define a transition system M_π that represents the behavior of π. A configuration of M_π is a pair (σ, ξ), in which σ is a state of the input and output channels and ξ is a stack. A state of an output channel is a finite sequence of values

[1] In the original definition of HBAC programs [1,21], one can specify a *grant set* and an *accept set* for each function call statement, which enable more flexible control of the current permission set. We omit these parameters to simplify the problem. With grant and accept sets, the current permission set is not necessarily determined uniquely by the set of already invoked functions.

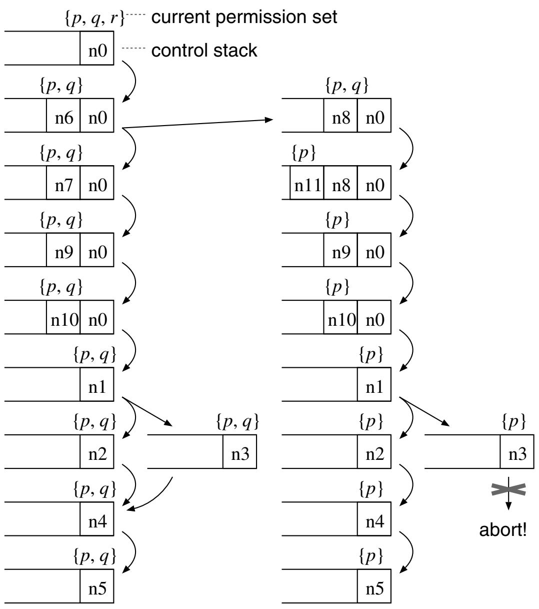


Fig. 2. Transition of the call stack and current permission set of π_0

that have been output so far. A state of an input channel is an infinite sequence of values that are going to be read out. A stack is a finite sequence of stack frames, and a stack frame is a triple $\langle n, \mu, C\rangle$ in which n is a program point, μ is a state of local variables (including formal parameters and the return value variable ret_f), and C is the current permission set. The leftmost stack frame of a stack is the stack top.

The transition relation $\Rrightarrow$ of M_π is the smallest relation that satisfies the following inference rules. As mentioned before, we assume that a program is represented as a control flow graph, and we write $n \to n'$ if there exists a control flow from a program point n to n'. The program statement of a program point n is denoted as $\lambda(n)$. For the end point n of a function, $\lambda(n) = \texttt{return}$. The formal parameter list of a function f is denoted as $\mathit{param}(f)$, and the initial program point of f is denoted as $IT(f)$. The concatenation of two sequences ξ_1 and ξ_2 is denoted as $\xi_1 : \xi_2$. The state in which every variable is undefined is denoted as $\perp$. For a state μ, $\mu[x \mapsto v]$ is the state that maps x to v and y to $\mu(y)$ for every $y \neq x$. We extend the domain of a state μ to expressions in the usual way, i.e., $\mu(\theta(e_1, \ldots, e_k)) = \theta(\mu(e_1), \ldots, \mu(e_k))$.

$$\frac{\lambda(n) = \mathit{out} := x, \quad n \to n', \quad \sigma' = \sigma[\mathit{out} \mapsto \sigma(\mathit{out}) : \mu(x)]}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma', \langle n', \mu, C\rangle : \xi)} \tag{1}$$

$$\frac{\begin{array}{l}\lambda(n) = x := \mathit{in}, \quad n \to n', \quad \sigma(\mathit{in}) = a : \zeta, \\ \mu' = \mu[x \mapsto a], \quad \sigma' = \sigma[\mathit{in} \mapsto \zeta]\end{array}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma', \langle n', \mu', C\rangle : \xi)} \tag{2}$$

$$\frac{\lambda(n) = x := e, \quad n \to n', \quad \mu' = \mu[x \mapsto \mu(e)]}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu', C\rangle : \xi)} \tag{3}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad param(f) = (x_1, \ldots, x_k),\\ \mu' = \bot[x_1 \mapsto \mu(e_1), \ldots, x_k \mapsto \mu(e_k)],\\ C' = C \cap SP(f)\end{array}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle IT(f), \mu', C'\rangle : \langle n, \mu, C\rangle : \xi)} \tag{4}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad \lambda(m) = \mathtt{return}, \quad n \to n',\\ \mu'' = \mu[x \mapsto \mu'(ret_f)]\end{array}}{(\sigma, \langle m, \mu', C'\rangle : \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu'', C'\rangle : \xi)} \tag{5}$$

$$\frac{\lambda(n) = \mathtt{check}[P], \quad P \subseteq C, \quad n \to n'}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \xi)} \tag{6}$$

$$\frac{\lambda(n) = \mathtt{if}\ e, \quad n \overset{\text{then}}{\to} n', \quad \mu(e) \neq \mathit{false}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \langle n, \mu, C\rangle : \xi)} \tag{7}$$

$$\frac{\lambda(n) = \mathtt{if}\ e, \quad n \overset{\text{else}}{\to} n', \quad \mu(e) = \mathit{false}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \langle n, \mu, C\rangle : \xi)} \tag{8}$$

$$\frac{\lambda(n) = \mathtt{fi}, \quad n \to n'}{(\sigma, \langle n, \mu, C\rangle : \langle m, \mu', C'\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \xi)} \tag{9}$$

Rule (4) means that when the control is at the program point n of a function call statement $x := f(e_1, \ldots, e_k)$, a new stack frame is pushed into the stack, the control moves to $IT(f)$, and the current values of $e_1, \ldots, e_k$ are assigned to the formal parameters of f. At the same time, the current permission set is updated to $C \cap SP(f)$. Rule (5) means that when the control is at the end point m of a function, the stack top is removed from the stack and the value of ret_f is returned to the caller. Rule (6) represents the transition for check statements. Rules (7) and (8) for `if` statements push a stack frame into the stack, which is inessential but keeps the soundness theorem (Theorem 1) simple.

An initial configuration of a program π is $(\sigma_0, \langle IT(f_0), \bot, SP(f_0)\rangle)$ where σ_0 is a state that maps every output channel to the empty sequence and f_0 is the main function. A stack frame $\langle n, \mu, C\rangle$ is said to be *reachable* if there exists a transitions $cnf_0 \Rrightarrow \cdots \Rrightarrow (\sigma, \langle n, \mu, C\rangle : \xi)$ for some initial configuration cnf_0 and some σ and ξ. The above sequence is called an *execution trace*. Similarly, a node n is *reachable* if there is a reachable stack frame $\langle n, \mu, C\rangle$ for some μ and C.

3 Information Flow Specification

An *information flow specification* is an assignment of security classes to the input and output channels.

We assume that the set of security classes, denoted as $\mathcal{SC}$, is an arbitrary finite semilattice partially ordered by a relation $\sqsubseteq$. The least element of $\mathcal{SC}$ is denoted as L (= Low), and the least upper bound of $a, b \in \mathcal{SC}$ is denoted as $a \sqcup b$.

A simple example of $\mathcal{SC}$ is $\{H, L\}$ (High and Low) such that $L \sqsubseteq H$. For this example, transmitting a value computed using a value from an H input channel into an L output channel is a violation of the information flow specification.

Absence of the violation of an information flow specification can formally be defined in terms of *noninterference*; i.e., no violation exists if values written to every L output channel do not change even when values read out from an H input channel change. However, it is well-known that noninterference is an undecidable property even if access control is absent. Therefore, we define an abstract system $M_\pi^\sharp$ from a program π and a specification Γ and define the notion of type safety by regarding each security class as a type. The soundness of this type system is guaranteed by Theorem 1, which states that type safety of $M_\pi^\sharp$ implies noninterference of π for Γ.

3.1 Derived Pushdown System and Type Safety

For a given program π and an information flow specification Γ, we define a transition system $M_\pi^\sharp$ as follows. Intuitively, $M_\pi^\sharp$ represents the behavior of a program that is the same as π except that each variable x keeps a security class instead of a value stored in x. Since the set $\mathcal{SC}$ of security classes is finite, $M_\pi^\sharp$ is a pushdown system (PDS), and we can compute the reachable set of stack frames of $M_\pi^\sharp$ [7].

A configuration of $M_\pi^\sharp$ is a stack. While a stack frame of M_π is a triple $\langle n, \mu, C\rangle$, that of $M_\pi^\sharp$ is $\langle n, sc, C\rangle$ in which sc is an assignment of security classes to local variables and permissions. The transition relation $\Rightarrow$ of $M_\pi^\sharp$ is the smallest relation that satisfies the following inference rules, in which $e^\sqcup$ is the expression obtained from an expression e by substituting $\sqcup$ for every built-in operator in e, and $\mathbb{L}$ is the assignment in which the security class of every variable and permission is L.

$$\frac{\lambda(n) = out := x, \quad n \to n'}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc, C\rangle : \xi} \tag{10}$$

$$\frac{\lambda(n) = x := in, \quad n \to n', \quad sc' = sc[x \mapsto \Gamma(in) \sqcup sc(\nu_{\mathrm{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \xi} \tag{11}$$

$$\frac{\lambda(n) = x := e, \quad n \to n', \quad sc' = sc[x \mapsto sc(e^\sqcup \sqcup \nu_{\mathrm{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \xi} \tag{12}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad param(f) = (x_1, \ldots, x_k), \\ C' = C \cap SP(f), \\ sc' = \mathbb{L}[x_1 \mapsto sc(e_1^\sqcup \sqcup \nu_{\mathrm{if}}), \ldots, x_k \mapsto sc(e_k^\sqcup \sqcup \nu_{\mathrm{if}}), \nu_{\mathrm{if}} \mapsto sc(\nu_{\mathrm{if}})] \\ \qquad\quad [p \mapsto sc(p \sqcup \nu_{\mathrm{if}}) \mid p \in C \setminus C'] \\ \qquad\quad [p \mapsto sc(p) \qquad\quad \mid p \in Prm \text{ and } p \notin C \setminus C']\end{array}}{\langle n, sc, C\rangle : \xi \Rightarrow \langle IT(f), sc', C'\rangle : \langle n, sc, C\rangle : \xi} \tag{13}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad \lambda(m) = \texttt{return}, \quad n \to n' \\ sc'' = sc[x \mapsto sc'(ret_f) \sqcup sc(\nu_{\mathrm{if}})][p \mapsto sc'(p) \mid p \in Prm]\end{array}}{\langle m, sc', C'\rangle : \langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc'', C'\rangle : \xi} \tag{14}$$

$$\frac{\lambda(n) = \texttt{check}[P],\ \ P \subseteq C,\ \ n \rightarrow n'}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc, C\rangle : \xi} \tag{15}$$

$$\frac{\lambda(n) = \texttt{if}\ e,\ \ (n \overset{\text{then}}{\rightarrow} n' \text{ or } n \overset{\text{else}}{\rightarrow} n'),\ \ sc' = sc[\nu_{\text{if}} \mapsto sc(e^{\sqcup} \sqcup \nu_{\text{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \langle n, sc, C\rangle : \xi} \tag{16}$$

$$\frac{\lambda(n) = \texttt{fi},\ \ n \rightarrow n',\ \ sc'' = sc[\nu_{\text{if}} \mapsto sc'(\nu_{\text{if}})]}{\langle n, sc, C\rangle : \langle m, sc', C'\rangle : \xi \Rightarrow \langle n', sc'', C\rangle : \xi} \tag{17}$$

An initial configuration and reachable stack frames of $M_\pi^\sharp$ are defined in the same way as M_π.

In the above definition of $M_\pi^\sharp$, each assignment statement is replaced with a calculation on security classes. For example, an assignment statement $z := x + y$ in π is replaced with $z := x \sqcup y$ in $M_\pi^\sharp$, because the security class of z after this assignment is the maximum of the security classes of x and y.

We use a special variable ν_{if} in $M_\pi^\sharp$ for representing the security class of implicit information flow [6,20]. The security class of ν_{if} increases at the beginning of a conditional branch, and the increase is canceled at the end of the branch (see Rules (16) and (17)). To save the security class of ν_{if} before a conditional branch, a new stack frame is pushed into the stack in Rule (16). In Rules (11) to (14), the security class of each updated variable becomes higher than or equal to the security class of implicit flow. In Rule (13), the security class of each permission removed from the current permission set is also updated.

We have to consider another kind of implicit flow caused by a check statement. For example, when the following compound statement is executed and $p \notin SP(f)$ for the callee function f, one can know whether $y = 0$ or not because if $y = 0$ then permission p is removed from the current permission set and the execution is aborted at the check statement.

$$\texttt{if}\ y = 0\ \texttt{then}\ x := f()\ \texttt{fi;}\ \texttt{check}[\{p\}]$$

In this case, the information on whether $y = 0$ or not flows into the current permission set, and then the information flows outside by the check statement (even when the execution is not aborted at the check statement). Hence we take the security class of each permission p into account as well as that of each variable. The security class of p represents the security class of information on whether or not p remains in the current permission set. Moreover, we consider that information on each permission contained in the argument of a check statement flows into an insecure output channel; that is, a type error exists if a permission p whose security class is not L is contained in the argument of a check statement (cf. type error E3 described below).

Type Error. We say that there exists a type error in $M_\pi^\sharp$ if there exists a reachable stack frame $\langle n, sc, C\rangle$ that satisfies any of the following conditions E1 to E4. If no type error exists in $M_\pi^\sharp$, then we say that π is *type-safe*.

E1) $\lambda(n) = out := x$ for an output channel out and $sc(x \sqcup \nu_{\text{if}}) \not\sqsubseteq \Gamma(out)$.
E2) $\lambda(n) = x := in$ for an input channel in and $sc(\nu_{\text{if}}) \not\sqsubseteq \Gamma(in)$.
E3) $\lambda(n) = \texttt{check}[P]$ and $sc(p) \neq L$ for some $p \in P$.
E4) $\lambda(n) = \texttt{check}[P]$ and $P \not\subseteq C$ and $sc(\nu_{\text{if}}) \neq L$.

E1 represents a situation in which a value of a security class higher than an output channel *out* is written to *out*. E2 represents an information leak through an input channel. We assume that an attacker can be aware of reading out a value from an input channel, and thus an implicit flow occurs if the reading out is performed in a branch of an `if`-statement. E3 and E4 represent an information leak through a check statement. E3 represents a situation in which information on the current permission set flows out. E4 represents a situation in which an attacker is aware of the current program point because of the abortion at the check statement.

Another kind of information flow may occur when a program does not terminate. For simplicity, we ignore this kind of information flow in this paper (termination insensitivity). Although we can modify our type-error detection method so that it becomes sound even for the above kind of information flow, the modified method may report more false positives because it considers that every loop may not terminate.

3.2 Soundness

The above type-error detection method using $M^\sharp_\pi$ is sound in the sense that π satisfies noninterference if $M^\sharp_\pi$ is type-safe (and if π always terminates). This soundness is guaranteed by Theorem 1 shown below. Note that in Item (2) (resp. (4)) of the theorem, each side of $\Rrightarrow^*$ (resp. $\Rightarrow^*$) is a stack with a single stack frame.

Theorem 1. *Let π be an HBAC program, Γ be a specification with $\mathcal{SC}$ as the set of security classes, n be a program point in π, sc be an assignment of security classes to variables and permissions, and C_1 and C_2 be subsets of permissions in π such that*

(1) there exists no type error in $M^\sharp_\pi$ if the initial configuration of $M^\sharp_\pi$ is either $\langle n, sc, C_1\rangle$ or $\langle n, sc, C_2\rangle$.

Assume the folloing three conditions hold for some n', σ_i, σ'_i, μ_i, μ'_i, C'_i $(i = 1, 2)$, sc', y, and τ.

(2) $(\sigma_i, \langle n, \mu_i, C_i\rangle) \Rrightarrow^* (\sigma'_i, \langle n', \mu'_i, C'_i\rangle)$ $(i = 1, 2)$.
(3) For every variable x, $sc(x) \sqsubseteq \tau$ implies $\mu_1(x) = \mu_2(x)$. For every $io \in In \cup Out$, $\Gamma(io) \sqsubseteq \tau$ implies $\sigma_1(io) = \sigma_2(io)$. For every permission p, $sc(p) \sqsubseteq \tau$ implies $p \in C_1 \Leftrightarrow p \in C_2$.
(4) For any sc' such that $\langle n, sc, C_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ $(i = 1$ or $2)$, $sc'(y) \sqsubseteq \tau$.

Then, the following two conditions hold.

(5) $\mu'_1(y) = \mu'_2(y)$ if y is a variable. $y \in C'_1 \Leftrightarrow y \in C'_2$ if y is a permission.
(6) For every $io \in In \cup Out$, $\Gamma(io) \sqsubseteq \tau$ implies $\sigma'_1(io) = \sigma'_2(io)$.

Proof (sketch). Let $\alpha_i = ((\sigma_i, \langle n, \mu_i, C_i\rangle) \Rrightarrow \cdots \Rrightarrow (\sigma'_i, \langle n', \mu'_i, C'_i\rangle))$ $(i = 1, 2)$. This theorem can be proved by induction on the length of α_1. □

4 Proposed Method

4.1 Permission-Check Statement Insertion Problem

Input. A program π and an information flow specification Γ. All check statements in π must be `check[∅]`.

Output. A type-safe program π' that is obtained from π by modifying the arguments of arbitrary number of check statements.

Example 2. Consider the program π_1 and the information flow specification (i.e. assignment of security classes to input and output channels) in Figure 3. If input channel *select* always gives a non-zero value at program points n_0 and n_5, then the execution trace of the program is (a prefix of) the following transition sequence of stacks, where the three components of each stack frame represent a program point, a state of variables and permissions, and the current permission set, respectively. The state of variables and permissions is a tuple of the security classes of variables x, y, ν_{if}, ret_f, and ret_g, and permissions p_f and p_g.

$$\begin{aligned}
&\langle n_0, (L, L, L, L, L, L, L), \{p_f, p_g\}\rangle \\
&\Rightarrow \langle n_1, (L, L, L, L, L, L, L), \{p_f, p_g\}\rangle \\
&\Rightarrow \langle n_3, (L, L, L, L, L, L, L), \{p_f, p_g\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{15}, (L, L, L, L, L, L, L), \{p_f\}\rangle : \langle n_3, (\ldots), \{\ldots\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{16}, (L, L, L, L, H, L, L), \{p_f\}\rangle : \langle n_3, (\ldots), \{\ldots\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_4, (L, H, L, L, L, L, L), \{p_f\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_5, (L, H, L, L, L, L, L), \{p_f\}\rangle \\
&\Rightarrow \langle n_6, (L, H, L, L, L, L, L), \{p_f\}\rangle \\
&\Rightarrow \langle n_7, (L, H, L, L, L, L, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_8, (L, H, L, L, L, L, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{11}, (L, H, L, L, L, L, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{12}, (L, H, L, L, L, L, L), \{p_f\}\rangle
\end{aligned}$$

In the above trace, a type error of E1 occurs in the statement $out_1 := y$ at n_8. To remove this error, we can add permission p_g to the argument of the check statement at n_7. After this addition, the execution along the above trace is aborted at n_7 since the current permission set $\{p_f\}$ at n_7 in that trace does not contain p_g, and the above error is removed. Moreover, this addition does not bring any other type errors of E3 or E4 because the security classes of p_g and ν_{if} at n_7 are L in every trace from n_0 to n_7.

Theorem 2. *The permission-check statement insertion problem is co-NP-hard.*

Proof (sketch). Reduction from 3UNSAT. For a given set $U = \{x_1, \ldots, x_n\}$ of variables and a set $C = \{c_1, \ldots, c_m\}$ of clauses over U, we construct a program π_2 shown in Figure 4. Variables x_{i0} and x_{i1} in π_2 represent the negative and positive

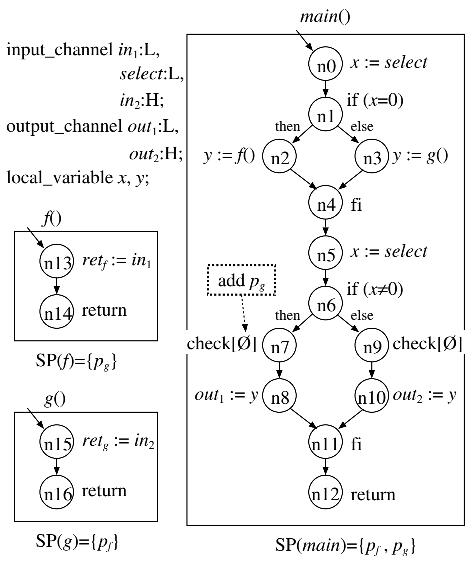


Fig. 3. Program π_1

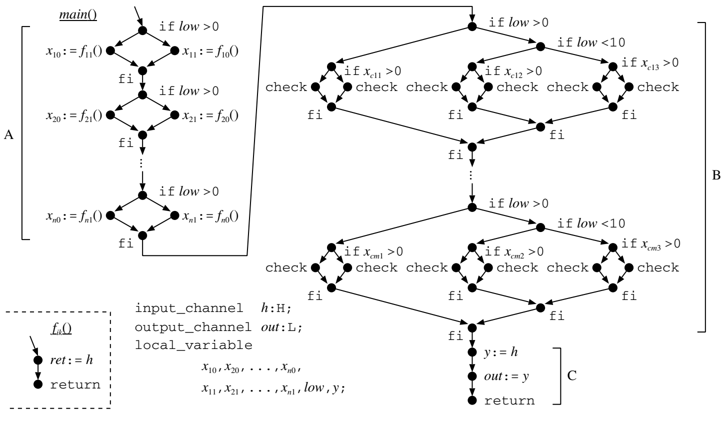


Fig. 4. Program π_2

literal of x_i, respectively, and the variables representing the three literals in c_j are denoted by $x_{c_{j1}}$, $x_{c_{j2}}$, and $x_{c_{j3}}$, respectively. In this program, there exists a path to the type-error statement in part C and $\nu_{\text{if}} \neq L$ at every check statement on the path if and only if C is satisfiable. Moreover, if C is unsatisfiable, then there is a setting of the arguments of check statements that never causes type errors and every path to part C is aborted at some check statement. □

4.2 Algorithm

We call a program point n a *check node* if $\lambda(n) = \texttt{check}[P]$ for some P. On the permission-check statement insertion problem, all we can do is to add permissions to the argument of check statements. When a program π' is obtained from π by adding permissions to the arguments of check statements, the transition relation of $M^\sharp_{\pi'}$ is a subset of that of $M^\sharp_\pi$, since the precondition of the inference rule (15) in Section 3.1 holds for $M^\sharp_\pi$ if it holds for $M^\sharp_{\pi'}$ and the other rules do not depend on check statements. Hence, we can design an algorithm for solving the problem as follows. For each reachable stack frame fr of $M^\sharp_\pi$ that causes a type error of E1 or E2, make fr unreachable by adding a permission to the argument of some check statement in each execution trace α from an initial configuration to fr. Let $\alpha = cnf_0 \Rightarrow \cdots \Rightarrow \langle n, sc, C\rangle : \xi \Rightarrow \cdots \Rightarrow fr : \xi'$ be such a trace where n is the check node whose argument is to be modified. If a permission $p \notin C$ is added to the argument of n, the execution will be aborted at n in α. So if we perform this modification for every α, then fr becomes unreachable. However, the above modification may introduce another type error of E3 or E4. Let us fix the node n for a while. The necessary and sufficient condition to avoid such a type error as a side effect is: $sc(p) = L$, $sc(\nu_{\text{if}}) = L$, and any other stack frame $\langle n, sc', C'\rangle$ for the same n for which p brings a type error (i.e. $sc'(p) \neq L$ or ($p \notin C'$ and $sc'(\nu_{\text{if}}) \neq L$)) is unreachable (by possibly modifying the argument of another check statement).

Based on the above observation, the algorithm consists of two phases:

(1) For each check node n, compute $SafeP(n)$, which is the set of all permissions that can be added to the argument of n without type error or with type error that can be removed by other check statements; ($SafeP(n)$ is formally defined in Phase (1) below.)

(2) For every type-error stack frame fr and a trace $\alpha = cnf_0 \Rightarrow \cdots \Rightarrow fr : \xi'$, find a configuration $\langle n, sc, C\rangle : \xi$ in α and a permission p such that n and p satisfy the condition mentioned in the previous paragraph, by using $SafeP(n)$. If the addition of p introduces a type error, then repeat Phase (2).

In the following, let cnf_0 be an initial configuration of $M^\sharp_\pi$, $top(fr : \xi) = fr$ be the function that answers the stack top, and $LP(sc) = \{\, p \mid sc(p) = L \,\}$ be the subset of permissions whose security class is L for a given sc.

Phase (1): Computation of *SafeP*(n) We define the following two inference rules (18) and (19).

$$\frac{\lambda(n) \neq \texttt{check[}P\texttt{]} \text{ or } sc(\nu_{\text{if}}) \neq L \text{ or } (SafeP(n) \cap LP(sc)) \setminus C = \emptyset}{\neg Stoppable(\langle n, sc, C\rangle)} \tag{18}$$

$$\frac{\begin{array}{l} cnf_0 \Rightarrow \ldots \Rightarrow cnf_\ell \Rightarrow \langle n, sc, C\rangle : \xi, \quad (sc(p) \neq L \text{ or} \\ (p \notin C,\ sc(\nu_{\text{if}}) \neq L)), \quad \neg Stoppable(top(cnf_i)) \text{ for } 0 \leq i \leq \ell \end{array}}{p \notin SafeP(n)} \tag{19}$$

Since every occurence of $\mathit{Stoppable}(\cdot)$ and $\mathit{SafeP}(\cdot)$ is negative, these two rules have the greatest fixpoints for these two predicates, which can be computed as follows. We say that a stack frame fr is stoppable if $\mathit{Stoppable}(\mathit{fr})$ holds during the computation.

(i) Let $\mathit{SafeP}(n) = \mathit{Prm}$ for each check node n (see Section 2.2 for Prm) and $M_\pi^{\sharp\prime}$ be $M_\pi^\sharp$ without any transitions.
(ii) Compute $\mathit{Stoppable}(\mathit{fr})$ for each stack frame fr of $M_\pi^\sharp$ by (18) according to the current $\mathit{SafeP}(\cdot)$, and add transitions to $M_\pi^{\sharp\prime}$ from each non-stoppable frame.
(iii) Compute $\mathit{SafeP}(n)$ for each check node n by (19) according to the current $\mathit{Stoppable}(\cdot)$. To do this, compute all reachable stack frames of $M_\pi^{\sharp\prime}$ based on a model checking method for pushdown systems. If a stack frame $\langle n, sc, C\rangle$ for a check node n is reachable, then remove every permission p satisfying the second precondition of (19) from $\mathit{SafeP}(n)$.
(iv) Repeat Steps (ii) and (iii) until no more change occurs.

Phase (2): Changing the arguments of check statements. Let $M_{\text{ex}}^\sharp$ be the pushdown system obtained from $M_\pi^\sharp$ by extending the stack frames to 4-tuples and substituting the following inference rules (20) and (21) for Rule (15) for check statements. The fourth component of a stack frame is a pair $\langle n, Q\rangle$ of a program point and a subset of permissions. This pair represents that $\mathit{top}(\mathit{cnf}) = \langle n, sc, C\rangle$ for some sc and C such that cnf is the last stoppable configuration on the execution trace to the current configuration and the execution will be actually aborted by adding an arbitrary element of Q to the argument of n.

$$\frac{\begin{array}{l}\lambda(n) = \mathtt{check}[P],\ \ P \subseteq C,\ \ n \to n',\ \ sc(\nu_{\text{if}}) = L,\\ Q = (\mathit{SafeP}(n) \cap \mathit{LP}(sc)) \setminus C \neq \emptyset\end{array}}{\langle n, sc, C, X\rangle : \xi \Rightarrow \langle n', sc, C, \langle n, Q\rangle\rangle : \xi} \tag{20}$$

$$\frac{\begin{array}{l}\lambda(n) = \mathtt{check}[P],\ \ P \subseteq C,\ \ n \to n',\ \ (sc(\nu_{\text{if}}) \neq L \text{ or}\\ \ (\mathit{SafeP}(n) \cap \mathit{LP}(sc)) \setminus C = \emptyset)\end{array}}{\langle n, sc, C, X\rangle : \xi \Rightarrow \langle n', sc, C, X\rangle : \xi} \tag{21}$$

By the following algorithm, the arguments of check statements are modified to remove type errors.

(i) Let the fourth component of the initial configuration of $M_{\text{ex}}^\sharp$ be $\perp$, and compute all reachable stack frames of $M_{\text{ex}}^\sharp$.
(ii) If a type-error stack frame $\langle n, sc, C, X\rangle$ is reachable and $X = \langle n', Q\rangle$, then add an arbitrary element of Q to the argument of the check statement at n'. Since any element of Q belongs to $\mathit{SafeP}(n)$ and can cause abortion at n', no backtracking is needed. If $X = \perp$, then notify a user that the given problem instance has no solution, and halt.
(iii) Repeat Steps (i) and (ii) until no more change occurs.

The algorithm finds one of the solutions if and only if the given problem instance has at least one solution.

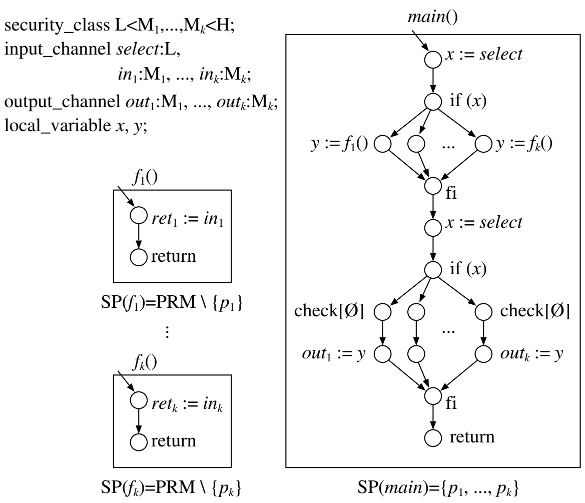


Fig. 5. Program $\pi_a(k)$

4.3 Experiment

We describe the results of an experiment in which a prototype implementation of the above algorithm is applied to the following two kinds of problem instances. Each input program is an extension of program π_1 in Figure 3 and represents a typical information flow control problem in which there are k pairs of input and output channels and a value from i-th input channel must not be written to the output channels other than i-th one.

(1) The input program $\pi_a(k)$ in Figure 5 has k functions f_i for $1 \leq i \leq k$, and the security class of the return value of function f_i is M_i. Security class M_i $(1 \leq i \leq k)$ satisfies $L \sqsubseteq M_i \sqsubseteq H$ and $M_i \not\sqsubseteq M_j$ and $M_j \not\sqsubseteq M_i$ for every $j \neq i$. Program $\pi_a(k)$ also has k check statements and k output channels, and the security class of each output channel out_i is M_i. Thus we have to modify the argument of each check statement so that the return value of f_i is written only to out_i.

(2) Program $\pi_b(k)$ is a program obtained from $\pi_a(k)$ by splitting the lower part of the main function into a separate function, which can be arbitrarily repeated by a tail call to itself.

Figure 6 shows the computation time for $\pi_a(k)$ and $\pi_b(k)$[2]. The computation time for $\pi_a(k)$ is approximately $O(k^2)$, and the time for $\pi_b(k)$ is approximately $O(k^3)$. Computation of reachable stack frames of $M^\sharp_{\text{ex}}$ is dominant in the proposed algorithm. We adopt an efficient method for computing the set of reachable stack frames described in [19, Section 4.4], whose computation time is approximately proportional to the number of reachable stack frames. The number of reachable stack frames for $\pi_a(k)$ and $\pi_b(k)$ is shown in Figure 7, which

[2] The prototype implementation is written in C (GCC 4.1.2). We use a computer with Intel Core 2 Duo 1.06 GHz, 2 GB RAM, and CentOS 5.3.

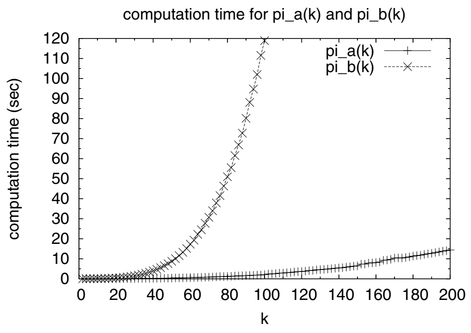


Fig. 6. Computation time for $\pi_a(k)$ and $\pi_b(k)$

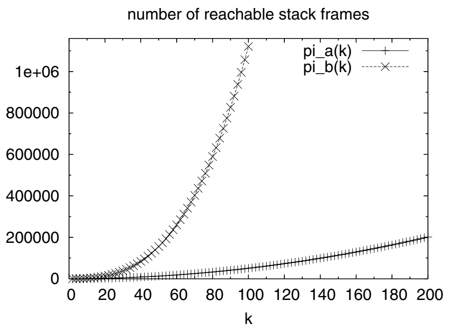


Fig. 7. The number of reachable stack frames for $\pi_a(k)$ and $\pi_b(k)$.

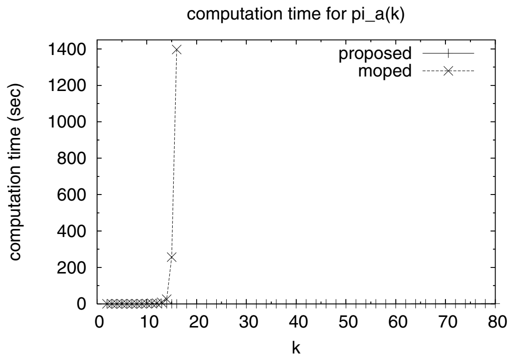


Fig. 8. Comparison between the proposed implementation and Moped

plots the same curves as Figure 6. Although Steps (i) and (ii) of Phase (2) of the algorithm are repeated if the addition to the argument of the check statement at n' in Step (ii) causes a new type error, for $\pi_a(k)$ and $\pi_b(k)$ Step (i) was required only twice. Moreover, the second computation of reachable stack frames needed much less time than the first one because the increase of the arguments of check statements much reduces the number of reachable stack frames. These observations suggest that the time complexity of the proposed algorithm is nearly the order of the number of reachable stack frames.

Comparison between the proposed implementation and Moped. The main part of the proposed algorithm is the computation of the set of reachable stack frames, and it can be performed using existing model checking tools for PDS. However, those are not optimized for analysis of HBAC programs and their suitability for the permission-check statement insertion problem is unknown. Hence we measured the computation time required by PDS model checking tool *Moped* (version 1.0.14) for computing the set of reachable stack frames of $\pi_a(k)$ (Figure 8).

While our implementation required at most two seconds when $k \leq 100$, the computation time of Moped rapidly increases, and it becomes more than several hours when $k \geq 17$. From these results, our implementation is more suitable for the permission-check statement insertion problem than Moped.

5 Conclusion

In this paper we studied on a problem to automatically insert permission-check statements for making a given program satisfy a given information flow specification. We showed that the problem is co-NP-hard. We also proposed an algorithm based on a model checking method of pushdown systems. Applying a prototype implementation to problem instances, we found that the complexity of the proposed algorithm is proportional to the number of reachable stack frames.

Future work includes the followings.

(1) A method for finding the optimal solution: Some problem instances have more than one solution, and the proposed algorithm does not necessarily answer an optimal one. We would like to investigate an algorithm to find the solution that minimize the total size of the argument of check statements.
(2) An algorithm for the original definition of HBAC programs: We would like to extend the proposed algorithm to the original definition of HBAC programs, where freedom of the static permission set of each function and the grant and accept set of each function call statement is imposed.

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In: Network & Distributed System Security Symp., pp. 107–121 (2003)
2. Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 27–48. Springer, Heidelberg (2005)
3. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. Journal of Functional Programming 5(2), 131–177 (2005)
4. Bartoletti, M., Degano, P., Ferrari, G.L.: Static analysis for stack inspection. In: ConCoord. ENTCS, vol. 54 (2001)
5. Besson, F., Blanc, T., Fournet, C., Gordon, A.D.: From stack inspection to access control: A security analysis for libraries. In: 17th IEEE CSFW, pp. 61–75 (2004)
6. Denning, D.E.: A lattice model of secure information flow. ACM Commun. 19(5), 236–243 (1976)
7. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model-checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
8. Esparza, J., Kučera, A., Schwoon, S.: Model-checking LTL with regular variations for pushdown systems. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 316–339. Springer, Heidelberg (2001)
9. Fong, P.W.: Access control by tracking shallow execution history. In: IEEE Symp. on Security & Privacy, pp. 43–55 (2004)

10. Heintze, N., Riecke, J.G.: The Slam calculus: Programming with secrecy and integrity. In: 25th ACM POPL, pp. 365–377 (1998)
11. Jensen, T., Le Métayer, D., Thorn, T.: Verification of control flow based security properties. In: IEEE Symp. on Security & Privacy, pp. 89–103 (1999)
12. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for Java. In: 17th ACM OOPSLA, pp. 359–372 (2002)
13. Leroy, X., Rouaix, F.: Security properties of typed applets. In: 25th ACM POPL, pp. 391–403 (1998)
14. Myers, A.C., Liskov, B.: Complete, safe information flow with decentralized labels. In: IEEE Symp. on Security & Privacy, pp. 186–197 (1998)
15. Nitta, N., Takata, Y., Seki, H.: An efficient security verification method for programs with stack inspection. In: 8th ACM CCS, pp. 68–77 (2001)
16. Pistoia, M., Banerjee, A., Naumann, D.A.: Beyond stack insepction: A unified acess-control and information-flow security model. In: IEEE Symp. on Security & Privacy, pp. 149–163 (2007)
17. Pottier, F., Skalka, C., Smith, S.: A systematic approach to access control. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 30–45. Springer, Heidelberg (2001)
18. Takata, Y., Seki, H.: Automatic generation of history-based access control from information flow specification. Tech. Rep. NAIST-IS-TR2010002, Nara Institute of Science and Technology (2010), `http://isw3.naist.jp/IS/TechReport/`
19. Takata, Y., Wang, J., Seki, H.: A formal model and its verification of history-based access control. IEICE Trans. on Information and Systems (Japanese Edition) J91-D(4), 847–858 (2008)
20. Volpano, D., Smith, G.: A type-based approach to program security. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 607–621. Springer, Heidelberg (1997)
21. Wang, J., Takata, Y., Seki, H.: HBAC: A model for history-based access control and its model checking. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 263–278. Springer, Heidelberg (2006)

Auxiliary Constructs for Proving Liveness in Compassion Discrete Systems*

Teng Long[1,2] and Wenhui Zhang[1]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] School of Information Science and Engineering
Graduate University of China Academy of Sciences, Beijing, China
{longteng,zwh}@ios.ac.cn

Abstract. For proving response properties in systems with compassion requirements, a deductive rule is introduced in [1]. In order to use the rule, auxiliary constructs are needed. They include helpful assertions and ranking functions defined on a well-founded domain. The work in [2] computes ranking functions for response properties in systems with justice requirements. This paper presents an approach which extends the work in [2] with compassion requirements. The approach is illustrated on two examples of sequential and concurrent programs.

1 Introduction

Model checking is a main verification technique for finite state systems, and has been successfully applied to proving the correctness of hardware and software designs. The concept of abstraction helps enhancing the applicability of model checking to infinite systems. Predicate abstraction [3,4,5], has been useful for the verification of safety properties in infinite systems. For the verification of liveness properties, ranking abstraction has been introduced in [6,7,8] recognizing that the usual state abstraction is often inadequate to capture liveness properties. Compassion requirements[1] are introduced into the abstract system so that the ranking abstraction preserves the liveness properties under consideration. One of the common features of these two methods is that we need to extract auxiliary constructs in order to make the methods successful in proving safety and liveness properties. In the former case, one needs to construct invariants and in the latter, one needs to construct ranking functions.

Our focus is on methods for computing ranking functions for proving liveness properties. Invisible ranking introduced in [9] is one such method for automatically generating helpful assertions and ranking functions for proving liveness

* Supported by the National Natural Science Foundation of China under Grant Nos. 60721061, 60833001, and the CAS Innovation Program.

[1] A compassion requirement is a pair of assertions requiring that in a computation, if the first assertion is satisfied infinitely often, then the second one must also be satisfied infinitely often.

properties in systems with justice requirements[2]. The method was then extended to handle a larger class of problems by relaxing restrictions requiring that the helpful assertions and ranking functions only depend on the local states of a process [10]. For proving liveness properties in systems with justice requirements, an approach is presented in [2] based on graph manipulation for generating helpful assertions and ranking functions.

Our approach presented in this paper extends that of [2] in order to be able to compute ranking functions for proving liveness properties in sequential and concurrent programs with compassion requirements. Our approach may as well be used for proving liveness properties with the use of predicate abstraction (when ranking abstraction does not provide additional useful compassion requirements).

The rest of this paper is organized as follows. In Section 2 we introduce the basic concepts used in the approach. It includes the computational model FDS (fair discrete system) and CDS (compassion discrete system) with its related notions of fairness, the rule RESPONSE [1] for the deductive proof of response properties of CDS. Section 3 presents the approach for computing the auxiliary constructs, and Section 4 illustrates the application of the approach on two examples of sequential and concurrent programs. Finally, concluding remarks are contained in Section 5.

2 Preliminaries

We introduce the computational model with fairness requirements [11], and the rule for proving response properties [1].

Computational Model. A fair discrete system (FDS) is a quintuple $D=\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ where the components are as follows.

- V : A finite set of typed *system variables*, containing data and control variables. A set of states (interpretation) over V is denoted by Σ. For a state s and a system variable $v \in V$, we denote by s[v] the value assigned to v by the state s.
- Θ : The *initial condition* - an assertion (state formula) characterizing the initial states.
- ρ : The *transition relation* - an assertion $\rho(V, V')$, relating the variables in state $s \in \Sigma$ to the V' in a D-successor state $s' \in \Sigma$.
- $\mathcal{J}$: A set of justice requirements (weak fairness). The justice requirement $J \in \mathcal{J}$ is an assertion which guarantee that every computation should include infinitely many states satisfying J.
- $\mathcal{C}$: A set of compassion requirements (strong fairness). The compassion requirement $\langle p, q\rangle \in \mathcal{C}$ is a pair of assertions, which guarantee that every computation should include either only finitely many p-states, or infinitely many q-states.

[2] A justice requirement is an assertion requiring that in a computation, this assertion must be satisfied infinitely often.

Computation. A computation of D is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$, satisfying the following requirements: (1) $s_0 \models \Theta$. (2) For each j = 0, 1, ..., the state s_{j+1} is in a D-successor of the state s_j. For each $v \in V$, we interpret v as $s_l[v]$ and v' as $s_{l+1}[v]$, that is $\langle s_l, s_{l+1}\rangle \models \rho(V, V')$.

Justice. A computation σ is *just*, if σ contains infinitely many occurrences of J-states for every $J \in \mathcal{J}$. A *justice discrete system* (JDS) is an FDS with no compassion requirements.

Compassion. A computation σ is *compassionate*, if σ contains only finitely many p-states, or σ contains infinitely many q-states, for every $\langle p, q\rangle \in \mathcal{C}$. A *compassion discrete system* (CDS) is an FDS with no justice requirements.

Proof Rule for Response Properties. For verifying response properties under the assumption of compassion (strong fairness) requirements over a CDS (since an FDS is equivalent to a CDS[3], it is sufficient to consider CDS only), the deductive rule RESPONSE which was presented and proved to be sound and complete in [1], was developed (this rule is hereafter referred to as C-RESPONSE for emphasizing that it involves compassion requirements). It is shown as follows.

Let p, q be assertions.
Let $\mathcal{A} : (W, \succ)$ be a well-founded domain.
Let $\{F_i = \langle p_i, q_i\rangle \mid i \in \{1, ..., n\}\}$ be a set of compassion requirements.
Let $\{\varphi_i \mid i \in \{1, ..., n\}\}$ be a set of assertions.
Let $\{\Delta_i : \Sigma \to W \mid i \in \{1, ..., n\}\}$ be a set of ranking functions.

$$\begin{array}{lll}
\text{R1} & \text{p} & \Rightarrow q \vee \bigvee_{j=1}^{n}(p_j \wedge \varphi_j) \\
\multicolumn{3}{l}{\forall i \leq n:} \\
\text{R2} & p_i \wedge \varphi_i \wedge \rho & \Rightarrow q' \vee \bigvee_{j=1}^{n}((p'_j \wedge \varphi'_j) \\
\text{R3} & \varphi_i \wedge \rho & \Rightarrow q' \vee (\varphi'_i \wedge \Delta_i = \Delta'_i) \vee \bigvee_{j=1}^{n}(p'_j \wedge \varphi'_j \wedge \Delta_i \succ \Delta'_j) \\
\text{R4} & \varphi_i & \Rightarrow \neg q_i \\
\hline
 & p & \Rightarrow \Diamond q
\end{array}$$

The use of the rule requires: a well-founded domain $\mathcal{A}$, and for each compassion requirement $\langle p_i, q_i\rangle$, a helpful assertion φ_i and a ranking function $\Delta_i : \Sigma \mapsto W$ mapping states of D to elements of $\mathcal{A}$.

R1 requires that any p-state is either a goal state (i.e., a q-state), or a $(p_i \wedge \varphi_i)$-state for some $i \in \{1, \ldots, n\}$. It means that the initial states must be a goal state or in a rank. R2 requires that any step from a $(p_i \wedge \varphi_i)$-state moves either directly to a q-state, or to another $(p_j \wedge \varphi_j)$-state, or stays at a state of the same type (i.e., a $(p_i \wedge \varphi_i)$-state). R3 requires that any step from a φ_i-state moves either directly to a q-state, or to a $(p_j \wedge \varphi_j)$-state with decreasing rank, or stay at a state of the same type with the same rank. R4 together with the previous rules guarantees that if an execution does not satisfy $\Diamond q$, then it

[3] The justice requirement can be expressed as the degenerate compassion requirement $\langle 1, J\rangle$, where 1 denotes the assertion *True* which holds at every state.

violates the compassion requirement. R3, R4 and the well-founded domain of the ranks together guarantee that a sequence of moves starting from a state cannot infinitely often decrease the rank or stay at some states with the same rank indefinitely, therefore it must go to the goal state.

The rule also implicitly requires a match among the number of compassion requirements, the number of assertions, and the number of ranking functions. To be more flexible, we extend the proof rule to allow a compassion requirement to be matched with more than one assertion and ranking function. The modified rule is presented as follows.

Let p, q be assertions.
Let $\mathcal{A} : (W, \succ)$ be a well-founded domain.
Let $\{F_i = \langle p_i, q_i\rangle \mid i \in \{1, ..., n\}\}$ be a set of compassion requirements.
Let $(\{F_i | i \in \{1, ..., n\}\}, \{(F_1, k_1), ..., (F_n, k_n)\})$ be a multiset, in which k_i is the number of instances of F_i in the multiset, such that $\Sigma_{i=1}^{n} k_i = m$.
Let $\{\varphi_i \mid i \in \{1, ..., m\}\}$ be a set of assertions.
Let $\{\Delta_i : \Sigma \rightarrow W \mid i \in \{1, ..., m\}\}$ be a set of ranking functions.

$$\begin{array}{lll}
\text{R1} & \text{p} & \Rightarrow q \vee \bigvee_{j=1}^{m}(p_{h(j)} \wedge \varphi_j) \\
\multicolumn{3}{l}{\forall i \leq m:} \\
\text{R2} & p_{h(i)} \wedge \varphi_i \wedge \rho & \Rightarrow q' \vee \bigvee_{j=1}^{m}((p'_{h(j)} \wedge \varphi'_j) \\
\text{R3} & \varphi_i \wedge \rho & \Rightarrow q' \vee (\varphi'_i \wedge \Delta_i = \Delta'_i) \vee \bigvee_{j=1}^{m}(p'_{h(j)} \wedge \varphi'_j \wedge \Delta_i \succ \Delta'_j) \\
\text{R4} & \varphi_i & \Rightarrow \neg q_{h(i)} \\
\hline
 & p & \Rightarrow \Diamond q
\end{array}$$

In which $h(i) = a$, such that $a \in \{1, \ldots, n\} \wedge 0 < i - \Sigma_{l=1}^{a-1} k_l \leq k_a$ holds.

The correctness follows from the original rule by viewing one compassion requirement as multiple identical compassion requirements.

3 Proving a Response Property

In order to be able to use the proof rule C-RESPONSE for proving a response property $\psi : p \Rightarrow \Diamond q$, we have to define a well-founded domain $\mathcal{A}$, and for each compassion requirement $\langle p_i, q_i\rangle$, define a helpful assertion φ_i and a ranking function $\Delta_i : \Sigma \mapsto W$ mapping states of CDS D to elements of $\mathcal{A}$. The phases for proving $D \models \psi$ including those of computing the helpful assertions and ranking functions are as follows:

1. Use ranking abstraction [6,7,2] and construct D^α and ψ^α from D and ψ, and then construct a pending graph [2] based on D^α.
2. Construct an initial rank for each node of the pending graph and a set of compassion requirements associated to each of these nodes.
3. Construct an abstract graph from the pending graph, such that each node in the abstract graph represents a subset of the nodes of the pending graph, then construct $\mathcal{A}$, and for each node, construct φ_i and Δ_i, and make an association of some compassion requirement F_i to the node. Note that according to the construction, one F_i may correspond to several abstract nodes.

3.1 Ranking Abstraction and Pending Graph

This step is carried out according to the technique of ranking abstraction [6,7,2] and pending graph [2].

Ranking abstraction, as explained in [2], is a method of augmenting the concrete program by a non-constraining progress monitor, which measures the progress of program execution, relative to a given ranking function. In order to distinguish this kind of ranking functions from the ranking functions in the proof rule C-RESPONSE, we call this kind of ranking functions ARFs (augmenting ranking functions) in the sequel. Once a program is augmented, a conventional state abstraction can be used. In such a way, the state abstraction can preserve the ability to monitor progress in the abstract system.

For a system $D = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ (in which $\mathcal{J}$ is empty when CDS is considered) and a well-founded domain $(W, \prec)$, let δ be an ARF over W, let dec_δ be a fresh variable, the augmentation of D by δ is

$$D + \delta : \langle V \cup \{dec_\delta\}, \Theta, \rho \wedge \rho_\delta, \mathcal{J}, \mathcal{C} \cup \{(dec_\delta > 0, dec_\delta < 0)\} \rangle$$

where ρ_δ is defined by

$$dec'_\delta = \begin{cases} 1 & \delta \succ \delta' \\ 0 & \delta = \delta' \\ -1 & otherwise \end{cases}$$

A system may be augmented with a set of ARFs $\{\delta_1, ..., \delta_k\}$. Then predicate abstraction may be applied. In the predicate abstraction, it is not necessary to abstract variables of the form dec_δ since it ranges over the finite domain $\{-1, 0, 1\}$, and the abstraction preserves the compassion requirement $(dec_\delta > 0, dec_\delta < 0)$.

Assuming that we have an abstract program D^α from D constructed by the above process with the abstraction map α, a pending graph is then constructed from D^α. Let us denote the graph by $G = \langle N, E \rangle$. The set of nodes N are those satisfying $pend \vee g$ where $pend$ characterizes the states reachable from a p-state by a q-free path, and g is a q^α-state reachable from a pending state in one step. The set of edges E consists of all transitions connecting two pending states and the edges connecting $pend$ nodes to the goal node g.

The set of nodes of G may be written as $\{S_0, S_1, ..., S_m\}$ where $S_0 = g$ is the goal state and $S_1, ..., S_m$ are pending states. This is the starting point of our algorithm for computing the auxiliary constructs for the proof rule.

3.2 Compassion Requirements and Initial Ranks

The ranking functions in the abstract program are represented as a mapping $N \rightarrow TUPLES$, where TUPLES is the type of lexicographic tuples whose elements are either natural numbers or ARFs. For simplicity, we call such a tuple as a “rank”. Let Δ_l and H_l be respectively the rank and the list of compassion requirements for $S_l \in N$. For convenience, we write q for q^α, and similarly for other formulas and constructions. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be a set of original compassion requirements and $\mathcal{F}_D = \{(dec_\delta > 0, dec_\delta < 0)\}$ be the set of

Algorithm 1. C-RANK(G)

```
1: decompose(G) into a set of MSCCs [$C_0, ..., C_k$];
2: for i=0; $i \leq k$; i++ do
3:    for each $S_l$ of $C_i$, append $i$ to $\Delta_l$;
4: end for
5: for i=0; $i \leq k$; i++ do
6:    if the goal state $g$ is in $C_i$ then
7:       continue;
8:    end if
9:    flag = 0;
10:   for each $F \in \mathcal{F} \cup \mathcal{F}_D$ do
11:      if not violate($C_i$,$F$) then
12:         continue;
13:      end if
14:      flag = 1 ;
15:      for each $S_l$ of $C_i$ do
16:         append $F$ to $H_l$;
17:      end for
18:      for each $F \in \mathcal{F}_D$ do
19:         for each $S_l$ of $C_i$ do
20:            append $term(F)$ to $\Delta_l$;
21:         end for
22:      end for
23:      if $C_i$ is a trivial MSCC then
24:         break;
25:      else
26:         remove_edge($C_i$,$F$); C-RANK($C_i$); break;
27:      end if
28:   end for
29:   if flag = 0 then
30:      terminate unsuccessfully;
31:   end if
32: end for
```

dec-requirements (the compassion requirements introduced by the ranking abstraction). The procedure for computing Δ_l and H_l (which are initially empty) is described in Algorithm 1. The way of dealing with MSCCs (maximal strongly connected components) and fairness follows the idea of [12] by Emerson and Lei. The main functions are explained as follows.

decompose(G). The graph G is decomposed into a set of MSCCs, denoted as $C_0, ..., C_k$. They are ordered so that if C_i is reachable from C_j, then $i < j$. For MSCCs not connected to each other, their indices may be in an arbitrary order.

violate(C_i, F). The MSCC (may be the trivial one) C_i violates the compassion requirement $F = (p, q)$, if p is satisfied by some node of the MSCC and q is not satisfied by any node of the MSCC.

Algorithm 2. C-GRAPH

```
1: call C-RANK(G);
2: for each $F_i \in \mathcal{F}$ do
3:     $W_i$=subgraph($G, F_i$);
4: end for
5: for each $W_i \in \{W_1, ..., W_n\}$ do
6:     create_merge_nodes($W_i, G'$);
7: end for
8: create_edges($G'$);
```

term(F). For a dec-requirement F of the form $\langle dec_\delta > 0, dec_\delta < 0\rangle$, $term(F) = \delta$.

remove_edge(C_i, F). Given an MSCC C_i and a compassion requirement $F = (p, q)$, this procedure modifies the MSCC in such a way that one node satisfying p is identified and all incoming edges of such a node in this MSCC are removed.

3.3 Abstract Nodes, Helpful Assertions and Ranks

According to H_l, we construct the abstract nodes as an assertion Φ by grouping together certain nodes that need to satisfy the same compassion requirements. Let G' be the abstract graph, initially empty, i.e., $G' = (\{\}, \{\})$. The procedure for constructing G' is described in Algorithm 2. The main functions are explained as follows.

subgraph(G, F_i). In the assignment $W_i = subgraph(G, F_i)$, the variable W_i is a local variable used to hold a subset of nodes (a subgraph). The nodes of the subgraph is constructed according to $F_i \in \mathcal{F}$ (the original compassion requirements) as follows: $S_l \in W_i \iff F_i \in H_l$. Then W_i is considered as a derived subgraph of G with the nodes as specified.

create_merge_nodes(W_i, G'). (1) W_i may contain several different MSCCs violating F_i. For each such MSCC in W_i, a node representing this MSCC is created and added to G'. The rank Δ_l of the abstract node Φ_l is assigned the rank obtained before the MSCC is split into smaller MSCCs. (2) For the nodes created previously, they are merges according to the following condition: the states that the nodes represent differ only in the dec-variables introduced in the ranking abstraction. Then the rank of the abstract node is assigned the lowest rank of the nodes represented by the abstract node. (3) For each node Φ_l in the final abstract graph G', the concrete helpful assertion $\varphi_l = \alpha^{-1}(\Phi_l)$ is obtained by concretizing the abstract nodes (viewed as abstract assertions, by making correspondence between formulas and sets of states).

create_edges(G'). For each pair of nodes Φ and Φ' such that $\Phi' \not\subset \Phi$, if some node of Φ is connected to some of Φ', an edge from Φ to Φ' is created.

3.4 Correctness of Auxiliary Constructs

For each compassion requirement (p_i, q_i), several abstract states may be associated. This has been made explicit in the modified C-RESPONSE rule, where we consider the set of original compassion requirements as a multiset, such that one compassion requirement (p_i, q_i) has the number of occurrences matching the number of associated abstract states. Then each (occurrence of a) compassion requirement corresponds to one abstract state (with a rank and a helpful assertion) associated with it.

Ranking Core. For the correctness, we assume that every ranking function in ranking abstraction is chosen to be a variable. Such a set of variables (representing a set of ranking functions) are called ranking core $\mathcal{R}$ [6]. It is easily seen that the proof of the correctness of the above algorithms with this assumption can be extended to ranking functions that are arithmetic terms. The abstraction of D according to the abstraction map α and the ranking core $\mathcal{R}$ is denoted $\mathcal{D}^{\mathcal{R},\alpha}$.

Theorem. *Let CDS $\mathcal{D}$, a ranking core $\mathcal{R}$, an abstraction mapping α and the property Ψ be given. Let the assertions φ_i and ranking functions Δ_i be that successfully extracted by C-GRAPH. If $\mathcal{D}^{\mathcal{R},\alpha} \models \Psi^\alpha$ then R_1-R_4 of the rule C-RESPONSE are provable with the extracted auxiliary constructs.*

The correctness is established by analyzing the different steps in the construction of the auxiliary construct. Firstly, two ranks are compared as follows [2]. Let $\Delta_i = (a_1, \ldots, a_r)$, $\Delta_j = (b_1, \ldots, b_r)$. Let $gt(\Delta_i, \Delta_j)$ be defined by:

$$gt(\Delta_i, \Delta_j) \equiv \bigvee_{k=1}^{r} (a_1 = b'_1) \wedge \cdots \wedge (a_{k-1} = b'_{k-1}) \wedge (a_k \succ b'_k)$$

The formula $gt(\Delta_i, \Delta_j)$ formalizes the condition for $\Delta_i \succ \Delta_j$ in the lexicographic order. We may not be able to decide whether $gt(\Delta_i, \Delta_j)$ is true or false immediately, because a_k, b_k may be functions such as $\delta_k = x$ or $\delta_k = y$. Let $\Delta \succ_E \Delta'$ denote that Δ appear after Δ' according to the lexicographic order with the following conditions: the lexicographic order is augmented by an environment E that specifies whether $\delta_k \succ \delta'_k$ or $\delta_k = \delta'_k$. The environment E may be replaced by a state S that reflects whether the value of a variable is decreased when the program moves to the state S. Let $\Delta > \Delta'$, where Δ, Δ' represent ranks, denote that Δ appear after Δ' according to the lexicographic order in the initial state.

Claim 1. *Let S_i and S_j be states in the pending graph. The following properties hold.*

- P_1. If $\Delta_i \succ_{S_j} \Delta_j$ and $\Delta_j > \Delta_k$, then $\Delta_i \succ_{S_j} \Delta_k$.
- P_2. If the states S_i and S_j agree on the values of their non-dec variables, then they have the same set of successors.

P_1 is true according to the definition. P_2 is true according to the construction of the pending graph. These properties are the same as those stated in [2].

Claim 2. *Let Δ_i and Δ_j be the associated ranks of S_i and S_j. Then on successful termination of C-RANK, the following properties hold.*

- P_3. For every two states S_i, S_j belonging to different MSCCs such that S_i is connected to S_j, there is a rank decrease $\Delta_i \succ_{S_j} \Delta_j$.
- P_4. For every two states S_i, S_j belonging to one MSCC such that S_i is connected to S_j, there is no rank decrease only if the MSCC violates some compassion requirement (p_k, q_k) (non-dec-requirements) and there is at least one state S_k which $S_k \models p_k$, or the MSCC does not violate any compassion requirement.

P_3 follows from the decomposition of the pending graph into MSCCs. P_4 follows from the way MSCCs being modified when they do not satisfy some compassion requirements.

Claim 3. *Let Δ_i and Δ_j be the associated ranks of Φ_i and Φ_j. Then on termination of C-GRAPH, the following properties hold.*

- P_5. If Φ_i is connected to Φ_j in the abstract graph and $S \in \Phi_j$, then there is a Φ_k such that $S \in \Phi_k$, $\Delta_i \succ_S \Delta_k$ and $S \models p_k$ where (p_k, q_k) is the compassion requirement violated by the MSCC represented by Φ_k.
- P_6. Let $s[\Delta]$ be Δ with the variables replaced by their value in the state s. If concrete states s, s' satisfy $s \models \varphi_i$, and $s' \models \varphi_j$, $i \neq j$ and s' is a D-successor of s, then there is a φ_k such that $s' \models \varphi_k \wedge p_k$ and $s[\Delta_i] \succ s'[\Delta_k]$.

P_5 follows from the construction of the abstract graph. Φ_k in P_5 is necessarily a superset of Φ_j (i.e., $\Phi_j \subseteq \Phi_k$), when Φ_k is considered as a set of nodes of the pending graph. P_6 follows from P_5 by the soundness of the abstraction.

Proof of the theorem. Let $\Phi_0 = g, \Phi_1, ..., \Phi_n$ be the nodes in the abstract graph, and $\varphi_i, \Delta_i, (p_i, q_i)$ be the helpful assertion, rank and compassion requirement (which has been reorganized into a multiset that matches the number of Φ_i) associated to Φ_i for $i = 1, ..., n$. Assume $D^\alpha \models \Psi^\alpha$. (1) Since Ψ^α is true, the disjunction of the abstract states $g, \Phi_1, ..., \Phi_n$ covers the states in the pending graph. By the correctness of the abstraction, $g \vee \bigvee_{i=1}^{l} \varphi_i$ covers the state space represented by the pending graph. The a p-state is either the goal state g or a state with a progress requirement, i.e. $p_i \wedge \varphi_i$ for some $i \in \{1, ..., n\}$. (2) Similarly, successor states of such a state (excluding g) also satisfy the same condition. (3) The correctness of R_3 follows from property P_6. (4) R_4 is guaranteed by the construction of φ_i, since each φ_i represents an MSCC (or a collection of MSCCs when they are merged) violating the compassion requirement (p_i, q_i).

3.5 Discussion

Previous works in this directive of research include using deduction rules with weak fairness (justice) requirements to prove liveness properties of sequential or simple concurrent programs. They depend on dec-requirements to decide the ranks of states in just MSCC. We concern deductive rule with strong fairness (compassion) requirements to prove liveness properties of more complex concurrent programs. It depends on compassion requirements to decide the ranks of states in MSCC.

4 Application Examples

We illustrate the application of the approach on two programs:

- COND-TERM, a sequential program with a non-deterministic choice of the values of a variable [1].
  This example is supposed to show the approach applied on a verification problem with ranking abstraction in which some dec-requirement is introduced in the abstraction phase.
- MUX-SEM, a concurrent program for mutual exclusion [13].
  This example is supposed to show the approach applied on a concurrent program.

4.1 Example 1: COND-TERM

The following is the program COND-TERM (conditional termination). The response property we wish to establish is $\Psi : at\_l_1 \Rightarrow \Diamond at\_l_4$. The just requirements are $\neg at\_l_i$ for $i = 1, 2, 3, 4$, and the compassion requirements is $F_1 = \langle at\_l_3 \wedge x = 0, 0\rangle$. Let F_{i+1} be $\langle 1, \neg at\_l_i\rangle$ representing the just requirements for $i = 1, 2, 3$.

```
x,y: natural init $x = 0$
  $l_1$: while $y > 0$ do
  $l_2$:     x:= {0,1}
  $l_3$:     $y := y + 1 - 2x$
  $l_4$:
```

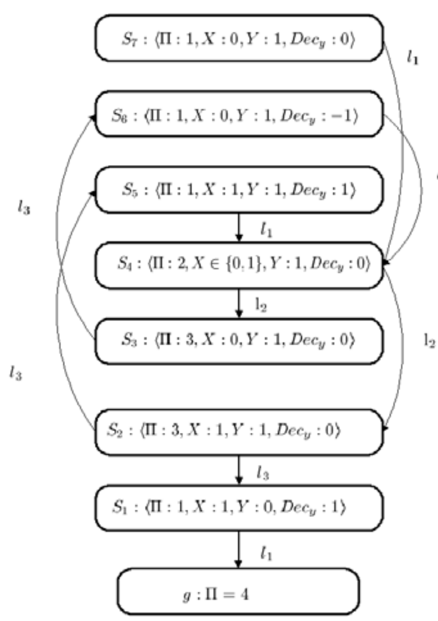
Phase 1 (ranking abstraction and pending graph). The ranking core in this case is chosen to be $\{y\}$ and the ARF (augmenting ranking function) y is associated with the natural numbers as the well-founded set. Let $Dec_y = sign(y - y')$ in which y denotes the value of y in the previous state and y' denotes the value of y in the current state. The abstraction mapping α is defined by :

$$\alpha : \Pi = \pi, X = (x > 0), Y = (y > 0), dec_y = Dec_y$$

where $\Pi = i$ denotes at_l_i is true. We construct the pending graph as showing in Fig. 1. The constraints of the graph include the additional compassion requirement $F_D = \langle dec_y > 0, dec_y < 0\rangle$ which is deduced from the condition of the while loop according to the rank abstraction process. We have $\mathcal{F}_D = \{F_D\}$ in this example. The pending graph includes two kinds of uncompassionate loops, one violating the given compassion and the other violating the compassion introduced in the abstraction process.

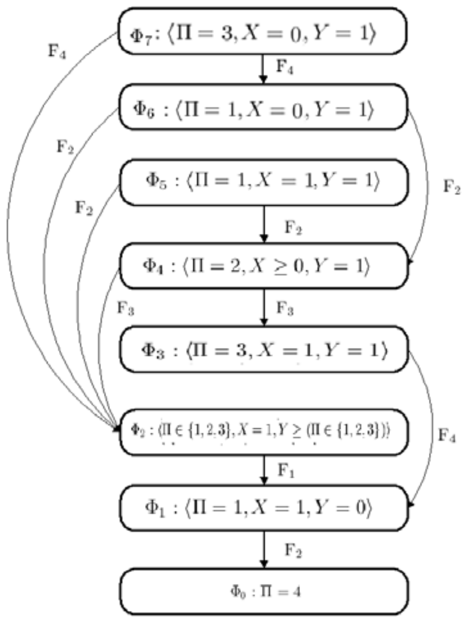
There are 8 states $\{S_0, S_1, ..., S_7\}$ with $S_0 = g$.

Phase 2 (compassion requirements and initial ranks). Let Δ_i and H_i be the rank of S_i and the set of compassion requirements associated to S_i, respectively, initially with $\Delta_i = []$ and $H_i = []$.



$S_7 : \langle \Pi : 1, X : 0, Y : 1, Dec_y : 0 \rangle$
$S_6 : \langle \Pi : 1, X : 0, Y : 1, Dec_y : -1 \rangle$
$S_5 : \langle \Pi : 1, X : 1, Y : 1, Dec_y : 1 \rangle$
$S_4 : \langle \Pi : 2, X \in \{0, 1\}, Y : 1, Dec_y : 0 \rangle$
$S_3 : \langle \Pi : 3, X : 0, Y : 1, Dec_y : 0 \rangle$
$S_2 : \langle \Pi : 3, X : 1, Y : 1, Dec_y : 0 \rangle$
$S_1 : \langle \Pi : 1, X : 1, Y : 0, Dec_y : 1 \rangle$
$g : \Pi = 4$
l_1
l_2
l_3

Fig. 1. The Pending Graph



$\Phi_7 : \langle \Pi = 3, X = 0, Y = 1 \rangle$
$\Phi_6 : \langle \Pi = 1, X = 0, Y = 1 \rangle$
$\Phi_5 : \langle \Pi = 1, X = 1, Y = 1 \rangle$
$\Phi_4 : \langle \Pi = 2, X \geq 0, Y = 1 \rangle$
$\Phi_3 : \langle \Pi = 3, X = 1, Y = 1 \rangle$
$\Phi_2 : \langle \Pi \in \{1, 2, 3\}, X = 1, Y \geq \langle \Pi \in \{1, 2, 3\} \rangle \rangle$
$\Phi_1 : \langle \Pi = 1, X = 1, Y = 0 \rangle$
$\Phi_0 : \Pi = 4$
F_1
F_2
F_3
F_4

Fig. 2. The Abstract Graph

In the first level of computation, we have 4 MSCCs:

$$\{g\}, \{S_1\}, \{S_2, S_3, S_4, S_5, S_6\}, \{S_7\}.$$

Then $\Delta_0 = [0], \Delta_1 = [1], \Delta_2 = \Delta_3 = \Delta_4 = \Delta_5 = \Delta_6 = [2], \Delta_7 = [3]$.

Let C_0, C_1, C_2, C_3 denote the 4 MSCCs. Since C_0 is the set of the goal state, we check C_1, C_2, C_3 against the compassion requirements, and obtain Table 1.

Table 1. The MSCCs C_1, C_2, C_3

Component	Violation
C_1	F_2
C_2	F_1
C_3	F_2

Table 2. The MSCCs C_{21}, C_{22}, C_{23}

Component	Violation
C_{21}	F_4
C_{22}	F_2
C_{23}	F_D

Then we add the respective compassion requirement to $H_1, ..., H_7$, and obtain $H_1 = H_7 = [F_2], H_2 = H_3 = H_4 = H_5 = H_6 = [F_1]$.

Since C_2 is not a non-trivial subgraph, we remove the edge $(S_4 \rightarrow S_3)$, which leads into the state satisfying $at\_l_3 \wedge x = 0$, from C_2, and compute again with the modified subgraph.

In the second level of computation, we have 3 MSCCs: $\{S_3\}$, $\{S_6\}$, $\{S_4, S_2, S_5\}$. Then $\Delta_3 = [2, 2], \Delta_6 = [2, 1], \Delta_2 = \Delta_4 = \Delta_5 = [2, 0]$.

Let C_{21}, C_{22}, C_{23} denote the 3 MSCCs. By checking the MSCCs against the compassion requirements, we obtain Table 2.

Then we add the respective compassion requirement to $H_2, ..., H_6$. In addition, since C_{23}=$\{S_4, S_2, S_5\}$ violates $F_D = \langle dec_y > 0, dec_y < 0\rangle$, we add y to $\Delta_2, \Delta_4, \Delta_5$, and obtain $\Delta_2 = \Delta_4 = \Delta_5 = [2, 0, y]$.

Then since C_{23} is not a non-trivial subgraph, we remove the edge $(S_2 \rightarrow S_5)$, which leads into the state satisfying $dec_y > 0$, from C_{23}, and compute again with the modified subgraph.

In the third level of computation, we have 3 MSCCs: $\{S_5\}$, $\{S_4\}$, $\{S_2\}$.

The rank to be assigned to the nodes in this level is $2, 1, 0$, and we obtain $\Delta_5 = [2, 1, y, 2], \Delta_4 = [2, 1, y, 1], \Delta_2 = [2, 1, y, 0]$.

Since $\{S_5\}$, $\{S_4\}$ and $\{S_2\}$ are trivial MSCCs, we add F_2, F_3, F_4 to H_5, H_4, H_2 respectively. The final value of Δ_i and H_i are as shown in Table 3.

Table 3. The Rank Table of Program COND-TERM

Index i	S_i	Δ_i	H_i
7	S_7	$[3]$	$[F_2]$
6	S_6	$[2, 1]$	$[F_1, F_2]$
5	S_5	$[2, 0, y, 2]$	$[F_1, F_D, F_2]$
4	S_4	$[2, 0, y, 1]$	$[F_1, F_D, F_3]$
3	S_3	$[2, 2]$	$[F_1, F_4]$
2	S_2	$[2, 0, y, 0]$	$[F_1, F_D, F_4]$
1	S_1	$[1]$	$[F_2]$
0	S_0	$[0]$	

Phase 3 (abstract nodes, helpful assertions and ranks). According to H_i, we construct the abstract nodes [4] by grouping together nodes that need satisfying the same compassion requirement merging S_6 - S_2 as an abstract state Φ_2 and by grouping together nodes that agree with the value of all variables except the dec-variable: merging S_7 and S_5 as another abstract state Φ_6. The abstract nodes with their respective ranks are listed in Table 4 and the abstract graph is shown in Fig. 2.

Finally, we obtain the concrete helpful assertions $\varphi_1, \ldots, \varphi_7$ by concretizing the abstract assertions $\Phi_1, \ldots, \Phi_7$, and obtain the ranks $\Delta_1, \ldots, \Delta_7$ by renumbering the respective ranks in Fig. 2. The helpful assertions and the ranks are shown in Table 5.

The validity of the premises of the rule C-RESPONSE for this example may be verified by using the constructed auxiliary constructs $\varphi_1, ..., \varphi_7$ and $\Delta_1, ..., \Delta_7$. The reader is referred to the technical report [5] for the details.

[4] Note that F_D is not involved in the construction of abstract nodes, since it is not one of the original system constraints.

Table 4. The Abstract Table of Program COND-TERM

Abstract Node	*Nodes*	*Rank*	*Compassion Req.*
Φ_7	S_3	$[2,2]$	F_4
Φ_6	S_6, S_7	$[2,1]$	F_2
Φ_5	S_5	$[2,0,y,2]$	F_2
Φ_4	S_4	$[2,0,y,1]$	F_3
Φ_3	S_2	$[2,0,y,0]$	F_4
Φ_2	$S_2,...,S_6$	$[2]$	F_1
Φ_1	S_1	$[1]$	F_2
Φ_0	S_0	$[0]$	

Table 5. The Concrete Table of Program COND-TERM

Index	*Helpful Assertion* φ_i	Δ_i
7	$at\_l_3 \wedge y > 0 \wedge x = 0$	$[2,2]$
6	$at\_l_1 \wedge y > 0 \wedge x = 0$	$[2,1]$
5	$at\_l_1 \wedge y > 0 \wedge x = 1$	$[2,0,y,2]$
4	$at\_l_2 \wedge y > 0 \wedge x \in \{0,1\}$	$[2,0,y,1]$
3	$at\_l_3 \wedge y > 0 \wedge x = 1$	$[2,0,y,0]$
2	$at\_l_{1..3} \wedge y \geqslant at\_l_{1,2,3} \wedge x \in \{0,1\}$	$[2]$
1	$at\_l_1 \wedge y = 0 \wedge x = 1$	$[1]$

4.2 Example 2: MUX-SEM

The following is the concurrent program MUX-SEM. Let $at\_l_i[j]$ denotes that process j is at l_i (of process j). The response property we wish to establish is $at\_l_2[1] \Rightarrow \Diamond at\_l_3[1]$. The just requirements are $\neg at\_l_4[j]$ and $\neg at\_l_3[j]$. The compassion requirement is $F_1 = \langle at\_l_2[1] \wedge y = 1, at\_l_3[1]\rangle$. The just requirements are special compassion requirements formulated as $\langle 1, \neg at\_l_4[i]\rangle$ and $\langle 1, \neg at\_l_3[i]\rangle$ for $i \in \{1, ..., n\}$.

$$\textbf{local}\; y : \textbf{boolean} \quad \textbf{init}\; y = 1;$$
$$\|_{i=1}^{n} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad l_1 : \textbf{Noncritical} \\ \quad l_2 : \textbf{request}\; y \\ \quad l_3 : \textbf{Critical} \\ \quad l_4 : \textbf{release}\; y \end{array}\right]$$

The abstraction mapping α is defined by:

$$\alpha : \Pi = \pi, \Pi_3 = \pi_3, \Pi_4 = \pi_4, Y = (y > 0)$$

where Π is a function with range $\{1, 2, 3, 4\}$ (and the domain being the system states). $\Pi = i$ denotes that $at\_l_i[1]$ is true, for $i \in \{1, ..., 4\}$. Π_k is a function with range $\{0, 1\}$ and it is 1 if and only if the following is true:

$$\bigvee_{j=2}^{n}(at\_l_k[j] \wedge \bigwedge_{i=2}^{n}((i \neq j) \Rightarrow \neg at\_l_k[i]))$$

The set of compassion requirements $\{\langle 1, \neg at\_l_4[i]\rangle \mid i = 1, ..., n\}$ and the set $\{\langle 1, \neg at\_l_3[i]\rangle \mid i = 1, ..., n\}$ induce two new compassion requirements $F_2 = \langle 1, \neg\pi_4 = 1\rangle$ and $F_3 = \langle 1, \neg\pi_3 = 1\rangle$. Then the set of compassion requirements of the abstract program is $\mathcal{F} = \{F_1, F_2, F_3\}$.

Let $-$ denote any value in the range of the respective position of the abstract states. For instance, $(1, -, -, -)$ denotes the abstract states where the value of (Π, Π_3, Π_4, Y) satisfying $\Pi = 1$ and the rest of the positions could be any value. And (2, 1, 0, 0) denotes that process 1 is at l_2 and there is another process j at l_3 meanwhile $y = 1$. Then the abstract states represented by the following tuples covers the reachable concrete states :

$$(1, -, -, -), (2, 0, 0, 1), (2, 1, 0, 0), (2, 0, 1, 0), (3, -, -, -), (4, -, -, -)$$

Let S_0, S_1, S_2, S_3 be the set of states represented by respectively

$$(3, -, -, -), (2, 0, 0, 1), (2, 0, 1, 0), (2, 1, 0, 0).$$

Then we construct the pending graph with these four states with $S_0 = g$, and proceed with computing the temporary Δ_i and H_i for each of the states S_1, S_2, S_3 using algorithm 1, and obtain $(\Delta_1, \Delta_2, \Delta_3) = ([1, 2], [1, 0], [1, 1])$. Then we compute the abstract states and their associated ranks using algorithm 2, and obtain three abstract nodes (not counting the node representing the goal state) $\Phi_1 = \{S_1, S_2, S_3\}, \Phi_2 = \{S_2\}, \Phi_3 = \{S_3\}$ with their respective ranks $[1], [1, 0], [1, 1]$ and associated compassion requirement F_1, F_2, F_3.

Finally, we obtain the concrete helpful assertions $\varphi_1, \varphi_2, \varphi_3$ by concretizing the abstract assertions Φ_1, Φ_2, Φ_3, and obtain the ranks $\Delta_1, \Delta_2, \Delta_3$ by renumbering the respective ranks. The concrete assertions $\varphi_1, \varphi_2, \varphi_3$ and the ranks $\Delta_1, \Delta_2, \Delta_3$ are shown as follows.

Index i	φ_i	Δ_i
3	$at\_l_2[1] \wedge \bigvee_{j=2}^{n}(at\_l_3[j] \wedge \bigwedge_{i=2}^{n}((i \neq j) \Rightarrow \neg at\_l_3[i])) \wedge y = 0$	[2]
2	$at\_l_2[1] \wedge \bigvee_{j=2}^{n}(at\_l_4[j] \wedge \bigwedge_{i=2}^{n}((i \neq j) \Rightarrow \neg at\_l_4[i])) \wedge y = 0$	[1]
1	$at\_l_2[1]$	[0]

5 Concluding Remarks

For proving a response property in systems with fairness based on the rule presented in [1], we need auxiliary constructs. We have presented a method for extracting such constructs. The method consists of phase 2 and phase 3 described in Section 3, while phase 1 is as same as that of [2]. The method extends that presented in [2] which aimed at proving a response property in systems with justice. The use of the method has been illustrated by examples of concurrent and sequential programs. When the system is restricted to only allowing justice requirements, the auxiliary constructs we obtained may be different from those obtained by using the method presented in [2]. For illustrating this, we have also tried our method on the example of [2] for proving the response property in a system with justice, the details can be found in the technical report [5].

References

1. Pnueli, A., Sa'ar, Y.: All you need is compassion. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 233–247. Springer, Heidelberg (2008)
2. Balaban, I., Pnueli, A., Zuck, L.D.: Modular ranking abstraction. Int. J. Found. Comput. Sci. 18(1), 5–44 (2007)
3. 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)
4. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI, pp. 203–213 (2001)
5. Long, T., Zhang, W.: Auxiliary constructs for proving liveness in compassion discrete systems. Technical Report, ISCAS–LCS–09–03, Institute of Sofware, Chinese Academy of Sciences (2009), `http://lcs.ios.ac.cn/~zwh/tr/`
6. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Inf. Comput. 163(1), 203–243 (2000)
7. Balaban, I., Pnueli, A., Zuck, L.D.: Ranking abstraction as companion to predicate abstraction. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 1–12. Springer, Heidelberg (2005)
8. Kesten, Y., Pnueli, A., Vardi, M.Y.: Verification by augmented abstraction: The automata-theoretic view. J. Comput. Syst. Sci. 62(4), 668–690 (2001)
9. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 223–238. Springer, Heidelberg (2004)
10. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with incomprehensible ranking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 482–496. Springer, Heidelberg (2004)
11. Manna, Z., Pnueli, A.: Completing the temporal picture. Theor. Comput. Sci. 83(1), 91–130 (1991)
12. Emerson, E.A., Lei, C.L.: Modalities for model checking: Branching time logic strikes back. Sci. Comput. Program. 8(3), 275–306 (1987)
13. 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)

Symbolic Unfolding of Parametric Stopwatch Petri Nets

Louis-Marie Traonouez[1], Bartosz Grabiec[2], Claude Jard[2], Didier Lime[3], and Olivier H. Roux[3,⋆]

[1] Università di Firenze, Dipartimento di Sistemi e Informatica, Italy
[2] ENS Cachan & INRIA, IRISA, Rennes, France
Université européenne de Bretagne
[3] École Centrale de Nantes & Université de Nantes, IRCCyN, Nantes, France

Abstract. This paper proposes a new method to compute symbolic unfoldings for safe Stopwatch Petri Nets (SwPNs), extended with time parameters, that symbolically handle both the time and the parameters.

We propose a concurrent semantics for (parametric) SwPNs in terms of timed processes *à la* Aura and Lilius. We then show how to compute a symbolic unfolding for such nets, as well as, for the subclass of safe time Petri nets, how to compute a finite complete prefix of this unfolding.

Our contribution is threefold: unfolding in the presence of stopwatches or parameters has never been addressed before. Also in the case of time Petri nets, the proposed unfolding has no duplication of transitions and does not require read arcs and as such its computation is more local. Finally the unfolding method is implemented (for time Petri nets) in the tool Romeo.

Keywords: unfolding, time Petri nets, stopwatches, parameters, symbolic methods.

1 Introduction

The analysis of concurrent systems is one of the most challenging practical problems in computer science. Formal specification using Petri nets has the advantage to focus on the tricky part of such systems, that is parallelism, synchronization, conflicts and timing aspects. Among the different analysis techniques, we chose to develop the work on unfoldings [9].

Unfoldings were introduced in the early 1980s as a mathematical model of causality and became popular in the domain of computer aided verification. The main reason was to speed up the standard model-checking technique based on the computation of the interleavings of actions, leading to a very large state space in case of highly concurrent systems. The seminal papers are [14] and [8]. They dealt with basic bounded Petri nets.

⋆ This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003).

Since then, the technique has attracted more attention, and the notion of unfolding has been extended to more expressive classes of Petri nets (Petri nets with read and inhibitor arcs [7,3], unbounded nets [1], high-level nets [12], and time Petri nets [6]).

Advancing this line of works, we present in this paper a method to unfold safe parametric stopwatch Petri nets. Stopwatch Petri nets (SwPNs) [5] are a strict extension of the classical time Petri nets *à la* Merlin (TPNs) [15,4] and provide a means to model the suspension and resumption of actions with a memory of the "work" done before the suspension. This is very useful to model real-time preemptive scheduling policies for example.

The contribution of this paper is a new unfolding algorithm addressing the problem for stopwatch and parametric models for the first time. When applied to the subclass of time Petri nets, it provides an alternative to [6] and improves on the latter method by providing a more compact unfolding and not requiring read arcs in the unfolding (if the TPN itself has no read arcs of course). We also provide a way to compute a finite complete prefix of the unfolding for (safe) TPNs. Note this is the best we can do as most interesting properties, such as reachability, are undecidable in time Petri nets in presence of stopwatches [5] or parameters [16].

While not extremely difficult from a theoretical point of view, we think that the handling of parameters is of utmost practical importance: adding parameters in specifications is a real need. It is often difficult to set them a priori: indeed, we expect from the analysis some useful information about their possible values. This feature of genericity clearly adds some "robustness" to the modeling phase. It is important to note that, as for time, we handle these parameters symbolically to achieve this genericity and the unfolding technique synthesizes all their possible values as linear constraint expressions.

Finally, note that the lack of existence of a finite prefix in the stopwatch or parametric cases is not necessarily prohibitive as several analysis techniques, such as supervision, can do without it [10]. Practical experience also demonstrates that even for very expressive models, such as Linear Hybrid Automata [11], the undecidability of the interesting problems still allows to analyze them in many cases.

Organization of the paper. Section 2 gives preliminary definitions and Section 3 propose an unfolding method of stopwatch parametric Petri nets based on an original way of determining conflicts in the net. Section 4 shows how to compute a complete finite prefix of the unfolding of a time Petri net. Finally in Section 5, we discuss open problems and future work.

2 Definitions

We denote by $\mathbb{N}$ the set of non-negative integers, by $\mathbb{Q}$ the set of rational numbers and $\mathbb{R}$ the set of real numbers. For $A \in \{\mathbb{Q}, \mathbb{R}\}$, $A_{\geq 0}$ (resp. $A_{>0}$) denotes the subset of non-negative (resp. strictly positive) elements of A. Given $a, b \in \mathbb{N}$ such that $a \leq b$, we denote by $[a..b]$ the set of integers greater or equal to a and less or equal to b. For any set X, we denote by $|X|$ its cardinality.

For a function f on a domain D and a subset C of D, we denote by $f_{|C}$ the restriction of f to C.

Let X be a finite set. A (rational) *valuation* of X is a function from X to $\mathbb{Q}$. A (rational) *linear expression* on X is an expression of the form $a_1x_1 + \cdots + a_nx_n$, with $n \in \mathbb{N}$, $\forall i, a_i \in \mathbb{Q}$ and $x_i \in X$. A *linear constraint* on X is an expression of the form $L_X \sim b$, where L_X is a linear expression on X, $b \in \mathbb{Q}$ and $\sim \in \{<, \leq, \geq, >\}$. Given a linear expression $L = a_1x_1 + \cdots + a_nx_n$ on X and a rational valuation v on X, we denote $v(L)$ the rational number $a_1v(x_1) + \cdots + a_nv(x_n)$. Similarly for a linear constraint $C = L \sim b$, we note $v(C)$ the Boolean expression $(v(L) \sim b)$. We extend this notation in the same way for conjunctions, disjunctions and negations of constraints.

For the sake of readability, when non-ambiguous, we will "flatten" nested tuples, *e.g.* $\langle\langle\langle B, E, F\rangle, l\rangle, v, \theta\rangle$ will be written $\langle B, E, F, l, v, \theta\rangle$.

2.1 Unfolding Petri Nets

Definition 1 (Place/transition net). *A place/transition net with read arcs (P/T net) is a tuple $\langle P, T, W, W_r\rangle$ where: P is a finite set of* places*, T is a finite set of* transitions*, with $P \cap T = \emptyset$, $W \subseteq (P \times T) \cup (T \times P)$ is the* transition incidence relation *and $W_r \subseteq P \times T$ is the* read incidence relation

This structure defines a directed bipartite graph such that $(x, y) \in W \cup W_r$ iff there is an arc from x to y.

We further define, for all $x \in P \cup T$, the following sets: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in W\}$, ${}^\diamond x = \{y \in P \cup T \mid (y, x) \in W_r\}$ and $x^\bullet = \{y \in P \cup T \mid (x, y) \in W\}$. These set definitions naturally extend by union to subsets of $P \cup T$.

A *marking* $m : P \to \mathbb{N}$ is a function such that (P, m) is a multiset. For all $p \in P$, $m(p)$ is the number of *tokens* in the place p. In this paper we restrict our study to *1-safe* nets, *i.e.* nets such that $\forall p \in P,\ m(p) \leq 1$. Therefore, in the rest of the paper, we will usually identify the marking m with the set of places p such that $m(p) = 1$. In the sequel we will call *Petri net* (with read arcs) a marked P/T net, *i.e.* a pair $\langle \mathcal{N}, m\rangle$ where $\mathcal{N}$ is a P/T net and m a marking of $\mathcal{N}$, called *initial marking.*

A transition $t \in T$ is said to be enabled by the marking m if ${}^\bullet t \cup {}^\diamond t \subseteq m$. We denote by $\mathsf{en}(m)$, the set of transitions enabled by m.

2.2 Semantics of True Concurrency

There is a path $x_1, x_2, \ldots, x_n$ in a P/T net iff $\forall i \in [1..n]$, $x_i \in P \cup T$ and $\forall i \in [1..n-1], (x_i, x_{i+1}) \in W \cup W_r$.

In a P/T net, consider $x, y \in P \cup T$. x and y are *causally related*, which we denote by $x < y$, iff there exists a path in the net from x to y. The causal past of a transition t is called *local configuration* and denoted by $\lceil t\rceil$, and is constituted by the transitions that causally precede t, *i.e.* $\lceil t\rceil = \{t' \in T \mid t' < t\}$.

The addition of the read arcs introduces another causal relation between two transitions $x, y \in T$, that is called *weak causality* and denoted by $x \nearrow y$, iff

$x < y \vee {}^{\diamond}x \cap {}^{\bullet}y \neq \emptyset$. This notion is already presented in [7]. The relation denotes that the firing of the transition x happens before the one of y.

The two causal relations induce a relation of conflicts between the transitions of the net. A set $X \subseteq T$ of transitions are said to be in conflict, noted $\#X$, when some transitions consumed the same token, or when the weak causality defines a cycle in this set. Formally:

$$\#X = \begin{cases} \exists x, y \in X \ : \ x \neq y \wedge {}^{\bullet}x \cap {}^{\bullet}y \neq \emptyset \ \vee \\ \exists x_0, x_1, \ldots, x_n \in X \ : \ x_0 \nearrow x_1 \nearrow \ldots x_n \nearrow x_0 \end{cases}$$

Definition 2 (Occurrence net). *An* occurrence net *is an acyclic P/T net* $\langle B, E, F, F_r \rangle$*:*

- *finite by precedence* ($\forall e \in E,\ \lceil e \rceil$ *is finite*),
- *such that each place has at most one input transition* ($\forall b \in B,\ |{}^{\bullet}b| \leq 1$),
- *and such that there is no conflicts in the causal past of each transition* ($\forall e \in E,\ \neg\#\{e \cup \lceil e \rceil\}$).

We use the classical terminology of *conditions* and *events* to refer to the places B and the transitions E in an occurrence net.

Definition 3 (Branching process). *A* branching process *of a Petri net* $\mathcal{N} = \langle P, T, W, W_r, m_0 \rangle$ *is a labeled occurrence net* $\beta = \langle \mathcal{O}, l \rangle$ *where* $\mathcal{O} = \langle B, E, F, F_r \rangle$ *is an occurrence net and* $l : B \cup E \rightarrow P \cup T$ *is the labeling function such that:*

- $l(B) \subseteq P$ *and* $l(E) \subseteq T$,
- *for all* $e \in E$, *the restriction* $l_{|{}^{\bullet}e}$ *of* l *to* ${}^{\bullet}e$ *is a bijection between* ${}^{\bullet}e$ *and* ${}^{\bullet}l(e)$,
- *for all* $e \in E$, *the restriction* $l_{|{}^{\diamond}e}$ *of* l *to* ${}^{\diamond}e$ *is a bijection between* ${}^{\diamond}e$ *and* ${}^{\diamond}l(e)$,
- *for all* $e \in E$, *the restriction* $l_{|e^{\bullet}}$ *of* l *to* $e^{\bullet}$ *is a bijection between* $e^{\bullet}$ *and* $l(e)^{\bullet}$,
- *for all* $e_1, e_2 \in E$, *if* ${}^{\bullet}e_1 = {}^{\bullet}e_2$, ${}^{\diamond}e_1 = {}^{\diamond}e_2$ *and* $l(e_1) = l(e_2)$ *then* $e_1 = e_2$.

E should also contain the special event $\perp$, *such that:* ${}^{\bullet}\perp = \emptyset$, ${}^{\diamond}\perp = \emptyset$, $l(\perp) = \emptyset$, *and* $l_{|\perp^{\bullet}}$ *is a bijection between* $\perp^{\bullet}$ *and* m_0.

Branching processes can be partially ordered by a *prefix relation*. For example, the process $\{e_1, e_2, e_3\}$ is a prefix of the branching process in Fig. 1b in which t_1 is fired only once. There exists the greatest branching process according to this relation for any Petri net $\mathcal{N}$, which is called the *unfolding* of $\mathcal{N}$. Let $\beta = \langle B, E, F, F_r, l \rangle$ be a branching process.

A co-*set* in β is a set $B' \subseteq B$ of conditions that are in concurrence, that is to say without causal relation or conflict, *i.e.* $\forall b, b' \in B', \neg(b < b')$ and $\neg\# \bigcup_{b \in B'} ({}^{\bullet}b \cup \lceil {}^{\bullet}b \rceil)$.

A *configuration* of β is a set of events $E' \subseteq E$ which is causally closed and conflict-free, that is to say $\forall e' \in E', \forall e \in E,\ e < e' \Rightarrow e \in E'$ and $\neg\#E'$. In particular the local configuration $\lceil e \rceil$ of an event e is a configuration.

A *cut* is a maximal co-set (inclusion-wise). For any configuration E', we can define the cut $\mathsf{Cut}(E') = E'^{\bullet} \setminus {}^{\bullet}E'$, which is the marking of the Petri net obtained after executing the sequence of events in E'.

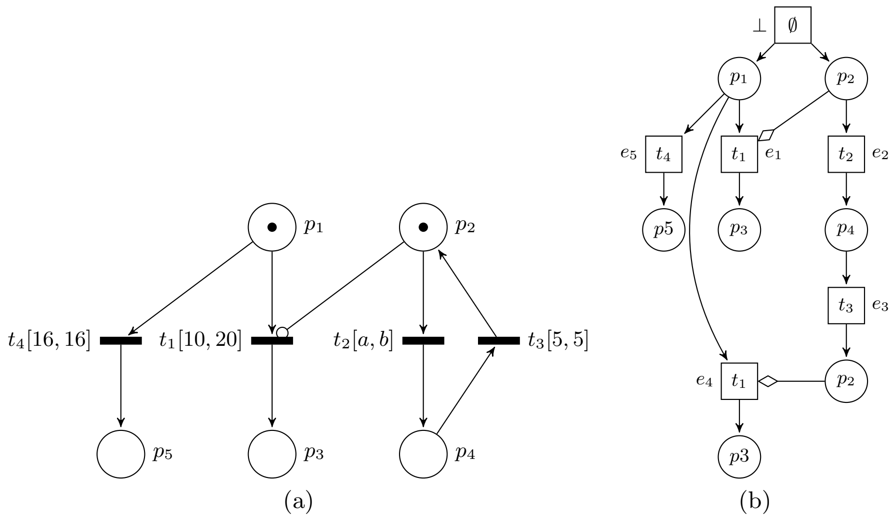


Fig. 1. A parametric stopwatch Petri net (a) and a branching processes of its underlying (untimed) Petri net (b). Stopwatch arcs are drawn with a circle tip and read arcs with a diamond tip.

An *extension* of β is a pair $\langle t, e\rangle$ such that e is an event not in E, ${}^\bullet e \cup {}^\diamond e \subseteq B$ is a co-set, the restriction of l to ${}^\bullet e$ is bijection between ${}^\bullet e$ and ${}^\bullet t$, the restriction of l to ${}^\diamond e$ is bijection between ${}^\diamond e$ and ${}^\diamond t$, and there is no $e' \in E$ s.t. $l(e') = t$, ${}^\bullet e' = {}^\bullet e$ and ${}^\diamond e' = {}^\diamond e$. Adding e to E and labeling e with t gives a new branching process.

Example 1. Fig. 1b shows a branching process obtained by unfolding the net presented in Fig. 1a (ignoring any timing or parameter information). The labels are figured inside the nodes. The branching process in Fig. 1b includes two firings of t_1 after executing the loop t_2, t_3. It could be repeated infinitely many times, leading to an infinite unfolding.

2.3 Stopwatch Petri Nets

A mainstream way of adding time to Petri nets is by equipping transitions with a time interval. This model is known as Time Petri nets (TPNs) [15,4]. We use a further extension of TPNs featuring stopwatches, called Stopwatch Petri nets (SwPNs) and originally proposed in [5]. Stopwatches allow the modelling of suspension / resumption of actions, which has many useful applications like modelling real-time preemptive scheduling policies [13].

The added expressivity comes at the expense of decidability: most interesting problems, such as reachability, liveness, etc. are undecidable for SwTPNs, even when bounded [5]. They are decidable however when restricting to bounded TPNs [4].

Definition 4 (Stopwatch Petri net). *A Stopwatch Petri net (with read arcs) SwPN is a tuple $\langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}\rangle$ where: $\langle P, T, W, W_r, m_0\rangle$ is a Petri net, $W_s \subseteq P \times T$ is the* stopwatch incidence relation, *and* $\mathsf{eft} : T \to \mathbb{Q}_{\geq 0}$ *and* $\mathsf{lft} : T \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$ *are functions satisfying $\forall t \in T$, $\mathsf{eft}(t) \leq \mathsf{lft}(t)$, and respectively called* earliest (eft) *and* latest (lft) *transition firing times.*

Given a SwPN $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}\rangle$, we denote by $\mathsf{Untimed}(\mathcal{N})$ the Petri net $\langle P, T, W, W_r \cup W_s, m_0\rangle$. Note that in $\mathsf{Untimed}(\mathcal{N})$ stopwatch arcs are transformed into read arcs. For any transition t, we define the set of its *activating* places as $^{\circ}t = \{p \in P \mid (p, t) \in W_s\}$. A transition is said to be *active* in marking M if it is enabled by M and $^{\circ}t \subseteq M$. An enabled transition that is not active is said to be *suspended*.

Intuitively, the semantics of TPN states that any enabled transition measures the time during which it has been enabled and an enabled transition can only fire if that time is within the time interval of the transition. Also, unless it is disabled by the firing of another transition, the transition must fire within the interval: a finite upper bound for the time interval then means that the transition will become urgent at some point. For SwPNs, the time during which the transition has been enabled progresses if and only if all its activating places are marked. Otherwise the stopwatch is "frozen" and keeps its current value.

More formally, we define the concurrent semantics of SwPNs using the time processes of Aura and Lilius [2]. Let us first recall the definition of these time processes:

Definition 5 (Time process). *A* time process *of a Stopwatch Petri net $\mathcal{N}$ is a pair $\langle E', \theta\rangle$, where E' is a configuration of (a branching process of)* $\mathsf{Untimed}(\mathcal{N})$ *and $\theta : E' \to \mathbb{R}_{\geq 0}$ is a timing function giving a firing date for any event of E'.*

Let $\langle E', \theta\rangle$ be a time process of a SwPN $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}\rangle$ and $\beta = \langle B, E, F, F_r, l\rangle$ be the associated branching process of $\mathsf{Untimed}(\mathcal{N})$. We note $^{*}e = {}^{\bullet}e \cup \{b \in {}^{\diamond}e \mid l(b) \in {}^{\diamond}l(e)\}$ the set of conditions that enabled an event e in the process E. These conditions are the consumed conditions and the read conditions due to read arcs, but it excludes the read conditions due to stopwatches.

Let $B' \subseteq E'^{\bullet}$ be a co-set and $t \in T$ be a transition enabled by $l(B')$. We define the *enabling date* of t by B' as: $\mathsf{TOE}(B', t) = \max(\{\theta({}^{\bullet}b) \mid b \in B' \wedge l(b) \in {}^{\bullet}t \cup {}^{\diamond}t\})$. This means that we measure the time during which the transition has been enabled. By extension, for any event e, we note $\mathsf{TOE}(e) = \mathsf{TOE}(^{*}e, l(e))$. We also define the set of events temporally preceding an event $e \in E'$ as: $\mathsf{Earlier}(e) = \{e' \in E' \mid \theta(e') < \theta(e)\}$, and we note $C_e = \mathsf{Cut}(\mathsf{Earlier}(e))$.

When dealing with stopwatches, the enabling date is not sufficient to determine the firing dates of the event, and is replaced by the notion of activity duration. For any co-set B', we define its *duration* up to some date θ as:

$$\mathsf{dur}(B', \theta) = \min\{\min_{e \in B'^{\bullet}}\{\theta(e)\}, \theta\} - \max_{b \in B'}\{\theta({}^{\bullet}b)\}$$

Then, for a transition t enabled by a co-set B', we define its *active* co*-sets* $\mathsf{Acos}(B', t)$ as all the co-sets A s.t.

– A is in the causal past of B',
– the conditions that enabled t in B also belong to A,
– t is active in A.

Finally the *activity duration* of the transition t at some date θ is:

$$\mathsf{adur}(B,t,\theta) = \sum_{A \in \mathsf{Acos}(B,t)} \mathsf{dur}(A,\theta)$$

By extension, for any event e, we note $\mathsf{Acos}(e) = \mathsf{Acos}({}^{*}e, l(e))$, and $\mathsf{adur}(e,\theta) = \mathsf{adur}({}^{*}e, l(e), \theta)$.

The semantics of a Stopwatch Petri net is then defined using the notion of *validity* of time processes.

Definition 6 (Valid time process for SwPNs). *A time process is* valid *iff* $\theta(\bot) = 0$ *and the following constraints are satisfied,* $\forall e \in E'$ $(e \neq \bot)$*:*

$$\theta(e) \geq \max(\{\theta({}^{\bullet}b) \mid b \in {}^{\bullet}e \cup {}^{\diamond}e\}) \tag{1}$$

$$\mathsf{adur}(e, \theta(e)) \geq \mathsf{eft}(l(e)) \tag{2}$$

$$\forall t \in \mathsf{en}(l(C_e)), \mathsf{adur}(C_e, t, \theta(e)) \leq \mathsf{lft}(t) \tag{3}$$

Condition 1 ensures that time progresses. Condition 2 states that to fire a transition $l(e)$ by an event e, it must have been active for at least a duration equal to $\mathsf{eft}(l(e))$ before being fired. Condition 3 states that at the firing date of an event e, the activity duration of no transition t can exceed its maximum firing time $\mathsf{lft}(t)$. Notice that if the former is purely local to the transition t, the latter refers to all enabled transitions in the net, which adds causality between events that are not causally related in the underlying untimed net.

It is easy to see that in the case of TPNs without stopwatches this definition reduces to the definition of Aura and Lilius [2] since, for any transition t enabled by a co-set B, we then have $\mathsf{Acos}(B,t) = B$ and $\forall\theta,\ \mathsf{dur}(B,\theta) = \theta - \mathsf{TOE}(B,t)$.

Note that, in this paper, we consider only Petri nets with non-zeno behavior.

Finally, we extend SwPNs with parameters, a model introduced in [16].

Definition 7 (Parametric Stopwatch Petri net). *A* Parametric *Stopwatch Petri net (PSwPN) is a tuple* $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ *where:* $\langle P, T, W, W_r, m_0 \rangle$ *is a Petri net,* W_s *is the stopwatch incidence relation as before,* Π *is a finite set of* parameters $(\Pi \cap (P \cup T) = \emptyset)$*,* D_Π *is a conjunction of linear constraints describing the set of* initial constraints *on the parameters, and* eft *and* lft *are functions on* T *such that for all* $t \in T$*,* $\mathsf{eft}(t)$ *and* $\mathsf{lft}(t)$ *are* rational linear expressions *on* Π *(or* $\mathsf{lft}(t)$ *is infinite).*

Definition 8 (Semantics of a PSwPN). *Let* $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$*. Given a rational valuation* v *on* Π *such that* $v(D_\Pi)$ *is true, we define the semantics of* $\mathcal{N}$ *as the SwPN* $\mathcal{N}_v = \langle P, T, W, W_r, W_s, m_0, v(\mathsf{eft}), v(\mathsf{lft}) \rangle$*.*

Example 2. Fig. 1a gives an example of a PSwPN. Notice that the time interval of transition t_2 refers to two parameters a and b. The only initial constraint is $D_\Pi = \{a \leq b\}$.

3 Unfolding

The method we propose to unfold parametric stopwatch Petri nets is based on an original way of determining conflicts in the net. In the non parametric timed case (no stopwatch), unfoldings built with this method differ in general from those of [6]. In [6], the emphasis is put on the on-line characteristic of the algorithm: it is a pessimistic approach that ensures that events and constraints put in the unfolding cannot be back into question. This leads possibly to unnecessary duplication of events. In contrast, we propose here an optimistic approach, which requires to dynamically compute the conflicts, and sometimes to backtrack on the constraints.

We propose to refine the conflict notion by defining a relation of direct conflict.

Definition 9 (Direct conflict). *Let $\mathcal{O} = \langle B, E, F, F_r \rangle$ be an occurrence net. Two events $e_1, e_2 \in E$ are in direct conflict, which we denote by $e_1 \ \mathsf{conf}\ e_2$, iff*

$$\begin{cases} \neg\#\{e_2 \cup \lceil e_2 \rceil \cup \lceil e_1 \rceil\} \\ \neg\#\{e_1 \cup \lceil e_1 \rceil \cup \lceil e_2 \rceil\} \\ {}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset \end{cases}$$

The first two conditions amount to say that ${}^\bullet e_1 \cup {}^\bullet e_2$ is a co-set. Direct conflicts are central to our study for they are at the root of all conflicts.

Example 3. The branching process presented in Fig. 1b contains direct conflicts $e_1 \ \mathsf{conf}\ e_5$, $e_4 \ \mathsf{conf}\ e_5$ and $e_1 \ \mathsf{conf}\ e_4$. e_1 and e_2 are only weakly ordered ($e_1 \nearrow e_2$).

3.1 Time Branching Processes

We shall now extend the notion of branching process with time information, allowing us to define the symbolic unfolding of PSwPNs. We do this in a way similar to extending configurations to time processes, by adding a function labeling events with their firing date. In a branching process however, some events may be in conflict, which means that some of them may not fire at all. We will account for this situation by labeling an event that never fires with $+\infty$.

The introduction of time in Petri nets reduces the admissible discrete behaviors, but induces new kinds of causal relations. For instance, in the TPN of Fig. 2(a), the firing of t_1 is only possible if t_3 is fired before t_2, which liberates the conflict between t_1 and t_2.

In the unfolding method of TPNs proposed in [6] these relations are handled by using read arcs in the unfolding, so that the firing of an event is duplicated according to the local state in which it is fired. The drawback in this approach is that it can lead to numerous unnecessary duplications of an event. For instance, considering now the TPN of Fig. 2(b), the firing of t_4 is possible in the states (p_1, p_4), (p_2, p_4) or (p_3, p_4), leading to a duplication of the event in each case.

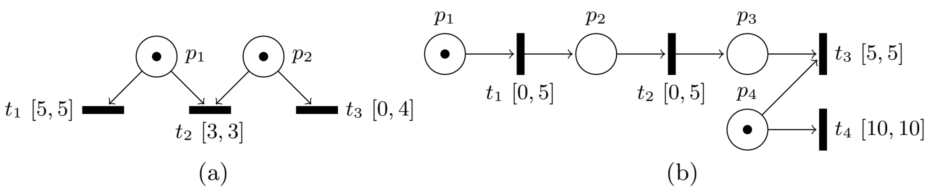


Fig. 2. Time-induced causality in time Petri nets

In our approach we try to express more local conditions by referring only to events in direct conflict. In the example of Fig. 2(b), this is expressed by the relation e_{t_3} conf e_{t_4} that allows the derivation of the constraints on the firing date of these two events. The cost of this approach is that until t_2 has not been fired, no restriction is put on the firing of t_4, and additional constraints are only added afterwards.

Definition 10 (Time branching process). *Given a SwPN $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}\rangle$, a* Time Branching Process *(TBP) of $\mathcal{N}$ is a tuple $\langle \beta, \theta\rangle$ where $\beta = \langle B, E, F, F_r, l\rangle$ is a branching process of* $\mathsf{Untimed}(\mathcal{N})$ *and $\theta : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a timing function giving a firing date for any event in E.*

As for time processes we define the notion of validity of the timing function of time branching process. In the sequel, we will say that a TBP is valid if its timing function is valid.

Definition 11 (Valid timing function for a TBP). *Given a PSwPN $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi\rangle$ and a valuation $v \in D_\Pi$ of the parameters, let $\Gamma = \langle B, E, F, F_r, l, \theta\rangle$ be a time branching process of $\mathcal{N}_v$. θ is a* valid *timing function for Γ iff $\theta(\bot) = 0$ and $\forall e \in E$ $(e \neq \bot)$,*

$$\Big[\theta(e) \neq \infty \;\wedge\; \theta(e) \geq \max(\{\theta({}^\bullet b) \mid b \in {}^\bullet e \cup {}^\diamond e\}) \tag{4}$$

$$\wedge\; \mathsf{adur}(e, \theta(e)) \geq v(\mathsf{eft}(l(e))) \tag{5}$$

$$\wedge\; \mathsf{adur}(e, \theta(e)) \leq v(\mathsf{lft}(l(e))) \tag{6}$$

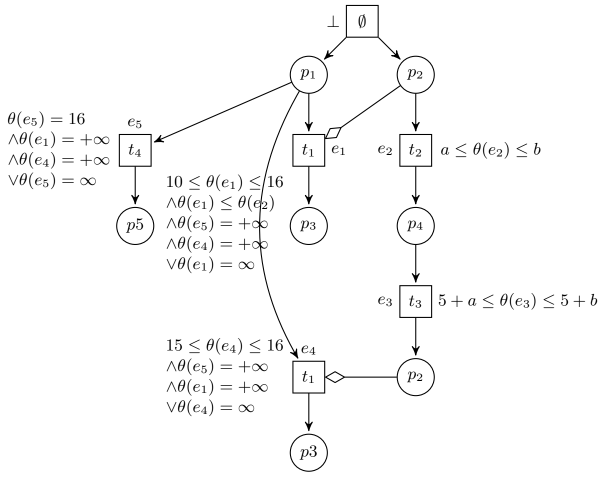
$$\wedge\; \forall e' \in E \text{ } s.t. \text{ } e' \mathrel{\mathsf{conf}} e,\; \theta(e') = \infty \tag{7}$$

$$\wedge\; \forall e' \in E \text{ } s.t. \text{ } e \nearrow e',\; \theta(e) \leq \theta(e')\Big] \tag{8}$$

$$\vee\; \Big[\theta(e) = \infty \;\wedge\; \exists b \in {}^\bullet e,\; \theta({}^\bullet b) = \infty\Big] \tag{9}$$

$$\vee\; \Big[\theta(e) = \infty \;\wedge\; \exists e' \in E \text{ } s.t. \text{ } (e \mathrel{\mathsf{conf}} e' \vee e \nearrow e')$$

$$\wedge\; \theta(e') \neq \infty \wedge \mathsf{adur}(e, \theta(e')) \leq v(\mathsf{lft}(l(e)))\Big] \tag{10}$$



$\perp$
$\emptyset$
p_1
p_2
$\theta(e_5) = 16$
$\wedge\theta(e_1) = +\infty$
$\wedge\theta(e_4) = +\infty$
$\vee\theta(e_5) = \infty$
e_5
t_4
t_1
e_1
e_2
t_2
$a \leq \theta(e_2) \leq b$
$10 \leq \theta(e_1) \leq 16$
$\wedge\theta(e_1) \leq \theta(e_2)$
$\wedge\theta(e_5) = +\infty$
$\wedge\theta(e_4) = +\infty$
$\vee\theta(e_1) = \infty$
$p5$
p_3
p_4
e_3
t_3
$5 + a \leq \theta(e_3) \leq 5 + b$
$15 \leq \theta(e_4) \leq 16$
$\wedge\theta(e_5) = +\infty$
$\wedge\theta(e_1) = +\infty$
$\vee\theta(e_4) = \infty$
e_4
t_1
p_2
$p3$

Fig. 3. A TBP with symbolic constraints for the PSwPN of Fig. 1a

Additionally, if $\exists\{e_0, e_1, \ldots, e_n\} \subseteq E$ *s.t.* $e_0 \nearrow e_1 \nearrow \cdots \nearrow e_n \nearrow e_0$ *then* $\exists i \in [0..n]$ *s.t* $\theta(e_i) = \infty$.

In these constraints, the usual operators are naturally extended to $\mathbb{R}_{\geq 0} \cup \{\infty\}$.

Eq. 4 ensures that time progresses. Eq. 5 constrains the earliest firing date and Eq. 6 the latest firing date of event e according to the parametric time interval associated to the transition $l(e)$. Also, an event e has a finite firing date iff it actually fires: this means that no other event e' in conflict with e can have a finite firing date e (Eq. 7). Finally with read arcs, in case the event e is weakly ordered before an event e', then with Eq. 8, e must fire before e'.

While Eqs. 5 to 7 define when an event can be fired, *i.e.* they give it a constrained but finite firing date, the last two equations define the cases in which an event cannot fire at all, giving it an infinite firing date. First, if one of the preconditions of event e has an infinite production date, then e has an infinite firing date (Eq. 9). Second, e may have an infinite firing date if it is in direct conflict with another event that has a finite firing date (Eq. 10). This implies that this event with a finite firing date will fire before e would have been forced to fire *i.e.* before its activity duration reaches the upper bound of the interval. Note that this is the only way to introduce infinite firing dates in the equation system. Those will then be propagated by Eq. 9.

Example 4. We consider the PSwPN of Fig. 1a. One of its TBP with symbolic constraints is presented on Fig. 3. For the values $a = 2$ and $b = 4$ of the parameters, a valid timing that verifies these constraints is $\theta(e_1) = \infty$, $\theta(e_2) = 3$, $\theta(e_3) = 8$, $\theta(e_4) = 15$ and $\theta(e_5) = \infty$.

3.2 Temporally Complete Time Branching Processes

Valid time branching processes as defined by Def. 10 and 11 do not necessarily contain correct executions, since a TBP is *a priori* incomplete in the sense that all timed constraints of the PSwPN may not be included yet in the TBP: by extending the TBP with additional events, new conflicts may appear that would add those constraints. We will therefore consider *temporally complete* TBP as defined below:

Definition 12 (Temporally complete TBP). *Let* $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ *be PSwPN and* v *be a valuation of its parameters. A valid TBP* $\langle B, E, F, F_r, l, \theta \rangle$ *of* $\mathcal{N}_v$ *is* temporally complete *if for all the extensions* $\langle t, e \rangle$ *of* $\langle B, E, F, F_r, l \rangle$,

$$\forall e' \in E \ s.t. \ \theta(e') \neq \infty, \ \mathsf{adur}({}^{*}e, t, \theta(e')) \leq v(\mathsf{lft}(t)) \tag{11}$$

This definition basically says that the firing date of all events in the TBP should be less or equal than the latest firing date of all possible extensions. Since the conflicts that have not yet been discovered will result from these extensions, this implies that all the events in the TBP are possible before these conflicts occur. It further ensures that all the parallel branches in the TBP have been unfolded to a same date. A similar condition can be stated for time processes.

Example 5. For the TBP of Fig. 3, the timing given in example 4, although valid, admits the firing of t_2 as an extension after e_3, and its maximal firing date is 13 which is inferior to the firing date of e_4. Thus, this TPB cannot be complete.

3.3 Extensions of a TBP

We now show how a given TBP can be extended with additional events, eventually leading to the construction of the whole unfolding.

Proposition 1. *Let* $\mathcal{N}$ *be a PSwPN and* v *a valuation of its parameters. Let* $\langle B, E, F, F_r, l, \theta \rangle$ *be a temporally complete TBP of* $\mathcal{N}_v$ *and let* $\langle t, e \rangle$ *be an extension of* $\beta = \langle B, E, F, F_r, l \rangle$*. Let* β' *be the branching process obtained by extending* β *by* $\langle t, e \rangle$*. Then there exists* θ' *such that* $\langle \beta', \theta' \rangle$ *is a valid TBP of* $\mathcal{N}_v$*.*

While the TBP obtained by the extension $\langle t, e \rangle$ is valid, it is not necessarily temporally complete: only the conflicts present in β' are considered but e could be prevented by conflicts that have not yet been added through other extensions. We have the following result however:

Proposition 2. *Let* $\langle \beta, \theta \rangle$ *be a temporally complete TBP of a PSwPN and let* $\langle t, e \rangle$ *be the extension of* β *with the smallest latest firing date. Then* $\langle \beta, \theta \rangle$ *extended by* $\langle t, e \rangle$ *is a temporally complete TBP.*

3.4 Symbolic Time Branching Processes

If we consider all the possible valuations of the parameters and all the possible valid timing functions for a given branching process of $\mathsf{Untimed}(\mathcal{N})$ we obtain what we call a *symbolic* TBP.

Definition 13 (Symbolic time branching process). *Let $\mathcal{N}$ be a PSwPN. A* symbolic time branching process *(STBP) Γ is a pair $\langle \beta, \mathcal{D} \rangle$ where $\beta = \langle B, E, F, F_r, l \rangle$ is a branching process of* $\mathsf{Untimed}(\mathcal{N})$*, $\mathcal{D}$ is a subset of $\mathbb{Q}^{|\Pi|} \times (\mathbb{R} \cup \{+\infty\})^{|E|}$ such that for all $\lambda = (v_1, \ldots, v_{|\Pi|}, \theta_1, \ldots, \theta_n, \ldots) \in \mathcal{D}$, if we note $E = \{e_1, \ldots, e_n, \ldots\}$, v_λ the valuation $(v_1, \ldots, v_{|\Pi|})$ and θ_λ the timing function such that $\forall i, \theta_\lambda(e_i) = \theta_i$, then $\langle \beta, \theta_\lambda \rangle$ is a valid TBP of $\mathcal{N}_{v_\lambda}$.*

In practice, the set $\mathcal{D}$ can be represented as a union of pairs $\langle \mathcal{E}_i, \mathcal{D}_i \rangle$ where $\mathcal{E}_i$ is a subset of the events of β and $\mathcal{D}_i$ is a rational convex polyhedron (possibly of infinite dimension) whose variables are the events in $\mathcal{E}_i$ plus the parameters of the net. Each point λ in $\mathcal{D}_i$ describes a value of the parameters and the finite values of the timing function on the elements of $\mathcal{E}_i$. For all elements not in $\mathcal{E}_i$, the timing function has value $+\infty$.

Now we can extend the notion of prefix to STBPs.

Definition 14 (Prefix of an STBP). *Let $\mathcal{N}$ be PSwPN whose set of parameters is Π. Let $\langle \beta, \bigcup_i \mathcal{E}_i, \bigcup_i \mathcal{D}_i \rangle$ and $\langle \beta', \bigcup_j \mathcal{E}'_j, \mathcal{D}' \rangle$ be two STBPs of $\mathcal{N}$. $\langle \beta, \mathcal{D} \rangle$ is a* prefix *of $\langle \beta', \mathcal{D}' \rangle$ if β is a prefix of β' and $\mathcal{D}$ is the projection of $\mathcal{D}'$ on the parameters plus the events of β.*

Finally, we can define the symbolic unfolding of a PSwPN.

Definition 15 (Symbolic unfolding). *The* symbolic unfolding *of a PSwPN $\mathcal{N}$ is the greatest STBP according to the prefix relation.*

This unfolding has the same size as the one computed for underlying Petri net. However, some events may not be able to take a finite firing date, in any circumstances. These events are not *possible* and will be useless. Thus, it will be sufficient to compute a prefix of the unfolding in which they are discarded.

3.5 Correctness and Completeness

In this subsection we give two results proving the correctness and completeness of our symbolic unfolding w.r.t. to the concurrent semantics of (P)SwPNs, that we have given in Section 2 as time processes.

We first establish a result on the configurations of TBP. For every TBP $\Gamma = \langle B, E, F, F_r, l, v, \theta \rangle$, we define the set $E_{<\infty} = \{e \in E \mid \theta(e) < \infty\}$ of all the events which may fire in the TBP.

Proposition 3. *Let $\Gamma = \langle B, E, F, F_r, l, v, \theta \rangle$ be valid TBP. Then $E_{<\infty}$ is a configuration.*

The correctness result for our approach states that all the time processes we can extract from our TBPs, and in particular those contained in the symbolic unfolding, are valid:

Theorem 1 (Correctness). *Let $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ be a parametric stopwatch Petri net and let $v \in D_\Pi$ be a valuation of its parameters. Let $\langle B, E, F, F_r, l, \theta \rangle$ be a temporally complete time branching process of $\mathcal{N}_v$. Let $E_{<\infty} = \{e \in E \mid \theta(e) < \infty\}$ and $\theta_{<\infty}$ is the restriction of θ to $E_{<\infty}$.*

$\langle E_{<\infty}, \theta_{<\infty} \rangle$ is a valid time process of $\mathcal{N}_v$.

Finally the following completeness result states that all valid time processes can be represented by a TBP. Therefore, since the symbolic unfolding contains all the valid TBPs, it also contains all the time processes of the PSwPN.

Theorem 2 (Completeness). *Let $\mathcal{N} = \langle P, T, W, W_r, W_s, m_0, \mathsf{eft}, \mathsf{lft}, \Pi, D_\Pi \rangle$ be a PSwPN and $v \in D_\Pi$ be a valuation of the parameters. Let $\langle B, E, F, F_r, l \rangle$ be a branching process of the underlying Petri net and $\langle E, \theta \rangle$ be a time process of the SwPN $\mathcal{N}_v$.*

There exists a temporally complete time branching process of $\mathcal{N}_v$, $\langle B', E', F', F'_r, l', \theta' \rangle$, such that $\forall e \in E, \exists e' \in E'$ s.t. $l(e) = l'(e')$ and $\theta(e) = \theta'(e')$.

The idea of the proof is to construct a TBP by adding all the events in conflict with some events of the time process.

4 Complete Prefixes of the Symbolic Unfolding

In this section, we show how to compute a complete prefix of the symbolic unfolding of a TPN. Consequently, from now we replace $v(\mathsf{eft}(t))$ by $\mathsf{eft}(t)$, $v(\mathsf{lft}(t))$ by $\mathsf{lft}(t)$, and we assume that $\mathsf{adur}(B, t, \theta) = \theta - \mathsf{TOE}(B, t)$, and ${}^\circ t = \emptyset$. In these conditions, we prove this prefix is finite.

A *consistent state* of the unfolding $\langle B, E, F, F_r, l, \mathcal{D} \rangle$ of a TPN $\mathcal{N} = \langle P, T, W, W_r, m_0, \mathsf{eft}, \mathsf{lft} \rangle$ is a pair $\langle A, \lambda \rangle$ such that $A \subseteq B$ is a cut and $\lambda \in \mathcal{D}$ and

– $\forall b \in A,\ \theta_\lambda({}^\bullet b) \neq \infty$,
– $\forall t \in T,\ {}^\bullet t \cup {}^\diamond t \subseteq l(A) \Rightarrow \max_{b \in A}\{\theta_\lambda({}^\bullet b)\} \leq \mathsf{TOE}(t, A) + \mathsf{lft}(t)$.

To compute a finite prefix we need to consider a finite number of states. However, the firing dates of the events grow continuously in the unfolding. Therefore, we define an equivalence relation between two consistent states by considering the age of the tokens (a reduced age since even ages can grow infinitely). Finally, we prove that the same transitions are firable from two equivalent states.

Definition 16 (reduced age of a condition). *For any* co*-set A, any timing function θ, and any condition $b \in A$, we define the* (reduced) age *of b in A as*

$$\mathsf{age}(b, \theta, A) = \min\{\max_{b' \in A}\{\theta({}^\bullet b')\} - \theta({}^\bullet b), \max\{K(t) \mid t \in T \wedge t \in l(b)^\bullet\}\}$$

where $K(t) = \begin{cases} \mathsf{eft}(t) & \text{if } \mathsf{lft}(t) = +\infty \\ \mathsf{lft}(t) & \text{otherwise.} \end{cases}$

Definition 17 (Equivalent consistent states). *Two consistent states $\langle A_1, \lambda_1 \rangle$ and $\langle A_2, \lambda_2 \rangle$ are equivalent iff $l(A_1) = l(A_2)$ and $\forall b_1 \in A_1, \forall b_2 \in A_2$, s.t. $l(b_1) = (b_2)$, $\mathsf{age}(b_1, \theta_{\lambda_1}, A_1) = \mathsf{age}(b_2, \theta_{\lambda_2}, A_2)$.*

Theorem 3 (Firing a transition in equivalent states). *Let $s_1 = \langle A_1, \lambda_1 \rangle$ and $s_2 = \langle A_2, \lambda_2 \rangle$ be two equivalent consistent states of the unfolding $\langle B, E, F, F_r, l, \mathcal{D} \rangle$ of a TPN $\mathcal{N} = \langle P, T, W, W_r, m_0, \mathsf{eft}, \mathsf{lft} \rangle$. If a transition t is firable from s_1 in an event e_1 at a date $\theta_{\lambda_1}(e_1) \geq \max_{b \in A_1}(\theta_{\lambda_1}({}^\bullet b))$, before all the other enabled transitions (i.e. $\forall t \in \mathsf{en}(l(A_1)) \theta_{\lambda_1}(e_1) \leq \mathsf{TOE}(t, A_1) + \mathsf{lft}(t)$), then*

1. *t is firable from s_2 in an event e_2 at the date $\theta_{\lambda_1}(e_1) - \max_{b \in A_1}(\theta_{\lambda_1}({}^\bullet b)) + \max_{b \in A_2}(\theta_{\lambda_2}({}^\bullet b))$, before all the other enabled transitions,*
2. *the states reached after the firing are equivalent.*

Knowing that the same behaviors are possible after equivalent states we can stop the construction of the unfolding by defining the notion of *cut-off* event.

Definition 18 (Cut-off event). *Let $\mathcal{N} = \langle P, T, W, W_r, m_0, \mathsf{eft}, \mathsf{lft} \rangle$ be a TPN. and let $\beta = \langle B, E, F, F_r, l, \mathcal{D} \rangle$ be a symbolic time branching process of $\mathcal{N}$. An event $e \in E$ is a* cut-off *event if there exists $e' \in E$ such that:*

- *$e' < e$,*
- *$l(e') = l(e)$,*
- *$\forall \lambda \in \mathcal{D}$, $\exists \lambda' \in \mathcal{D}$ s.t. $\langle C_{e'}, \lambda' \rangle$ and $\langle C_e, \lambda \rangle$ are equivalent.*

Definition 19 (Cut-off-free maximal prefix). *Let $\mathcal{N}$ be a TPN and let $\Gamma = \langle \beta, D \rangle$ be its symbolic unfolding. The* cut-off-free maximal prefix *CFP($\mathcal{N}$). is the greatest prefix of Γ that does not contain any cut-off events.*

We prove that the prefix computed contains at least the firing of each fireable transition of the unfolding, ad we show that this prefix is finite.

Theorem 4 (Completeness of the prefix). *Let $\mathcal{N} = \langle P, T, W, W_r, m_0, \mathsf{eft}, \mathsf{lft} \rangle$ be a TPN whose symbolic unfolding is $\langle B, E, F, F_r, l, \mathcal{D} \rangle$. Let CFP($\mathcal{N}$) $= \langle B^*, E^*, F^*, F_r^*, l^*, \mathcal{D}^* \rangle$. Then $\forall \lambda \in \mathcal{D}$, $\forall e \in E$ s.t. $\theta_\lambda(e) \neq \infty$, $\exists \lambda^* \in \mathcal{D}^*$, $\exists e^* \in E^*$, s.t. $\theta_{\lambda^*}(e^*) \neq \infty$ and $l(e^*) = l(e)$.*

Theorem 5 (Finiteness of the prefix). *For any (1-safe) time Petri net $\mathcal{N}$, the cut-off-free maximal prefix CFP($\mathcal{N}$) is finite.*

5 Conclusion

In this paper we have proposed a new technique for the unfolding of safe parametric stopwatch Petri nets that allow a symbolic handling of both time and parameters. To the best of our knowledge, this is the first time that the parametric or stopwatch cases are addressed in the context of unfoldings. Moreover, when restricting to the subclass of safe time Petri nets, our technique compares well with the previous approach of [6]. It indeed provides a more compact unfolding, by eliminating the duplication of transitions, and also removes the need for read arcs in the unfolding. As a tradeoff, the constraints associated with the firing times of events may seem slightly more complex.

We have partly implemented the technique in our tool, Romeo, whose 2.9.0 version can currently compute unfoldings of safe time Petri nets. The computation of the finite prefix is however not yet implemented. We propose instead to couple the method with a supervision technique that makes the unfolding finite based on a finite set of observations. This approach, that also works with parameters and stopwaches, is detailled in [10] with a case study.

Further work includes investigating non-safe bounded models and application of the unfolding technique to revisit the problems of model-checking and control.

References

1. Abdulla, P.A., Iyer, S.P., Nylen, A.: Unfoldings of unbounded Petri nets. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 495–507. Springer, Heidelberg (2000)
2. Aura, T., Lilius, J.: A causal semantics for time Petri nets. Theoretical Computer Science 243(2), 409–447 (2000)
3. Baldan, P., Busi, N., Corradini, A., Pinna, G.M.: Functorial concurrent semantics for Petri nets with read and inhibitor arcs. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 442–457. Springer, Heidelberg (2000)
4. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. on Soft. Eng. 17(3), 259–273 (1991)
5. Berthomieu, B., Lime, D., Roux, O.H., Vernadat, F.: Reachability problems and abstract state spaces for time Petri nets with stopwatches. Journal of Discrete Event Dynamic Systems - Theory and Applications (DEDS) 17(2), 133–158 (2007)
6. Chatain, T., Jard, C.: Complete finite prefixes of symbolic unfoldings of safe time Petri nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 125–145. Springer, Heidelberg (2006)
7. Chatain, T., Jard, C.: Sémantique concurrente symbolique des réseaux de Petri saufs et dépliages finis des réseaux temporels. In: Proceedings of NOTERE, Tozeur, Tunisia. IEEE Computer Society Press, Los Alamitos (May-June 2010)
8. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming 23, 151–195 (1994)
9. Esparza, J., Heljanko, K.: Unfoldings, A Partial-Order Approach to Model Checking. In: Monographs in Theoretical Computer Science. Springer, Heidelberg (2008)
10. Grabiec, B., Traonouez, L.-M., Jard, C., Lime, D., Roux, O.H.: Diagnosis using unfoldings of parametric time Petri nets. In: Proceedings of FORMATS, Vienna, Austria. LNCS. Springer, Heidelberg (to appear, September 2010)
11. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences 57, 94–124 (1998)
12. Khomenko, V., Koutny, M.: Branching processes of high-level Petri nets. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 458–472. Springer, Heidelberg (2003)
13. Lime, D., Roux, O.(H.): Formal verification of real-time systems with preemptive scheduling. Journal of Real-Time Systems 41(2), 118–151 (2009)
14. McMillan, K.L.: Using unfolding to avoid the state space explosion problem in the verification of asynchronous circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993)
15. Merlin, P.M.: A study of the recoverability of computing systems. PhD thesis, Dep. of Information and Computer Science, University of California, Irvine, CA (1974)
16. Traonouez, L.-M., Lime, D., Roux, O.H.: Parametric model-checking of stopwatch Petri nets. Journal of Universal Computer Science (J.UCS) 15(17), 3273–3304 (2009)

Recursive Timed Automata

Ashutosh Trivedi and Dominik Wojtczak

Computing Laboratory, Oxford University, UK

Abstract. We study *recursive timed automata* that extend timed automata with recursion. Timed automata, as introduced by Alur and Dill, are finite automata accompanied by a finite set of real-valued variables called clocks. Recursive timed automata are finite collections of timed automata extended with special states that correspond to (potentially recursive) invocations of other timed automata from their collection. During an invocation of a timed automaton, our model permits passing the values of clocks using both *pass-by-value* and *pass-by-reference* mechanisms. We study the natural reachability and termination (reachability with empty invocation stack) problems for recursive timed automata. We show that these problems are decidable (in many cases with the same complexity as the reachability problem on timed automata) for recursive timed automata satisfying the following condition: during each invocation either all clocks are passed by reference or none is passed by reference. Furthermore, we show that for recursive timed automata that violate this condition reachability/termination problems are undecidable for automata with as few as three clocks. We also establish similar results for two-player game extension of our model against reachability/termination objective.

1 Introduction

Recursion is one of the central ideas in mathematics and computer science. Informally, recursion is a process in which objects are defined in terms of other objects of same type. For instance, *recursive state machines* [1] are defined as collection of rather peculiar state machines whose states, in addition to being states in usual sense, are allowed to be other state machines, including themselves; or in other words, some states may correspond to potentially recursive invocation of other state machines. Similarly, *recursive Markov decision processes* [14] are collection of special Markov decision processes whose states may correspond to the invocation of other Markov decision processes. Following this line of work, we define recursive extension of timed automata [2] and study reachability and termination problems for this model.

Timed automata are finite automata—a finite set of locations and a finite set of transitions—coupled with a finite set of continuous variables, called clocks, which grow with uniform slope. Simple form of constraints on clocks are allowed to appear as guards on the transitions and as location invariants. Syntax of timed automata also permits resetting the clocks to zero. The reachability problem for timed automata with at least three clocks is known to be PSPACE-complete, while the reachability problem is NLOGSPACE-complete for timed automata with one clock.

Recursive timed automaton consists of finite number of *components* where each component is a special form of timed automaton with specially marked entry and exit locations. Moreover components can also have special form of locations, called *boxes*, that correspond to recursive invocation of other components. We allow passing the values of clocks to the invoked component in the sense that the values of these clocks are available to the invoked component, and passed clocks grow normally while the invoked component is under execution. Moreover, the passed clocks can be used in guards and location invariants inside the component, and transitions of the component may reset these clocks to zero. We allow two different mechanisms of passing the clocks: 1) *pass-by-value*, where upon returning from the invoked component clocks assume the value prior to the invocation; and 2) *pass-by-reference*, where upon returning from the invoked component their value is unaltered (it is as if a copy of these clocks is restored in the calling component). We say that a clock is *global* if it is always passed by reference, and it is *local* if it is always passed by value. Notice that, since there is no bound on the depth of recursive calls in our model, we will need to be able to analyse potentially infinitely many clocks, but at any point of the execution only a fixed number of them is not stopped.

We study reachability and termination (reachability of one of the exits with the empty calling context) problems on recursive timed automata. We show that the reachability problem of recursive timed automata is decidable, and EXPTIME-complete, if for every component, either all clocks of that component are passed by reference or none is passed by reference. Moreover, we study reachability games on recursive time automata, where the control state determines which of the two players picks the action to be performed. The objective of one player is reaching a particular subset of the control states, while the objective of the other player is complementary, i.e. avoiding them forever throughout the run of the recursive time automaton. We show that determining the winner of such games is in 2EXPTIME.

Applications. Much in the same way as recursive state machines can model *Boolean programs* [4] (or more general software systems using predicate abstraction [17]), it can be argued that recursive timed automata can model *hard real-time software systems* [8]. The need to use the dense semantics of time is more pressing in the case of real-time distributed software systems, i.e., computer programs that run on multiple autonomous computers communicating through computer network. Even after disallowing concurrency, verifying the correctness of real-time distributed software is a fantastic challenge as each participating computer has its own physical clock of varying frequency, while no global clock is available. Under such circumstances it is impossible to model system using discrete semantics of time without knowing the clock frequencies of participating computers. Hence it is natural to study these systems with dense semantics of time.

In [9], the authors study the problem of automatic generation of an optimal controller for an oil pump by defining this model as a 2-player game played on a time automaton. The actual controller used in practice for controlling this oil pump was a 400 lines long C program. Most C programs, apart from the simplest ones, make use of functions and recursive invocations of one function by another. Parameters to such functions are either passed by value or by reference (which in C language is done explicitly by passing a pointer). These kind of controllers operate on variables that are constantly growing in real-time, e.g. total time, oil pressure, temperature etc. The natural model to study correctness of such a system is a game played on recursive timed automaton with a safety critical objective, i.e. the aim for the controller is to avoid a certain set of bad states, while the aim of the other player, the "malicious environment", is trying to reach one of these states. If the controller has a winning strategy, i.e. no matter what how the environment behaves, none of the unsafe states will ever be reached, then the implementation of the controller is correct.

Related work. All the work with pushdown timed automata, see e.g. [10,11], has considered only global clocks. Bouajjani, Echahed, and Robbana [5] studied linear hybrid automata with pushdown stack and counters, and showed decidability of reachability in pushdown timed systems. Emmi and Majumdar [13] showed the decidability of language inclusion for implementation as timed pushdown automata and specification as timed automata with one clock; however they proved that it is undecidable when the specification is visibly pushdown timed automata even with one clock [13]. The work on timed automata with counters [7] studies extending time automata with multiple counters. The reachability problem for such systems is already undecidable without clocks, so the authors study several decidable subclasses of this model. Context-Free Timed Systems, studied in [6], are less expressive than our model, and [6] shows decidability of various verification problems for context-free timed systems with linear-hybrid observers (a variable that cannot be used in the constraints used on any edge, similar to prices/cost variables in timed automata).

The paper is organised as follows. In the next section we set definitions of key concepts like labelled transition systems, games, and recursive state machines. In Section 3 we introduce our model and define problems studied in this paper. In Section 4 we prove the undecidability of termination problem and games on the general model, while in Section 5 we discuss decidable subclasses and give complexity results.

2 Definitions

2.1 Preliminaries

Notation. We assume, the sets $\mathbb{N}$ of non-negative integers, $\mathbb{R}$ of reals and $\mathbb{R}_\oplus$ of non-negative reals. For $n \in \mathbb{N}$, let $[\![n]\!]_\mathbb{N}$ and $[\![n]\!]_\mathbb{R}$ denote the sets $\{0, 1, \ldots, n\}$, and $\{r \in \mathbb{R} \,|\, 0 \leq r \leq n\}$ respectively.

Labelled Transition System. A *labelled transition system* (LTS) is a tuple $\mathcal{L} = (S, A, X)$ where S is the set of *states*, A is the set of *actions*, and $X : S \times A \to S$ is the *transition function*. We say that an LTS $\mathcal{L}$ is *finite* (*discrete*) if both S and A are finite (countable). We write $A(s)$ for the set of actions available at $s \in S$, i.e., the set of actions $a \in A$ for which $X(s, a)$ is non-empty.

We say that $(s, a, s') \in S \times A \times S$ is a transition of $\mathcal{L}$ if $s' = X(s, a)$ and a *run* of $\mathcal{L}$ is a sequence $\langle s_0, a_1, s_1, \ldots \rangle \in S \times (A \times S)^*$ such that (s_i, a_{i+1}, s_{i+1}) is a transition of $\mathcal{L}$ for all $i \geq 0$. We write $\mathit{Runs}^\mathcal{L}$ ($\mathit{FRuns}^\mathcal{L}$) for the sets of infinite (finite) runs and $\mathit{Runs}^\mathcal{L}(s)$ ($\mathit{FRuns}^\mathcal{L}(s)$) for the sets of infinite (finite) runs starting from state s. For a finite run $r = \langle s_0, a_1, \ldots, s_n \rangle$ we write $\mathit{last}(r) = s_n$ for the last state of the run. A *strategy* in $\mathcal{L}$ is a function $\sigma : \mathit{FRuns}^{\mathcal{L}} \to A$ such that for all runs $r \in \mathit{FRuns}$ we have that $\sigma(r) \in A(\mathit{last}(r))$. We write $\Sigma^{\mathcal{L}}$ for the set of strategies in $\mathcal{L}$. For a state $s \in S$ and a strategy $\sigma \in \Sigma^{\mathcal{L}}$, we write $\mathrm{Run}(s, \sigma)$ for the unique run $\langle s_0, a_1, s_1, \ldots \rangle \in \mathit{Runs}^{\mathcal{L}}(s)$ such that $s_0 = s$ and for every $i \geq 0$ we have that $\sigma(r_n) = a_{n+1}$, where $r_n = \langle s_0, a_1, \ldots, s_n \rangle$ (here $r_0 = \langle s_0 \rangle$). For a set $F \subseteq S$ and a run $r = \langle s_0, a_1, \ldots \rangle$ we define $\mathit{Stop}(F)(r) = \inf \{ i \in \mathbb{N} \, : \, s_i \in F \}$.

Given a state $s \in S$ and a set of final states $F \subseteq S$ we say that a final state is reachable from s_0 if there is a strategy $\sigma \in \Sigma^{\mathcal{L}}$ such that $\mathit{Stop}(F)(\mathrm{Run}(s, \sigma)) < \infty$. A *reachability problem* is to decide whether in a given LTS a final state is reachable from a given initial state.

Games on Labelled Transition Systems. A *game arena* G is a tuple $(\mathcal{L}, S_{\text{Ach}}, S_{\text{Tor}})$, where $\mathcal{L} = (S, A, X)$ is an LTS, $S_{\text{Ach}} \subseteq S$ is the set of states controlled by player Achilles, and $S_{\text{Tor}} \subseteq S$ is the set of states controlled by player Tortoise. Moreover, sets S_{Ach} and S_{Tor} form a partition of the set S.

A strategy of player Achilles is a partial function $\alpha : \mathit{FRuns}^{\mathcal{L}} \to A$ such that for a run $r \in \mathit{FRuns}^{\mathcal{L}}$ we have that $\alpha(r)$ is defined if $\mathit{last}(r) \in S_{\text{Ach}}$, and $\alpha(r) \in A(\mathit{last}(r))$ for every such r. A strategy of player Tortoise is defined analogously. Let $\Sigma^{\mathcal{L}}_{\text{Ach}}$ and $\Sigma^{\mathcal{L}}_{\text{Tor}}$ be the set of strategies of player Achilles and Tortoise, respectively. The unique run $\mathrm{Run}(s, \alpha, \tau)$ from a state s when players use strategies $\alpha \in \Sigma^{\mathcal{L}}_{\text{Ach}}$ and $\tau \in \Sigma^{\mathcal{L}}_{\text{Tor}}$ is defined in a straightforward manner.

In a *reachability game* on G, rational players Achilles and Tortoise take turns to move a token along the states of $\mathcal{L}$. The decision to choose the successor state is made by the player controlling the current state. The objective of Achilles is to eventually reach certain states, while the objective of Tortoise is to avoid them forever. For an initial state s and a set of final states F, the lower value $\underline{\mathrm{Val}}^{\mathcal{L}}_F(s)$ of the reachability game is defined as the upper bound on the number of transitions that Tortoise can ensure before the game visits a state in F irrespective of the strategy of Achilles, and is equal to $\sup_{\tau \in \Sigma^{\mathcal{L}}_{\text{Tor}}} \inf_{\alpha \in \Sigma^{\mathcal{L}}_{\text{Ach}}} \mathit{Stop}(F)(\mathrm{Run}(s, \alpha, \tau))$. The concept of upper value is $\overline{\mathrm{Val}}^{\mathcal{L}}_F(s)$ is analogous and defined as $\inf_{\alpha \in \Sigma^{\mathcal{L}}_{\text{Ach}}} \sup_{\tau \in \Sigma^{\mathcal{L}}_{\text{Tor}}} \mathit{Stop}(F)(\mathrm{Run}(s, \alpha, \tau))$. If $\underline{\mathrm{Val}}^{\mathcal{L}}_F(s) = \overline{\mathrm{Val}}^{\mathcal{L}}_F(s)$ then we say that the reachability game is determined, or the value $\mathrm{Val}^{\mathcal{L}}_F(s)$ of the reachability game exists and it is such that $\mathrm{Val}^{\mathcal{L}}_F(s) = \underline{\mathrm{Val}}^{\mathcal{L}}_F(s) = \overline{\mathrm{Val}}^{\mathcal{L}}_F(s)$. We say that Achilles wins the reachability game if $\mathrm{Val}^{\mathcal{L}}_F(s) < \infty$. A *reachability game problem* is to decide whether in a given game arena G, an initial state s and a set of final states F, player Achilles has a strategy to win the reachability game.

2.2 Recursive State Machines

Recursive state machines (RSMs) generalise LTSs, and can be used to model systems exhibiting recursion and non-deterministic behaviour.

Definition 1 ([1]). *A* recursive state machine $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k)$ *is a tuple of components, where for each* $1 \leq i \leq k$ *component* $\mathcal{M}_i = (N_i, En_i, Ex_i, B_i, Y_i, A_i, X_i)$ *consists of:*

– *a finite set* N_i *of* nodes*, including the set* En_i *of* entry nodes *and the (disjoint from* En_i*) set* Ex_i *of* exit nodes.

- *a finite set B_i of* boxes.
- boxes-to-components mapping $Y_i : B_i \to \{1, 2, \ldots, k\}$ *that assigns every box to a component. To each box $b \in B_i$ we associate a set of* call ports $Call(b)$*, and a set of return ports $Ret(b)$:*

  $$Call(b) = \left\{(b, en) \,:\, en \in En_{Y_i(b)}\right\}, \text{ and } Ret(b) = \left\{(b, ex) \,:\, ex \in Ex_{Y_i(b)}\right\}.$$

  Let $Call_i = \cup_{b \in B_i} Call(b)$ and $Ret_i = \cup_{b \in B_i} Ret(b)$ be the set of call and return ports of component $\mathcal{M}_i$. We write $Q_i = N_i \cup Call_i \cup Ret_i$ for the union of the set of nodes, call ports and return ports, and we collectively refer to them as the vertices *of the component $\mathcal{M}_i$.*
- *a finite set A_i of* actions.
- *a* transition function $X_i : Q_i \times A_i \to Q_i$ *with a condition that call ports and exit nodes do not have any outgoing transitions.*

For the sake of simplicity, we assume that the set of boxes $B_1, \ldots, B_k$ and set of nodes $N_1, N_2, \ldots, N_k$ are mutually disjoint. We use symbols N, B, A, Q, X, etc. to denote the union of the corresponding symbols over all components. For example, $N = \cup_{i=1}^{k} N_i$.

Example 2. The visual presentation of a finite recursive state machine with three components M_1, M_2, and M_3 is depicted in Figure 1. Components are shown as thinly framed rectangles with their labels written close to upper right corner, e.g. see component M_1. Nodes of the components are shown as circles with their labels written inside them, e.g. see node u_1. Entry nodes of a component appear on the left of the component (see u_1), while exit nodes appear on the right (see u_4). Boxes are shown as thickly framed rectangles inside components labelled $b : M$, where b is the label of the box and M is the component it is mapped to. Call ports of boxes are drawn as small circles on the left of the box, while return ports are on the right. We omit labelling the call and return ports as these labels are clear from their position on the boxes. For example, call port (b_1, v_1) is the top small circle on the left-hand side of box b_1, since box b_1 is mapped to M_2 and v_1 is the top node on its left-hand side.

Intuitively, a run of an RSM starts at one of the entries of its components and proceeds via the edges from one state to another until it reaches an entry port of a box or an exit of the current component. In the former, this box is pushed onto the *stack of pending (recursive) calls* and the run starts from the corresponding entry of the component this box is mapped to. In the latter, if the stack of pending calls is empty then the run

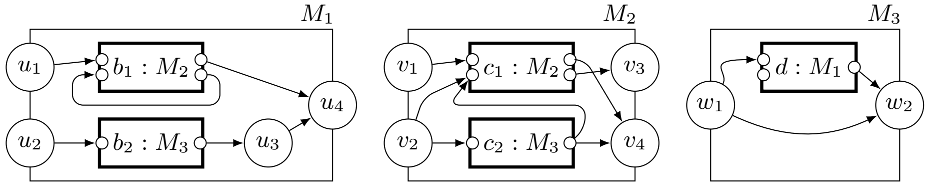


Fig. 1. Example recursive state machine taken from [1]

terminates; otherwise, it pops the box from the top of the stack and jumps to the exit port (of the just popped box) corresponding to the just reached exit of the component.

Formally, the semantics of a recursive state machine is given by a discrete LTS, whose states are pairs consisting of a sequence of boxes, called the context, and a vertex. The context corresponds to the sequence of unreturned component calls, and the vertex is a vertex of the current component.

Definition 3 (RSM semantics). *Let $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k)$ be an RSM where the component $\mathcal{M}_i$ is $(N_i, En_i, Ex_i, B_i, Y_i, A_i, X_i)$. The semantics of $\mathcal{M}$ is the countable labelled transition system $[\![\mathcal{M}]\!] = (S_\mathcal{M}, A_\mathcal{M}, X_\mathcal{M})$ where:*

- *$S_\mathcal{M} \subseteq B^* \times Q$ is the set of states;*
- *$A_\mathcal{M} = \cup_{i=1}^{k} A_i$ is the set of actions;*
- *$X_\mathcal{M} : S_\mathcal{M} \times A_\mathcal{M} \to S_\mathcal{M}$ is the transition function such that for $s = (\langle\kappa\rangle, q) \in S_\mathcal{M}$ and $a \in A_\mathcal{M}$, we have that $s' = X_\mathcal{M}(s, a)$ if and only if one of the following holds:*
  1. *the vertex q is a call port, i.e. $q = (b, en) \in Call$, and $s' = (\langle\kappa, b\rangle, en)$;*
  2. *the vertex q is an exit node, i.e. $q = ex \in Ex$ and $s' = (\langle\kappa'\rangle, (b, ex))$ where $(b, ex) \in Ret(b)$ and $\kappa = (\kappa', b)$;*
  3. *the vertex q is any other kind of vertex, and $s' = (\langle\kappa\rangle, q')$ and $q' \in X(q, a)$.*

A given $\mathcal{M}$ and a subset $Q' \subseteq Q$ of its nodes we define the set $[\![Q']\!]_\mathcal{M}$ as the set $\{(\langle\kappa\rangle, v') : \kappa \in B^* \text{ and } v' \in Q'\}$. We also define the set of terminal configurations *Term*$_\mathcal{M}$ as the set $\{(\langle\varepsilon\rangle, ex) : ex \in Ex\}$.

Given a recursive state machine $\mathcal{M}$, an initial node v, and a set of *final vertices* $F \subseteq Q$ the *reachability problem* on $\mathcal{M}$ is defined as the reachability problem on the LTS $[\![\mathcal{M}]\!]$ with the initial state $(\langle\varepsilon\rangle, v)$ and the set of final states $[\![F]\!]$. We also define *termination problem* as the reachability problem of one of the exits with the empty context. Hence, given a recursive state machine $\mathcal{M}$ and an initial node v, the termination problem on $\mathcal{M}$ is defined as the reachability problem on LTS $[\![\mathcal{M}]\!]$ with the initial state $(\langle\varepsilon\rangle, v)$ and the set of final states *Term*$_\mathcal{M}$. It is easy to show that the reachability problem is at least as hard as the termination problem. We can see this on the example in Figure 1: if we can decide whether state u_3 is reachable from $(\langle\varepsilon\rangle, u_2)$, we will also know whether it is possible to terminate from $(\langle\varepsilon\rangle, w_1)$ (simply because it is impossible to reach node u_3 from $(\langle\varepsilon\rangle, w_1)$). Hence, all the complexity upper bounds for the reachability problem in this paper apply also to the termination problem, and all the complexity lower bounds for the termination problem apply also to the reachability problem.

Games on Recursive State Machines. A partition $(Q_{\text{Ach}}, Q_{\text{Tor}})$ of vertices Q of an RSM $\mathcal{M}$ (between players Achilles and Tortoise) gives rise to recursive game arena $G = (\mathcal{M}, Q_{\text{Ach}}, Q_{\text{Tor}})$. Given an initial state, v, and a set of final states, F, the reachability game on $\mathcal{M}$ is defined as the reachability game on the game arena $([\![\mathcal{M}]\!], [\![Q_{\text{Ach}}]\!]_\mathcal{M}, [\![Q_{\text{Tor}}]\!]_\mathcal{M})$ with the initial state $(\langle\varepsilon\rangle, v)$ and the set of final states $[\![F]\!]_\mathcal{M}$. Also, the termination game $\mathcal{M}$ is defined as the reachability game on the game arena $([\![\mathcal{M}]\!], [\![Q_{\text{Ach}}]\!]_\mathcal{M}, [\![Q_{\text{Tor}}]\!]_\mathcal{M})$ with the initial state $(\langle\varepsilon\rangle, v)$ and the set of final states *Term*$_\mathcal{M}$. It is a well known result (see, e.g. [22,15]) that reachability games and termination games on RSMs are determined.

Complexity results for RSMs and their subclasses. The two most natural subclasses of RSMs are *1-box RSMs* and *1-exit RSMs*. 1-box RSMs are these RSMs that have just a single component and a single box inside of it (this box of course has to be mapped to that single component). On the other hand, an RSM is a 1-exit RSM iff each of its components has just one exit (and hence also all of its boxes), i.e. Ex_i is a singleton for all possible i-s. The general class of RSMs is sometimes referred to as *multi-exit RSMs*. Table 1 summarises some key results for RSMs and their subclasses.

Table 1. Complexity results for reachability objective for RSMs

# Players	1-box RSMs	1-exit RSMs	multi-exit RSMs
1	NLogSpace-complete [12]	PTime-complete [3]	PTime-complete [3]
2	PSpace-complete [20,18]	PTime-complete [22,15]	ExpTime-complete [22]

The results for 1-box RSMs are derived from the corresponding results for one-counter automata (for their definition, see, e.g. [16]), due to their exact correspondence: the counter value is equal to the number of boxes in the calling context, calling a box results in increasing the counter by 1, while reaching an exit corresponds to decreasing the counter by 1.

3 Recursive Timed Automata

Recursive timed automata (RTAs) extend classical timed automata [2] (TAs) with recursion feature similar to RSMs. Instead of defining TAs explicitly, we directly define RTAs whose degenerate case corresponds to TAs. Just as a TA is a finite automaton with a finite set of clocks (continuous variables), a recursive timed automaton is an RSM with a finite set of clocks which can be passed to components during invocation either by value or by reference. Before formally defining the syntax and semantics of RTA we need to introduce the concept of clock valuations, regions and zones.

3.1 Clocks, Clock Valuations, Regions and Zones

Let $\mathcal{C}$ be a finite set of *clocks*. In the definition of recursive timed automata (and timed automata [2]) constraints on clocks may appear in the guards on the transitions, where a clock or the difference of two clocks can be compared against natural numbers (in general with rational numbers). Let K be the largest such number. The set of *clock constraints* over $\mathcal{C}$ is the set of conjunctions of *simple constraints*, which are constraints of the form $c \bowtie i$ or $c - c' \bowtie i$, where $c, c' \in \mathcal{C}$, $i \in [\![K]\!]_{\mathbb{N}}$, and $\bowtie \in \{<, >, =, \leq, \geq\}$. Let SCC be the finite set of simple clock constraints.

A *clock valuation* on $\mathcal{C}$ is a function $\nu : \mathcal{C} \to \mathbb{R}_{\oplus}$ and we write V for the set of clock valuations. For a clock valuation $\nu \in V$ and delay $t \in \mathbb{R}_{\oplus}$ we write $\nu + t$ for the clock valuation defined by $(\nu + t)(c) = \nu(c) + t$, for all $c \in \mathcal{C}$. For a subset of clocks $C \subseteq \mathcal{C}$ and a clock valuation $\nu' \in V$, we write $\nu[C{:=}\nu']$ for the clock valuation where $\nu[C{:=}\nu'](c) = \nu'(c)$ if $c \in C$, and $\nu[C{:=}\nu'](c) = \nu(c)$ otherwise.

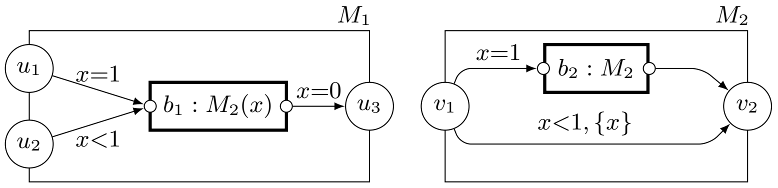


Fig. 2. Example recursive timed automaton

Clock valuation $\mathbf{0} \in V$ is a special valuation such that $\mathbf{0}(c) = 0$ for all $c \in \mathcal{C}$. Hence, for $C \subseteq \mathcal{C}$, we write $\nu[C{:=}\mathbf{0}]$ for the clock valuation where $\nu[C{:=}\mathbf{0}](c) = 0$ if $c \in C$, and $\nu[C{:=}\mathbf{0}](c) = \nu(c)$ otherwise.

A *clock region* is a maximal set $\zeta \subseteq V$, such that SCC(ν)=SCC(ν') for all $\nu, \nu' \in \zeta$. We write $\mathcal{R}$ for the finite set of clock regions. Every clock region is an equivalence class of the indistinguishability-by-clock-constraints relation, and vice versa. Note that ν and ν' are in the same clock region if and only if the integer parts of the clocks and the partial orders of the clocks, determined by their fractional parts, are the same in ν and ν'. We write $[\nu]$ for the clock region of ν. For a clock region ζ, a subset of clocks $C \subseteq \mathcal{C}$, and a clock valuation ν, we write $\zeta[C{:=}\nu]$ for the set $\{[\nu'[C{:=}\nu]] : \nu' \in \zeta\}$. Observe that if $\nu = \mathbf{0}$ then the set $\zeta[C{:=}\mathbf{0}]$ is a singleton, and we sometimes abuse the notation to write $\zeta[C{:=}\mathbf{0}]$ for the unique region.

A *clock zone* is a convex set of clock valuations, which is a union of a set of clock regions. We write $\mathcal{Z}$ for the set of clock zones. A set of clock valuations is a clock zone if and only if it is definable by a clock constraint.

3.2 Syntax

Definition 4 (Syntax). *A* recursive timed automaton $\mathcal{T} = (\mathcal{C}, (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k))$ *is a pair made of a set of clocks $\mathcal{C}$ and a collection of components $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$. Each component $\mathcal{T}_i = (N_i, En_i, Ex_i, B_i, Y_i, A_i, X_i, P_i, Inv_i, E_i, \rho_i)$ consists of:*

- *a finite set N_i of* nodes*, including the set En_i of entry nodes and the (disjoint from En_i) set Ex_i of exit nodes;*
- *a finite set B_i of* boxes*;*
- *boxes-to-components mapping $Y_i : B_i \to \{1, 2, \ldots, k\}$ that assigns every box to a component; (Call ports $Call(b)$ and return ports $Ret(b)$ of a box $b \in B_i$, and call ports $Call_i$ and return ports Ret_i of a component $\mathcal{T}_i$ are defined as before. We set $Q_i = N_i \cup Call_i \cup Ret_i$ and refer to this set as the set of vertices of $\mathcal{T}_i$.)*
- *a finite set A_i of* actions*;*
- *the* transition function $X_i : Q_i \times A_i \to Q_i$ *is with the condition that call ports and exit nodes do not have any outgoing transitions;*
- *pass-by-value mapping $P_i : B_i \to 2^{\mathcal{C}}$ that assigns every box the set of clocks that are passed by value to the component mapped to the box; (The rest of the clocks are assumed to be passed by reference.)*
- *the* invariant condition $Inv_i : Q_i \to \mathcal{Z}$*;*
- *the* action enabledness function $E_i : Q_i \times A_i \to \mathcal{Z}$*; and*
- *the* clock reset function $\rho_i : A_i \to 2^{\mathcal{C}}$.

We assume that the sets of boxes, nodes, etc. are mutually disjoint and we use symbols (N, B, Y, Q, P, X, etc.) without a subscript, to denote the union of the corresponding objects over all components. When we consider an RTA as an input of an algorithm, its size should be understood as the sum of the sizes of encodings of Q, $\mathcal{C}$, Inv, A, E, and X. Analogously as for RSMs, we define special subclasses of RTAs: *1-exit RTAs*, for which each component is allowed to have just one exit, and *1-box RTAs*, that just consist of a single component with a single box inside of them.

We say that a recursive timed automaton is *glitch-free* if for every box either all clocks are passed by value or none is passed by value, i.e. for each $b \in B$ we have that either $P(b) = \mathcal{C}$ or $P(b) = \emptyset$. Any general recursive timed automaton with one clock is trivially glitch-free.

Example 5. The visual presentation of a recursive timed automaton with two components M_1 and M_2, and one clock x is shown in Figure 2. The visual representation is similar to that in RSMs. However, each transition is labelled with a guard and the clocks to be reset, (e.g. transition from node v_1 to v_2 can be taken only when clock $x{<}1$, and after taking this transition, clock x is reset), and a box is labelled as $b : M(C)$ to denote that box b is mapped to M and all the clocks in the set C are passed by value, and the rest of the clocks are passed by reference. When the set C is empty, we just write $b : M$ for $b : M(\emptyset)$.

3.3 Semantics

A *configuration* of an RTA $\mathcal{T}$ is a tuple $(\langle\kappa\rangle, q, \nu)$, where $\kappa \in (B \times V)^*$ is (possibly empty) sequence of pairs of boxes and clock valuations, $q \in Q$ is a vertex and $\nu \in V$ is a clock valuation over $\mathcal{C}$ such that $\nu \in \mathit{Inv}(q)$. The sequence $\langle\kappa\rangle \in (B \times V)^*$ denotes the stack of pending recursive calls and the valuation of all the clocks at the moment that call was made, and we refer to this sequence as the context of the configuration. Technically, it suffices to store the valuation of clocks passed by value, because other clocks retain their value after returning from a call to a box, but storing all of them simplifies the notation. We denote the the empty context by $\langle\epsilon\rangle$. For any $t \in \mathbb{R}$, we let $(\langle\kappa\rangle, q, \nu){+}t$ equal the configuration $(\langle\kappa\rangle, q, \nu{+}t)$. Informally, the behaviour of an RTA is as follows. In configuration $(\langle\kappa\rangle, q, \nu)$ time passes before an available action is triggered, after which a discrete transition occurs. Time passage is available only if the invariant condition $\mathit{Inv}(q)$ is satisfied while time elapses, and an action a can be chosen after time t elapses only if it is enabled after time elapse, i.e., if $\nu{+}t \in E(q, a)$. If the action a is chosen then the successor state is $(\langle\kappa\rangle, q', \nu')$ where $q' \in \delta(q, a)$ and $\nu' = (\nu + t)[\rho(a) := \mathbf{0}]$. Formally, the semantics of an RTA is given by an LTS which has both an uncountably infinite number of states and transitions.

Definition 6 (RTA semantics). *Let $\mathcal{T} = (\mathcal{C}, (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k))$ be an RTA where each component is of the form $\mathcal{T}_i = (N_i, En_i, Ex_i, B_i, Y_i, A_i, X_i, P_i, \mathit{Inv}_i, E_i, \rho_i)$. The semantics of $\mathcal{T}$ is a labelled transition system $[\![\mathcal{T}]\!] = (S_\mathcal{T}, A_\mathcal{T}, X_\mathcal{T})$ where:*

- *$S_\mathcal{T} \subseteq (B \times V)^* \times Q \times V$, the set of states, is such that $(\langle\kappa\rangle, q, \nu) \in S_\mathcal{T}$ if $\nu \in \mathit{Inv}(q)$.*
- *$A_\mathcal{T} = \mathbb{R}_\oplus {\times} A$ is the set of* timed actions*;*

– *$X_\mathcal{T} : S_\mathcal{T} \times A_\mathcal{T} \to S_\mathcal{T}$ is the transition function such that for $(\langle\kappa\rangle, q, \nu) \in S_\mathcal{T}$ and $(t, a) \in A_\mathcal{T}$, we have $(\langle\kappa'\rangle, q', \nu') = X_\mathcal{T}((\langle\kappa\rangle, q, \nu), (t, a))$ if and only if the following condition holds:*
  1. *if the vertex q is a call port, i.e. $q = (b, en) \in Call$ then $t = 0$, the context $\langle\kappa'\rangle = \langle\kappa, (b, \nu)\rangle$, $q' = en$, and $\nu' = \nu$.*
  2. *if the vertex q is an exit node, i.e. $q = ex \in Ex$, $\langle\kappa\rangle = \langle\kappa'', (b, \nu'')\rangle$, and let $(b, ex) \in Ret(b)$, then $t = 0$; $\langle\kappa'\rangle = \langle\kappa''\rangle$; $q' = (b, ex)$; and $\nu' = \nu[P(b){:=}\nu'']$.*
  3. *if vertex q is any other kind of vertex, then $\nu{+}t' \in Inv(q)$ for all $t' \in [0, t]$; $\nu{+}t \in E(q, a)$; and $\langle\kappa'\rangle = \langle\kappa\rangle$, $q' \in X(q, a)$, and $\nu' = (\nu + t)[\rho(a) := \mathbf{0}]$.*

3.4 Reachability (Termination) Problems and Games

For a subset $Q' \subseteq Q$ of states of recursive timed automaton $\mathcal{T}$ we define the set $[\![Q']\!]_\mathcal{T}$ as the set $\{(\langle\kappa\rangle, q, \nu) \in S_\mathcal{T} \;:\; q \in Q'\}$. We also define the set of terminal configuration $Term_\mathcal{T}$ as the set $Term_\mathcal{T} = \{(\langle\varepsilon\rangle, q, \nu) \in S_\mathcal{T} \;:\; q \in Ex\}$.

Given a recursive timed automaton $\mathcal{T}$, an initial node q and valuation $\nu \in V$, and a set of *final vertices* $F \subseteq Q$, the *reachability problem* on $\mathcal{T}$ is defined as the reachability problem on LTS $[\![\mathcal{T}]\!]$ with the initial state $(\langle\varepsilon\rangle, q, \nu)$ and the set of final states $[\![F]\!]_\mathcal{T}$. As with RSMs, we also define *termination problem* as reachability of one of the exits with the empty context. Hence, given an RTA $\mathcal{T}$ and an initial node q and a valuation $\nu \in V$, the termination problem on $\mathcal{T}$ is defined as the reachability problem on LTS $[\![\mathcal{T}]\!]$ with the initial state $(\langle\varepsilon\rangle, q, \nu)$ and the set of final states $Term_\mathcal{T}$.

Example 7. Consider the RTA shown in Figure 2. From the vertex u_1 of M_1 there is no path that visits the exit node u_3 with the empty calling context, as the only transition available form u_1 is to wait until clock $x = 1$, and then invoking component M_2 which recursively calls itself forever if the value of clock $x = 1$. On the other hand, from node u_2 there are infinitely many paths that reach u_3 with the empty context. Notice that termination at u_3 is possible only when delay at u_2 is 0 time-units, as upon exiting box b clock x is tested against 0. Since clock x was passed by value to component M_2, the current value of clock x is the one before the invocation of M_2, and hence the clock reset inside M_2 does not help.

A partition $(Q_{\text{Ach}}, Q_{\text{Tor}})$ of vertices Q of an RTA $\mathcal{T}$ gives rise to a recursive timed game arena $\Gamma = (\mathcal{T}, Q_{\text{Ach}}, Q_{\text{Tor}})$. Given an initial vertex q, a valuation $\nu \in V$ and a set of final states F, the reachability game on Γ is defined as the reachability game on the game arena $([\![\mathcal{T}]\!], [\![Q_{\text{Ach}}]\!]_\mathcal{T}, [\![Q_{\text{Tor}}]\!]_\mathcal{T})$ with the initial state $(\langle\varepsilon\rangle, (q, \nu))$ and the set of final states $[\![F]\!]_\mathcal{T}$. Also, termination game on $\mathcal{T}$ is defined as the reachability game on the game arena $([\![\mathcal{T}]\!], [\![Q_{\text{Ach}}]\!]_\mathcal{T}, [\![Q_{\text{Tor}}]\!]_\mathcal{T})$ with the initial state $(\langle\varepsilon\rangle, (q, \nu))$ and the set of final states $Term_\mathcal{T}$.

4 Undecidability Results

The following is one of the key results of this paper.

Theorem 8. *Termination problem is undecidable for recursive timed automata with at least three clocks. Moreover, termination game problem is undecidable for recursive timed automata with at least two clocks.*

For the undecidability proofs we use reduction from the halting problem of two-counter *Minsky machines* [19]. A Minsky machine $\mathcal{A}$ is a tuple (L, C, D) where: $L = \{\ell_0, \ell_1, \dots, \ell_n\}$ is the set of states including the distinguished terminal state ℓ_n; $C = \{c_1, c_2\}$ is the set of two *counters*; $D = \{\delta_0, \delta_1, \dots, \delta_{n-1}\}$ is the set of transitions of the following type:

1. (increment c) $\delta_i : c := c + 1$; goto ℓ_k,
2. (test-and-decrement c) δ_i : if $(c > 0)$ then $(c := c - 1$; goto $\ell_k)$ else goto ℓ_m,

where $c \in C$, $\delta_i \in D$ and $\ell_k, \ell_m \in L$.

A configuration of a Minsky machine is a tuple (ℓ, c, d) where $\ell \in L$ and c, d are natural numbers that specify the value of counters c_1 and c_2, respectively. The initial configuration is $(\ell_0, 0, 0)$. A run of Minsky machine is a (finite or infinite) sequence of configurations $\langle s_0, s_1, \dots\rangle$ where s_0 is the initial configuration, and the relation between subsequent configurations is governed by transitions at their respective states. The run is a finite sequence if and only if the last configuration is the terminal state ℓ_n. Note that a Minsky machine has only one run starting from the initial configuration. *Termination problem* for a Minsky machine asks whether its unique run is finite. It is well known ([19]) that the termination problem for a two-counter Minsky machine is undecidable. In the rest of the section we show a reduction from the halting problem of Minsky machines to the termination games on RTA with two clocks. The reduction to the termination problem for (1-player) RTAs with three clocks is in the technical report version of this paper [21].

We fix the clocks set $\mathcal{C} = \{x, y\}$, and we describe the construction of the central component $\mathsf{HALT}^{\mathcal{A}}$ with nodes $\ell_0, \ell_1, \dots, \ell_n$ with the entry node ℓ_0 and the exit node ℓ_n. A configuration (ℓ_i, c, d) of a Minsky machine corresponds to the configuration $(\langle\varepsilon\rangle, \ell_i, \nu)$ such that $\nu(x) = 2^{-c} \cdot 3^{-d}$ and $\nu(y) = 0$. Decrementing and incrementing counter c is simulated by doubling and halving, resp., of the clock x, while decrementing and incrementing the counter d is simulated by tripling, and *thirding*[1], resp., the value of clock x. Testing counter c (resp. d) against 0 can be simulated by multiplying clock x by some power of 3 (resp., 2) and then comparing it against 3 (resp. 2). The components for doubling (DB) and halving (HF) the value of clock x, and testing whether the value of clock x is of the form 2^{-i} or 3^{-i} ($\mathsf{P2O}$ or $\mathsf{P3O}$, resp.) are given in Figure 3. Due to space constraints, we omit the description of components for tripling, TR, and thirding, TH, of clock x. However such components are very similar to components DB and HF. All components function as intended only when upon entering them, the value of clock y is 0. The vertices of these components are partitioned between Achilles and Tortoise: the only vertex controlled by Tortoise (shown as black squares) is the return port (B_8, ex_9) in component M_8. The component DB', invoked from inside the component M_8, is similar to gadget DB, however it doubles the value of clock y, while assuming that clock x is set to 0. We assume that the node labelled ☹ has no outgoing transitions. The behaviour of components $\mathsf{P2O}$ and $\mathsf{P3O}$ is as follows: if clock x is of the form 2^{-i} and 3^{-i}, resp., a run starting at that component's entry will terminate at its bottom exit and if clock x is not of that form then such a run will terminate at that component's top exit. Since the precise construction

[1] Dividing by three.

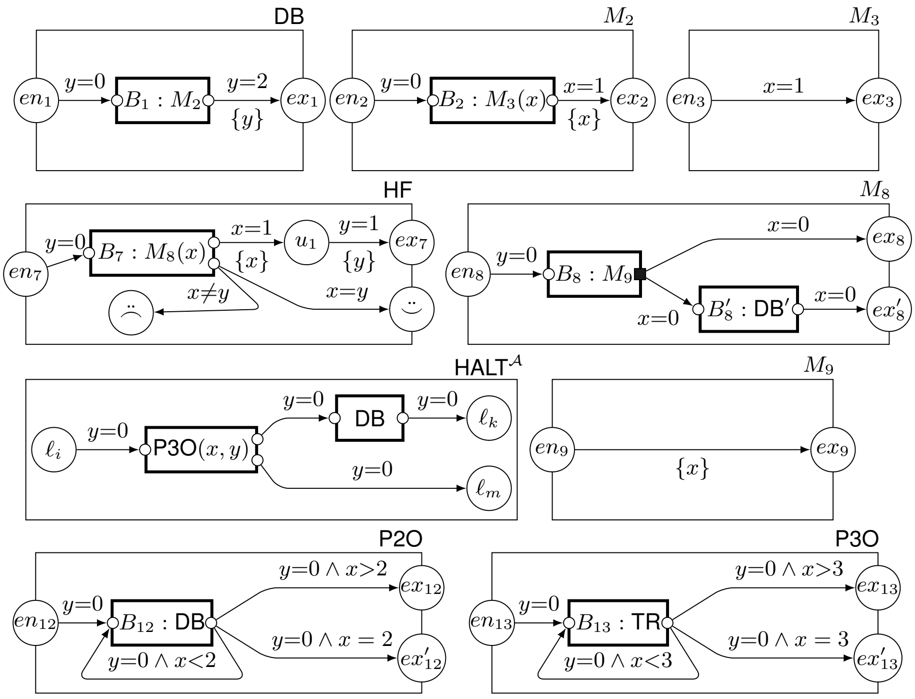


Fig. 3. Components for doubling DB and halving HF the value of clock x, and checking whether x is of the form 2^{-i} and 3^{-i}

of HALT$^{\mathcal{A}}$ is straightforward, we just present in Figure 3 a schema that simulates the test-and-decrement c operation: δ_i : if $(c > 0)$ then $(c := c - 1;$ goto $\ell_k)$ else goto ℓ_m. Whenever a run reaches node ℓ_i inside HALT$^{\mathcal{A}}$, a box mapped to P3O is called that tests whether the value of counter c is zero. After returning from P3O both clocks are restored and the exit port indicates whether clock c is zero or not. If clock c is zero then the run proceeds straight to node ℓ_m; otherwise the value of the counter c is decremented by 1 by multiplying clock x by two using component DB and the run proceeds to node ℓ_k. It should be easy to see now how to encode the increment c operation and how to combine them all into the HALT$^{\mathcal{A}}$ component.

To make the proofs more comprehensible, we show a run in an RTA using three different forms of transitions $s \xrightarrow{g,C}_t s'$, $s \rightsquigarrow s'$, and $s \xrightarrow{M(C)}_* s'$ defined in the following way.

1. The transitions of form $s \xrightarrow{g,C}_t s'$, where $s = (\langle\kappa\rangle, n, \nu), s' = (\langle\kappa\rangle, n', \nu')$ are configurations of an RTA, g is a clock constraint, C is a set of clocks, and t is a real number, holds if there is a transition in the RTA from vertex n to n' with guard g and clock reset set C, moreover $\nu' = (\nu + t)[C := \mathbf{0}]$.

2. The transitions of the form $s \rightsquigarrow s'$, where $s = (\langle\kappa\rangle, n, \nu), s' = (\langle\kappa'\rangle, n', \nu')$, correspond to the following cases:
   (a) transitions from a call port to an entry node, i.e. , $n = (b, en)$ for some box $b \in B$ and $\kappa' = \langle\kappa, (b, \nu)\rangle$ and $n' = en \in En$, while $\nu' = \nu$.
   (b) transition from an exit node to a return port (which also restore the value of clocks passed by value), i.e. $\langle\kappa\rangle = \langle\kappa'', (b, \nu'')\rangle$, $n = ex \in Ex$, and $n' = (b, ex) \in Ret(b)$ and $\kappa' = \kappa''$, while $\nu' = \nu[P(b) = \nu'']$.
3. The transitions of the form $s \xrightarrow{M(C)}_* s'$, called *summary edges*, where $s = ((\langle\kappa\rangle, n, \nu)), s' = (\langle\kappa'\rangle, n', \nu')$ are such that $n = (b, en)$ and $n' = (b, ex)$ are call and return ports, resp., of a box b mapped to M which passes by value to M the clocks in C.

For the sake of simplicity, in this section instead of presenting the context information in the form $(b, \nu) \in B \times V$, we write $(b, (\nu(x), \nu(y)))$ if some clock is passed by value to b, else we just write b. We also write a configuration $(\langle\kappa\rangle, n, \nu)$ as $(\langle\kappa\rangle, n, (\nu(x), \nu(y)))$.

Proposition 9. *For any context $\kappa \in (B \times V)^*$, any box $b \in B$, and $x_0 \in [0,1]$ we have that $(\langle\kappa\rangle, (b, en_1), (x_0, 0)) \xrightarrow{\mathsf{DB}}_* (\langle\kappa\rangle, (b, ex_1), (2 \cdot x_0, 0))$.*

Proof. Component DB, shown in Figure 3, uses components M_2 and M_3. The following is the unique run starting from the configuration $(\langle\kappa\rangle, (b, en_1), (x_0, 0))$ terminating at the configuration $(\langle\kappa\rangle, (b, ex_1), (2 \cdot x_0, 0))$.

$$\begin{aligned}
&(\langle\kappa\rangle, (b, en_1), (x_0, 0)) \rightsquigarrow (\langle\kappa, b\rangle, en_1, (x_0, 0))\\
\xrightarrow{y=0}_0\ &(\langle\kappa, b\rangle, (B_1, en_2), (x_0, 0)) \rightsquigarrow (\langle\kappa, b, B_1\rangle, en_2, (x_0, 0))\\
\xrightarrow{y=0}_0\ &(\langle\kappa, b, B_1\rangle, (B_2, en_3), (x_0, 0)) \rightsquigarrow (\langle\kappa, b, B_1, B_2(x_0, 0)\rangle, en_3, (x_0, 0))\\
\xrightarrow{x=1}_{(1-x_0)}\ &(\langle\kappa, b, B_1, B_2(x_0, 0)\rangle, ex_3, (1, 1-x_0)) \rightsquigarrow (\langle\kappa, b, B_1\rangle, (B_2, ex_3), (x_0, 1-x_0))\\
\xrightarrow{x=1,\{x\}}_{(1-x_0)}\ &(\langle\kappa, b, B_1\rangle, ex_2, (0, 2-2\cdot x_0)) \rightsquigarrow (\langle\kappa, b\rangle, (B_1, ex_2), (0, 2-2\cdot x_0))\\
\xrightarrow{y=2,\{y\}}_{(2\cdot x_0)}\ &(\langle\kappa, b\rangle, ex_1, (2\cdot x_0, 0)).
\end{aligned}$$

The intermediate steps of this sequence of transitions can be easily verified. □

Proposition 10. *For any context $\kappa \in (B \times V)^*$, any box $b \in B$, and $x_0 \in [0,1]$, there exists a unique strategy of Achilles such that*

$$\textit{either } (\langle\kappa\rangle, (b, en_7), (x_0, 0)) \xrightarrow{\mathsf{HF}}_* (\langle\kappa\rangle, (b, ex_7), (\frac{x_0}{2}, 0)),$$

$$\textit{or } (\langle\kappa\rangle, (b, en_7), (x_0, 0)) \xrightarrow{\mathsf{HF}}_* (\langle\kappa\rangle, (b, \ddot\smile), (x_0, x_0))).$$

Moreover, for other strategies of Achilles there exists a strategy of Tortoise such that component HF does not terminate.

Proof. The main observation here is that, in component HF, starting from the configuration $(\langle\kappa\rangle, (b, en_7), (x_0, 0))$ Achilles has a strategy to terminate only if he chooses to delay the time by $\frac{x_0}{2}$ in component M_9 (called via box B_8). The evolution of the run from $(\langle\kappa\rangle, (b, en_7), (x_0, 0))$ to $(\langle\kappa, b, B_7(x_0, 0), B_8\rangle, en_9, (x_0, 0))$ is straightforward. Now, in component M_9 Achilles can wait for an arbitrary amount of time before taking a transition to ex_9 and resetting clock x. Let us assume that he waits for t time units, and hence $(\langle\kappa, b, B_7(x_0, 0)\rangle, (B_8, ex_9), (0, t))$ is reached which is controlled by Tortoise. Now Tortoise has a choice between making a transition to ex_8 (believing that $t = \frac{x_0}{2}$) or invoking the component B'_8 (when suspecting that $t \neq \frac{x_0}{2}$).

If Tortoise believes that $t = \frac{x_0}{2}$ then he makes a transition to ex_8 and thus the system reaches the configuration $(\langle\kappa, b\rangle, (B_7, ex_8), (x_0, t))$ giving rise to the following run:

$$(\langle\kappa, b\rangle, (B_7, ex_8), (x_0, t)) \xrightarrow{x=1,\{x\}}_{(1-x_0)} (\langle\kappa, b\rangle, u_1, (0, 1 - x_0 + t))$$
$$\xrightarrow{y=1,\{y\}}_{(x_0-t)} (\langle\kappa, b\rangle, ex_7, (x_0 - t, 0)) \rightsquigarrow (\langle\kappa\rangle, (b, ex_7), (x_0 - t, 0)).$$

Hence if $t = \frac{x_0}{2}$ then the run terminates at configuration $(\langle\varepsilon\rangle, ex_7, (\frac{x_0}{2}, 0))$.

On the other hand if Tortoise believes that $t \neq \frac{x_0}{2}$, then he invokes the component DB' to double the value of clock y (while keeping the value of clock x equal to 0), and makes a transition, via exit ex'_8, to the configuration $(\langle\kappa, b, B_7(x_0, 0)\rangle, ex'_8, (0, 2 \cdot t))$. Since x_0 was passed by value, it is restored upon exiting from box B_7 and the configuration reached is $(\langle\kappa, b\rangle, (B_7, ex'_8), (x_0, 2 \cdot t))$. If Tortoise's suspicion was right and $t \neq \frac{x_0}{2}$ then the only transition available to Achilles is to move to the ☹ vertex which never terminates. Otherwise Achilles can only move to configuration $(\langle\kappa\rangle, (b, ☺), (x_0, x_0))$ and terminate. Hence, it is clear that the only winning strategy for Achilles is to choose $t = \frac{x_0}{2}$. □

Proposition 11. *For any context $\kappa \in (B \times V)^*$, any box $b \in B$, and $x_0 \in [0, 1]$ we have that starting from configuration $(\langle\kappa\rangle, (b, en_{12}), (x_0, 0))$ the component* $\mathsf{P2O}$ *terminates at $(\langle\kappa\rangle, (b, ex'_{12}), (x_0, 0))$ only when $x_0 = 2^{-i}$ for some $i \in \mathbb{N}$ and otherwise it terminates at $(\langle\kappa\rangle, (b, ex_{12}), (x_0, 0))$.*

From Propositions 9, 10, and 11 (and similar results related to other components) it follows that Achilles has a strategy to terminate at ℓ_n in component $\mathsf{HALT}^{\mathcal{A}}$ if and only if the Minsky machine $\mathcal{A}$ terminates.

5 Decidability Results

5.1 Region Abstraction

For every RTA $\mathcal{T}$ we define regional equivalence relation $\mathcal{E}_R \subseteq S_\mathcal{T} \times S_\mathcal{T}$ in the following way: For configurations $s = (\langle\kappa\rangle, q, \nu)$ and $s' = (\langle\kappa'\rangle, q', \nu')$ we have that $s, s' \in \mathcal{E}_R$, or equivalently we write $[s] = [s']$, if $q = q'$, $[\nu] = [\nu']$, and $\kappa = (b_1, \nu_1), (b_2, \nu_2), \ldots, (b_n, \nu_n)$ and $\kappa' = (b'_1, \nu'_1), (b'_2, \nu'_2), \ldots, (b'_n, \nu'_n)$ are such that for every $1 \leq i \leq n$ we have $[\nu_i] = [\nu'_i]$ and $b_i = b'_i$.

A relation $B \subseteq S_\mathcal{T} \times S_\mathcal{T}$ defined over the set of configurations $S_\mathcal{T}$ of a recursive timed automaton is a *time-abstract bisimulation* if for every pair of configurations $s_1, s_2 \in S_\mathcal{T}$ such that $(s_1, s_2) \in B$, for every timed action $(t, a) \in A_\mathcal{T}$ such that $X_\mathcal{T}(s_1, (t, a)) = s'_1$, there exists a timed action $(t', a) \in A_\mathcal{T}$ such that $X_\mathcal{T}(s_2, (t', a)) = s'_2$ and $(s'_1, s'_2) \in B$.

Proposition 12. *Regional equivalence relation for glitch-free recursive timed automata is a time-abstract bisimulation.*

Proof. Let us fix configurations $s = (\langle\kappa\rangle, q, \nu)$ and $s' = (\langle\kappa'\rangle, q', \nu')$ such that $[s] = [s']$, timed action $(t, a) \in X_{\mathcal{T}}$ such that $X_{\mathcal{T}}(s, (t, a)) = s_a (= (\kappa_a, (q_a, \nu_a)))$. We need to find (t', a) such that $X_{\mathcal{T}}(s', (t', a)) = s'_a (= (\kappa'_a, (q'_a, \nu'_a)))$ and $[s_a] = [s'_a]$. There are following three cases.

1. The vertex q is a call port, i.e. $q = (b, en) \in Call$. In this case $t = 0$, the context $\langle\kappa_a\rangle = \langle\kappa, (b, \nu)\rangle$, $q_a = en$, and $\nu_a = \nu$. Since $q' = q (= (b, en))$ is then also a call port, we have that $t' = 0$, and $\langle\kappa'_a\rangle = \langle\kappa', (b, \nu')\rangle$, $q'_a = en$, and $\nu'_a = \nu_a$. It is trivial to show that $[s_a] = [s'_a]$.
2. The vertex q is an exit node, i.e. $q = ex \in Ex$, and let $\langle\kappa\rangle = \langle\kappa_*, (b, \nu_*)\rangle$ and $(b, ex) \in Ret(b)$. In this case $t = 0$; context $\langle\kappa_a\rangle = \langle\kappa_*\rangle$; $q_a = (b, ex)$; and $\nu_a = \nu[P(b){:=}\nu_*]$. Let the context $\langle\kappa'\rangle$ be $\langle\kappa'_*, (b, \nu'_*)\rangle$. Since again $q' = q (= ex)$ is also an exit node we have that $t' = 0$, $\langle\kappa'_a\rangle = \langle\kappa'_*\rangle$ and $\nu'_a = \nu'[P(b){:=}\nu'_*]$. We need to show that $[\nu_a] = [\nu'_a]$. Notice that for glitch-free RTAs there are exactly two cases to consider:
   - $P(b) = \mathcal{C}$. In this case $\nu_a = \nu_*$ and $\nu'_a = \nu'_*$, and since $[\nu_*] = [\nu'_*]$ we get that $[\nu_a] = [\nu'_a]$.
   - $P(b) = \emptyset$. In this case $\nu_a = \nu$ and $\nu'_a = \nu'$, and since $[\nu] = [\nu']$ we get that $[\nu_a] = [\nu'_a]$.
3. if vertex q is of any other kind, then the result follows by classical region equivalence relation.

The proof is now complete. □

The following proposition follows from the 2nd case in the proof of Proposition 12.

Proposition 13. *For general (non glitch-free) RTA with two clocks the successors of regionally equivalent configurations are not necessarily regionally equivalent.*

By using two boxes mapped to DB in a sequence, one is able to construct a new component D_1 that multiplies the value of clock x by $2 \cdot 2 = 2^{2^0} \cdot 2^{2^0} = 2^{2^1} = 4$. (See, e.g. how component DB is exploited in component P2O in Figure 3.) In general, by using two boxes mapped to D_i, one is able to construct a new component D_{i+1} that multiplies the value of clock x by $2^{2^i} \cdot 2^{2^i} = 2^{2^{i+1}}$. So, to solve reachability problem for general RTA with two clocks, one needs to consider doubly-exponentially many (in the size of the RTA) partitions of a region.

Proposition 12 allows us to extend the concept of region abstraction in the setting of glitch-free RTA. Before we introduce the abstraction, we need to define some notations.

For $\zeta, \zeta' \in \mathcal{R}$, we say that clock region ζ' is in the future of clock region ζ, or that ζ is in the past of ζ', if there are $\nu \in \zeta$, $\nu' \in \zeta'$ and delay $d \in \mathbb{R}_{\oplus}$ such that $\nu' = \nu + d$; we then write $\zeta \to_* \zeta'$. We say that ζ' is the *time successor* of ζ if $\zeta \to_* \zeta'$, $\zeta \neq \zeta'$, and $\zeta \to_* \zeta'' \to_* \zeta'$ implies $\zeta'' = \zeta$ or $\zeta'' = \zeta'$ and write $\zeta \to \zeta'$ and $\zeta' \leftarrow \zeta$. Time successor definition is extended to n-th time successor in a natural way: we say that ζ' is the n-th successor of ζ, and write $\zeta \to_{+n} \zeta'$, if there is a sequence of regions $\langle\zeta_1, \zeta_2, \ldots, \zeta_n\rangle$ such that $\zeta_1 = \zeta$, $\zeta_n = \zeta'$ and $\zeta_i \to \zeta_{i+1}$ for every $1 \leq i < n$. In this case we also write $[\zeta_1, \zeta_n]$ for the union of regions $\zeta_1, \ldots \zeta_n$.

Definition 14 (Region Abstraction). *Let $\mathcal{T} = (\mathcal{C}, (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k))$ be a glitch-free RTA, where each $\mathcal{T}_i$ is the tuple $(N_i, En_i, Ex_i, B_i, Y_i, A_i, X_i, P_i, Inv_i, E_i, \rho_i)$. The region abstraction of $\mathcal{T}$ is a* finite *RSM $\mathcal{T}^{\text{RG}} = ({\mathcal{T}_1}^{\text{RG}}, {\mathcal{T}_2}^{\text{RG}}, \ldots, {\mathcal{T}_k}^{\text{RG}})$ where for each $1 \le i \le k$, component ${\mathcal{T}_i}^{\text{RG}} = ({N_i}^{\text{RG}}, {En_i}^{\text{RG}}, {Ex_i}^{\text{RG}}, {B_i}^{\text{RG}}, {Y_i}^{\text{RG}}, {A_i}^{\text{RG}}, {X_i}^{\text{RG}})$ consists of:*

- *a finite set ${N_i}^{\text{RG}} \subseteq (N_i \times \mathcal{R})$ of* nodes *such that $(n, \zeta) \in {N_i}^{\text{RG}}$ if $\zeta \in Inv(n)$. Moreover, ${N_i}^{\text{RG}}$ includes the sets of entry nodes ${En_i}^{\text{RG}} \subseteq En_i \times \mathcal{R}$ and exit nodes ${Ex_i}^{\text{RG}} \subseteq Ex_i \times \mathcal{R}$;*
- *a finite set ${B_i}^{\text{RG}} = B_i \times \mathcal{R}$ of* boxes*;*
- boxes-to-components mapping ${Y_i}^{\text{RG}} \colon {B_i}^{\text{RG}} \to \{1, 2, \ldots, k\}$ *is such that ${Y_i}^{\text{RG}}(b, \zeta) = Y_i(b)$. To each box $(b, \zeta) \in {B_i}^{\text{RG}}$ we associate a set of call ports $Call^{\text{RG}}(b, \zeta)$, and a set of return ports $Ret^{\text{RG}}(b, \zeta)$:*

$$Call^{\text{RG}}(b, \zeta) = \left\{(((b, \zeta), en), \zeta') \;:\; \zeta' \in \mathcal{R} \text{ and } en \in En_{Y_i(b)}\right\}, \text{ and}$$
$$Ret^{\text{RG}}(b, \zeta) = \left\{(((b, \zeta), ex), \zeta') \;:\; \zeta' \in \mathcal{R} \text{ and } ex \in Ex_{Y_i(b)}\right\}.$$

  Let ${Call_i}^{\text{RG}}$ and ${Ret_i}^{\text{RG}}$ be the set of call and return ports of component ${\mathcal{T}_i}^{\text{RG}}$. We write ${Q_i}^{\text{RG}} = {N_i}^{\text{RG}} \cup {Call_i}^{\text{RG}} \cup {Ret_i}^{\text{RG}}$ for the vertices *of the component ${\mathcal{T}_i}^{\text{RG}}$.*
- *${A_i}^{\text{RG}} \subseteq \mathbb{N} \times A_i$ is the set of actions, such that if $(h, a) \in {A_i}^{\text{RG}}$ (here h is number of region hops before taking a) then $h \le (2 \cdot |\mathcal{C}|)^K$, where $K \in \mathbb{N}$ is the largest constant that appears in one of the clock constraints in E or Inv;*
- *a* transition function *${X_i}^{\text{RG}} : {Q_i}^{\text{RG}} \times {A_i}^{\text{RG}} \to {Q_i}^{\text{RG}}$ with the natural condition that call ports and exit nodes do not have any outgoing transitions. Moreover, for $q, q' \in {Q_i}^{\text{RG}}$, $(h, a) \in {A_i}^{\text{RG}}$ we have that $q' = {X_i}^{\text{RG}}(q, (h, a))$ if one of the following conditions holds:*
  1. *$q = (n, \zeta) \in {N_i}^{\text{RG}}$, there exists a region ζ_a such that $\zeta \to_{+h} \zeta_a$, $[\zeta, \zeta_a] \subseteq Inv_i(n)$, $\zeta_a \in E_i(n, a)$, and*
     - *if $q' = (n', \zeta')$ then $\zeta' = \zeta_a[\rho_i(a) := \mathbf{0}]$ and $X_i(n, a) = n'$.*
     - *if $q' = (((b, \zeta'), en), \zeta'')$ then $\zeta' = \zeta'' = \zeta_a[\rho_i(a) := \mathbf{0}]$ and $X_i(n, a) = (b, en)$.*
  2. *$q = (((b, \zeta_{\text{Saved}}), ex), \zeta_{\text{Curr}})$ is a return port of ${\mathcal{T}_i}^{\text{RG}}$. Let $\zeta = \zeta_{\text{Saved}}$ if $P_i(b) = \mathcal{C}$ and $\zeta = \zeta_{\text{Curr}}$ otherwise. There exists a region ζ_a such that $\zeta \to_{+h} \zeta_a$, and $[\zeta, \zeta_a] \subseteq Inv_i((b, ex))$, $\zeta_a \in E_i((b, ex), a)$, and*
     - *if $q' = (n', \zeta')$ then $\zeta' = \zeta_a[\rho_i(a) := \mathbf{0}]$ and $X_i(n, a) = n'$.*
     - *if $q' = (((b, \zeta'), en), \zeta'')$ then $\zeta' = \zeta'' = \zeta_a[\rho_i(a) := \mathbf{0}]$ and $X_i(n, a) = (b, en)$.*

The following proposition is a direct consequence of Proposition 12 and the definition of region abstraction.

Proposition 15. *Reachability (termination) problems and games on glitch-free RTA $\mathcal{T}$ can be reduced to solving reachability (termination) problems and games, respectively, on the corresponding region abstraction $\mathcal{T}^{\text{RG}}$.*

5.2 Computational Complexity

All the results stated here concern glitch-free recursive timed automata only and their formal proofs can be found in [21]. First, we summarise the complexity results for the reachability problem for glitch-free RTAs in Table 2.

Table 2. Complexity results for glitch-free RTAs

# Players	RTAs with 1 clock	RTAs with at least 2 clocks
1	PTIME-complete	EXPTIME-complete
2	EXPTIME-complete	2EXPTIME

By examining the reduction of RTAs to the corresponding RSMs via region abstraction in the previous section, it can be observed that in the case where all the clocks are passed by reference (i.e. they are global) only the number of internal nodes and exits grows exponentially, not the number of boxes. It is simply because the clocks values are never being restored to the value they had before the box was called and hence the valuation of the clocks does not have to be stored at the boxes in the region abstraction. This observation allows us to provide better complexity upper and lower bounds for the reachability problem and games on 1-box RTAs with global clocks, summarised in Table 3, because 1-box RSMs can be analysed a lot more efficiently than multi-exit RSMs. Since the number of exits can grow arbitrarily large after region abstraction is applied to a 1-exit RTA with just a single global clock, no similar improvement can be obtained for 1-exit RSMs with only global clocks.

Table 3. Complexity results for 1-box RTAs with only global clocks

# Players	1-box RTAs with 1 global clock	1-box RTAs with at least 2 global clocks
1	PTIME-complete	PSPACE (PSPACE-complete for 3+ clocks)
2	PSPACE-complete	EXPSPACE (and EXPTIME-hard)

On the other hand, if all clocks are local then only the number of boxes grows exponentially, not the number of control states (in particular the number of exit ports of each box does not increase). This allows us to provide a much better complexity upper and lower bounds for 1-exit RTAs with only local clocks. We summarise the results for the reachability problem for this subclass of RTAs in Table 4. Again, even if there is only one single local clock, the number of boxes can grow arbitrarily large after the region abstraction is applied to such a system, hence no similar improvements can be achieved when restricting the model to 1-box RTAs with local clocks.

Table 4. Complexity results for 1-exit RTAs with only local clocks

# Players	1-exit RTAs with 1 local clock	1-exit RTAs with at least 2 local clocks
1	PTIME-complete	EXPTIME-complete
2	PTIME-complete	EXPTIME-complete

6 Conclusion

We defined a natural extension of boolean programs with real-time clocks. These clocks, among others, may either correspond to physical time or other continuous values read from sensors. Just like in any advanced imperative programming language, we allow to pass these clocks by value or by reference. We showed that unfortunately arbitrary mixing of these two kinds of variable passing leads to undecidability. On the other hand, if we disallow it, the model becomes decidable and for many special subclasses of this model, the computational complexity is not higher than PSPACE for 1 player setting and EXPTIME for 2 players setting, which is the same as the respective reachability analysis of ordinary finite-state timed automata.

Acknowledgment

The authors are supported in part by EPSRC grants EP/F001096 and EP/G050112/1.

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Transactions on Programming Languages and Systems 27, 786–818 (2005)
2. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. 126 (1994)
3. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. In: ACM SIGSOFT 1998, pp. 175–188 (1998)
4. Ball, T., Rajamani, S.: The slam toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Bouajjani, A., Echahed, R., Robbana, R.: On the automatic verification of systems with continuous variables and unbounded discrete data structures. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 64–85. Springer, Heidelberg (1995)
6. Bouajjani, A., Echahed, R., Robbana, R.: Verification of context-free timed systems using linear hybrid observers. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 118–131. Springer, Heidelberg (1994)
7. Bouchy, F., Finkel, A., Sangnier, A.: Reachability in timed counter systems. Electronic Notes in Theoretical Computer Science 239, 167–178 (2009); Joint Proceedings of the 8th, 9th, and 10th International Workshops on Verification of Infinite-State Systems (INFINITY 2006, 2007, 2008)
8. Buttazzo, G.C.: Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, Santa Clara (2004)
9. Cassez, F., Jessen, J.J., Larsen, K.G., Raskin, J.-F., Reynier, P.-A.: Automatic synthesis of robust and optimal controllers — an industrial case study. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 90–104. Springer, Heidelberg (2009)
10. Dang, Z.: Binary reachability analysis of pushdown timed automata with dense clocks. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 506–517. Springer, Heidelberg (2001)
11. Dang, Z.: Pushdown timed automata: a binary reachability characterization and safety verification. Theor. Comput. Sci. 302(1-3), 93–121 (2003)

12. Demri, S., Gascon, R.: The Effects of Bounding Syntactic Resources on Presburger LTL. J. Logic Computation 19(6), 1541–1575 (2009)
13. Emmi, M., Majumdar, R.: Decision problems for the verification of real-time software. In: Hybrid Systems: Computation and Control, pp. 200–211 (2006)
14. Etessami, K., Yannakakis, M.: Recursive markov decision processes and recursive stochastic games. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 891–903. Springer, Heidelberg (2005)
15. Etessami, K.: Analysis of recursive game graphs using data flow equations. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 282–296. Springer, Heidelberg (2004)
16. Etessami, K., Wojtczak, D., Yannakakis, M.: Quasi-Birth-Death processes, Tree-like QBDs, Probabilistic 1-Counter Automata, and Pushdown Systems. Performance Evaluation 67(9), 837–857 (2010); Special Issue of QEST 2008 (2008)
17. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
18. Jancar, P., Sawa, Z.: A note on emptiness for alternating finite automata with a one-letter alphabet. Inf. Process. Lett. 104(5), 164–167 (2007)
19. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Englewood Cliffs (1967)
20. Serre, O.: Parity games played on transition graphs of one-counter processes. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 337–351. Springer, Heidelberg (2006)
21. Trivedi, A., Wojtczak, D.: Recursive timed automata. Oxford University Computing Laboratory technical report, RR-10-09 (2010)
22. Walukiewicz, I.: Pushdown processes: Games and model checking, pp. 62–74 (1996)

Probabilistic Contracts for Component-Based Design[⋆]

Dana N. Xu, Gregor Gössler, and Alain Girault

INRIA, France

Abstract. We define a probabilistic contract framework for the construction of component-based embedded systems, based on the theory of Interactive Markov Chains. A contract specifies the assumptions a component makes on its context and the guarantees it provides. Probabilistic transitions allow for uncertainty in the component behavior, e.g. to model observed black-box behavior (internal choice) or reliability. An interaction model specifies how components interact.

We provide the ingredients for a component-based design flow, including (1) contract satisfaction and refinement, (2) parallel composition of contracts over disjoint, interacting components, and (3) conjunction of contracts describing different requirements over the same component. Compositional design is enabled by congruence of refinement.

1 Introduction

Typical embedded and distributed systems often encompass unreliable software or hardware components, as it may be technically or economically impossible to make a system entirely reliable. As a result, system designers have to deal with probabilistic specifications such as "the probability that this component fails at this point of its behavior is less than or equal to 10^{-4}". More generally, uncertainty in the observed behavior is introduced by abstraction of black-box — or simply too complex — behavior of components, the environment, or the execution platform. In this paper we introduce a framework for the design of correct systems from probabilistic, interacting components.

Figure 1(a) shows a Link system that transmits data between a Client and a Server. The Link receives a request from the Client and encodes the request before sending it to the Server. The encoding process fails with probability 0.02. After receiving a response from the Server, it decodes the data before delivering it to the Client. To model components, we adopt the discrete time Interactive Markov Chain (IMC) semantic model [8], which combines Labeled Transition System (LTS) and Markov Chain. Figure 1(b) shows an IMC describing the network Link of Figure 1(a). From its initial state l_0, the Link goes to state l_1 as soon as it receives (rec) a request from a Client; the probability that it delivers (del') this request to the Server is 0.98 and the probability that it fails before delivering to the Server is 0.02. The Link goes to state l_4 immediately after

[⋆] Supported by the European project COMBEST no. 215543.

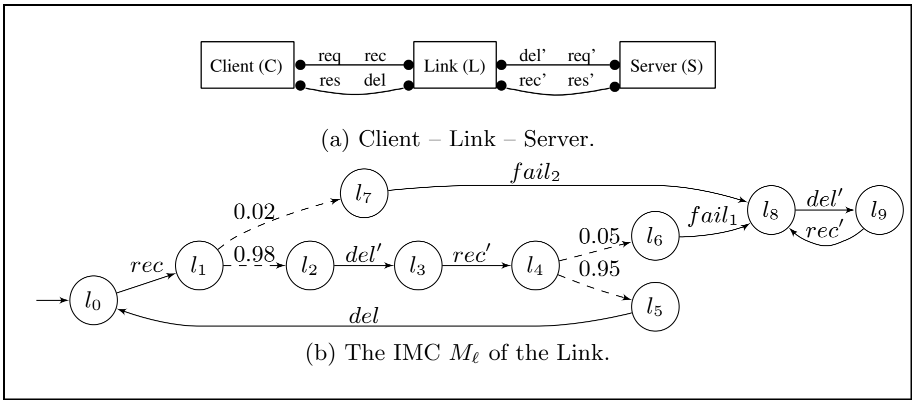


Fig. 1. An example of IMC: a Client-Link-Server

receiving a response (rec') from the Server; the probability that it delivers (del) the response to the Client is 0.95 and the probability of failing to do so is 0.05. In state l_8, the Link may still communicate with the Server regarding other services, but will not deliver any response to the Client.

Components communicate through interactions, that is, synchronized action transitions. Interactions are essential in component frameworks, as they allow the modeling of how components cooperate and communicate. We use the BIP framework [7] to model interactions between components.

Since the deploying context of a component is not known at design time, we use probabilistic *contracts* to specify and reason about correct behaviors of a component. Contracts have first been introduced in [11]. They allow us to specify what a component can expect from its context, what it must guarantee, and explicitly limit the responsibilities of both.

The framework we propose here allows us to model components, their interactions, and uncertainty in their observed behavior (§2). It supports different steps in a design flow: refinement, satisfaction, and abstraction (§3), parallel composition (§4.1), and conjunction (shared refinement) (§4.2). We prove that these operations satisfy the desired properties of *independent implementability* and *congruence* for parallel composition, and *soundness* for conjunction. Thus,

- refinement is compositional, that is, contracts over different components can be refined and implemented independently;
- the parallel composition of two contracts is satisfied by the parallel composition of any two implementations of the contracts; and
- several contracts C_i over the same component may be used to independently specify different requirements, possibly over different subsets of the component interactions. The conjunction is a common refinement of all C_i.

As pointed out in [2], conjunction of probabilistic specifications is non trivial, as a straight-forward approach would introduce spurious behaviors.

2 Components and Contracts

We give a formal definition to the discrete-time Interactive Markov Chains described in [8], used to model the behavior of components.

Definition 1 (Probability distribution). *A* probability distribution *over a set X is a function $f : X \to [0,1]$ such that $\sum_{x \in X} f(x) = 1$.*

Definition 2 (Interactive Markov Chain (IMC)). *An IMC is a tuple $(\mathcal{Q}, \mathcal{A}, \to, \pi, s_0)$ where:*

- *$\mathcal{Q}$ is a nonempty finite set of states, partitioned into $\mathcal{Q}^{\mathsf{p}}$, the set of probabilistic states, and $\mathcal{Q}^{\mathsf{a}}$, the set of action states;*
- *A is a finite alphabet of actions;*
- *$\to \subseteq \mathcal{Q}^{\mathsf{a}} \times \mathcal{A} \times \mathcal{Q}$ is an action transition relation;*
- *$\pi : \mathcal{Q}^{\mathsf{p}} \to (\mathcal{Q} \to [0,1])$ is a transition probability function such that, for each $s \in \mathcal{Q}^{\mathsf{p}}$, $\pi(s)$ is a probability distribution over $\mathcal{Q}$;*
- *s_0 is the initial state.*

IMC may interact with each other by synchronizing on action transitions (details in §4). Each action state in $\mathcal{Q}^{\mathsf{a}}$ has outgoing action transitions like those in an LTS. Each probabilistic state in $\mathcal{Q}^{\mathsf{p}}$ has outgoing probabilistic transitions like those in a Markov Chain. Probability distributions on states are memoryless, i.e., the future of an IMC depends only on the current state, but not on past choices. For example, in Figure 1(b), the probabilistic choice that the Link delivers the response to the Client (i.e., $\pi(l_4)(l_5) = 0.95$) is independent of the probabilistic choice of delivering a request to the Server (i.e., $\pi(l_1)(l_2) = 0.98$).

Notation: For convenience, we sometimes write the transition probability function π as a transition relation $\dashrightarrow \subseteq \mathcal{Q}^{\mathsf{p}} \times [0,1] \times \mathcal{Q}$ such that

$$\dashrightarrow = \{(s, p, s') \mid s \in \mathcal{Q}^{\mathsf{p}} \wedge s' \in \mathcal{Q} \wedge p = \pi(s)(s')\}$$

We introduce contracts as a finite specification for a possibly infinite number of IMCs. In contrast to IMCs, the probabilistic transitions of a contract are labeled with probability *intervals*, similar to [9,15]. Moreover, a distinct $\top$ state is used to distinguish assumptions on the use of the component from the guarantees it provides.

Definition 3 (Contract). *A contract is a tuple $(\mathcal{Q}, \mathcal{A}, \to, \sigma, t_0)$ where:*

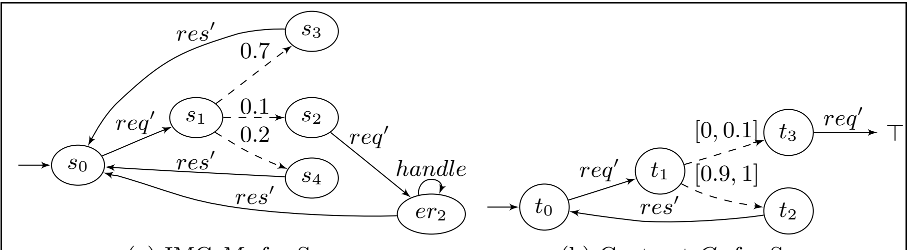
- *$\mathcal{Q}$ is a nonempty finite set of states, partitioned into $\mathcal{Q} = \mathcal{Q}^{\mathsf{p}} \cup \mathcal{Q}^{\mathsf{a}} \cup \{\top\}$, where $\mathcal{Q}^{\mathsf{p}}$ is the set of probabilistic states, $\mathcal{Q}^{\mathsf{a}}$ is the set of action states, and $\top$ is a distinct state without any outgoing transitions;*
- *A is a finite alphabet of actions;*
- *$\to \subseteq \mathcal{Q}^{\mathsf{a}} \times \mathcal{A} \times \mathcal{Q}$ is the action transition relation;*

- *$\sigma : \mathcal{Q}^{\mathsf{p}} \rightarrow (\mathcal{Q} \rightarrow 2^{[0,1]})$ is a transition probability predicate, associating with each pair of states $(s, s') \in \mathcal{Q}^{\mathsf{p}} \times \mathcal{Q}$ an interval of probabilities;*
- *t_0 is the initial state.*

Notations: We also write σ as a transition relation $\dashrightarrow \subseteq \mathcal{Q}^{\mathsf{p}} \times 2^{[0,1]} \times \mathcal{Q}$ such that $\dashrightarrow = \{(s, P, s') \mid s \in \mathcal{Q}^{\mathsf{p}} \wedge s' \in \mathcal{Q} \wedge P = \sigma(s)(s')\}$. We write $q \overset{>0}{\dashrightarrow} q'$ if $\exists p > 0 : p \in \sigma(q, q')$, and $\overset{>0}{\dashrightarrow}^{+}$ for the transitive closure of $\overset{>0}{\dashrightarrow}$. We extend arithmetic operations to intervals: $[\ell_1, u_1] + [\ell_2, u_2] = [\ell_1 + \ell_2, u_1 + u_2]$.

The meaning of a contract over a component C is the following:

- a transition $s \xrightarrow{a} \top$ specifies the *assumption* of the component C that an interaction involving action a does not occur in state s;
- in an action state s, an action a labeling a transition not leading to $\top$ specifies the *guarantee* of the component C that a is enabled in s; conversely, the absence of any outgoing transition labeled with a specifies the guarantee that an interaction involving a will not occur;
- the $\top$ state represents the fact that the assumption has been violated, and henceforth, the component C can show arbitrary, uncontrollable behavior;
- a transition $s \overset{[a,b]}{\dashrightarrow} t$ specifies an interval of allowed transition probabilities.



(a) IMC M_s for Server (b) Contract C_s for Server

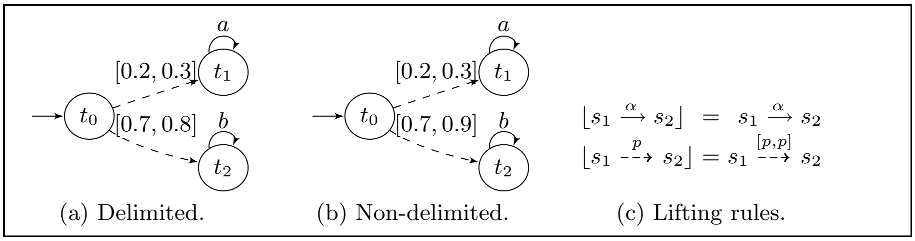
Fig. 2. Contract examples

Example 1. The contract C_s in Figure 2(b) specifies that, after the Server receives a request *req'*, the probability that it reaches state t_3 is within $[0, 0.1]$; in state t_3, it *assumes* that the environment does not give *req'* again; if this occurs, its implementation is not bound by C_s any more; the probability that it reaches t_2 from t_1 is within $[0.9, 1]$; in state t_2, it *guarantees* to send a response (*res'*). In §3, we show how to check that M_s (in Figure 2(a)) satisfies C_s.

From the definitions of IMC and contract, we can see that an IMC can be trivially converted to a contract. For this, we define a lifting operator $\lfloor . \rfloor$ (Figure 3 (c)). For the sake of simplicity, we use the same notation $\dashrightarrow$ to represent both kinds of probabilistic transitions (i.e., those in an IMC and in a contract).

Definition 4 (Delimited contract). *A contract* $C = (\mathcal{Q}, \mathcal{A}, \rightarrow, \sigma, t_0)$ *is delimited* [5] *iff* $\forall s \in \mathcal{Q}^{\mathsf{p}}\ \forall s' \in \mathcal{Q}\ \forall p \in \sigma(s)(s') : 1 - p \in \sum_{s'' \in \mathcal{Q} \setminus \{s'\}} \sigma(s)(s'')$.

Example 2. Figure 3 (a) shows a delimited contract: for all $p \in [0,2, 0.3]$, we can find $p' \in [0.7, 0.8]$ such that $p + p' = 1$ and vice versa. Figure 3 (b) shows a contract that is not delimited. However, we can *cut* [5] the redundant sub-interval [0.8,0.9] from the interval [0.7,0.9] to obtain a delimited contract.



a
$[0.2, 0.3]$
t_1
t_0
$[0.7, 0.8]$
b
t_2
(a) Delimited.
a
$[0.2, 0.3]$
t_1
t_0
$[0.7, 0.9]$
b
t_2
(b) Non-delimited.
$\lfloor s_1 \xrightarrow{\alpha} s_2 \rfloor = s_1 \xrightarrow{\alpha} s_2$
$\lfloor s_1 \overset{p}{\dashrightarrow} s_2 \rfloor = s_1 \overset{[p,p]}{\dashrightarrow} s_2$
(c) Lifting rules.

Fig. 3. Delimited contract and rules for lifting IMC to contract

3 Contract Refinement

System synthesis involves refining a contract several times until an implementation is obtained. We therefore define formally the notion of contract refinement.

3.1 Refinement and Satisfaction

We first define contract refinement, and give thereafter some explanations.

Definition 5 (Contract refinement). *Let* $C_1 = (\mathcal{Q}_1, \mathcal{A}, \rightarrow_1, \sigma_1, s_0)$ *and* $C_2 = (\mathcal{Q}_2, \mathcal{A}, \rightarrow_2, \sigma_2, t_0)$ *be two contracts.* C_1 *refines* C_2 *(written* $C_1 \leq C_2$*) iff* $s_0 \leq t_0$*, where* $\leq \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ *is the greatest relation s.t. for all* $s \leq t$ *we have:*

1. $s = \top \implies t = \top$;
2. *If* $(s, t) \in \mathcal{Q}_1^{\mathsf{a}} \times (\mathcal{Q}_2^{\mathsf{a}} \cup \{\top\})$ *then*
   (a) $\forall t' \neq \top \in \mathcal{Q}_2,\ (t \xrightarrow{\alpha}_2 t') \implies (\exists s' \in \mathcal{Q}_1,\ s \xrightarrow{\alpha}_1 s' \wedge s' \leq t')$;
   (b) $\forall s' \in \mathcal{Q}_1,\ (s \xrightarrow{\alpha}_1 s') \implies (t = \top\ \vee\ \exists t' \in \mathcal{Q}_2,\ t \xrightarrow{\alpha}_2 t' \wedge s' \leq t')$.
3. *If* $(s, t) \in \mathcal{Q}_1^{\mathsf{p}} \times \mathcal{Q}_2^{\mathsf{p}}$ *then there exists a function* $\delta : \mathcal{Q}_1 \times \mathcal{Q}_2 \rightarrow [0, 1]$*, which, for each* $s' \in \mathcal{Q}_1$*, gives a probability distribution* $\delta(s')$ *over* $\mathcal{Q}_2$*, such that for every probability distribution* f *over* $\mathcal{Q}_1$ *with* $f(s') \in \sigma_1(s)(s')$ *and* $\forall t' \in \mathcal{Q}_2$*,*
$$\sum_{s' \in \mathcal{Q}_1} f(s') * \delta(s')(t') \in \sigma_2(t)(t') \quad \textit{and} \quad \forall s' \in \mathcal{Q}_1 : \big(\delta(s')(t') > 0 \implies s' \leq t'\big)$$
4. *If* $(s, t) \in \mathcal{Q}_1^{\mathsf{a}} \times \mathcal{Q}_2^{\mathsf{p}}$ *then* $\exists t^{\mathsf{a}} \in \mathcal{Q}_2^{\mathsf{a}} : t \overset{>0}{\dashrightarrow}_2^{+} t^{\mathsf{a}} \wedge s \leq t^{\mathsf{a}}$ *and* $\forall t' \in \mathcal{Q}_2$*,* $\big(t \overset{>0}{\dashrightarrow}_2 t' \implies s \leq t'\big)$.

5. *If* $(s,t) \in \mathcal{Q}_1^{\mathsf{p}} \times \mathcal{Q}_2^{\mathsf{a}}$ *then* $\exists s^{\mathsf{a}} \in \mathcal{Q}_1^{\mathsf{a}} : s \overset{>0}{\dashrightarrow}{}_1^{+} s^{\mathsf{a}} \wedge s^{\mathsf{a}} \leq t$ *and* $\forall s' \in \mathcal{Q}_1$, $\big(s \overset{>0}{\dashrightarrow}_1 s' \implies s' \leq t\big)$.

In Definition 5, condition (1) ensures that C_1 makes no stronger assumptions on the context than C_2. Condition (2a) says that any transition accepted by C_2 must also be accepted by C_1. Similarly, condition (2b) says that each action transition of C_1 must also be enabled in C_2, unless C_2 is in the $\top$ state. Condition (3), adapted from [9], deals with refinement among probabilistic states. Intuitively, $s \leq t$ if there exists a function δ which distributes the probabilities of transitions from s to s' onto the transitions from t to t', such that the sum of the fractions is in the range $\sigma(t)(t')$, as illustrated in Example 3 below. Condition (4) says that an action state s refines a probabilistic state t if it refines all action states reachable with a path of positive probability from t. Finally, condition (5) is symmetrical to condition (4).

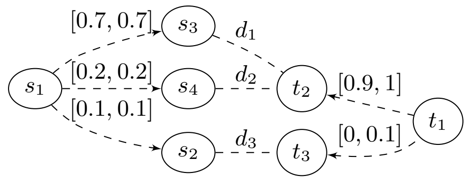
Before giving an example of refinement, we define the satisfaction of a contract by an implementation (an IMC) as the refinement of the contract by the lifted IMC (i.e., written in the form of a contract).

Definition 6 (Contract satisfaction). *An IMC M satisfies a contract C (written $M \models C$) iff $\lfloor M \rfloor \leq C$.*

Example 3. We illustrate how to check $\lfloor M_s \rfloor \leq C_s$, in particular, $s_1 \leq t_1$. It is easy to check $s_3 \leq t_2$, $s_4 \leq t_2$, and $s_2 \leq t_3$. Dashed lines stand for non-negative distributions δ. Condition (3) in Definition 5 states that $s_1 \leq t_1$ if for each successor state of s_1 there is a function δ (i.e., d_1, d_2, d_3) such that, for each tuple (p_2, p_3, p_4) satisfying constraints (1) to (4) in Figure 4, the constraints (5) and (6) are implied. Condition (3) can be checked efficiently by requiring the set inclusion to hold for the bounds of interval $\sigma(s)(s')$, using a linear programming solver. As $\delta(s')$ is a probability distribution, we obtain for our example $d_1 = d_2 = d_3 = 1$. (Note that if we had $s_2 \leq t_2$ as well, say, we had d_4 from s_2 to t_2, we would have another constraint $d_3 + d_4 = 1$.)

Definition 7 (Models of contracts). *The set of models of a contract C (written $\mathcal{M}(C)$) is the set of IMCs that satisfy C: $\mathcal{M}(C) = \{M \mid M \models C\}$.*

Definition 8 (Semantical equivalence). *Contracts C_1 and C_2 are semantically equivalent (written $C_1 \equiv C_2$) iff $\mathcal{M}(C_1) = \mathcal{M}(C_2)$.*



(1) $p_2 \in [0.1, 0.1]$
(2) $p_3 \in [0.7, 0.7]$
(3) $p_4 \in [0.2, 0.2]$
(4) $p_2 + p_3 + p_4 = 1$
(5) $p_3 * d_1 + p_4 * d_2 \in [0.9, 1]$
(6) $p_2 * d_3 \in [0, 0.1]$

Fig. 4. Left: Contract refinement $s_1 \leq t_1$. Right: Constraints to be checked.

Lemma 1 (Monotonicity of satisfaction). *For all IMC M and contracts C_1 and C_2, if $M \models C_1$ and $C_1 \leq C_2$, then $M \models C_2$.*

Lemma 2 (Refinement and model inclusion). *For all contracts C_1 and C_2, $C_1 \leq C_2 \implies \mathcal{M}(C_1) \subseteq \mathcal{M}(C_2)$.*

The proofs for all lemmas and theorems in this paper can be found in [14].

3.2 Bisimulation

We adapt the usual notion of bisimulation to contracts, and define reduction of a contract with respect to bisimulation.

Definition 9 (Bisimulation $\simeq$). *Given a contract $C = (\mathcal{Q}, \mathcal{A}, \rightarrow, \dashrightarrow, s_0)$, let $\simeq \subseteq \mathcal{Q} \times \mathcal{Q}$ be the greatest relation such that if $s \simeq t$ then:*

1. $s = \top \iff t = \top$;
2. *If* $(s,t) \in \mathcal{Q}^{\mathsf{a}} \times \mathcal{Q}^{\mathsf{a}}$ *then*
   (a) $\forall \alpha \in \mathcal{A} \quad \forall s' \in \mathcal{Q}, (s \xrightarrow{\alpha} s' \implies \exists t' \in \mathcal{Q}, (t \xrightarrow{\alpha} t' \wedge s' \simeq t'))$
   (b) $\forall \alpha \in \mathcal{A} \quad \forall t' \in \mathcal{Q}, (t \xrightarrow{\alpha} t' \implies \exists s' \in \mathcal{Q}, (s \xrightarrow{\alpha} s' \wedge s' \simeq t'))$
3. *If* $(s,t) \in \mathcal{Q}^{\mathsf{p}} \times \mathcal{Q}^{\mathsf{p}}$ *then*
   (a) *there is a function* $\delta : \mathcal{Q} \times \mathcal{Q} \to [0,1]$, *which for each* $s' \in \mathcal{Q}$ *gives a probability distribution* $\delta(s')$ *on* $\mathcal{Q}$, *s.t. for every probability distribution* f *over* $\mathcal{Q}$ *with* $f(s') \in \sigma(s)(s')$ *and* $\forall t' \in \mathcal{Q}$
   $$\sum_{s' \in \mathcal{Q}} f(s') * \delta(s',t') \in \sigma(t)(t') \quad \textit{and} \quad \forall s' \in \mathcal{Q} : \big(\delta(s',t') > 0 \implies s' \simeq t'\big)$$
   (b) *symmetric to (3a);*
4. *If* $(s,t) \in \mathcal{Q}^{\mathsf{a}} \times \mathcal{Q}^{\mathsf{p}}$ *then* $\exists t^{\mathsf{a}} \in \mathcal{Q}^{\mathsf{a}} : t \overset{>0}{\dashrightarrow}{}^{+} t^{\mathsf{a}} \wedge s \simeq t^{\mathsf{a}}$ *and* $\forall t' \in \mathcal{Q}$, $t \overset{>0}{\dashrightarrow} t' \implies s \simeq t'$;
5. *If* $(s,t) \in \mathcal{Q}^{\mathsf{p}} \times \mathcal{Q}^{\mathsf{a}}$ *then* $\exists s^{\mathsf{a}} \in \mathcal{Q}^{\mathsf{a}} : s \overset{>0}{\dashrightarrow}{}^{+} s^{\mathsf{a}} \wedge s^{\mathsf{a}} \simeq t$ *and* $\forall s' \in \mathcal{Q}$, $s \overset{>0}{\dashrightarrow} s' \implies s' \simeq t$.

In Definition 9, condition (2) is the standard definition for bisimulation. Conditions (3a) and (3b) deal with the probabilistic transitions. Finally, conditions (4) and (5) say that an action state is bisimilar with a probabilistic state if it is bisimilar with all its successors with non-zero probability, and some action state is reachable from this probabilistic state.

Definition 10 (Reduction modulo $\simeq$). *Let $C = (\mathcal{Q}, \mathcal{A}, \rightarrow, \sigma, s_0)$ be a contract. For all $s \in \mathcal{Q}$, let $\mathcal{C}_s = \{q \in \mathcal{Q} \mid s \simeq q\}$ be the equivalence class of s. Let $\mathcal{C} = \{\mathcal{C}_s \mid s \in \mathcal{Q}\}$. The reduced contract, written $\overline{C}$, is $(\mathcal{C}, \mathcal{A}, \rightarrow_{\simeq}, \sigma_{\simeq}, \mathcal{C}_{s_0})$ such that, $\forall s = \{s_1, \ldots, s_m\}, t = \{t_1, \ldots, t_n\} \in \mathcal{C}$, we have: (1) $s \xrightarrow{a}_{\simeq} t$ iff $\exists i,j : s_i \xrightarrow{a} t_j$, and (2) $\sigma_{\simeq}(s,t) = \sum_{1 \leq j \leq n} \sigma(s_1, t_j)$).*

Notice that an equivalence class may contain both action and probabilistic states. By Definition 9, except for probabilistic transitions with probability interval $[0, 0]$, either all transitions leaving an equivalence class are action transition and Definition 9 (2) applies, or they are all probabilistic transitions and Definition 9 (3) applies as follows. For each probabilistic state $s_i \in s$, the probabilities of transitions to states $t_j \in t$ are summed up (it does not matter which of the transitions is taken since all successors t_j are equivalent). This sum is the transition probability from s_i to some state in t. By definition of $\simeq$, the sum is the same for all $s_i \in s$, thus we pick $\sigma(s_1, t_j)$. For example, we can reduce the contract C_2 of Figure 7 (right) by combining the bisimilar states t_2 and t_3 into one: $t_1 \overset{[0.2,0.6]}{\dashrightarrow} \{t_2, t_3\}$.

Lemma 3 (Model equivalence). *For all delimited contracts C, $\overline{C} \equiv C$.*

3.3 Contract Abstraction

The need of abstraction arises naturally in contract frameworks. We abstract actions in $\mathcal{A} \setminus \mathcal{B}$ that we do not care about by renaming them into internal τ actions. The contract over the alphabet $\mathcal{B} \cup \{\tau\}$ is then projected on the sub-alphabet $\mathcal{B}$ by using the standard determinization algorithm (see e.g. [1]).

Definition 11 (Projection). *Let $C = (\mathcal{Q}, \mathcal{A}, \rightarrow_1, \sigma, s_0)$ be a contract and $\mathcal{B} \subseteq \mathcal{A}$. Let $C' = (\mathcal{Q}, \mathcal{B} \cup \{\tau\}, \rightarrow_1, \sigma, s_0)$ be the contract where all transition labels in $\mathcal{A} \setminus \mathcal{B}$ are replaced with τ. The projection of C on $\mathcal{B}$ (written $\pi_{\mathcal{B}}(C)$) is obtained by τ-elimination (determinization) of C'.*

Example 4. In Figure 2, if we do not care how the implementation handles failure cases, we can check that $\pi_{\mathcal{A}_s \setminus \{handle\}}(M_s) \models C_s$.

4 Contract Composition

We introduce two composition operations for contracts: parallel composition $||$, parametrized with an interaction set $\mathcal{I}$, and conjunction $\wedge$ (shared refinement).

4.1 Parallel Composition of Contracts

Parallel composition allows us to build complex models from simpler components in a stepwise and hierarchical manner. In order to reason about the composition of components at the contract level, we introduce parallel composition of contracts. As in the BIP component framework [7], parallel composition is parametrized with a set of interactions, where each interaction is a set of component actions occurring simultaneously. For instance, an interaction set $\{a, a|b, c\}$ says that action a can interleave or synchronize with b; action b must synchronize with a; action c is a singleton interaction that always interleaves. The symbol "$|$" is commutative, which means that $a|b$ is identical to $b|a$. In Figure 5, the interactions α and β are of the form c, $a|b$, or $a|b|d$, and so on.

Definition 12 (Parallel composition of contracts). *Let $C_1 = (\mathcal{Q}_1, \mathcal{A}_1, \rightarrow_1, \dashrightarrow_1, s_0)$ and $C_2 = (\mathcal{Q}_2, \mathcal{A}_2, \rightarrow_2, \dashrightarrow_2, t_0)$ be two contracts. The parallel composition of C_1 and C_2 on interaction set $\mathcal{I}$ (written $C_1||_{\mathcal{I}}C_2$) is the contract $\big(\mathcal{Q}, \mathcal{I}, \rightarrow, \dashrightarrow, (s_0, t_0)\big)$ where:*

1. *$\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ with $\top = (\mathcal{Q}_1 \times \{\top_2\}) \cup (\{\top_1\} \times \mathcal{Q}_2)$ — that is, $\top$ of $C_1||_{\mathcal{I}}C_2$ is an aggregate state reached as soon as C_1 or C_2 reaches its $\top_i$ state —, $\mathcal{Q}^{\mathsf{a}} = \mathcal{Q}_1^{\mathsf{a}} \times \mathcal{Q}_2^{\mathsf{a}}$, and $\mathcal{Q}^{\mathsf{p}} = \mathcal{Q} \setminus (\mathcal{Q}^{\mathsf{a}} \cup \top)$;*
2. *$\rightarrow$ is the least relation satisfying the rules* [R1]–[R3] *in Figure 5; and*
3. *$\dashrightarrow$ is the least relation satisfying the rules* [R4]–[R6] *in Figure 5.*

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \quad \alpha \in \mathcal{I} \qquad q_2 \in \mathcal{Q}_2^{\mathsf{a}}}{(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2)}\ [\text{R1}] \qquad \frac{q_2 \xrightarrow{\alpha}_2 q_2' \quad \alpha \in \mathcal{I} \qquad q_1 \in \mathcal{Q}_1^{\mathsf{a}}}{(q_1, q_2) \xrightarrow{\alpha} (q_1, q_2')}\ [\text{R2}]$$

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \quad q_2 \xrightarrow{\beta}_2 q_2' \quad \alpha|\beta \in \mathcal{I}}{(q_1, q_2) \xrightarrow{\alpha|\beta} (q_1', q_2')}\ [\text{R3}] \qquad \frac{q_1 \overset{[p1,p2]}{\dashrightarrow}_1 q_1' \qquad q_2 \overset{[p3,p4]}{\dashrightarrow}_2 q_2'}{(q_1, q_2) \overset{[p_1 * p_3, p_2 * p_4]}{\dashrightarrow} (q_1', q_2')}\ [\text{R4}]$$

$$\frac{q_1 \overset{P}{\dashrightarrow}_1 q_1' \qquad q_2 \in \mathcal{Q}_2^{\mathsf{a}}}{(q_1, q_2) \overset{P}{\dashrightarrow} (q_1', q_2)}\ [\text{R5}] \qquad \frac{q_2 \overset{P}{\dashrightarrow}_2 q_2' \qquad q_1 \in \mathcal{Q}_1^{\mathsf{a}}}{(q_1, q_2) \overset{P}{\dashrightarrow} (q_1, q_2')}\ [\text{R6}]$$

Fig. 5. Rules for the parallel composition of contracts

Rules [R1] to [R3] are the usual parallel composition rules for interactive processes, while the rule [R4] is similar to the typical parallel composition for Markov chains but on probability intervals. Finally, rules [R5] and [R6] state that probabilistic transition, usually modeling hidden internal behavior, have priority over action transitions.

Example 5. Figure 6 illustrates the parallel composition of contracts C_s (from Figure 2(b)) and $C_\ell = \lfloor M_\ell \rfloor$ (where M_ℓ is given in Figure 1(b)), with $\mathcal{I} = \{rec, del, req'|del', res'|rec', fail_1, fail_2\}$. The composed contract $C_s \;||_{\mathcal{I}}\; C_\ell$ states that a failure in the Link component does not prevent it from continuing to deliver the request req' to the Server and receiving response from the Server, but the failure prevents it from delivering the response res' back to the Client.

We end the section on parallel composition with two essential properties.

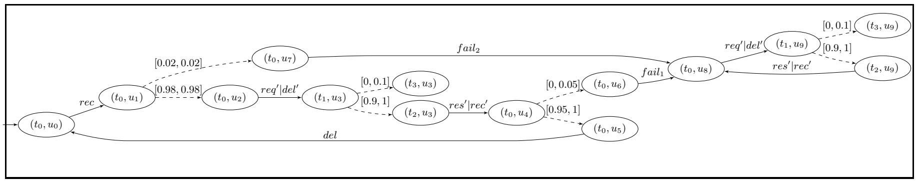


Fig. 6. Parallel composition of C_s and C_ℓ

Theorem 1 (Independent implementability). *For all IMCs M, N, contracts C_1, C_2, and interaction set $\mathcal{I}$, if $M \models C_1$ and $N \models C_2$, then $M||_{\mathcal{I}}N \models C_1 ||_{\mathcal{I}}C_2$.*

Theorem 2 (Congruence of refinement). *For all contracts C_1, C_2, C_3, and interaction set $\mathcal{I}$, if $C_1 \leq C_2$, then $C_1||_{\mathcal{I}}C_3 \leq C_2||_{\mathcal{I}}C_3$.*

4.2 Conjunction of Contracts

A single component may have to satisfy several contracts that are specified independently, each of them specifying different requirements on the component, such as safety, reliability, and quality of service aspects. Therefore, the contracts may use different, possibly overlapping, sub-alphabets of the component. The *conjunction* of contracts computes a common refinement of all contracts. Prior to conjunction, we define *similarity* of contracts as a test whether a common refinement exists.

Definition 13 (Similarity ($\sim$)). *Let $C_1 = (\mathcal{Q}_1, \mathcal{A}_1, \rightarrow_1, \dashrightarrow_1, s_0)$ and $C_2 = (\mathcal{Q}_2, \mathcal{A}_2, \rightarrow_2, \dashrightarrow_2, t_0)$ be two contracts. $\sim \subseteq Q_1 \times Q_2$ is the largest relation such that $\forall (s,t) \in \mathcal{Q}_1 \times \mathcal{Q}_2$, $s \sim t$ iff $(s = \top \vee t = \top)$ or conditions (1) to (4) below hold:*

1. *If $(s,t) \in \mathcal{Q}_1^{\mathsf{a}} \times \mathcal{Q}_2^{\mathsf{a}}$ then*
   - *(a) for all $s' \in \mathcal{Q}_1$, if $s \xrightarrow{a} s'$, then either $t \xrightarrow{a} t'$ for some $t' \in \mathcal{Q}_2$ and $s' \sim t'$, or $a \notin \mathcal{A}_2$ and $s' \sim t$; and*
   - *(b) for all $t' \in \mathcal{Q}_2$, if $t \xrightarrow{a} t'$, then either $s \xrightarrow{a} s'$ for some $s' \in \mathcal{Q}_1$ and $s' \sim t'$, or $a \notin \mathcal{A}_1$ and $s \sim t'$;*
2. *If $(s,t) \in \mathcal{Q}_1^{\mathsf{p}} \times \mathcal{Q}_2^{\mathsf{p}}$ then*
   - *(a) for all $s' \in \mathcal{Q}_1$, if $s \overset{P_1}{\dashrightarrow} s'$, then $t \overset{P_2}{\dashrightarrow} t'$ for some $t' \in \mathcal{Q}_2$ with $P_1 \cap P_2 \neq \emptyset$ and $(s' \sim t' \ \vee \ 0 \in P_1 \cap P_2)$;*
   - *(b) for all $t' \in \mathcal{Q}_2$, if $t \overset{P_2}{\dashrightarrow} t'$, then $s \xrightarrow{P_1} s'$ for some $s' \in \mathcal{Q}_1$ with $P_1 \cap P_2 \neq \emptyset$ and $(s' \sim t' \ \vee \ 0 \in P_1 \cap P_2)$;*
3. *If $(s,t) \in \mathcal{Q}_1^{\mathsf{p}} \times \mathcal{Q}_2^{\mathsf{a}}$ then for all $s' \in \mathcal{Q}_1$ with $s \overset{P}{\dashrightarrow}_1 s'$, $(s' \sim t \quad \vee \quad 0 \in P)$;*
4. *If $(s,t) \in \mathcal{Q}_1^{\mathsf{a}} \times \mathcal{Q}_2^{\mathsf{p}}$ then for all $t' \in \mathcal{Q}_2$ with $t \overset{P}{\dashrightarrow}_2 t'$, $(s \sim t' \quad \vee \quad 0 \in P)$.*

Finally, C_1 and C_2 are similar, written $C_1 \sim C_2$, iff $s_0 \sim t_0$.

The P_i in Definition 13 refers to a probabilistic interval in the form of $[\ell_i, u_i]$. Any state is similar with the top state (where the contract does not constrain the implementation in any way). Two action states are similar if they agree on the enabled actions in the common alphabet, and the successor states are similar again. Two probabilistic states are similar if the probabilistic transitions can be matched such that the intervals overlap, and the successor states are either similar, or can be made unreachable by refining the probability interval to $[0, 0]$.

Definition 14 (Unambiguous contract). *A contract $C = (\mathcal{Q}, \mathcal{A}, \rightarrow, \dashrightarrow, s_0)$ is unambiguous iff for all $r, s, t \in \mathcal{Q}$, if $r \overset{>0}{\dashrightarrow} s \wedge r \overset{>0}{\dashrightarrow} t \wedge s \sim t$, then $s = t$.*

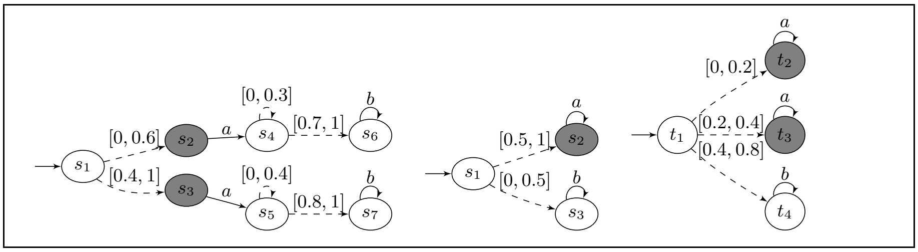


Fig. 7. Left: ambiguous contract C_a. Middle: C_1. Right: C_2 where $C_1 \not\sim C_2$.

A contract is *unambiguous* if the reachable successor states of any probabilistic state are pairwise non-similar. In Figure 7 (left), the contract C_a is not unambiguous: $s_2 \sim s_3$ (highlighted in gray) but $s_2 \neq s_3$.

We are now ready to define the conjunction of contracts.

Definition 15 (Conjunction of contracts ($\wedge$)). *For unambiguous contracts $C_1 = (\mathcal{Q}_1, \mathcal{A}_1, \rightarrow_1, \dashrightarrow_1, s_0)$ and $C_2 = (\mathcal{Q}_2, \mathcal{A}_2, \rightarrow_2, \dashrightarrow_2, t_0)$ such that C_1 and C_2 are similar, let $C_1 \wedge C_2$ be the contract $\big(\mathcal{Q}_1 \times \mathcal{Q}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \rightarrow, \dashrightarrow, (s_0, t_0)\big)$ where $\top = (\top_1, \top_2)$ and*

1. *$\rightarrow$ is the least relation satisfying the rules* [C1] – [LiftR] *in Figure 8, and*
2. *$\dashrightarrow$ is the least relation satisfying the rules* [C3] – [C4R] *in Figure 8 (where for all other probabilistic transitions $(q_1, q_2) \overset{P}{\dashrightarrow} (q_1', q_2')$, $P = [0, 0]$).*

Rule [C1] requires the contracts to agree on action transitions over the common alphabet. According to rule [C2L] (resp. [C2R]), the conjunction behaves like the first (resp. second) contract as soon as the other contract is in $\top$. Rules [LiftL] and [LiftR] allow the interleaving of action transitions that are not

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \quad q_2 \xrightarrow{\alpha}_2 q_2'}{(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2')}\ [\text{C1}] \qquad \frac{q_1 \xrightarrow{\alpha}_1 q_1'}{(q_1, \top) \xrightarrow{\alpha} (q_1', \top)}\ [\text{C2L}] \qquad \frac{q_2 \xrightarrow{\alpha}_1 q_2'}{(\top, q_2) \xrightarrow{\alpha} (\top, q_2')}\ [\text{C2R}]$$

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \quad \alpha \notin \mathcal{A}_2 \quad q_2 \in \mathcal{Q}_2^{\mathsf{a}}}{(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2)}\ [\text{LiftL}] \qquad \frac{q_2 \xrightarrow{\alpha}_2 q_2' \quad \alpha \notin \mathcal{A}_1 \quad q_1 \in \mathcal{Q}_1^{\mathsf{a}}}{(q_1, q_2) \xrightarrow{\alpha} (q_1, q_2')}\ [\text{LiftR}]$$

$$\frac{q_1 \overset{P_1}{\dashrightarrow}_1 q_1' \quad q_2 \overset{P_2}{\dashrightarrow}_2 q_2' \quad q_1' \sim q_2'}{(q_1, q_2) \overset{P_1 \cap P_2}{\dashrightarrow} (q_1', q_2')}\ [\text{C3}]$$

$$\frac{\begin{array}{c} q_1 \overset{P}{\dashrightarrow}_1 q_1' \quad q_2 \in \mathcal{Q}_2^{\mathsf{a}} \cup \{\top\} \\ q_1' \sim q_2 \end{array}}{(q_1, q_2) \overset{P}{\dashrightarrow} (q_1', q_2)}\ [\text{C4L}] \qquad \frac{\begin{array}{c} q_2 \overset{P}{\dashrightarrow}_2 q_2' \quad q_1 \in \mathcal{Q}_1^{\mathsf{a}} \cup \{\top\} \\ q_1 \sim q_2' \end{array}}{(q_1, q_2) \overset{P}{\dashrightarrow} (q_1, q_2')}\ [\text{C4R}]$$

Fig. 8. Rules for conjunction of contracts

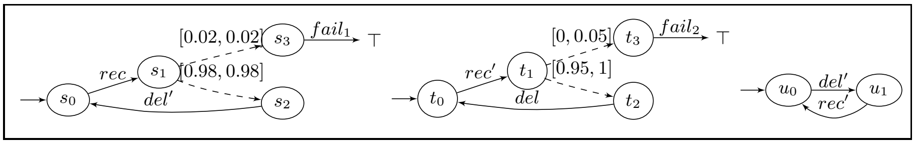


Fig. 9. Left: $C_{\ell 1}$. Middle: $C_{\ell 2}$. Right: $C_{\ell 3}$

in the common alphabet. Rules [C3] – [C4R] define the probabilistic transitions whose successor states are similar. For non-similar successor states, the probability interval is refined to $[0, 0]$, according to Definition 15.

Example 6. Figure 9 shows three contracts for the Link component: $C_{\ell 1}$ specifies that the implementation should receive a request from the Client and deliver it to the Server; $C_{\ell 2}$ specifies that the implementation should receive a response from the Server and deliver it to the Client; $C_{\ell 3}$ requires the response (rec') received from the Server to occur after the request (del') delivered to the Server. We can verify that $M_\ell \models (C_{\ell 1} \wedge C_{\ell 3}) \wedge (C_{\ell 2} \wedge C_{\ell 3})$ (where M_ℓ is in Figure 1(b)).

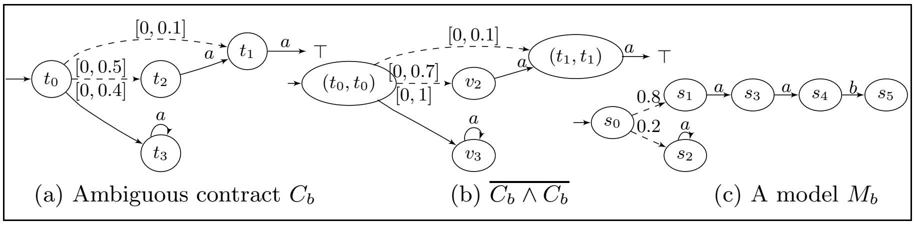


Fig. 10. Example where $M_b \models C_b \wedge C_b$ but $M_b \not\models C_b$

Example 7. As a contract that is not in reduced form is not unambiguous, contracts should be reduced before performing conjunction. In Figure 7 (left), contract C_2 is non unambiguous, but $t_2 \simeq t_3$. If we reduce C_2 by applying Definition 10, we get $t_1 \overset{[0.2,0.6]}{\dashrightarrow} \{t_2, t_3\} \xrightarrow{a} \{t_2, t_3\}$. The reduced contract is unambiguous and $s_1 \sim t_1$, such that conjunction yields a common refinement of C_1 and C_2.

Theorem 3 (Soundness of conjunction). *For any IMC M and unambiguous contracts C_i with alphabet $\mathcal{A}_i$, $i = 1, 2$ such that C_1 and C_2 are similar, if $M \models C_1 \wedge C_2$ then $\pi_{\mathcal{A}_i}(M) \models C_i$, $i = 1, 2$.*

Example 8. Figure 10 motivates the requirement of conjunction (Definition 15) for unambiguous contracts. The resulting contract $C_b \wedge C_b$ is reduced such that the model relation can be seen easily. The node v_2 denotes the equivalent class $\{(s_1, s_2), (s_2, s_1), (s_2, s_2)\}$; the node v_3 denotes the equivalent class $\{(s_1, s_3), (s_2, s_3), (s_3, s_1), (s_3, s_2), (s_3, s_3)\}$. As $t_1 \sim t_2 \sim t_3$, duplicated intervals lead to an unsound result.

It is interesting to note the similarity of conjunction with discrete controller synthesis [13], in the sense that conjunction is a refinement of both contracts making bad states (i.e., pairs of states where both contracts are contradictory) unreachable. In this analogy, both action transitions and probabilistic transitions with strictly positive intervals amount to uncontrollable transitions, whereas transitions whose probability interval contains 0 amount to controllable transitions that can be refined to $[0, 0]$ so as to make bad states unreachable.

5 Case Study

We study a dependable computing system with time redundancy. The system specification is expressed by the contract C_S of Figure 11 (top left), which specifies that the computation *comp* should have a success probability of at least 0.999. If the computation fails, then nothing is specified (state $\top$).

The processor is specified by the contract C_P of Figure 11 (top right). Following an execution request *exe*, either the processor succeeds and replies with *ok* (with a probability at least p), or fails and replies with *nok* (with a probability at most $1 - p$). The failure rates for successive executions are independent. The probability p is a parameter of the contract.

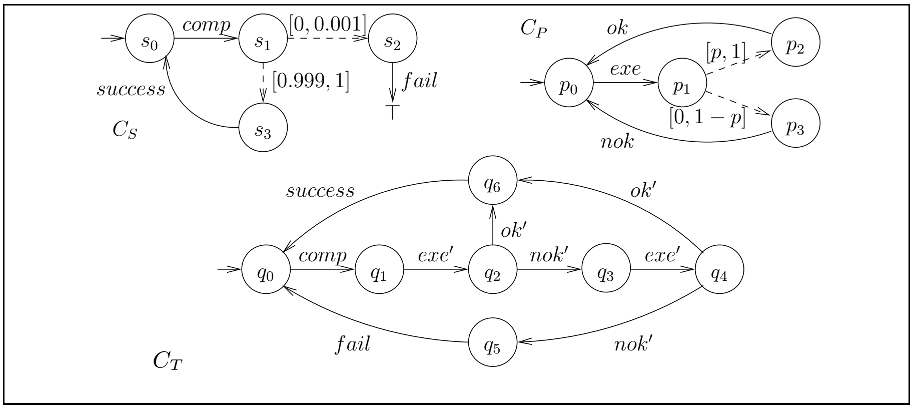


Fig. 11. Specification C_S; processor contract C_P; time redundancy contract C_T

We place ourselves in a setting where the reliability level guaranteed by C_P alone (as expressed by p) cannot fulfill the requirement of C_S (that is, 0.999), and hence some form of redundancy must be used. We propose to use time redundancy, as expressed by the contract C_T of Figure 11 (bottom). Each computation *comp* is first launched on the processor (*exe*), either followed by a positive (*ok*) or negative (*nok*) answer from the processor. In the latter case, the execution is launched a second time, therefore implementing time redundancy. The contract C_T finally answers with *success* if *either* execution is followed by *ok*, or with *fail* is *both* executions are followed by *nok*.

In terms of component-based design for reliability, we wonder what is the minimum value of p to guarantee the reliability level of C_S. To compute this minimum value, we first compute the parallel composition $C_T||_{\mathcal{I}}C_P$, with the interaction set $\mathcal{I} = \{comp, exe|exe', ok|ok', nok|nok', success, fail\}$. The reduction modulo bisimulation of this parallel composition is shown in Figure 12 (top), where the interactions $exe|exe'$, $ok|ok'$, and $nok|nok'$ have been replaced for conciseness by **exe**, **ok**, and **nok**, respectively. We call this new contract $C_{T||P}$. We then compute the projection of $C_{T||P}$ onto the set $\mathcal{B} = \{comp, success, fail\}$. The result $C_\pi = \pi_{\mathcal{B}}(C_{T||P})$ is shown in Figure 12 (bottom left).

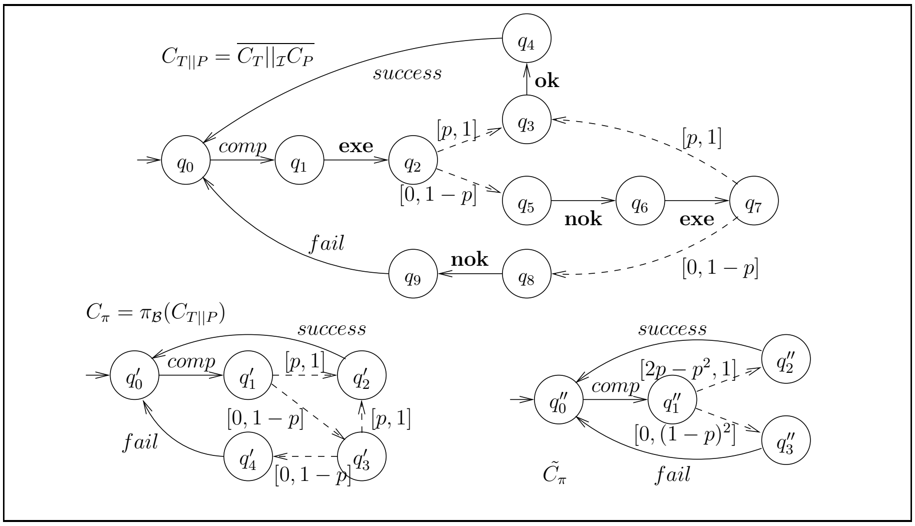


Fig. 12. Parallel composition $C_{T||P}$; projection C_π; transitive closure $\tilde{C_\pi}$

We are thus faced with a contract C_π having *sequences* of probabilistic transitions; more precisely, since some probabilistic states have several outgoing transitions, we have DAGs of probabilistic transitions. We therefore compute the transitive closure for each such DAG, that is, the equivalent probabilistic transitions from the initial state of the DAG (e.g., q'_1 in C_π) to its final states (e.g., q'_2 and q'_4 in C_π). Without entering into the details of this computation, we show the resulting contract $\tilde{C_\pi}$ in Figure 12 (bottom right).

The last step involves checking under which condition on p the contract $\tilde{C_\pi}$ refines the specification C_S. We have $\tilde{C_\pi} \leq C_S \Leftrightarrow (1-p)^2 \leq 0.001$. This means that, with time redundancy and a processor with a reliability level of at least 0.969, we are able to ensure an overall reliability level of 0.999.

6 Discussion

We have introduced a design framework based on probabilistic contracts, and proved essential properties for the use in component-based design. Our definition of contracts is based on the ideas from [9,15,5], although the frameworks in [9,5] do not support interactions.

Shared refinement of interfaces, and conjunction of modal specifications over possibly different alphabets have been defined in [4] and [12]. A framework over modal assume/guarantee-contracts is introduced in [6], for which both parallel composition and conjunction are defined. Probabilistic assume/guarantee-contracts have been introduced in [3] in terms of traces. [10] introduces a compositional framework based on continuous time IMCs, adopting a similar interaction model as done in this paper. This framework supports abstraction, parallel and symmetric composition, but not conjunction. The recently introduced Constraint Markov Chains (CMC) [2] generalize Markov chains by introducing constraints on state valuations and transition probability distributions, aiming at a similar goal of providing a probabilistic component-based design framework. Whereas CMCs do not support explicit interactions among components, they allow one to expressively specify constraints on probability distributions. Conjunction is shown to be sound and complete in this framework.

Future work will encompass implementing the framework and carrying out case studies. A particularly interesting application would be the design of adaptive systems where the probabilistic behavior of components may change over time, while the overall system must at any time satisfy a set of safety, reliability, and quality of service contracts.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Caillaud, B., Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M., Wasowski, A.: Compositional design methodology with constraint markov chains. Research Report 6993, INRIA (2009)
3. Delahaye, B., Caillaud, B.: A model for probabilistic reasoning on assume/guarantee contracts. Research Report 6719, INRIA (2008)
4. Doyen, L., Petrov, T., Henzinger, T.A., Jobstmann, B.: Interface theories with component reuse. In: Proc. EMSOFT 2008, pp. 79–88. ACM, New York (2008)
5. Fecher, H., Leucker, M., Wolf, V.: Don't know in probabilistic systems. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 71–88. Springer, Heidelberg (2006)
6. Gössler, G., Raclet, J.-B.: Modal contracts for component-based design. In: Proc. SEFM 2009, pp. 295–303. IEEE, Los Alamitos (2009)
7. Gössler, G., Sifakis, J.: Composition for component-based modeling. Science of Computer Programming 55(1-3), 161–183 (2005)
8. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality. LNCS, vol. 2428, p. 57. Springer, Heidelberg (2002)
9. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277. IEEE Computer Society, Los Alamitos (1991)
10. Katoen, J.-P., Klink, D., Neuhäußer, M.R.: Compositional abstraction for stochastic systems. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 195–211. Springer, Heidelberg (2009)

11. Meyer, B.: Design by Contract. In: Advances in Object-Oriented Software Engineering, pp. 1–50. Prentice Hall, Englewood Cliffs (1991)
12. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Passerone, R.: Why modalities are good for interface theories? In: Proc. ACSD 2009. IEEE, Los Alamitos (2009)
13. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control and Optimization 25(1) (1987)
14. Xu, D.N., Gössler, G., Girault, A.: Probabilistic contracts for component-based design. Research Report 7328, INRIA (2010)
15. Yi, W.: Algebraic reasoning for real-time probabilistic processes with uncertain information. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 680–693. Springer, Heidelberg (1994)

Model-Checking Web Applications with Web-TLR*

María Alpuente[1], Demis Ballis[2], Javier Espert[1], and Daniel Romero[1]

[1] DSIC-ELP, Universidad Politécnica de Valencia
{alpuente,jespert,dromero}@dsic.upv.es
[2] DIMI, University of Udine
demis@dimi.uniud.it

Abstract. WEB-TLR is a software tool designed for model-checking Web applications which is based on rewriting logic. Web applications are expressed as rewrite theories which can be formally verified by using the Maude built-in LTLR model-checker. WEB-TLR is equipped with a user-friendly, graphical Web interface that shields the user from unnecessary information. Whenever a property is refuted, an interactive slideshow is generated that allows the user to visually reproduce, step by step, the erroneous navigation trace that underlies the failing model checking computation. This provides deep insight into the system behavior, which helps to debug Web applications.

1 Introduction

In recent years, the automated verification of Web applications has become a major field of research. Nowadays, a number of corporations (including book retailers, auction sites, travel reservation services, *etc.*) interact primarily through the Web by means of Web applications that combine static content with dynamic data produced "on-the-fly" by the execution of Web scripts (e.g. Java servlets, Microsoft ASP.NET and PHP code). The inherent complexity of such highly concurrent systems has turned their verification into a challenge [1,6,9].

In [2], we formulated a rich and accurate navigation model that formalizes the behavior of Web applications in rewriting logic. Our formulation allows us to specify critical aspects of Web applications such as concurrent Web interactions, browser navigation features (i.e., forward/backward navigation, page refreshing, and window/tab openings), and Web script evaluations by means of a concise, high-level rewrite theory. Our formalization is particularly suitable for verification purposes since it allows in-depth analyses of several subtle aspects of Web interactions to be carried out. We have shown how real-size, dynamic

* This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2007-68093-C02-02 project, by Generalitat Valenciana, under grant Emergentes GV/2009/024, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004. Daniel Romero is also supported by FPI–MEC grant BES–2008–004860.

Web applications can be efficiently model-checked using the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic specifically designed to model-check rewrite theories that combines all the advantages of the state-based and event-based logics, while avoiding their respective disadvantages [8].

This paper describes WEB-TLR, which is a model-checking tool that implements the theoretical framework of [2]. WEB-TLR is written in Maude and is equipped with a freely accessible graphical Web interface (GWI) written in Java, which allows users to introduce and check their own specification of a Web application, together with the properties to be verified. In the case when the property is proven to be false (refuted), an online facility can be invoked that dynamically generates a counter-example (expressed as a navigation trace), which is ultimately responsible for the erroneous Web application behavior. In order to improve the understandability and usability of the system and since the textual information associated to counter-examples is usually rather large and poorly readable, the checker has been endowed with the capability to generate and display on-the-fly slideshows that allow the erroneous navigation trace to be visually reproduced step by step. This graphical facility, provides deep insight into Web application behavior and is extremely effective for debugging purposes.

WEB-TLR focuses on the Web application tier (business logic, and thus handles server-side scripts; no support is given for GUI verification with Flash technology or other kinds of client-side computations.

2 An Overview of the Web Verification Framework

In this section, we briefly recall the main concepts of the Web verification framework proposed in [2], which are essential for understanding this tool description.

A Web application is thought of as a collection of related Web pages that are hosted by a Web server and contain a mixture of (X)HTML code, executable code (Web scripts), and links to other Web pages. A Web application is accessed over a network such as the Internet by using a Web browser which allows Web pages to be navigated by clicking and following links. Interactions between Web browsers and the Web server are driven by the HTTP protocol.

A Web application is specified in our setting by means of a rewrite theory, which accurately formalizes the entities in play (e.g., Web server, Web browsers, Web scripts, Web pages, messages) as well as the dynamics of the system (that is, how the computation evolves through HTTP interactions). More specifically, the Web application behavior is formalized by using labeled rewrite rules of the form $\mathsf{label}\colon \mathsf{WebState} \Rightarrow \mathsf{WebState}$, where $\mathsf{WebState}$ is a triple[1] $\_\|\_\|\_ :(\mathsf{Browsers} \times \mathsf{Message} \times \mathsf{Server}) \rightarrow \mathsf{WebState}$ that can be interpreted as a snapshot of the system that captures the current configurations of the active browsers (i.e., the browsers currently using the Web application), together with the server and the channel through which the browsers and the server communicate via message-passing. Given an initial Web state st_0, a computation is

[1] A detailed specification of Browsers, Message, and Server can be found in [2].

a rewrite sequence starting from st_0 that is obtained by non-deterministically applying (labeled) rewrite rules to Web states.

Also, formal properties of the Web application can be specified by means of the *Linear Temporal Logic of Rewriting* (LTLR), which is a temporal logic that extends the traditional Linear Temporal Logic (LTL) with *state predicates*[8], i.e, atomic predicates that are locally evaluated on the states of the system. Let us see some examples. Assume that forbid is a session variable that is used to establish whether a login event is possible at a given configuration. In LTLR, we can define the state predicate userForbidden(bid), which holds in a Web state when a browser bid[2] is prevented from logging on to the system, by simply inspecting the value of the variable forbid appearing in the server session that is recorded in the considered state. More formally,

$$\mathsf{browsers}\|\mathsf{channel}\|\mathsf{server}(\mathsf{session}(\underline{(\mathsf{bid}, \{\mathsf{forbid} = \mathsf{true}\})})) \;|\!= \mathsf{userForbidden}(\mathsf{bid}) = \mathsf{true}$$

In LTLR, we can also define the following state predicates as boolean functions: failedAttempt(bid,n), which holds when browser bid has performed n failed login attempts (this is achieved by recording in the state a counter n with the number of failed attempts); curPage(bid,p), which holds when browser bid is currently displaying the Web page p; and inconsistentState, which holds when two browser windows or tabs of the same browser refer to distinct user sessions. These elementary state predicates are used below to build more complex LTLR formulas expressing mixed properties that include dependencies among states, actions, and time. These properties intrinsically involve both action-based and state-based aspects which are either not expressible or are difficult to express in other temporal logic frameworks.

3 The Web-TLR System

Our verification methodology has been implemented in the Web-TLR system using the high-performance, rewriting logic language Maude [4] (around 750 lines of code not including third-party components). Web-TLR is available online via its friendly Web interface at `http://www.dsic.upv.es/~dromero/web-tlr.html`. The Web interface frees users from having to install applications on their local computer and hides a lot of technical details of the tool operation. After introducing the (or customizing a default) Maude specification of a Web application, together with an initial Web state st_0 and the LTLR formula φ to be verified, φ can be automatically checked at st_0. Once all inputs have been entered in the system, we can automatically check the property by just clicking the button Check, which invokes the Maude built-in operator tlr check[3] that supports model checking of rewrite theories w.r.t. LTLR formulas.

In the case when φ is refuted by the model-checker, a counter-example is provided that is expressed as a model-checking computation trace starting from st_0. The counter-example is graphically displayed by means of an interactive

[2] We assume that the browser identifier uniquely identifies the user.

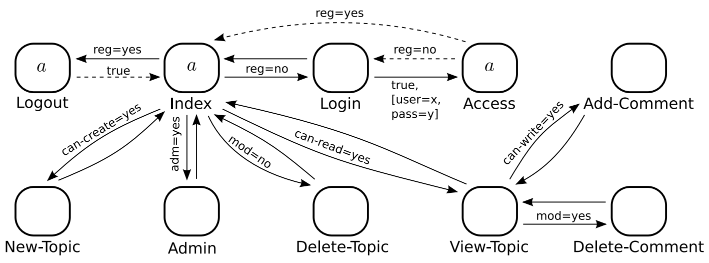


Fig. 1. The navigation model of an Electronic Forum

slideshow that allows forward and backward navigation through the computation's Web states. Each slide contains a graph that models the structure of (a part of) the Web application. The nodes of the graph represent the Web pages, and the edges that connect the Web pages specify Web links or Web script continuations[3]. The graph also shows the current Web page of each active Web browser. The graphical representation is combined with a detailed textual description of the current configurations of the Web server and the active Web browsers.

A Case Study of Web Verification. We tested our tool on several complex case studies that are available at the WEB-TLR web page and distribution package. In order to illustrate the capabilities of the tool, in the following we discuss the verification of an electronic forum equipped with a number of common features, such as user registration, role-based access control including moderator and administrator roles, and topic and comment management.

The navigation model of such an application is formalized by means of the graph-like structure given in Figure 1. Web pages are modeled as graph nodes. Each navigation link l is specified by a solid arrow that is labeled by a condition c and a query string q. l is enabled whenever c evaluates to $true$, while q represents the input parameters that are sent to the Web server once the link is clicked. For example, the navigation link connecting the Login and Access Web pages is always enabled and requires two input parameters (user and pass). The dashed arrows model Web application continuations, that is, arrows pointing to Web pages that are automatically computed by Web script executions. Each dashed arrow is labeled by a condition, which is used to select the continuation at runtime. For example, the Access Web page has got two possible continuations (dashed arrows) whose labels are reg=yes and reg=no, respectively. The former continuation specifies that the login succeeds, and thus the Index Web page is delivered to the browser; in the latter case, the login fails and the Login page is sent back to the browser.

[3] To obey the stateless nature of the Web, the structure of Web applications has traditionally been "inverted", resembling programs written in a continuation–passing style [6].

Using the state predicates given in Section 2, we are able to define and check sophisticated LTLR properties w.r.t. the considered Web application model. In the following, we discuss a selection of the properties that we considered.

Concise and Parametric Properties. We can define and verify the login property *"Incorrect login attempts are allowed only k times; then login is denied"*, which is defined parametrically w.r.t. the number of login attempts:

$$\Diamond(\mathsf{curPage}(\mathsf{A},\mathsf{Login}) \wedge \bigcirc(\Diamond\mathsf{failedAttempt}(\mathsf{A},\mathsf{k}))) \rightarrow \Box\mathsf{userForbidden}(\mathsf{A})$$

Note the sugared syntax (which is allowed in LTLR) when using relational notation for the state predicates which were defined as boolean functions above.

Unreachability Properties. Unreachability properties can be specified as LTLR formulas of the form $\Box\neg\langle\mathsf{State}\rangle$, where State is an undesired configuration that the system should not reach. This unreachability pattern allows us to specify and verify a wide range of interesting properties such as the absence of conflict due to multiple windows, mutual exclusion, link accessibility, *etc.*

- Mutual exclusion: *"No two administrators can access the administration page simultaneously"*.
  $\Box\neg(\mathsf{curPage}(\mathsf{A},\mathsf{Admin}) \wedge \mathsf{curPage}(\mathsf{B},\mathsf{Admin}))$.
- Link accessibility: *"All links refer to existing Web pages"* (absence of broken links).
  $\Box\neg\,\mathsf{curPage}(\mathsf{A},\mathsf{PageNotFound})$.
- No multiple windows problem: *"We do not want to reach a Web application state in which two browser windows refer to distinct user sessions"*.
  $\Box\neg\,\mathsf{inconsistentState}$.

The detailed specification of the electronic forum, together with some example properties are available at `http://www.dsic.upv.es/~dromero/web-tlr.html`

4 Conclusion

Web-TLR is the first verification engine based on the versatile and well-established Rewriting Logic/LTLR tandem for specifying Web systems and properties. Web-TLR distinguishes itself from related tools in a number of salient aspects: (*i*) The rich Web application core model which considers the communication protocol underlying Web interactions as well as common browser navigation features; (*ii*) Efficient and accurate model–checking of dynamic properties– e.g., reachability of Web pages generated by means of Web script executions– at low cost. Verification includes both analysis (checking whether properties are satisfed) and diagnostic traces demonstrating why a property holds or does not hold; (*iii*) Visualization of counter-examples via an interactive slideshow, which allows the user to explore the model performing forward and backward transitions. At each slide, the interface shows the values of relevant variables of the Web state. This on–the–fly exploration does not require installation of the checker itself and is provided entirely by the GWI.

In recent years, the modeling and verification of Web applications have received increasing attention (for a thorough review, please refer to [2]). On the one hand, a number of model checkers and temporal logics have been proposed to formally check properties of Web systems [5,7,10]. These approaches are generally equipped with a coarse, static state structure, whereas states in Web-TLR are generated on-the-fly by evaluating Web scripts, which makes the Web-TLR's Web application model more precise and suitable for the verification of real, dynamic Web systems. On the other hand, a number of new Web languages have been proposed that allow safe Web applications to be built [6,11]. Unfortunately, such languages are often based on nonstandard communication infrastructures and —albeit rather powerful— are hence of limited use.

As future work, we plan to extend Web-TLR by considering the problem of synthesizing correct-by-construction Web applications. We also plan to deal with client-side scripts defined for example by JavaScript-like languages.

References

1. Alalfi, M.H., Cordy, J.R., Dean, T.R.: Modelling Methods for Web Application Verification and Testing: State of the Art. Software Testing, Verification and Reliability 19, 265–296 (2009)
2. Alpuente, M., Ballis, D., Romero, D.: Specification and Verification of Web Applications in Rewriting Logic. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 790–805. Springer, Heidelberg (2009)
3. Bae, K., Meseguer, J.: A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In: Proc. of the 9th International Workshop on Rule-Based Programming (RULE 2008). ENTCS. Elsevier, Amsterdam (2008)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Flores, S., Lucas, S., Villanueva, A.: Formal verification of websites. In: Proc. 4th Int'l Workshop on Automated Specification and Verification of Web Sites (WWV 2008). ENTCS, vol. 200(3), pp. 103–118 (2008)
6. Graunke, P., Findler, R., Krishnamurthi, S., Felleisen, M.: Modeling web interactions. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 238–252. Springer, Heidelberg (2003)
7. Haydar, M., Sahraoui, H., Petrenko, A.: Specification patterns for formal web verification. In: ICWE 2008: Proc. of the 2008 Eighth International Conference on Web Engineering, pp. 240–246. IEEE CS, Los Alamitos (2008)
8. Meseguer, J.: The Temporal Logic of Rewriting: A Gentle Introduction. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer, Heidelberg (2008)
9. Message, R., Mycroft, A.: Controlling control flow in web applications. In: Proc. 4th Int'l Workshop on Automated Specification and Verification of Web Sites (WWV 2008). ENTCS, vol. 200(3), pp. 119–131 (2008)
10. Miao, H., Zeng, H.: Model checking-based verification of web application. In: ICECCS 2007: Proc. of the 12th IEEE Int'l Conf. on Engineering Complex Computer Systems (ICECCS 2007), Washington, DC, USA, pp. 47–55. IEEE CS, Los Alamitos (2007)
11. Queinnec, C.: Continuations and web servers. Higher-Order and Symbolic Computation 17(4), 277–295 (2004)

GAVS: Game Arena Visualization and Synthesis

Chih-Hong Cheng[1], Christian Buckl[2],
Michael Luttenberger[1], and Alois Knoll[1]

[1] Department of Informatics, Technischen Universität München
Boltzmann Str. 3, Garching D-85748, Germany
[2] fortiss GmbH, Guerickestr.25, D-80805 München, Germany
{chengch,knoll}@in.tum.de, buckl@fortiss.org, luttenbe@model.in.tum.de

Abstract. Reasoning on the properties of computer systems can often be reduced to deciding the winner of a game played on a finite graph. In this paper, we introduce GAVS, an open-source tool for the visualization of some of the most fundamental games on finite graphs used in theoretical computer science, including, e.g., reachability games and parity games. The main purpose of GAVS is educational, a fact which is emphasized by the graphical editor for both defining game graphs and also visualizing the computation of the winning sets. Nevertheless, the underlying solvers are implemented with scalability in mind using symbolic techniques where applicable.

1 Introduction

We present GAVS[1], a tool which allows to visualize and solve some of the most common two-player games encountered in theoretical computer science, amongst others reachability, Büchi and parity games.

The importance of these games results from the reduction of different questions regarding the analysis of computer systems to two-player games played on finite graphs. For example, liveness (safety) properties can easily be paraphrased as a game play on the control-flow graph of a finite program: will the system always (never) visit a given state no matter how the user might interact with the system? The resulting games are usually called (co-)reachability if only finite program runs are considered. Similarly, one obtains (co-)Büchi games when considering infinite runs. Another well-known example is the class of parity games which correspond to the model-checking problem of μ-calculus. Advantages of the game theoretic reformulation of these analysis problems are the easier accessibility and the broadened audience.

The main goal of GAVS is to further enhance these advantages by providing educational institutions with a graphical tool for both constructing game graphs and also visualizing standard algorithms for solving them step-by-step. Still, symbolic methods, where applicable, have been used in the implementation in order to ensure scalability.

[1] Short for "Game Arena Visualization and Synthesis".

GAVS is released under the GNU General Public License (v3) and allows for an easy extension to novel algorithms. The software package is available at `http://www6.in.tum.de/~chengch/gavs`

2 Preliminaries and Supported Games

We briefly recapitulate the most important definitions regarding two-player games on finite graphs before explicitly enumerating the games supported by GAVS.

A *game graph* or *arena* is a directed graph $G = (V_0 \uplus V_1, E)$ whose nodes are partitioned into two classes V_0 and V_1. We only consider the case of two players in the following and call them player 0 and player 1 for simplicity. A *play* starting from node v_0 is simply a maximal path $\pi = v_0 v_1 \ldots$ in G where we assume that player i determines the *move* $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$ $(i \in \{0,1\})$. With $\mathrm{Occ}(\pi)$ / $\mathrm{Inf}(\pi)$ we denote the set of nodes visited / visited infinitely often by a play π. A *winning condition* defines when a given play π is *won* by player 0; if π is not won by player 0, it is won by player 1. A node v is won by player i if player i can always choose his moves in such a way that he wins any resulting play starting from v; the sets of nodes won by player i are denoted by W_i $(i \in \{0,1\})$.

GAVS supports the computation of W_0, W_1 for the following games:

1. Games defined w.r.t. a set of target states F:
   - *Reachability game*: player 0 wins a play π if $\mathrm{Occ}(\pi) \cap F \neq \emptyset$.
   - *Co-reachability (safety) game*: player 0 wins a play π if $\mathrm{Occ}(\pi) \cap F = \emptyset$.
   - *Büchi game*: player 0 wins a play π on G if $\mathrm{Inf}(\pi) \cap F \neq \emptyset$.
2. Games defined w.r.t. a coloring $c : V \to \mathbf{N}$ of G:
   - *Weak-parity game*: player 0 wins a play π if $\max(c(\mathrm{Occ}(\pi)))$ is even.
   - *Parity game*: player 0 wins a play π if $\max(c(\mathrm{Inf}(\pi)))$ is even.
3. Games defined w.r.t. finite families $E = \{E_1, \ldots, E_k\}$ and $F = \{F_1, \ldots, F_k\}$ of subsets of V:
   - *Staiger-Wagner game*: player 0 wins a play π if $\mathrm{Occ}(\pi) \in F$.
   - *Muller game*: player 0 wins a play π if $\mathrm{Inf}(\pi) \in F$.
   - *Streett game*: player 0 wins a play π if $\bigwedge_{j=1}^{k} \mathrm{Inf}(\pi) \cap F_i \neq \emptyset \Rightarrow \mathrm{Inf}(\pi) \cap E_i \neq \emptyset$.

It is well-known that all these games are determined, i.e., $W_0 \cup W_1 = V$. We refer the reader to [4] for a thorough treatment of these games and their relationship to each other.

We close this section by recalling the definition of attractor which is at the heart of many algorithms for solving the games mentioned above: for $i \in \{0,1\}$ and $X \subseteq V$, the map $\mathrm{attr}_i(X)$ is defined by

$$\mathrm{attr}_i(X) := X \cup \{v \in V_i \mid vE \cap X \neq \emptyset\} \cup \{v \in V_{1-i} \mid \emptyset \neq vE \subseteq X\},$$

i.e., $\mathrm{attr}_i(X)$ extends X by all those nodes from which either player i can move to X within one step or player $1-i$ cannot prevent to move within the next step. (vE denotes the set of successors of v.) Then $\mathrm{Attr}_i(X) := \bigcup_{k \in \mathbf{N}} \mathrm{attr}_i^k(X)$ contains all nodes from which player i can force any play to visit the set X.

3 Software Architecture

GAVS consists of three major parts: (a) a graphical user interface (GUI), (b) solvers for different winning conditions, and (c) a two-way translation function between graphical representations and internal formats acceptable by the engine. The GUI is implemented using the JGraphX library [3] and acts as a front-end for the different game solvers. Every game solver is implemented as a separate back-end. We give a brief description of their implementation where we group back-ends sharing similar implementation approaches:

- Symbolic techniques: Algorithms of this type are implemented using JDD [2], a Java-based package for binary decision diagrams (BDDs). The supported winning conditions include reachability, safety, Büchi, weak-parity, and Staiger-Wagner.
- Explicit state operating techniques: Algorithms of this type are implemented based on direct operations over the graph structure.
  - **Parity game.** In addition to an inefficient version which enumerates all possibilities using BDDs, we have implemented the discrete strategy improvement algorithm adapted from [5]; the algorithm allows the number of nodes/vertices to exceed the number of colors in a game.
- Reduction techniques: Algorithms of these games are graph transformations to other games with different types of winning conditions.
  - **Muller game.** The algorithm performs reductions to parity games using the latest appearance record (LAR) [4].
  - **Streett game.** The algorithm performs reductions to parity games using the index appearance record (IAR) [4].

GAVS can easily be extended by additional game solvers. We briefly sketch how this can be done.

Every game engine interacts with the GUI via the method `EditorAction.java`. When invoked, an intermediate object of the data type `mxGraph` (the predefined data structure for a graph in JGraphX) is retrieved. We then translate the graph from `mxGraph` to a simpler structure (defined in `BuechiAutomaton.java`) which offers a simple entry for users to extend GAVS with new algorithms.

For symbolic algorithms, we offer methods for translating the graph to BDDs. After the algorithm is executed, GAVS contains mechanisms to annotate the original game graph via the winning region encoded by a BDD, and visualize the result. To redirect the result of synthesis back to the GUI, a map structure with the type `HashMap<String, HashSet<String>>` is required, where the key is the source vertex and the value set contains destination vertices, describing the edges that should be labeled by GAVS. For explicit state operating algorithms, mechanisms follow analogously.

4 Example: Working with GAVS

Due to page limits, we give two small yet representative examples on using GAVS. We refer the reader to the GAVS homepage [1] for a full-blown tutorial, and serveral examples.

4.1 Example: Safety Games

We give a brief description of how to use GAVS for constructing a safety game and solving it step-by-step with the assist of Figure 1.

In the first step, the user constructs the game graph by simply drawing it using the graphical interface, similar to Figure 1-a: the states V_1 are called plant states and are of rectangular shape, while the states V_0 are called control states and are of circular shape.

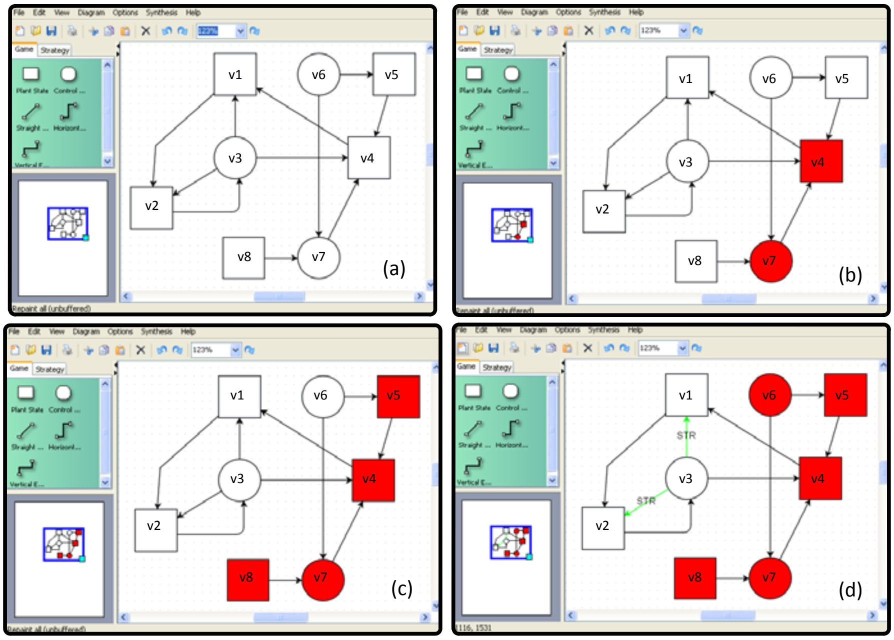


Fig. 1. An example for constructing and executing a safety game

Next, the user specifies the target nodes F, i.e., the nodes which player 0 tries to avoid. GAVS supports both graphical and textual methods[2]. In Figure 1-b, states v_4 and v_7 are painted by the user with red color, offering an graphical description of risk states.

[2] Graphical specification is only available with reachability, safety, and Büchi winning conditions; for weak-parity and parity games, colors of vertices can also be labeled directly on the game graph.

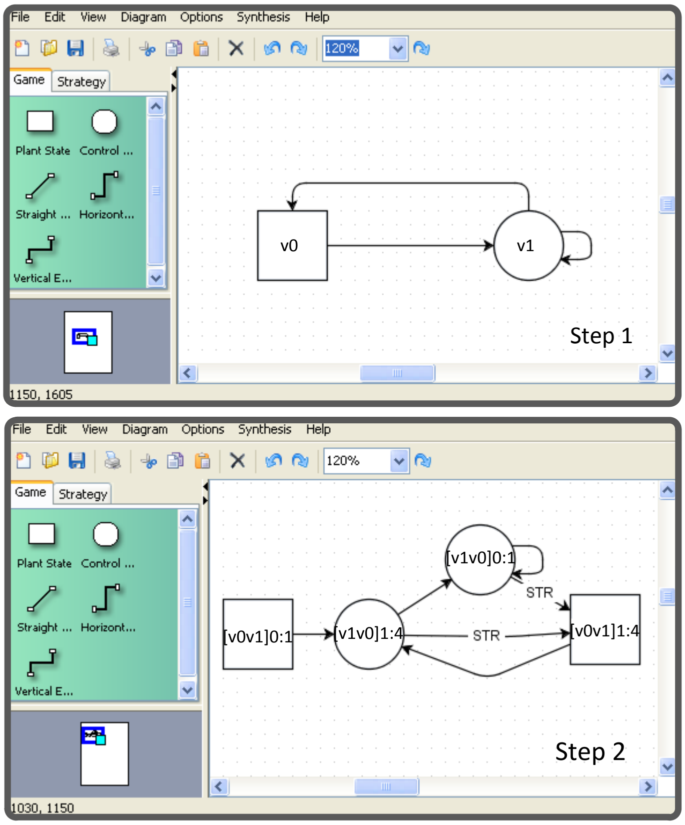


Fig. 2. A Muller game and its synthesized result in the form of the parity game

Finally, GAVS can be used to either compute the winning set W_1 immediately or to guide the user through the computation of W_1 step-by-step: the computation of $\text{Attr}_1(\{v_4, v_7\}) = \text{attr}_1^2(\{v_4, v_7\})$ is shown in Figures 1-b to 1-d with the corresponding nodes highlighted in red. For games with positional strategies, a winning strategy is shown automatically on the graph (edges labeled with "STR"), for instance, Figure 1-d shows the winning strategy for the safety game: edges (v_3, v_2) and (v_3, v_1) are highlighted as safe transitions.

4.2 Example: Muller Games

For Muller and Streett games, instead of generating strategies directly, game reductions are implemented for clearer understanding regarding the meaning of strategies. This is due to the fact that the generated FSM strategies for Muller and Streett games require memory which is factorial to the number of states, making them difficult for users to comprehend.

We indicate how Muller game reduction is applied in GAVS. Consider Figure [2], where a Muller game is shown in step 1 with the winning condition $\{\{v_0, v_1\}\}$. The reduced parity game generated by GAVS is shown in step 2, where each vertex is of the format `"[Vertex Permutation] LAR index : Color"`. By interpreting the strategy using LAR, it is clear that for player 0, the generated strategy is a positional move (v_1, v_0) on vertex v_1.

5 Concluding Remarks

As the name of the tool suggests, Game Arena Visualization and Synthesis (GAVS), which is served for both research and educational purposes, is designed to provide a unified platform to connect visualization and synthesis for games. We are also interested in the visualization and the synthesis of pushdown games, which will be our next step.

Acknowledgments. We thank Javier Esparza, Michael Geisinger, Jia Huang, and Shun-Ching Yang for their preliminary evaluations on GAVS. The first author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

References

1. GAVS: Game Arena Visualization and Synthesis, `http://www6.in.tum.de/~chengch/gavs/`
2. JDD: Java BDD and Z-BDD library, `http://javaddlib.sourceforge.net/jdd/`
3. JGraphX: Java Graph Drawing Component, `http://www.jgraph.com/jgraph.html`
4. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002)
5. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)

CRI: Symbolic Debugger for MCAPI Applications*

Mohamed Elwakil[1], Zijiang Yang[1], and Liqiang Wang[2]

[1] Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008
[2] Department of Computer Science, University of Wyoming, Laramie, WY 82071
{mohamed.elwakil,zijiang.yang}@wmich.edu, wang@cs.uwyo.edu

Abstract. We present a trace-driven SMT-based symbolic debugging tool for MCAPI (Multicore Association Communication API) applications. MCAPI is a newly proposed standard that provides an API for connectionless and connection-oriented communication in multicore applications. Our tool obtains a trace by executing an instrumented MCAPI. The collected trace is then encoded into an SMT formula such that its satisfiability indicates the existence of a reachable error state such as an assertion failure.

Keywords: MCAPI, Message Race, Symbolic Analysis, Satisfiability Modulo Theories.

1 Introduction

As multicore-enabled devices are becoming ubiquitous, development of multicore applications is inevitable, and debugging tools that target multi-core applications will be in demand. Inter-core communication, in which data is passed between cores via messages, is an essential part of multicore applications. The Multicore Association has developed the MCAPI standard [1] and a reference runtime implementation for it to address inter-core communication needs. In an MCAPI application, a core is referred to as a *node*. Communication between nodes occurs through endpoints. A node may have one or more endpoints. An endpoint is uniquely defined by a node identifier, and a port number. The MCAPI specification supplies APIs for initializing nodes, creating endpoints, obtaining addresses of remote endpoints, and sending and receiving messages. Fig. 1 shows a snippet from an MCAPI application in which four cores communicate via messages. For brevity, the variables declarations and initialization calls are omitted. In this application, each node has one endpoint; hence there is no need for different port numbers per node. Endpoints are created by issuing the *create_ep* calls (e.g. lines 5 and 24). To obtain the address of a remote endpoint, the *get_ep* calls are used (e.g. lines 6 and 25). Messages are sent using the *msg_send* call (e.g. lines 8 and 27). Messages are received using the *msg_recv* call (e.g. lines 16 and 34).

* The work was funded in part by NSF grant CCF-0811287 and ONR Grant N000140910740.

```
#define PORT 1
void* C1_routine (void *t)
{
  int Msg=1;
  My_ep=create_ep(PORT);
  C2_ep = get_ep(2,PORT);
  C4_ep = get_ep(4,PORT);
  msg_send(My_ep,C2_ep,Msg); //M0
  msg_send(My_ep,C4_ep,Msg); //M1
}
void* C2_routine (void *t)
{
  int X,Y,Z;
  My_ep = create_ep(PORT);
  C4_ep = get_ep(4,PORT);
  msg_recv(My_ep,X);
  msg_recv(My_ep,Y);
  Z=X-Y;
  msg_send(My_ep,C4_ep,Z); //M2
}
void* C3_routine (void *t)
{
  int Msg=10;
  My_ep = create_ep(PORT);
  C2_ep = get_ep(2,PORT);
  C4_ep = get_ep(4,PORT);
  msg_send(My_ep,C2_ep,Msg); //M3
  msg_send(My_ep,C4_ep,Msg);//M4
}
void* C4_routine (void *t)
{
  int U;
  My_ep = create_ep(PORT);
  msg_recv(My_ep,U);
  assert(U>0);
}
```

Fig. 1. A snippet of an MCAPI application

The MCAPI runtime provides each endpoint with FIFO buffers for incoming and outgoing messages. Messages sent from an endpoint S to another endpoint D, are delivered to D according to their order of transmission from S. However, the order at which a destination endpoint D, receives messages originating from endpoints $S1$, and $S2$, is *non-deterministic*, even if endpoints $S1$ and $S2$ belong to the same node. Two or more messages are said to be *racing* if their order of arrival at a destination (i.e. a core) is non-deterministic [2]. The *msg_recv* calls specify only the receiving endpoint, which is the reason for the possibility of message races. In Fig. 1, messages *M0* and *M3* are racing towards core *C2* and messages *M1, M2*, and *M4* are racing towards core *C4*. There are twelve possible scenarios for the order of arrival of messages at *C2* and *C4*. In only two scenarios (when *M0* beats *M3,* and *M2* beats *M1* and *M4*), there will be an assertion failure at *C4*.

Testing the application in Fig. 1 by multiple executions does not necessarily expose the single scenario that leads to an assertion failure. Even if an assertion failure takes place during testing, it is very difficult to find out the specific order of messages arrival that caused the assertion failure.

In this paper we present our tool (CRI) that *symbolically* explores all possible orders of messages arrival in an MCAPI application execution. Our approach is based on encoding the trace of an execution as an SMT formula in quantifier-free first order logic. This formula can be decided efficiently using any of the available SMT solvers such as Yices [3].

Our main contributions are 1) the modeling of MCAPI constructs as SMT constraints, and 2) developing a tool that generates these constraints from an MCAPI application execution trace. Our tool predicts errors that may not be discovered by testing and guides an execution to reproduce the same error when an error is detected. The rest of this paper is organized as follows: Section 2 describes our tool in detail. Section 3 reviews related work. We conclude in Section 4.

2 CRI

Fig. 2 shows the workflow of our tool. An application source code is instrumented so that an execution will produce a trace that contains all the statements that has been executed such as MCAPI functions calls and C statements.

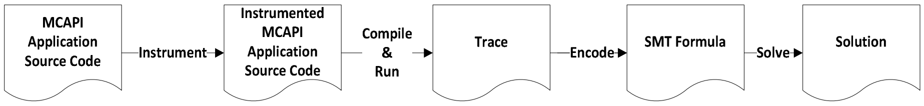

MCAPI Application Source Code
Instrument
Instrumented MCAPI Application Source Code
Compile & Run
Trace
Encode
SMT Formula
Solve
Solution

Fig. 2. CRI Workflow

The trace is then encoded as an SMT formula. If the formula is satisfiable, then there is a reachable error state (e.g. an assertion failure) and the SMT solver solution will provide enough information to guide a controlled execution to reach this error state. Otherwise, we can conclude that for all possible execution scenarios involving the same statements in the input trace, no error state is reachable. In the following we describe the structure of the trace, the variables used in the encoding, and present the encoding of some statements.

A trace consists of a list of nodes: $\mathbb{N} = \{N_1, \dots, N_{|\mathbb{N}|}\}$, and for every node N_{nid}, a set of local variables: $\mathbb{L}_{nid} = \{L_{nid,1}, \dots, L_{nid,|\mathbb{L}_{nid}|}\}$, a set of endpoints used in this node: $\mathbb{EP}_{nid} = \{EP_{nid,1}, \dots, EP_{nid,|\mathbb{EP}_{nid}|}\}$, and an ordered list of statements: $\mathbb{S}_{nid} = \{S_{nid,1}, \dots, S_{nid,|\mathbb{S}_{nid}|}\}$.

For a trace with B statements, there will be $B+1$ symbolic states ($\mathcal{s}^0$, $\mathcal{s}^1$, …, $\mathcal{s}^i, \dots \mathcal{s}^B$), such that $\mathcal{s}^0$ is the state before carrying out any statement, and $\mathcal{s}^i$ is the state at the ith time instant, after carrying out the ith statement. A state $\mathcal{s}^i$ is a valuation of all symbolic variables at time instant i. To capture the $B+1$ states, we create $B+1$ copies for the variables in the trace. For example, $L^i_{nid,x}$ denotes the copy of variable $L_{nid,x}$ at the ith time instant.

At any instant of time, one statement, called the *pending statement*, at one node, called the *active node*, will be *symbolically* carried out. The node selector variable NS^i identifies the node that will be active at time instant i. At any time instant i, the value of NS^i is selected by the SMT solver. The selection of NS^i value is not totally random, but is governed by scheduling constraints.

The pending statement in a node N_{nid} is identified using the node counter variable NC_{nid}. The domain of a NC_{nid} is $\{1 \dots |\mathbb{S}_{nid}|, \perp\}$. $NC_{nid}=x$ indicates that the pending statement in the node N_{nid} is $S_{nid,x}$. $NC_{nid} = \perp$ means that all statements in node N_{nid}, has been symbolically executed.

The MCAPI runtime buffers associated with endpoints are modeled as queues. For a receiving endpoint $EP_{n,x}$ that receives a message or more, there will be a corresponding queue $Q_{n,x}$. $\mathbb{Q}_n$ is the set of all queues needed for the receiving endpoints at node N_{nid}. A queue $Q_{n,x}$ is encoded as an array with two variables $head_{n,x}$ and $tail_{n,x}$ that indicate the head and tail positions in the array.

The MCAPI standard provides non-blocking send and receive calls: *msg_send_i*, and *msg_recv_i*, respectively. MCAPI runtime uses request objects to track the status of a non-blocking call. A non-blocking call initiates an operation (i.e. a send or a receive operation), sets a request object to pending, and returns immediately. The completion of a non-blocking call could be checked by issuing the blocking call *wait*, and passing to it the request object associated with the non-blocking call. The *wait* call will return when the non-blocking call has completed. A non-blocking send is completed when the message has been delivered to the MCAPI runtime. A non-blocking receive is completed when a message has been retrieved from the MCAPI runtime buffers. A request object will be encoded as a symbolic variable with three possible values: NLL, PND, and CMP.

Four constraints formulas make up the SMT formula: the initial constraint ($\mathbb{I}$), the statements constraint ($\mathbb{S}$), the scheduling constraint ($\mathbb{C}$), and the property constraint ($\mathbb{P}$). The initial constraint ($\mathbb{I}$) assigns the values of the symbolic variables at time instant *0*. All node counters are initialized to 1. $\mathbb{I}$ is expressed as $\bigwedge_{n=1}^{|\mathbb{N}|}\left((NC_n^0 = 1) \wedge \left(\bigwedge_{v=1}^{|\mathbb{L}_n|} L_{n,v}^0 = iv_{n,v}\right) \wedge \left(\bigwedge_{q=1}^{|\mathbb{Q}_n|} head_{n,q}^0 = tail_{n,q}^0 = 0\right)\right)$, where $iv_{n,v}$ is the initial value for the variable $L_{n,v}$. Note that the request variables used in a node N_{nid}, are among the node local variables ($\mathbb{L}_n$), and that they are initialized to NLL in the initial constraint.

The statements constraint ($\mathbb{S}$) mimics the effect of carrying out a pending statement. It is a conjunction of B constraints ($\mathbb{S} = \bigwedge_{i=1}^{B} \mathbb{s}^i$), such that $\mathbb{s}^i$ corresponds to the statement chosen to be carried out at time instant i. The $\mathbb{s}^i$ constraint is dependent on the statement type. Our tool handles eight types of statements: assignment, conditional, assert, blocking send, blocking receive, non-blocking send, non-blocking receive, and wait statements. In this paper we present the encoding of five types, and omit the rest.

1) For an assignment statement in the format of S=('assign', nid, sn, sn', v, $Expr$), where nid is the identifier of the node to which S belongs, sn is the node counter of S, sn' is the node counter of the statement to be executed next in this node ($\perp$ if S is the last statement in $\mathbb{S}_{nid}$), and $Expr$ is an expression whose valuation is assigned to variable v, the corresponding constraint formula is $\left(NS^i = nid \wedge NC_{nid}^i = sn\right) \rightarrow (NC_{nid}^{i+1} = sn' \wedge v^{i+1} = Expr^i \wedge \delta(\{v^i\}))$. This formula states that, at time instant i , if node N_{nid} is the active node ($NS^i = nid$) and node N_{nid}'s node counter is equal to sn ($NC_{nid}^i = sn$), then the node counter in the time instant $i+1$ is set to sn' ($NC_{nid}^{i+1} = sn'$), the value of variable v in the time instant $i+1$ is set to the valuation of the expression $Expr$ at time instant i ($v^{i+1} = Expr^i$), and that all local variables but v and all queues heads and tails should have in time instant $i+1$, the same values they had in time instant i ($\delta(\{v^i\})$).

2) For a conditional statement in the format of S=('condition', nid, sn, sn', $Expr$), the corresponding constraint formula is $\left(NS^i = nid \wedge NC_{nid}^i = sn\right) \rightarrow (Expr^i \wedge NC_{nid}^{i+1} = sn' \wedge \delta(\phi))$. Note that we enforce the condition $Expr$ to be true so only the executions with the same control flow in this node are considered. $\delta(\phi)$ states that all variables retain their values from time instant i to $i+1$.

3) For a blocking send statement in the format of S=('send_b', nid, sn, sn', $SrcEP_{nid,x}, DestEP_{uid,y}, Exp$), where $SrcEP_{nid,x}$ is the source endpoint, $DestEP_{uid,y}$ is the destination endpoint, and Exp is the expression whose valuation is being sent, the corresponding constraint formula is $(NS^i = nid \wedge NC^i_{nid} = sn) \rightarrow (NC^{i+1}_{nid} = sn' \wedge Q_{uid,y}[tail^i_{uid,y}] = Exp^i \wedge tail^{i+1}_{uid,y} = tail^i_{uid,y} + 1 \wedge \delta(\{tail^i_{uid,y}\}))$. This formula states that, at time instant i, if node N_{nid} is the active node and N_{nid}'s node counter is equal to sn, then the node counter in the time instant $i+1$ is set to sn', the valuation of the sent expression is enqueued to the destination endpoint queue $(Q_{uid,y}[tail^i_{uid,y}] = Exp^i \wedge tail^{i+1}_{uid,y} = tail^i_{uid,y} + 1)$, and all local variables and all queues heads and tails but $tail^i_{uid,y}$ should have in time instant $i+1$, the same values they had in time instant i $(\delta(\{tail^i_{uid,y}\}))$.

4) For a blocking receive statement in the format of S=('recv_b', nid, sn, sn', $RecvEP_{nid,x}, v$), the corresponding constraint formula is $(NS^i = nid \wedge NC^i_{nid} = sn) \rightarrow (NC^{i+1}_{nid} = sn' \wedge v^{i+1} = Q_{nid,x}[head^i_{nid,x}] \wedge head^{i+1}_{nid,x} = head^i_{nid,x} + 1 \wedge \delta(\{head^i_{nid,x}, v^i\}))$. In this formula, the relevant queue is dequeued, and the dequeued value is assigned to the receiving variable $(v^{i+1} = Q_{nid,x}[head^i_{nid,x}] \wedge head^{i+1}_{nid,x} = head^i_{nid,x} + 1)$.

5) For a non-blocking send statement in the format of S=('send_nb', nid, sn, sn', $SrcEP_{nid,x}, DestEP_{uid,y}, Exp, R$), such that R is the request variable associated with S, the corresponding constraint formula is $(NS^i = nid \wedge NC^i_{nid} = sn) \rightarrow (NC^{i+1}_{nid} = sn' \wedge Q_{uid,y}[tail^i_{uid,y}] = Exp^i \wedge tail^{i+1}_{uid,y} = tail^i_{uid,y} + 1 \wedge R^{i+1} = PND \wedge \delta(\{R^i, tail^i_{uid,y}\}))$. In addition to enqueuing the valuation of the sent expression, the value of the request variable is set to pending $(R^{i+1} = PND)$.

Like the statements constraint, the scheduling constraint ($\mathbb{C}$) is the conjunction of B constraints ($\mathbb{C} = \bigwedge_{i=1}^{B} \mathbb{c}^i$). Each $\mathbb{c}^i$ constraint consists of four parts which ensure that 1) a node that is done carrying out all its statements, will not be an active node, 2) the variables of an inactive node will not change, 3) a blocking receive will not take place if the relevant queue is empty, and 4) a wait associated with a non-blocking receive, will not take place if the relevant queue is empty. Due to the limited space, we present only the first ($\mathbb{c}^i_{done}$) and the third ($\mathbb{c}^i_{blocked\_recvb}$) parts of the scheduling constraint. $\mathbb{c}^i_{done}$ is expressed as: $\bigwedge_{n=1}^{|\mathbb{N}|}[(NC^i_n = \perp) \rightarrow (NS^i \neq n)$. This formula states that: when all the statements in a node have been executed $(NC^i_n = \perp)$, then this node can't be an active node $(NS^i \neq n)$. $\mathbb{c}^i_{blocked\_recv}$ is expressed as $\bigwedge_{n=1}^{|\mathbb{N}|}[((NC^i_n = A_{n,x}) \wedge (A_{n,x} = ('\text{m\_recv\_b}, A_{n,x}, EP_{recv}, v)) \wedge (head^i_{n,EP_{recv}} = tail^i_{n,EP_{recv}})) \rightarrow (NS^i \neq n)]$. This formula states that: if the pending action in a node is a blocking receive $((NC^i_n = A_{n,x}) \wedge (A_{n,x} = ('\text{m\_recv\_b}, A_{n,x}, EP_{recv}, v)))$ and the relevant queue is empty $(head^i_{n,EP_{recv}} = tail^i_{n,EP_{recv}})$, then this node can't be the active node $(NS^i \neq n)$.

The property constraint ($\mathbb{P}$) is derived from either a user-supplied application-specific property expressed as an assert statement, or from a built-in safety property such as "No message races exists". For an assert statement in the format of S=('assert', nid, sn, sn' ,$Expr$), the corresponding property constraint, is $(NS^i = nid \wedge NC^i_{nid} = sn) \rightarrow (Expr^i = true)$.

The overall formula that is passed to the SMT solver is the conjunction of the initial constraint, the statements constraint, the scheduling constraint and the negation of property constraint, and is expressed as $\mathbb{I} \wedge \mathbb{S} \wedge \mathbb{C} \wedge \sim \mathbb{P}$.

3 Related Work

To the best of our knowledge, the only other tool for analyzing MCAPI applications is MCC [4]. MCC explores all possible orders of messages arrival by repeated executions. It creates a scheduling layer above the MCAPI runtime, which allows MCC to discover all potentially matching send/receive pairs by intercepting calls to the MCAPI runtime. In [5], C. Wang et al. introduce a symbolic algorithm that detects concurrency errors in all feasible permutations of statements in an execution trace. They use concurrent static single assignment (CSSA) based encoding to construct an SMT formula. The algorithm has been applied to detect concurrency errors in shared memory multithreaded C programs.

4 Conclusion

We have presented CRI, a tool for symbolically debugging MCAPI applications. Our tool builds an SMT formula that encodes the semantics of an MCAPI application execution trace. By such analysis we are able to detect and reproduce errors that may not be discovered by traditional testing approaches. Due to the lack of publicly available MCAPI benchmarks, we performed experiments on MCAPI applications developed by ourselves. For example, the full code of the application in Fig. 1 was found to have an assertion failure in 0.03 seconds using Yices [3] as the SMT solver. We plan to extend our tool to support connection-oriented calls, and investigate optimizations to improve the performance.

References

1. The Multicore Association Communications API, `http://www.multicore-association.org/workgroup/mcapi.php`
2. Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K.: Debugging Race Conditions in Message-Passing Programs. In: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, SPDT 1996, PA, USA, pp. 31–40 (1996)
3. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
4. Sharma, S., Gopalakrishnan, G., Mercer, E., Holt, J.: MCC: A runtime verification tool for MCAPI applications. In: 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, Austin, Texas, USA, November 15-18 (2009)
5. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)

MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming*

Gal Katz and Doron Peled

Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel

Abstract. We present MCGP - a tool for generating and correcting code, based on our synthesis approach combining deep **M**odel **C**hecking and **G**enetic **P**rogramming. Given an LTL specification, genetic programming is used for generating new candidate solutions, while deep model checking is used for calculating to what extent (i.e., not only whether) a candidate solution program satisfies a property. The main challenge is to construct from the result of the deep model checking a fitness function that has a good correlation with the distance of the candidate program from a correct solution. The tool allows the user to control various parameters, such as the syntactic building blocks, the structure of the programs, and the fitness function, and to follow their effect on the convergence of the synthesis process.

1 Introduction

With the growing success of model checking for finding bugs in hardware and software, a natural challenge is to generate automatically correct-by-design programs. This is in particular useful in intricate protocols, which even skillful programmers may find difficult to implement. Automatically constructing a reactive system from LTL specification was shown to be an intractable problem (in 2EXPTIME) in [12]. For concurrent systems, the situation is even worse, as synthesizing a concurrent system with two processes from LTL specification is already undecidable [13]. A related (and similarly difficult) problem is correcting a piece of code that fails to satisfy its specification.

Genetic programming [1] is a method for automatically constructing programs. This is a directed search on the space of syntactically limited programs. Mutating and combining candidate solutions is used for generating new candidates. The search progresses towards a correct solution using a fitness function that is calculated for the newly generated candidates. Traditionally, the fitness is based on testing. Recently, we studied the use of model checking as a basis for providing fitness values to candidates [7]. Experimentally, we learned that the success of the model checking based genetic programming to synthesize correct programs is highly dependent on using fitness functions that are designed for the specific programming goal. One of the main features of our tool is the ability

* This research was supported in part by ISF Science Foundation grant 1262/09.

to provide the user with flexible ways for constructing the fitness function based on model checking and various parameters of the desired code.

A central building block of our tool is using *deep model checking*, which does not only checks whether a candidate program satisfies a property or not, but *to what extent* it does so. This provides a finer analysis, and consequently helps the convergence of the genetic search. Several levels of satisfactions that we use are described later. Furthermore, one may want to assign different priorities to properties (e.g., if a basic safety property does not hold, there is no gain in considering other satisfied liveness properties), and include other code parameters (e.g., decreasing fitness according to the length of the code). Our experience with the method shows that experimenting with fine-tuning the fitness function is a subtle task; its goal is to anticipate a good correlation between the fitness of a candidate program and its distance from a correct program, thus helping to steer the search in the right direction.

In previous work, we used our tool for the synthesis of various protocols. However, this was done in an ad hoc manner, requiring to dynamically change the tool for every new synthesis problem. The current presented tool is a generalization of our previous experience. It allows the user to automatically synthesize new protocols by only providing and tuning a set of definitions, without a need to change the tool itself.

2 Tool Architecture and Configuration

The tool is composed of a server side (written in C++) responsible for the synthesis process, and a user interface used for specifying the requirements, and interactively observing the synthesis results. The tool architecture and information flow are depicted in Fig. 1. The server side combines a deep model checker and a genetic programming engine, both enhanced in order to effectively integrate with each other. This part can be used directly from the command line, or by a more friendly Windows based user interface. In order to synthesize programs, the user first has to fill in several definitions described throughout this section, and then initiate the synthesis process described in section 3.

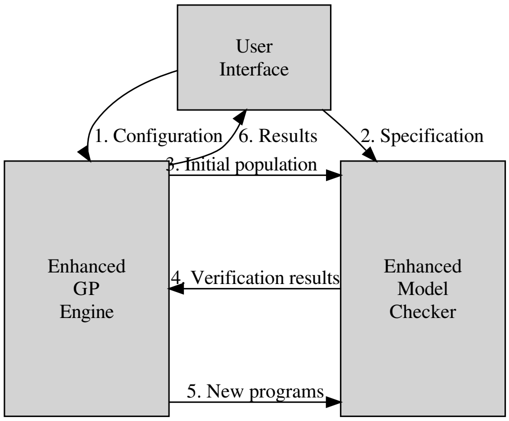


Fig. 1. Tool architecture

Genetic Programming Definitions. The first step in using the tool is to define the various process types that will comprise the synthesized programs.

A program consists of one or more process types containing the execution code itself, and an *init* process responsible for creating instances of the other process types, and initializing global variables. For each process type, the user then has to choose the appropriate building blocks from which its code can be generated. The tool comes with a library of building blocks used by common protocols and algorithms, such as variables, control structures, inter-process communication primitives, and more. Furthermore, the user can define new building blocks by combining existing building blocks, and optionally by writing additional C++ code. These building blocks can vary from simple expressions, macros and functions to complex statements that will be later translated into atomic transitions.

Each process type can contain both static parts set by the user, and dynamic parts, which the tool is required to generate and evolve. Depending on these settings, the tool can be used for several purposes:

– Setting all parts as *static* will cause the tool to simply run the deep model checking algorithm on the user-defined program, and provide its detailed results, including the fitness score assigned to each LTL property.
– Setting the *init* process as *static*, and all or some of the other processes as *dynamic*, will order the tool to synthesize code according to the specified architecture. This can be used for synthesizing programs from scratch, synthesizing only some missing parts of a given partial program, or trying to correct or improve a complete given program.
– Setting the *init* process as *dynamic*, and all other processes as *static*, is used when trying to falsify a given parametric program. In this case the tool will reverse its goal and automatically search for configurations that violate the specification (see [9]).
– Setting both the *init* and the program processes as *dynamic*, is used for synthesizing parametric programs. It causes the tool to alternatively evolve various programs, and configurations under which these programs have to be satisfied [9].

Specification Definitions. At this stage the user should provide a full system specification from which the tool can later derive the fitness function assigned to each generated program. The specification is mainly based on a list of LTL properties. The atomic propositions used by these properties are defined as C expressions representing Boolean conditions. The properties and the atomic propositions are then compiled into executable code that is later used by the model checking process (this is inspired by the way the Spin model checker [4] handles LTL properties). The user can set up a hierarchy between the checked properties, instructing the tool to start with some basic properties, and only then gradually check more advanced properties [7]. In addition to the LTL properties, other quantitative goals can be added, such as minimizing the size of generated programs. Further advanced model checking options (such as the DFS search depth, and the use of partial order reduction) can be tuned as well.

3 The Synthesis Process

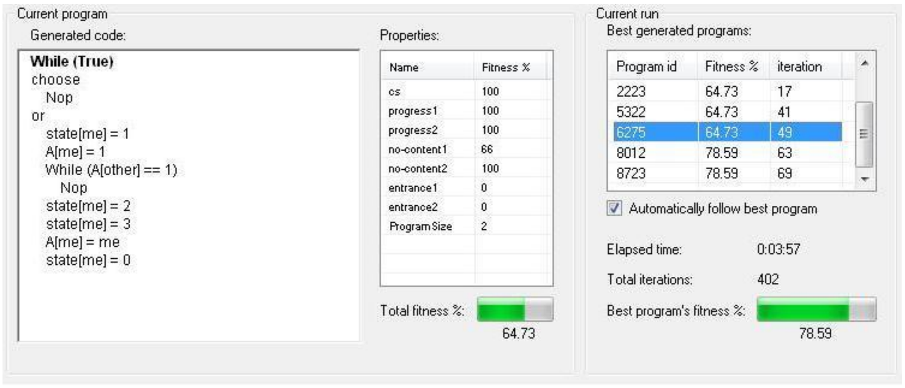
After completing the definitions step, the synthesis process can be initiated. First, using the skeletons and building blocks provided by the user for each process type, an initial set P of programs is randomly generated. The code of each process type is stored as a syntactic tree whose nodes are instances of the various building blocks, representing variables, statements, functions, constants, etc.

Next, an iterative process of improvements begins. At each iteration, a small subset of μ programs is selected randomly from P. Then, various genetic operations are performed on the selected programs, leading to the generation of new λ modified programs. The main operation in use is *mutation*, which basically adds, removes or changes the program code, by manipulating its syntactic tree [7]. Deep model checking is then performed on each new program, in order to measure the degree on which it satisfies the specification.

A probabilistic selection mechanism is then used in order to select new μ programs that will replace the originally selected μ programs, where a program has a chance to survive proportionally to its fitness score. The iterative process proceeds until either a perfect solution is found, or after the maximal allowed iterations number is reached.

During the process, the user can watch the gradual generation of solutions, by following the best generated programs, and by navigating through the chain of improvements. For each selected program, the generated code is displayed, as well as the deep model checking results, including the fitness score assigned to each specification property. Often, watching these results leads to some insights regarding tuning required to the specification, the building blocks, or other parameters, and the fitness function based on the above. Fig. 2 shows an example of the tool's screen during the synthesis of a mutual exclusion protocol described in [7].

Deep Model Checking. The main challenge in making the model checking based genetic approach work in practice is to obtain a fitness function that



```
While (True)
choose
  Nop
or
  state[me] = 1
  A[me] = 1
  While (A[other] == 1)
    Nop
  state[me] = 2
  state[me] = 3
  A[me] = me
  state[me] = 0
```

Name	Fitness %
cs	100
progress1	100
progress2	100
no-content1	66
no-content2	100
entrance1	0
entrance2	0
ProgramSize	2

Total fitness %: 64.73

Program id	Fitness %	iteration
2223	64.73	17
5322	64.73	41
6275	64.73	49
8012	78.59	63
8723	78.59	69

☑ Automatically follow best program

Elapsed time: 0:03:57
Total iterations: 402
Best program's fitness %: 78.59

Fig. 2. The user interface during synthesis of a mutual exclusion algorithm

correlates well with the potential of the candidate programs. In order to achieve this, it is not enough just to count the number of LTL properties that hold, as done in [5]; this gives a coarse indication that behaves poorly under experimentations. With this tool, we introduce different levels or *modes* of satisfaction for each LTL property; a property does not necessarily have to be satisfied by all the executions in order to contribute to the fitness value of the checked candidate.

We consider the following modes of satisfaction:

– None of the program's executions satisfy the property (level 0),
– some of the program's executions satisfy the property (level 1),
– each prefix of a program execution can be extended into an execution satisfying the property (level 2), and
– all of the program's executions satisfy the property (level 3).

Model checking algorithms usually only check membership for levels 3. By using also the formula itself, and not only its negation, we can also check membership for level 0, leading to a third possible level, where neither level 0 nor level 3 holds. Further distinction between levels 1 and 2 requires a different kind of analysis, and in fact, the complexity of checking it is higher than simple model checking: a reduction in [10] can be used to show that it is in EXPTIME-Complete. One can use a logic that permits specifying, separately, the LTL properties, and the modes of satisfaction [11] or develop separate optimized algorithms for each level as we did [7].

A main concern is the model checking efficiency: while we often try to synthesize just basic concurrent programs, with small as possible requirements, they can still have a large number of states (e.g., tens of thousands). Moreover, model checking is performed here a large number of times: we may have thousands of candidates before the synthesis process terminates or fails. Accordingly, in the latest version of the tool, we decided to slightly change the definition of the intermediate fitness levels, by adopting a technique similar to probabilistic qualitative LTL model checking. We treat the nondeterministic choices as having some probabilities, and base the distinction between the new levels on the probability p that the program satisfies the *negation* of the checked property ($p > 0$ implies new level 1, and $p = 0$ implies new level 2). While this new algorithm occasionally shifts the boundary between the two levels (compared to the original ones), it has the advantage of having PSPACE complexity [2].

In order to compute the fitness function, the following is applied to each LTL property. The property, and its negation are first translated into standard Büchi Automata, by running the LTL2BA code [3]. Then these automata are further extended (in order to fit into our specialized algorithm), and combined with the program's automaton, yielding the fitness levels mentioned above. Additional factors such as implication properties, and deadlocks can affect the fitness scoring as well [7]. The scoring for all of the LTL properties are finally summed up, possibly in conjunction with other quantitative measures (such as the program size) into a final fitness score assigned to the program.

Tool Evaluation. Using our tool, we successfully synthesized correct solutions to a series of problems, including known [7], and novel [6] two-process mutual exclusion algorithms, and parametrized leader election protocols [8]. Recently we used the tool's ability of synthesizing architectures, for the automatic discovery and correction of a subtle bug in the complicated α-core protocol [9]. The synthesis duration usually varies from seconds to hours, depending on many aspects of the problems, such as the number of processes, the solutions size, and the model checking time. The main synthesis algorithm has a large amount of parallelism, and thus execution time can be greatly improved when running on multi-core servers. A weakness of the genetic programming based approach is its probabilistic nature, which does not guarantee convergence into perfect solutions. The duration and success of the synthesis depend on the choices made by the user for the various building blocks and parameters. Additional information about the tool, including its freely available latest version, and some running examples, can be found at: http://sites.google.com/site/galkatzzz/mcgp-tool

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications, 3rd edn. Morgan Kaufmann/Dpunkt Verlag (2001)
2. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)
3. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
4. Holzmann, G.J.: The SPIN Model Checker. Pearson Education, London (2003)
5. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007)
6. Katz, G., Peled, D.: Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 33–47. Springer, Heidelberg (2008)
7. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008)
8. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: HVC (2009)
9. Katz, G., Peled, D.: Code mutation in verification and automatic code correction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 435–450. Springer, Heidelberg (2010)
10. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
11. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)
12. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
13. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)

ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems

Alexandre David[1], Kim. G. Larsen[1], Axel Legay[2], Ulrik Nyman[1], and Andrzej Wąsowski[3]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France
[3] IT University of Copenhagen, Denmark

Abstract. We present ECDAR a new tool for compositional design and verification of real time systems. In ECDAR, a component interface describes both the behaviour of the component and the component's assumptions about the environment. The tool supports the important operations of a good compositional reasoning theory: composition, conjunction, quotient, consistency/satisfaction checking, and refinement. The operators can be used to combine basic models into larger specifications to construct comprehensive system descriptions from basic requirements. Algorithms to perform these operations have been based on a game theoretical setting that permits, for example, to capture the real-time constraints on communication events between components. The compositional approach allows for scalability in the verification.

1 Overview

The context. Contemporary IT systems are assembled out of multiple independently developed components. Component providers operate under a contract on what the interface of each component is. Interfaces are typically described using textual documents or models in languages such as UML or WSDL. Unfortunately, such specifications are subject to interpretation. To avoid the risk of ambiguity, we recommend mathematically sound formalisms, such as interface theories, whenever possible. A good interface theory supports *refinement checking* (whether an interface can be replaced by another one), *satisfaction checking* (whether an implementation satisfies the requirements expressed with the interface), *consistency checking* (whether the interface can be implemented), a *composition operator* (structurally combining interfaces), a *conjunction operator* (computing a specification whose implementations are satisfying both operands), and a *quotient operation* that is the adjoint for composition. It should also guarantee important properties such as independent implementability [10].

It has been argued [7,10] that *games* constitute a natural model for interface theories: each component is represented by an automaton whose transitions are typed with *input* and *output* modalities. The semantics of such an automaton is given by a two-player game: the *input* player represents the environment, and the *output* player represents the component. Contrary to the input/output model proposed by Lynch [13], this semantic offers (among many other advantages) an optimistic treatment of composition (two interfaces can be composed if there exists at least one environment in which they can interact together in a safe way) and refinement (the refined system should accepts at least the same inputs and not produce more outputs). Game-based interfaces were first developed for *untimed systems* [10,8] and the composition and refinement operations were implemented in tools such as TICC [1] or CHIC [5].

Example. We will demonstrate our tool, ECDAR, by executing a compositional verification process. To that end we introduce a running example based on a modified real-time version of Milner's scheduler [14]. Fig. 1 (left) shows a single node, which can receive a start signal on rec_i. The node subsequently begins external work by outputting on w_i. In parallel to this it can forward the token by outputting on rec_{i+1}, but only after a delay between d and D time units. Fig. 1 (right) illustrates a ring of such nodes M_i in which some nodes have been grouped together. This grouping exemplifies a part of the specification, which we will later be able to replace with an abstraction SS_i in order to execute a compositional proof.

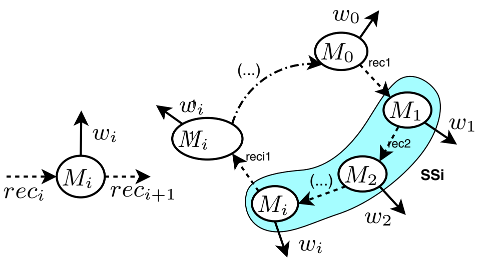


Fig. 1. Overview of Milner's scheduler example and the sub-specification SS_i

The timed case. The above example contains timing requirements that cannot be handled with tools such as TICC or CHIC, designed for untimed systems. There exist timed formalisms but they do not provide a satisfactory notions of composition and refinement. We have recently proposed the first complete *timed* interface theory based on timed games [6]. The idea is similar to the untimed case: components are modelled using timed input/output automata (TIOAs) with a timed game semantics [4]. Our theory is rich in the sense that it captures all the good operations for a compositional design theory. In this paper we go one step further and present ECDAR, a tool that implements the theory of [6]. We thus propose the first complete game-based tool for timed interfaces in the dense time setting. ECDAR implements checkers such as satisfaction/consistency, refinement, and satisfaction of TCTL formulas. The tool also supports the classical compositional reasoning operations of conjunction and composition. To the best of our knowledge, ECDAR is the first tool to propose an implementation of quotient. In addition, it comes with a user-friendly interface, where errors are reported in an intelligible way.

2 An Integrated Environment for Design and Analysis

The user interface of ECDAR is divided into two parts: 1) the *specification interface* where automata are specified in a graphical manner, and 2) the *query interface* where one can ask verification questions.

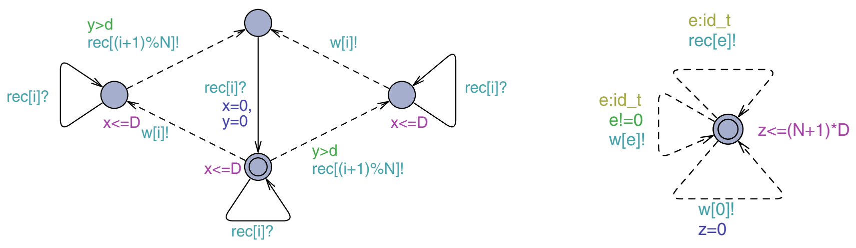


Fig. 2. Left: Template for a single node M_i. Right: Template for the overall specification.

Specification Interface. The specification interface of ECDAR uses the language of UPPAAL-TIGA to describe timed I/O automata (instead of timed game automata) with *input* and *output* modalities that are essential in this case. TIOAs communicate via broadcast channels. Global (shared) variables are not permitted. The user specifies whether the TIOA should be viewed as an implementation or as a specification. For implementations, the tool checks on-the-fly if every state respects the *independent progress* property [6], that progress must be ensured by the implementation. Details are available at ecdar.cs.aau.dk. The tool has its own engine, which reuses components of the game engine of UPPAAL-TIGA to support the new operators.

We model the scheduler using templates, in an entirely modular way. One only needs to instantiate more nodes to make a larger instance of the system. A single node of our scheduler is shown in the left side of Fig. 2. In the initial location of the specification, it is ready to receive a message on the channel `rec[i]?`. After this there are two ways to return to the initial state depending on the order in which it starts its work (`w[i]!`) and passes on the token (`rec[(i+1)%N]!`). The first node of the system M_0 is instantiated with a different initial location (the bottom-most one), reflecting the fact that it holds the token initially. The right side of Fig. 2 shows the overall specification S_0 of the system. It requires that `w[0]!` occurs at least every $(N+1) * D$ time units. Remaining actions can be executed freely.

Query Interface. The query interface provides two main checkers, the *refinement* checker and the *consistency* checker. The refinement checker is used to decide if an implementation satisfies a given specification or if a specification refines another one. As stated in [6], refinement checking reduces to solving a safety timed game between the two components. For our example one way to verify that the scheduler is correct is to verify a property of the type:

```
refinement: ( M0 || M1 || M2 || M3 || M4 ) <= S0
```

We call this type of verification monolithic, since it constructs a specification precisely representing the entire system. The tool provides a strategy to prove or disprove the property, which can be used to refine the model. The strategy can be played interactively. The consistency checker is used to check whether a specification admits at least one implementation. This question reduces to the one of deciding if there exists a strategy for the output player to avoid reaching *bad states* in the specification, i.e., states that do not satisfy the *independent progress* property. A *pruning* facility removes all the states not covered by the strategy. It can drastically reduce the state-space of the system. Following a similar principle, it is possible to constrain an interface with a TCTL* formula. For example, like in [11], one can use a Büchi objective to remove states allowing Zeno behaviours. This is the first time that a tool for compositional reasoning proposes this feature in the dense time setting.

3 Illustration and Experiment

In our example we have a ring of N nodes. It is natural to verify the monolithic property in order to show that the composed system refines the overall specification. Unfortunately, this strategy fails due to state-space explosion. As the number of components is increased, the state space grows, and more nondeterminism and interleaving is introduced in the system.

In order to combat the problem we apply compositional verification. The idea is to create N sub-specifications that are used in a series of refinement steps. First one shows that $M_1 \leq SS_1$. After this it is proved for increasing indexes, 1 to N that $SS_i||M_{i+1} \leq SS_{i+1}$. Finally the property $SS_n||M_0 \leq S_0$ is checked. Fig. 3 gives the properties for five nodes. The sub-specification aims at capturing the important aspect of the subsystem needed for the next step in the verification process of the overall property. It is very important to notice that the sub-specification is like all the other components in the system created as a template and that thus it is modelled only once and then instantiated with different indices.

```
refinement: M1 <= SS1
refinement: ( SS1 || M2 ) <= SS2
refinement: ( SS2 || M3 ) <= SS3
refinement: ( SS3 || M4 ) <= SS4
refinement: ( SS4 || M0 ) <= S0
```

Fig. 3. Incremental verification

Here the sub-specification SS_i, as shown in Fig. 4, is a model for a sequence of nodes $M_1||\ldots||M_i$ (see *Fig.* 1). Informally SS_i is expressed as following, noting that the relevant ports for this subsystem are `rec[1]?`, `w[e]!` `(0<e<=i)` and `rec[i+1]!`: Under the assumption that a) the time elapsing between two `rec[1]?` is more than $N * d$ time-units and b) there are no two consecutive `rec[1]?` without a `rec[i+1]!`, then it is guaranteed that `rec[i+1]!` will occur within $[i * d, i * D]$ time units from `rec[1]?`.

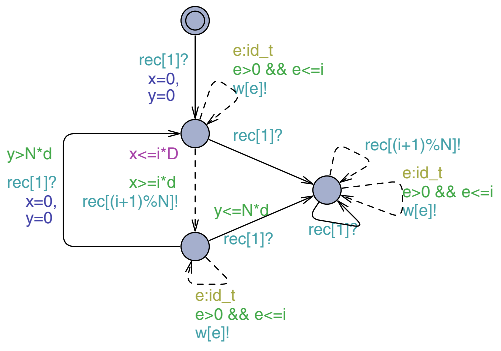


Fig. 4. The sub-specification SS_i that abstracts the the sub-system $M_1||\ldots||M_i$

We have performed experiments for different values of N, number of nodes in the ring, and d the minimum time delay before passing on the token. We have fixed the upper time limit for passing the token to 30. The results of the experiments are shown in Table 1. The table shows the time used to check a given property measured in seconds. For each value of N we have two rows. The top one represents the verification of all the steps in the compositional verification while the bottom row represents the verification of one monolithic property. If the verification took more than 600 seconds we stopped it. We had one instance where ECDAR ran out of memory which is indicated by `om`. The time results that are written in *italics* are the cases in which the compositional verification gave a negative result. In these cases one needs to propose more precise sub-specifications in order to make the compositional verification work. The monolithic method gives positive results in these cases.

In the case where d is close to D there is very little interleaving in the system and in this case the verification of the monolithic property is the fastest. The smaller the d value the more interleaving appears in the system and in these complex cases the compositional verification shows its strength. The cases where the compositional verification beats the monolithic are marked by **boldface**.

4 Related Work

In the untimed setting multiple contributions of Alfaro et al. focus on the operations of composition and refinement. Hence, tools such as TICC or CHIC only provide these operations. Theories exists for quotient [3] and conjunction [12], but they have not been implemented neither in TICC nor in CHIC. More recently, Bauer et al. have proposed a new extension of interface automata with new definition for composition/compatibility and refinement; these results are

Table 1. Results of the verification experiments

	$d = 29$	20	10	9	8	6	4
$n = 5$	0.080	0.097	0.191	*0.169*	*0.172*	*0.151*	*0.205*
monolithic	0.034	0.034	0.073	1.191	1.189	64.933	> 600
$n = 6$	0.102	0.133	0.231	*0.228*	*0.238*	*0.238*	*0.294*
monolithic	0.040	0.043	0.095	6.786	6.791	> 600	> 600
$n = 8$	0.225	0.349	0.516	**0.515**	*0.540*	*0.600*	*0.582*
monolithic	0.076	0.076	0.230	88.542	88.642	> 600	> 600
$n = 12$	0.830	1.414	1.802	**1.895**	**1.831**	*2.079*	*2.181*
monolithic	0.220	0.223	0.843	> 600	> 600	> 600	> 600
$n = 20$	4.990	9.739	12.377	**11.923**	**12.041**	**12.438**	*12.764*
monolithic	1.038	1.030	4.523	> 600	> 600	> 600	> 600
$n = 30$	22.053	45.709	55.728	**55.345**	**55.112**	**54.702**	*56.164*
monolithic	3.791	3.778	17.652	> 600	> 600	> 600	`om`

implemented in the MIO Workbench [2]. This work remains at the level of untimed systems, not considering operations such as quotient or pruning.

A first dense time extension of the theory of interface automata has been developed in [11]. The theory in [11] focuses exclusively on reducing composition and consistency checking to solving timed games and does not provide any definition and algorithms for refinement, conjunction, and quotient. In [9], Alfaro and Faella proposed an efficient implementation of the algorithm used to solve the timed games introduced in [11], but for the discretized time domain only. In addition, they also proposed an extension of TICC to the timed setting. This version of TICC does not provide the same services as Ecdar does. First, timed TICC only supports consistency checking and composition; the usefulness of the tool for compositional design of real time systems is thus limited. Second, the tool does not offer a user friendly interface and the interactions with the user are extremely limited. Last, the tool works on the discretized time domain only. Hence, all the complications introduced by the dense setting are not studied.

References

1. Adler, B.T., de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Raman, V., Roy, P.: Ticc: A tool for interface compatibility and composition. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 59–62. Springer, Heidelberg (2006)
2. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the mio workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010)
3. Bhaduri, P.: Synthesis of interface automata. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 338–353. Springer, Heidelberg (2005)
4. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
5. Chic (2003), http://www-cad.eecs.berkeley.edu/~tah/chic/
6. David, A., Larsen, K., Legay, A., Nyman, U., Wąsowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC (accepted 2010)
7. de Alfaro, L.: Game models for open systems. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 269–289. Springer, Heidelberg (2004)
8. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
9. de Alfaro, L., Faella, M.: An accelerated algorithm for 3-color parity games with an application to timed games. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 108–120. Springer, Heidelberg (2007)
10. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Marktoberdorf Summer School. Kluwer Academic Publishers, Dordrecht (2004)
11. de Alfaro, L., Henzinger, T.A., Stoelinga, M.I.A.: Timed interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
12. Doyen, L., Henzinger, T.A., Jobstman, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT, pp. 79–88. ACM Press, New York (2008)
13. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. Technical Report MIT/LCS/TM-373, The MIT Press (November 1988)
14. Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)

Developing Model Checkers Using PAT

Yang Liu, Jun Sun, and Jin Song Dong

School of Computing
National University of Singapore
{liuyang,sunj,dongjs}@comp.nus.edu.sg

Abstract. During the last two decades, model checking has emerged as an effective system analysis technique complementary to simulation and testing. Many model checking algorithms and state space reduction techniques have been proposed. Although it is desirable to have dedicated model checkers for every language (or application domain), implementing one with effective reduction techniques is rather challenging. In this work, we present a generic and extensible framework PAT, which facilitates users to build customized model checkers. PAT provides a library of state-of-art model checking algorithms as well as support for customizing language syntax, semantics, state space reduction techniques, graphic user interfaces, and even domain specific abstraction techniques. Based on this design, model checkers for concurrent systems, real-time systems, probabilistic systems and Web Services are developed inside the PAT framework, which demonstrates the practicality and scalability of our approach.

1 Introduction

After two decades' development, model checking has emerged as a promising and powerful approach for automatic verification of hardware and software systems. It has been used successfully in practice to verify complex circuit design [3], communication protocols [5] and driver software [2]. Till now, model checking has become a wide area including many different model checking algorithms catering for different properties (e.g., explicitly model checking, symbolic model checking, probabilistic model checking, etc.) and state space reduction techniques (e.g., partial order reduction, binary decision diagrams, abstraction, symmetry reduction, etc.).

Unfortunately, several reasons prevent many domain experts, who may not be experts in the area of model checking, from successfully applying model checking to their application domains. Firstly, it is nontrivial for a domain expert to learn a general purpose model checker (e.g., NuSMV [3], SPIN [5] and so on). Secondly, general purpose model checkers may be inefficient (or insufficient) to model domain specific applications, due to lack of language features, semantic models or data structures. For example, multi-party barrier synchronization or broadcasting is difficult to achieve in the SPIN model checker. Lastly, the level of knowledge and effort required to create a model checker for a specific domain is even higher than applying existing ones.

To meet the challenges of applying model checking in new application domains, we propose a generic and extensible framework called PAT (Process Analysis Toolkit) [1], which facilitates effective incorporation of domain knowledge with formal verification using model checking techniques. PAT is a self-contained environment to support

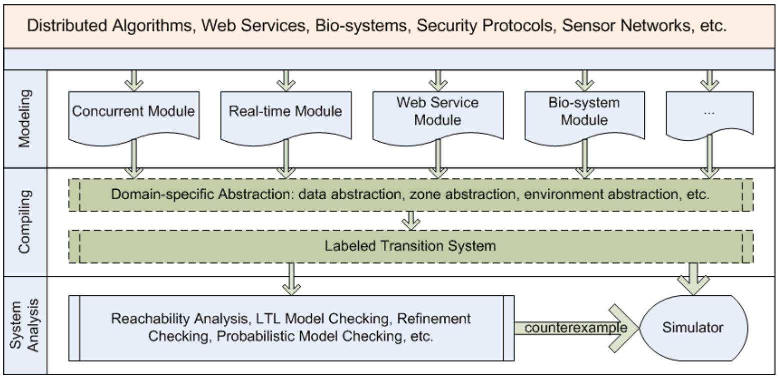

Distributed Algorithms, Web Services, Bio-systems, Security Protocols, Sensor Networks, etc.
Modeling
Concurrent Module
Real-time Module
Web Service Module
Bio-system Module
...
Compiling
Domain-specific Abstraction: data abstraction, zone abstraction, environment abstraction, etc.
Labeled Transition System
System Analysis
Reachability Analysis, LTL Model Checking, Refinement Checking, Probabilistic Model Checking, etc.
counterexample
Simulator

Fig. 1. PAT Architecture

composing, simulating and reasoning of system models. It comes with user friendly interfaces, a featured model editor and an animated simulator. Most importantly, PAT implements a library of model checking techniques catering for checking deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions [13], refinement checking [11] and probabilistic model checking. Advanced optimization techniques are implemented in PAT, e.g., partial order reduction, process counter abstraction [16], bounded model checking [14], parallel model checking [8] and probabilistic model checking. PAT supports both explicit state model checking and symbolic model checking (based on BDD or SAT solver). We have used PAT to model and verify a variety of systems [6]. Previously unknown bugs have been discovered [7]. The experiment results show that PAT is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in some cases.

2 Architecture Overview

PAT was initially designed to support a unified way of model checking under fairness [13]. Since then, PAT has been extended significantly and completely re-designed. We have adopted a layered design to support analysis of different systems/languages, which can be implemented as plug-in modules. Fig. 1 shows the architecture design of PAT. For each supported domain (e.g., distributed system, real-time system, service oriented computing and so on), a dedicated module is created in PAT, which identifies the (specialized) language syntax, well-formness rules as well as formal operational semantics. For instance, the CSP module is developed for the analysis of concurrent system modeled in CSP# [12]. The operational semantics of the target language translates the behavior of a model into LTS (Labeled Transition Systems)[1] at runtime. LTS serves as an implicitly shared internal representation of the input models, which can be automatically explored by the verification algorithms or used for simulation. To perform model checking on LTSs, the number of states in the LTSs needs to be finite. For

[1] To be precise, it is a Markov Decision Process when probabilistic choices are involved.

systems with infinite behavior (e.g., real time clocks or infinite number of processes), abstraction techniques are needed. Examples of abstraction techniques include data abstraction, process counter abstraction, clock zone abstraction, environment abstraction, etc. The verification algorithms perform on-the-fly exploration of the LTSs. If a counterexample is identified during the exploration, then it can be animated in the simulator. This design allows new modules to easily be plugged in and out, without recompiling the core system, and the developed model checking algorithms (mentioned in Section 1) to be shared by all modules. This design achieves *extensible architecture* as well as *module encapsulation*. We have successfully applied this framework in development of four different modules, each of which targets a different domain. 1). The concurrent system module is designed for analyzing general concurrent systems using a rich modeling language CSP# [12], which combines high-level modeling operators with programmer-favored low-level features. 2). The real-time system module supports analysis of real-time systems with compositional behavioral patterns (e.g. $timeout$, $deadline$) [15]. Instead of explicitly manipulating clock variables, time related constructs are designed to build on implicit clocks and discretized using clock zone abstraction [15]. 3). The Web Services (WS) module offers practical solutions to the conformance checking and prototype synthesis between WS Choreography and WS Orchestration. 4). The probabilistic module supports the modeling and verification of systems exhibiting random or probabilistic behavior.

3 Manufacturing Model Checkers

In this section, we discuss how to create a customized model checker for a new domain using the PAT framework with the help of its predefined APIs, examples and software packages. A domain often has its specific model description language. It is desirable that the domain experts can input their models using their own languages. There are three different ways of supporting a new language in PAT.

- The easiest way is to create a syntax rewriter from the domain specific language to an existing language. This is only recommended if the domain language is less expressiveness than the existing languages. For example, we have developed translators from Promela/UML state diagram to CSP#. Comparing with other tools, programming a translator is straightforward in PAT. Because PAT has open APIs for its language constructs, users only need to generate the language constructs objects using these APIs, which can guarantee that the generated syntax is correct. This approach is simple and requires little interaction with PAT codes. However, translation may not be optimal if special domain specific language features are present. Furthermore, reflecting analysis results back to the domain model is often non-trivial.
- The second way is to extend an existing module if the input languages are similar and yet with a few specialized features. For example, the probabilistic module is designed to extend the concurrent system module with one additional language feature, i.e., probabilistic choices. Knowledge about existing modules is required and a new parser may be created for the extended language features.

- The third way is to create a new module in PAT. In this case, users firstly need to develop a parser according to the syntax. The parser should generate a model consisting of ASTs of language construct classes, which encode their operational semantics. Abstract classes[2] for system states, language construct classes and system model are pre-defined in PAT with (abstract) signature methods for communications with verification algorithms and user interface interactions. Users only need to develop concrete classes in the new module by inheriting the abstract classes. This approach is the most complicated compared with the first two. Nevertheless, this approach gives the most flexibility and efficiency. It is difficult to quantify the effort required to build a high-quality module in PAT. Experiences suggest that a new module can be developed in months or even weeks in our team. This approach is feasible for domain experts who have only the basic knowledge on model checking. This is because model checking algorithms and state space reduction techniques are separated from the syntax and semantics of the modeling language.

It is possible that a domain may have its own specialized properties to verify and specified model checking algorithms. Our design allows seamless integration of new model checking algorithm and optimization techniques by inheriting base assertion class and implementing its API. Furthermore, supporting functions, like LTL to Büchi, Rabin, Streett automata conversion, are provided in PAT to ease the development of new algorithms. For instance, we have successfully developed the algorithms for divergence checking, timed refinement checking in real-time system module and new deadlock and probabilistic reachability checking. Furthermore, PAT facilitates customized state encoding by defining the interfaces methods in system state class. Different verification algorithms using different state encoding are developed. Currently, PAT supports explicitly state encoding using hash table and symbolic state representation using BDD. The choice of the encoding is made by the users in the user interface at runtime.

4 Performance Evaluation

PAT is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in some cases. Experimental results for LTL verification under fairness and refinement checking are presented in Fig. 2 as an indication of our effort on optimizing the model checking algorithms.

The table on the left shows the verification results on recently developed leader election protocols with different topologies, where the correctness (modeled using LTL formula) of these protocols requires different notions of fairness. Firstly, PAT usually finds counterexamples quickly. Secondly, verification under event-level strong fairness (ESF) is more expensive than verification with no fair, event-level weak fairness (EWF) or strong global fairness (SGF). Lastly, PAT outperforms SPIN for the fairness verifications. SPIN increases the verification time under weak fairness by a factor that is linear in the number of processes. SPIN has no support for strong fairness or SGF. PAT offers comparably better performance on verification under weak fairness and makes it feasible to verify under strong fairness or SGF.

[2] Detailed explanation and usages of the abstract classes are available in PAT's user manual.

Model	Size	EWF			ESF		SGF	
		Res.	PAT	SPIN	Res.	PAT	Res.	PAT
LE_C	5	Yes	4.7	35.7	Yes	4.7	Yes	4.1
LE_C	6	Yes	26.7	229	Yes	26.7	Yes	23.5
LE_C	7	Yes	152	1190	Yes	152	Yes	137
LE_C	8	Yes	726	5720	Yes	739	Yes	673
LE_T	7	Yes	1.4	7.6	Yes	1.4	Yes	1.4
LE_T	9	Yes	10.2	62.3	Yes	10.2	Yes	9.6
LE_T	11	Yes	68.1	440	Yes	68.7	Yes	65.1
LE_T	13	Yes	548	3200	Yes	573	Yes	529
LE_OR	3	No	0.2	0.3	No	0.2	Yes	11.8
LE_OR	5	No	1.3	8.7	No	1.8	-	-
LE_OR	7	No	15.9	95	No	21.3	-	-
LE_R	4	No	0.3	<0.1	No	0.7	Yes	19.5
LE_R	5	No	0.8	<0.1	No	2.7	Yes	299
LE_R	6	No	1.8	0.2	No	4.6	-	-
LE_R	7	No	4.7	0.6	No	9.6	-	-
LE_R	8	No	11.7	1.7	No	28.3	-	-
TC_R	3	Yes	<0.1	<0.1	Yes	<0.1	Yes	<0.1
TC_R	5	No	<0.1	<0.1	No	<0.1	Yes	0.6
TC_R	7	No	0.2	0.1	No	0.2	Yes	13.7
TC_R	9	No	0.4	0.2	No	0.4	Yes	640

Models	N	Property	Result	PAT	FDR
Dining Philosophers	6	P refines S	true	0.86	0.07
Dining Philosophers	8	P refines S	true	13.7	0.07
Dining Philosophers	10	P refines S	true	430	0.11
Reader/Writers	12	P refines S	true	< 1	0.81
Reader/Writers	14	P refines S	true	< 1	6.91
Reader/Writers	16	P refines S	true	< 1	81.2
Reader/Writers	200	P refines S	true	77.5	-
Milner's Cyclic Scheduler	11	P refines S	true	< 1	89.4
Milner's Cyclic Scheduler	12	P refines S	true	< 1	419
Milner's Cyclic Scheduler	13	P refines S	true	< 1	-
Milner's Cyclic Scheduler	200	P [T= S	true	60.4	-
5-valued *register*	2	P refines S	true	44.9	NA
6-valued *register*	2	P refines S	true	297	NA
stack of size 14	2	P refines S	true	99.4	NA
stack of size 2	3	P refines S	true	4321	NA
buggy queue of size 10	2	P refines S	false	6.87	NA
buggy queue of size 20	2	P refines S	false	41.1	NA
mailbox of 3 operations	2	P refines S	true	27.8	NA
mailbox of 4 operations	2	P refines S	true	954	NA
SNZI of size 2	2	P refines S	true	322	NA
SNZI of size 3	3	P refines S	true	6214	NA

Fig. 2. Experiment results on LTL verification under fairness assumption and refinement checking

In addition to temporal logic verification, PAT offers capability of refinement checking (i.e. language inclusion checking). The table on the right shows the performance using benchmark systems as well as newly developed concurrent algorithms. In the classic readers/writers problem, reduction in PAT is very effective so that PAT can handle a few hundreds readers/writers. In the Milner's cyclic scheduling algorithm, multiple processes are scheduled in a cyclic fashion. PAT is effective for this model to handle hundreds of processes. For models with complicated data variables (like scalable non-zero indicator SNZI, see [6] for the details of the examples), PAT is able to show the linearizability of these examples using refinement checking [6]. FDR [10] performs extremely well for Dining Philosophers because of the compression strategy developed for some specialized models. For other examples, PAT is much faster than FDR. Limited by the modeling language, FDR is rather difficult to model distributed systems like Stack, mailbox and SNZI. In addition, PAT supports timed refinement checking, which is beyond existing refinement checkers. In summary, PAT offers a set of well-optimized model checking languages as well as a framework for developing new model checkers.

5 Discussion and Summary

As a temporal logic model checker, PAT is related to the tools like NuSMV [3] and SPIN [5]. Compared to these tools, PAT serves a generic framework for manufacturing model checkers. It complements existing model checkers with specialized algorithms for (timed/untimed) refinement checking [11], verification under fairness constraints (with counter abstraction [16]), etc. PAT has a comparative performance with existing state-of-art tools, and even out-performs them on some cases. Bogor [4] and LTSA [9] are two extensible model checker developed as a plug-in of Eclipse. Bogor allows users to extend the base language to support new language features, but cannot be fully customized with desired syntax and semantic models. LTSA compiles the input language FSP (based on Process Algebra) into LTS, which is similar to PAT. However all the modules in LTSA adopt the translation approach to convert the input model (e.g., Message Sequence Chart and Web Service) into FSP models. Compared with these two, our approach takes one step further to allow the development of fully customized model checkers. Furthermore, the supported libraries in PAT offer user advanced model checking techniques like real-time verification and probabilistic model checking, which are absent in Bogor and LTSA.

Compared to [13], we redesigned the system to separate the GUI, verification algorithms and modeling languages. Each modeling language is encapsulated into a standalone package, which makes the system extensible. Furthermore, we have added the support for real-time and probabilistic systems. The enhancement is dramatic. Starting from 2007, PAT has come to a stable stage with solid testing and various applications. More than 60 built-in examples and hundreds of test cases are embedded in PAT. PAT has been used by a number of institutions as a research or educational tool. The main objective of PAT is to bring sophisticated model checking techniques to a variety of domains. The existing modules and on-going modules under development have shown the usefulness and feasibility of this framework.

References

1. PAT: Process Analysis Toolkit, http://pat.comp.nus.edu.sg/
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
3. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
4. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 148–152. Springer, Heidelberg (2005)
5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Wiley, Chichester (2003)
6. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model Checking Lineariability via Refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
7. Liu, Y., Pang, J., Sun, J., Zhao, J.: Efficient Verification of Population Ring Protocols in PAT. In: TASE 2009, pp. 81–89 (2009)
8. Liu, Y., Sun, J., Dong, J.S.: Scalable Multi-Core Model Checking Fairness Enhanced Systems. In: ICFEM 2009, pp. 426–445 (December 2009)
9. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. Wiley, Chichester (1999)
10. Roscoe, A.W.: Model-checking CSP. In: A Classical Mind: Essays in Honour of C.A.R. Hoare, pp. 353–378 (1994)

11. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In: ISoLA 2008, pp. 307–322. Springer, Heidelberg (2008)
12. Sun, J., Liu, Y., Dong, J.S., Chen, C.Q.: Integrating Specification and Programs for System Modeling and Verification. In: TASE 2009, pp. 127–135. IEEE Computer Society, Los Alamitos (2009)
13. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 702–708. Springer, Heidelberg (2009)
14. Sun, J., Liu, Y., Dong, J.S., Sun, J.: Bounded Model Checking of Compositional Processes. In: TASE 2008, pp. 23–30. IEEE Computer Society, Los Alamitos (2008)
15. Sun, J., Liu, Y., Dong, J.S., Wang, H.H.: Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction. In: ICFEM 2009, pp. 581–600 (December 2009)
16. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair Model Checking of Parameterized Systems. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 123–139. Springer, Heidelberg (2009)

YAGA: Automated Analysis of Quantitative Safety Specifications in Probabilistic B

Ukachukwu Ndukwu* and AK McIver

Department of Computing, Macquarie University, NSW 2109 Sydney, Australia
{ukachukwu.ndukwu,annabelle.mciver}@mq.edu.au

Abstract. Probabilistic B (pB) [2,8] extends classical B [7] to incorporate probabilistic updates together with the specification of quantitative safety properties. As for classical B, probabilistic B formulates safety as inductive invariants which can be checked mechanically relative to the program code. In the case that the invariants cannot be shown to be inductive, classical B uses model checking to allow experimental investigation, returning a counterexample execution trace in the case that the safety condition is violated. In this paper we introduce YAGA which provides similar support for probabilistic B and quantitative safety specifications. YAGA automatically interprets quantitative safety and the pB machine as a model checking problem to investigate the presence of counterexamples. Since inductive invariants characterise a strong form of safety, we are able to identify the specific point at which failure occurs as individual counterexample traces, which can then be ranked for importance, for example according to the probability of occurrence.

Keywords: Probabilistic B, quantitative safety, rewards, failures, diagnostic information.

1 Introduction

Probabilistic B (or pB) [2,8] extends classical B [7] by incorporating probability; like classical B it permits the specification of abstract systems which can be incrementally refined by introducing algorithmic detail. Correctness is characterised by the specification of safety conditions and the refinement relation ensures that those properties are inherited by the refinements. Where probability is an issue, *quantitative* as well as standard safety may be specified. Both types of safety are characterised by inductive invariants which the pB's automated prover is able to check relative to pB's modeling language. It does this by generating and discharging a number of *proof obligations* in much the same way as classical B but using real- rather than Boolean-valued expressions of the program state. More details of pB and its application to probabilistic systems development can be found in [8].

* The author acknowledges support from the Australian Commonwealth Endeavor International Postgraduate Research Scholarship (E-IPRS) Fund.

```
MACHINE Faulty
SEES Int_TYPE, Real_TYPE
VARIABLES cc
INVARIANT cc : INT
EXPECTATIONS real(0) ⇛ cc
INITIALISATION cc := 0
OPERATIONS
    OpX = BEGIN
              PCHOICE frac (1, 2) OF cc := cc + 1
              OR cc := cc − 1 END;
    OpY = cc := 0
END
```

Fig. 1. A Faulty pB Machine

When pB's mechanical prover fails to discharge the proof obligations it generates, the verifier is left with the problem of determining the cause of failure. There are two possibilities to explore. First is that the invariant needs to be strengthened, to make it inductive. Second is that the safety property it characterises does not in fact hold because there exist execution traces of the program model which violate it. Classical B provides diagnostic information by recasting a classical safety property as a model checking problem and using exhaustive search to identify faulty execution traces should they exist [3]. Our contribution in this paper is to do the same for pB and quantitative safety; we have implemented a prototype tool YAGA which provides such diagnostic information using probabilistic model checking.

1.1 A Brief Introduction to pB and Quantitative Safety

Probabilistic systems in pB are specified by a collection of *pB machines* which consist of several operations describing possible program executions, together with a collection of statements which specify the intended behaviour and context. The example of Fig. 1 illustrates key features of the language. There are two operations — *OpX* and *OpY* which can update a variable *cc*. In general operations can only execute if their corresponding preconditions hold, but in this example either operation can execute at any time, although execution of any operation is atomic. If *OpX* executes, for example, variable *cc* is incremented or decremented with probability 1/2 or (1−1/2) respectively, and *OpY* cannot interrupt it. Note that it can happen that the preconditions (if they do exist) of the operations overlap, and in which case the choice of which operation executes is made nondeterministically.

The SEES field ascribes more information to the data type of the variables specified in the VARIABLES field. But in general it is used to define the scope of a machine's visibility [8]. The INVARIANT field specifies invariant properties which are maintained by the operations from the given INITIALISATION. The proof obligations generated by pB's prover are designed to check that the operations comply with the invariants.

Here we concentrate on the EXPECTATIONS clause, which was introduced by Hoang [8] as a way to express quantitative invariant properties. The form of an EXPECTATIONS clause is highlighted in Fig. 1; on the left hand side of the inequality is a threshold of 0, on the right hand side is an expression of the program variables, in this case cc. This specifies a safety property that the expected value of cc as a random variable should *never* fall below the threshold 0. Here the expected value is relative to the distribution over program states determined by any (interleaved) execution of the operations OpX and OpY. In pB the proof obligations for the EXPECTATIONS clause are generated for each operation and the initialisation in the form of probabilistic Hoare-style triples, in this case:

$$\{0\} \;\; cc := 0 \;\; \{cc\}, \qquad \{cc\} \;\; Op \;\; \{cc\} \; . \tag{1}$$

The first triple represents an obligation for the initialisation: in this case that the initialisation of the variable cc should ensure that the value of cc is at least the threshold 0. The second triple captures the notion that the expression cc must be an inductive invariant relative to each operation. For probabilistic operations this means that given any state s from which the operation Op may be executed, the expected value of cc treated as a random variable must be at least the value of cc from s. For example if Op is taken to be OpX in Fig. 1, then from any initial value of the variable cc (for example when $cc = 0$), it generates a distribution over final states — $cc = -1$ with probability 1/2 and $cc = 1$ with probability 1/2. This means that the expected value of the random variable cc is $1/2\times(-1) + 1/2\times 1 = 0$, and this shows that the expected value of cc has not decreased (since it was 0 at the initial state $cc = 0$). In this sense cc is inductively invariant for OpX after this single transition, and pB can discharge OpX's proof obligation.

However the proof obligation for OpY cannot be discharged, suggesting that there is a problem with this specification. We shall return to this example in Sec. 3 to show how YAGA locates a particular execution sequence of OpX and OpY, providing diagnostic information to illustrate the circumstances under which the expected value of cc can fall below the threshold 0. Since inductive invariance is a strong form of safety, we have shown elsewhere [5,9] that failure corresponds exactly to when there is an "execution trace" which can occur with non-zero probability (from the initial state) to a state s such that the probabilistic triple with respect to a particular operation does not hold from s. In the example of Fig. 1, a counterexample is a sequence of (operation, state) pairs where the final operation in the list fails the right-hand triple in Eqn. 1. Note that other definitions of counterexample exist for various forms of probabilistic temporal properties, but because they are not based on an inductive interpretation, a set of paths is required [1].

A summary of YAGA's capabilities is as follows:

(a) YAGA first automatically translates the pB model to the PRISM [10] modeling language, and represents the EXPECTATIONS clause as a PRISM reward structure [4]. PRISM is a probabilistic model checker which is able to analyse quantitative safety with real-valued rewards, and as such provide us with a critical pre-processing stage whereby the quantitative safety property can be investigated experimentally. Note that PRISM is not able to identify individual counterexample paths, however it does produce important input to the counterexamples generator module discussed below.

(b) If faulty behaviour exists, YAGA returns a most useful diagnostic information as an (operation, state) pair listing where the final operation breaks the inductive invariance from its execution point.

(c) Finally, YAGA enables a verifier to "drill-down" to finer details of the faulty traces by ranking them with respect to either their total probability masses or their total expected reward.

The rest of this paper is structured as follows: Sec. 2 is an overview of the architecture of YAGA, and a discussion of its core modules; Sec. 3 summarises the effect of YAGA on the example of Fig. 1; and finally we conclude in Sec. 4.

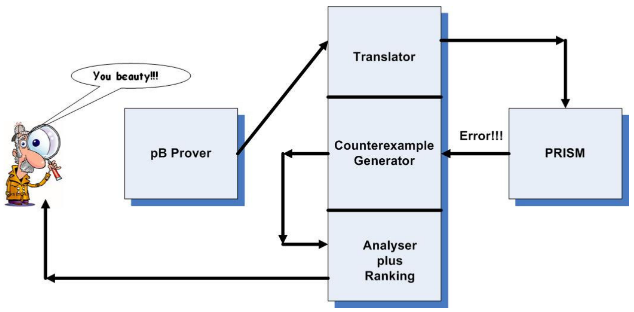


Fig. 2. YAGA - Architectural Process Flow

2 YAGA: Architectural Overview

In this section we summarise YAGA and its core modules as shown in Fig.2. YAGA[1] is a Java-based implementation that inputs a pB machine, and generates its representation in the PRISM modeling language. The underlying feature of YAGA is its reward structure interpretation of the quantitative safety specifications of the pB machine framework. Details of this interpretation can be found in [5,9].

The PRISM temporal logic specification or PCTL [4] formulas over the reward structure can then be checked by conducting experiments on the transformed

[1] The acronym YAGA, coined from an Igbo (a language largely spoken in southeast Nigeria) word — YAGAzie, which literally means "may it go well ...", is a designer's prayer to positively summarise the fear of impending faults within a pB machine design. However YAGA could be visualized as Yet Another Gangling Automation.

model. The experimental results are sufficient to validate (or refute) the quantitative safety property specified within the pB specification. In the event that a faulty behaviour (or counterexample) violating the property of interest is captured, the verifier is alerted to the results of a k-shortest path analysis identifying finite traces leading from the root of the resultant failure tree to a faulty state (*i.e.* where the invariant is found not to be inductive).

Finally, YAGA implements a rank engine which supplies a verifier with more specific information about the total probabilities or expectations corresponding to the faulty execution traces. Below we explain the modules in more detail.

2.1 Translator Module

The translator module provides an exact interpretation of a pB machine in the PRISM modeling language. A key strategy is to ensure that the pB machine clauses of interest are given a Markov decision process (MDP) semantic equivalence in PRISM. The resultant PRISM model is then composed of the following:

- Machine module: Encapsulates the following features of the resultant PRISM model: constants, formula list, module name, variables and update statements from the abstract machine's framework. In addition, its update statements are synchronised with similar update statements constructed from a counter module (see below).
- Counter module: This is a special module that enables a pB verifier to conduct experiments over resulting finite probability distributions of the execution traces generated by a pB machine, and corresponding to its specific *kth* execution. Apart from its own local variables, the effect of synchronising its update statements with that of the machine module is to record (in finite steps) any impending faulty behaviour within the machine, where a MAX_COUNT parameter provides details for probing the depth of the tree.
- Reward structure: The associated reward structure captures the exact interpretation of the safety specification embedded within the pB machine EXPECTATIONS clause [5].

Given this model representation (see Fig.3 for example), the verifier is able to investigate the safety specification by conducting preliminary experiments. The experimental results can reveal succinct information about impending faulty behaviours in the pB machine (as in Fig.4). On discovery of a faulty behavioural execution, the next two modules provide an exact analysis sufficient to report a most useful diagnostic information to the pB machine verifier.

2.2 Counterexamples Generator Module

After it inputs a MDP representing a faulty pB machine and constructs its equivalent reachability reward, the rest of the functionality of the counterexample generator module could further be divided into two:

- Model checking the resultant MDP, it then ouputs an *adv.tra* file which contains the optimal scheduling strategy of the MDP violating a given PCTL formula. YAGA loads the *adv.tra* file and constructs a transition probability matrix representation of the optimal scheduling strategy identifying the schedule which exhibits the faulty behaviour [9]. Practical details of generating this strategy are in [10].
- A key analysis phase of the generator involves the automatic construction of an additional *state-value.txt* file from the model's state space. The *state-value.txt* file marks every state of the matrix with (i) the valuation of the program's variables occurring in the reward structure, and (ii) a corresponding action that is enabled therein. Finally, YAGA analyzes the optimal scheduling strategy via a k-shortest path technique, using the *state-value.txt* file, to generate at least a single finite trace as a sequence of actions and their corresponding state valuations leading from the initial state to a state where the property is violated.

2.3 Ranking Module

As a utility module, the rank engine enables a verifier "drill-down" to finer details of the faulty traces by querying YAGA for information relating to: the total probability masses (from a maximal to a minimal order), and the total expected values of the rewards (from a minimal to a maximal order) of the traces themselves. Traces with high probability masses are possibly more useful for debugging compared with traces with low probability masses. Traces whose expected values are minimally deviated from the lower bound of the expected value of the reward properties would constitute more useful diagnostic information than traces that are maximally deviated from the bound.

3 Experimental Results

Given the PRISM model of Fig.3, we present experimental results that summarize the pB machine of Fig.1. We also interpret the results directly in relation to the faulty pB machine.

3.1 Preliminary Experiments

The result of the experimental investigation of the EXPECTATIONS clause of the faulty pB machine is shown in Fig. 4. Since by the safety specification of Fig. 1, if the machine complies with the statement in the EXPECTATIONS clause, then the expected value of *cc* is always at least zero. However, Fig. 4 reveals otherwise. It reports a decreasing value in the expected value of *cc* right from the second execution step. This behaviour continues as long as the machine executes. In the sections that follow, we present a detailed analysis of this faulty behaviour as well the execution traces corresponding to it.

```
const $MIN$;
const $MAX$;
const $MAX\_COUNT$;

module Faulty
$cc : [MIN..MAX]$   init   0;

[OpX] $(MIN < cc < MAX)$ $\rightarrow 0.5 : (cc' = cc + 1) + 0.5 : (cc' = cc - 1)$;
[OpY] true $\rightarrow (cc' = 0)$;
endmodule

module Counter
$count : [0..MAX\_COUNT + 1]$   init   0;
$terminate$ : bool init false;
$action : [0..3]$   init   0;

[OpX] $(count + 1 \leq MAX\_COUNT + 1)$ $\rightarrow (count' = count + 1)$ & $(action' = 1)$;
[OpY] $(count + 1 \leq MAX\_COUNT + 1)$ $\rightarrow (count' = count + 1)$ & $(action' = 2)$;
[] $(count + 1 \leq MAX\_COUNT + 1)$ $\rightarrow (count' = count + 1)$ & $(action' = 3)$;
[T] $(count = MAX\_COUNT + 1)$ $\rightarrow (terminate' =$ true);
endmodule

rewards
[T] $(count = MAX\_COUNT + 1)$ : $cc + MAX\_COUNT$;
endrewards
```

Fig. 3. A PRISM representation of the faulty pB model specified in Fig.1

3.2 A Most Informative Diagnostic Trace Located

As a result of the counterexamples generator module's analysis of the resultant *adv.tra* file, it reports a single trace (shown below) responsible for the property violation. The summary of the report is that, after the machine's initialisation `INIT`, a single machine sequence of the

```
************ Starting Error Reporting  ... ***************
       Faulty path located after 2 step(s)
    Sequence of operations and state valuations ::>>>
        [{INIT} (0), {OPX} (1), {OPY} (0)]
        Probability mass of path is:>>>> 0.5
************ Finished Error Reporting  ... ***************
```

operations `OPX` and `OPY` respectively would result in the machine's faulty behaviour. Note that the value (cc) denotes the state valuation of the random variable *cc* at a state where each of the operations is enabled. This gives the verifier a snapshot of the first faulty execution history. We see that after `OpX`, if *cc* is 1 then `OpY` may be executed to set *cc* to 0, and this is exactly a failure of the triple at Eqn. 1, since at $cc = 1$ the value of the expression *cc* is 1, and the expected value of *cc* after executing `OpY` from there is 0 ($= 1 \times 0$). This trace can occur with probability 0.5. Note that in general there can be several traces where inductive invariance can fail.

3.3 Diagnostic Trace "Expectations" Information

Since YAGA has only reported a single faulty trace whose probability mass is given above, the ranking module also reports a single expectations information as shown below.

```
******** Starting Ranking Information By "expectations" ******
            Path plus expectations Information::>>>
             [{INIT} (0), {OPX} (1), {OPY} (0)]
            Path residual expectations is:>>>> -1.0
************ Finished Ranking Information  ... **************
```

The interpretation is that after execution of the trace, the expected value of *cc* is strictly decreased below the specified threshold of zero, hence violating the statement in the EXPECTATIONS clause. This intuition is enough to guide the machine designer with improving the correctness of his design. Details of the underlying theory of YAGA, as well as the algorithmic interpretation of its core modules *i.e.* the translator, counterexamples generator and rank modules are clearly set out at [9].

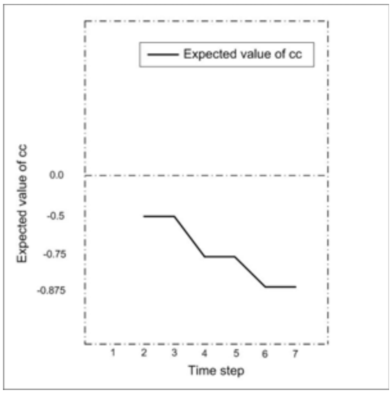


Fig. 4. Experiment over the EXPECTATIONS clause of Fig. 1

This graph was generated in the PRISM pre-processing stage, and displays the expected value of the expression *cc* treated as a random variable under operations of the pB machine. It shows for example that the quantitative safety property is first violated on the second step, and that the longer the machine can execute the more negative becomes that expected value.

4 Conclusion and Future Work

So far, we have been able to demonstrate a range of techniques that enable a pB machine designer explore experimentally, the consequence of the failure of a prover to discharge proof obligations arising from the specification of safety properties within the EXPECTATIONS clause of a pB machine framework. Elsewhere [9], we have used YAGA to analyse a probabilistic library machine, as well as the mincut algorithm [6] whose pB machine constructions were originally provided by Hoang [8].

Ultimately, the big picture for us is to develop YAGA into a complete and independent plug-in to be deployed as part of the successor to event-B, the Rodin toolkit [11].

References

1. Han, T., Katoen, J.-P., Damman, B.: Counterexample Generation in Probabilistic Model Checking. IEEE Trans. Software Eng. 35(2), 241–257 (2009)
2. Hoang, T.S., Jin, Z., Robinson, K., McIver, A.K., Morgan, C.C.: Probabilistic Invariants for Probabilistic Machines. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 240–259. Springer, Heidelberg (2003)
3. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
4. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
5. Ndukwu, U.: Quantitative Safety: Linking Proof-based Verification with Model Checking for Probabilistic Systems. In: Proceedings of First International Workshop on Quantitative Formal Methods (QFM 2009), Eindhoven, Netherlands (2009)
6. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
7. Abrial, J.-R.: The B-Book: Assigning Programs to Meaning. Cambridge University Press, Cambridge (1996)
8. Hoang, T.S.: Developing a Probabilistic B-Method and a Supporting Toolkit. PhD Thesis, University of New South Wales, Australia (2005)
9. Ndukwu, U.: Generating Counterexamples for Quantitative Safety Specifications in Probabilistic B. Submitted to the Journal of Logic and Algebraic Programming, JLAP (May 2010), `http://web.science.mq.edu.au/~ukndukwu/counterexamples.pdf`
10. PRISM: Probabilistic Symbolic Model Checker, `http://www.prismmodelchecker.org/`
11. Deploy: `http://www.deploy-project.eu/`

COMBINE: A Tool on Combined Formal Methods for Bindingly Verification

An N. Nguyen, Tho T. Quan, Phung H. Nguyen, and Thang H. Bui

Faculty of Computer Science and Engineering,
Hochiminh City University of Technology, Vietnam
annguyen2210@gmail.com,
{qttho,phung,thang}@cse.hcmut.edu.vn

Abstract. Theorem proving and model checking are two well-known formal methods emerging recently for software verification. Each of them has its own advantages and disadvantages. As an attempt to combine the verification capabilities of these two methods, in this paper we introduce a verification tool known as COMBINE (Combined fOrmal Methods for BINdingly vErification). Suggested by its name, COMBINE can verify imperative programs in a bindingly manner comprising of two phases: static verification and dynamic verification. In fact, COMBINE has been developed as a published Web-based system currently being used for teaching programming for students at Hochiminh City University of Technology (HCMUT), Vietnam.

1 Introduction

A major goal in software engineering today is to enable software systems to be developed in a reliable manner regardless their complexity. Formal methods have been emerging as a potential approach to fulfill this desire [1]. In the field of software verification, there are two popular approaches nowadays having been attracting much attention, which are *theorem proving* and *model checking*.

Theorem proving, or *automated theorem proving* (ATP) [2] is a technique that is based on Hoare logic to prove the correctness of a program by means of axiomatically processing. However, due to the theoretical complexity of logic-based reasoning, those provers suffer major obstacles when dealing with loop-based programs as they need to infer the *invariants* of the loops, which is a tough task currently far from being completely solved [3]. The other approach, model checking [4], is an automatic verification technique for finite state concurrent systems such as sequential circuit designs and communication protocols. The major advantage of model checking is its capability of producing traceable counter examples when detecting errors. However, the current model checkers suffer the state space explosion problem. In addition, the generated counter examples are not familiar and convenient for popular users.

In this paper, we introduce a verification tool known as COMBINE (Combined fOrmal Methods for BINdingly vErification) as an attempt to combine those two approaches of theorem proving and model checking for program verification in two phases namely *static verification* and *dynamic verification*. The rest of the paper is organized as follows. Section 2 gives the overall architecture of COMBINE. Section 3 presents examples on how COMBINE works. Section 4 summarizes the current status of COMBINE. Section 5 presents some related works and Section 6 draws some conclusions for the paper.

2 COMBINE Architecture

As presented in Figure 1, COMBINE comprises of two phases: static verification and dynamic verification, which respectively deploy theorem provers and model checkers for verification. When static verification cannot affirm the correctness of an input program, dynamic checker is then invoked to check the program again in a dynamic manner. Counter-examples can be generated in both phases if necessary when an error is detected.

2.1 Static Verification

Static verification module performs three main tasks, namely *Static Analysis*, *Correctness Proving* and *Counter-Example Generation*. Static Analysis will translate the original source code with formal specification to axiomatic description. Formal specification is written in a dedicated language such as ANSI/ISO C Specification Language (ACSL) [5] which is a behavioral specification language for C program. Axiomatic description is performed in C Intermediate Language (CIL) [6] . Then it is translated to appropriate verification conditions which are used later for Correctness Proving. In order to be compatible with multiple provers employed in the Correctness Proving task, these conditions are stored in an Abstract Syntax Tree (AST) and each prover will parse them to its own supported language if necessary. Based on verification conditions, provers will verify and give the result to programmer. If an error is detected, Counter-Example Generation will generate some helpful counter-examples as a predicate.

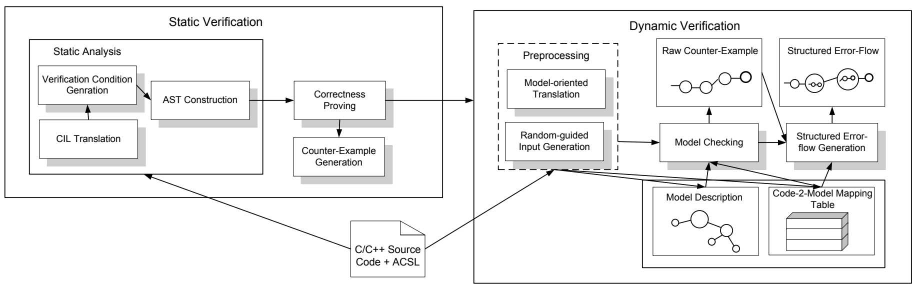


Fig. 1. COMBINE architecture

```
// Description: Find the absolute value of an integer.
# pragma JessieIntegerModel(math)
/*@ ensures (i >= 0 && \result == i) || (i < 0 && \result == -i);*/
int AbsNumber(int i) {
   if (i < 0) return -i;
   else return i;
}
```

(a) Correct solution of a given problem

```
  // Description: Find the max number between two integers.
  # pragma JessieIntegerModel(math)
  /*@ ensures \result >= x && \result >= y && \forall int z; z >= x
&& z >= y ==> z >= \result;
  */
  int MaxNumber(int x, int y) {
     if (x > y) return y;
     else return y;}
```

(b) Wrong solution of a given problem

```
// Description: In the given array, return the index of the 1st
element whose value equals v
/*@
requires \valid_range (t, 0, n-1);
ensures
   (0 <= \result < n ==> t[\result] == v) &&
   (\result == n ==> \forall int i; 0 <= i < n ==> t[i] != v) ; */
int SearchArray(int t[], int n, int v) {
   int j = 0;
  /*@ loop invariant 0<=j && \forall integer k; 0<=k<j ==> t[k] != v;
     @ loop variant n – j > 0; */
   while (j < n) {
      if (t[j] == v) break;
      j = j+2;
   }
   return j;
}
```

(c) Loop-based program with an unproved solution

Listing 1. C codes augmented with formal specifications

2.2 Dynamic Verification

After the phase of static verification, dynamic verification will be invoked when necessary. As presented in Figure 1, dynamic verification consists of three major tasks including *Preprocessing*, *Model Checking*, *Structured Error-Flow Generation*. *Preprocessing* consists of two subtasks, namely *Model-oriented Translation*, which basically translates the original source code into a formalism of a model checker description; and *Random-Guided Input Generation*, which generates a reduced input space meaningfully in a random–guided manner. In addition, this step will also generate a mapping table between elements of the targeted model checker with those of the original program. This mapping table, known as *Code-2-Model Mapping Table*, will be then used later to retrieve the error-flow in case an error is detected. Meanwhile, *Model Checking* performs the normal tasks of a model checker: verifying the model against a property, and producing a model-based counter example, known as *raw counter-example*, when recognizing some erroneous problems. Finally, based on the raw counter-example and the Code-2-Model Mapping Table, *Structured Error-Flow Generation* generates a structured error-flow represented in the original language of the verified program.

3 Examples

3.1 Axiomatic Problem Description

Intended input problem of COMBINE is an imperative program augmented by some axiomatic description. In Listing 1 there are 3 C programs to be verified with formal specifications annotated in their comments. In Listing 1(a), we define the problem of finding the absolute value of a given integer with the correct solution submitted to COMBINE. In Listing 1(b), the program submitted to COMBINE is a solution of finding the maximal number between two given integers, which is logically wrong as the programmer accidentally made a mistake in the last statement. For loop-based program, ACSL annotations are more complicated. For instance, the program in Listing 1(c) aims at finding an identified element in the given array. The requires-clause of this program implies that the lower bound and upper bound of the array index must be 0 and n-1 respectively. The ensures-clause is also used to specify the expected returned value of program. Moreover, we define *invariants* right before each loop iteration. There are two invariants given. The first invariant states that when the loop still executes, the searched value does not occur among the processed array elements. The second invariant just simply tells that the counter variable will never exceed the array size. Those invariants are essentially required by all provers to verify the loops. In this example, the loop body has an easily observed mistake.

3.2 Static Verification Phase: Correctness Proving and Counter-Example Generation

After affirming that the input programs have no syntax error, COMBINE will then perform static analysis and transfer these programs to the provers to prove if they satisfy the pre-defined requirements or not. Based on the static analysis results, the provers produce verification conditions accordingly. For example, with the program given in Listing 1(a), there are 2 verification conditions generated, corresponding to 2 cases of $i >= 0$ and $i < 0$. When verified, all conditions are passed by the provers integrated in COMBINE and hence the correctness of the program is confirmed[1]. In Listing 1(b), the source code yields six conditions to be verified, among them one is failed when verified. The failure case will be then moved to Counter-Example Generation step to be further processed. The counter-example is generated in a form of a Lisp-like predicate as `(AND (EQ return y) (< return x))`.

[1] For the sake of combined proving capability, there are many provers concurrently adopted in COMBINE, but only one prover confirming the program correctness is sufficient.

By means of axiomatic processing, a program correctness can be easily proven in a formal manner if its pre-conditions, post-conditions and invariants are well-written. However, in case of incorrectness, things become far complicated, especially for loop-based programs. For example, in Listing 2(c), all of our tested provers did not recognize the mistake in the last increasing statement in the loop body.

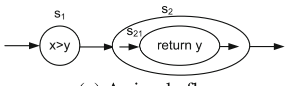
3.3 Dynamic Verification Phase: Model Checking and Structured Error-Flow Generation

In dynamic verification, we firstly translate the original source code into a form of a model description language of a model checker. Then, we have the model checker verify the translated model. Especially, in COMBINE we developed some tactics of random-guided generation of input values. In order to perform random-guided basis our strategy is to try to divide the input space into subdomains based on some heuristic rules some of which are illustrated in Table 1 [7]. For example, Rule 1 states that when encountering a statement of $x > a$, the system will generate three values of x for testing as: {a+ϵ, RANDOM, MAX} where ϵ indicating a very small value, MAX the possible maximal value of x and RANDOM a random value between a+ϵ and MAX. While the first five rules give examples of strategies to generate the testing input, the last three ones gives examples of generating subdomains. For example, in Rule 7, the corresponding subdomain (M_i) is generated by taking into account the condition E and other subdomains generated from other if-else clauses previously declared.

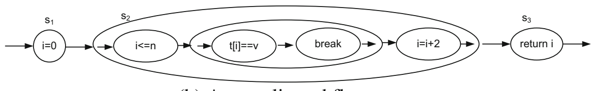
Table 1. Heuristic rules for guided-random input generation

Rule	Statement	Subdomain	Suggested Sample
Rule 1	$x>a$	$(a, +\infty)$	a+ϵ, RANDOM, MAX
Rule 2	$x>=a$	$[a, +\infty)$	a, RANDOM, MAX
Rule 3	$x<a$	$(-\infty, a)$	MIN, RANDOM, a-ϵ
Rule 4	$x<=a$	$(-\infty, a]$	MIN, RANDOM, a
Rule 5	x==a	{a}	a
Rule 6	if E …	M_1=Subset(E)	
Rule 7	… elseif E	M_i=Subset(E)$\cap(\neg M_{i-1})\cap \ldots \cap(\neg M_1)$	
Rule 8	… else E	$M_i=(\neg M_{i-1})\cap(\neg M_{i-2})\cap\ldots\cap(\neg M_1)$	

Remarkably, besides the counter-examples generated in static verification, dynamic verification will generate a higher conceptual level of error representation, known as *structured error-flow* [8]. Basically, a structured error-flow is a nested graph-based structure reflecting the execution path performed by a program with a certain input. In order to generate an error-flow, first we analyze the raw counter-example returned by the employed model checker when recognizing a system error. This raw counter-example just simply gives us all of ordered internal states visited the model checker to achieve the error. Using the Code-2-Table mapping generated when translating the original program into model description, we are able to retrieve the ordered path of the original statements that cause the error. Based on the structural analysis of the retrieved statements, we produce the corresponding final structured error-flow. For example, in Figure 2(a) is the structured error-flow generated when testing the program in Listing 1(b) with the input of x and y being 1 and -1 respectively. With these given input, the condition of if-statement is verified and then the return statement is executed. Note that in Figure 2(a), node s_{21} representing the return statement is a subnode inside node s_2, corresponding to the structural analysis stating that this return statement is inside a body of an if-clause. Information in this error-flow can be presented to user in a meaningful manner as depicted in Figure 3.



(a) A simple flow



(b) A complicated flow

Fig. 2. Graph-based representations of structured error-flows

```
Input:  x = 1; y = -1
Obtained Result: = -1
Desired Result: = 1
Line 3: If statement
        Line 3: [x > y] = true
        Line 4: return y;
```

Fig. 3. Error-flow generated for the program given in Listing 1(b)

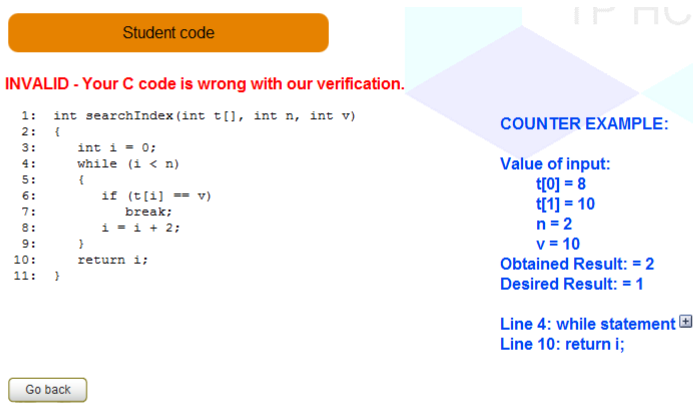
4 Status

The introduced COMBINE tool is being implemented as a Web-based system at Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam[2]. In the implemented system, we have adopted numerous provers for static verification, including Z3 [9], Simplify [10] and Alt-Ergo [11]; in which Simplify is particularly in charge of generating counter-examples. The provers are combined using Frama-C platform [12] with Jessie plugin [13] for static verification. Meanwhile, Spin [14], a well-known model checker, is adopted for dynamic verification. We have handled all programs involving if-clause and arithmetic types, including integer and floating-point. Pointer, loop-statement and array are also considered.

The system is intended to improve education performance in the course of Programming Methodologies as it can help student to verify their own solutions for the given programming exercises. Figure 4 illustrates a screenshot of the system generating error-flow for user. In this case, the generated error-flow is quite complicated, as given in Figure 2(b). In Figure 4, there is a '+' symbol appearing at Line 4 notification, which is corresponding to node s_2 in Figure 2(b). When clicking on this symbol, users can continue exploring the statements executions corresponding to the nested subnodes of s_2.

When using this tool to verify students' works, we categorized the verified problems into four types as indicated in Table 2. Type 1 includes the problems involving integer type, linear arithmetic and some simple functions on array of

[2] One can visit this system as http://www.cse.hcmut.edu.vn/provegroup/prove



INVALID - Your C code is wrong with our verification.

```
 1:  int searchIndex(int t[], int n, int v)
 2:  {
 3:     int i = 0;
 4:     while (i < n)
 5:     {
 6:        if (t[i] == v)
 7:           break;
 8:        i = i + 2;
 9:     }
10:     return i;
11:  }
```

COUNTER EXAMPLE:

Value of input:
t[0] = 8
t[1] = 10
n = 2
v = 10
Obtained Result: = 2
Desired Result: = 1

Line 4: while statement
Line 10: return i;

Fig. 4. COMBINE is being employed as a Web-based system

Table 2. Verification results on students work (√ indicates successful verification, whereas ∗ failed cases)

Problems	Static	Dynamic	Combination
Type 1	√	√	√
Type 2	∗	√	√
Type 3	√	∗	√
Type 4	∗	∗	∗

integers. Both static and dynamic verifications worked well in this case. However, static verification was not able to prove the functions in Type 2 including non-linear arithmetic and loop-based programs, which dynamic verification was able to handle. Type 3 contains some simple functions involving floating-point number or recursive algorithm. Static verfication performed well in this case, but dynamic verifcation could not solve them. Our system still experiences problems with functions of Type 4 which involves pointer or complex structured types.

5 Related Works

There are many theorem provers introduced specially for software verification, notably including Z3 [9], Simplify [10], Coq [15], Isabelle [16], Alt-Ergo [11] and Redlog [17]. In particular, some systems, such as Frama-C [12], Why [18] and HIP [19] have been developed to deploy multi-provers in order to make full use of their combined proving capabilities. Our tool is an extension of the emerging Frama-C platform which not only combines multi-provers in our systems but also employ model checking for dynamic verification.

Spin is adopted in our system for dynamic verification. Apart from this model checker, another emerging model checkers include Java Pathfinder (JPF) [20], NuSMV [21] and PAT [22]. Especially, in BLAST [23], model checking method is also used for testing C program to reduce the input space, BLAST relies on *lazy* *predicate abstraction* technique, which means that false alarm may be alerted due to the abstraction. Moreover, counter-examples generated in BLAST are still more or less of state-based formalism, which may cause difficulty for ordinary programmers to follows. COMBINE is not hindered by those limitations.

Our approach on combining static analysis and dynamic analysis is similar to Flanagan's hybrid type checking [24] where both static and dynamic checking are combined and are decided by the system to be executed resepectively when necessary. However, Flanagan's system mainly aim at resolving type checking problem whereas our COMBINE tool is meant to generally verify if a program fulfills a properties semantically represented as logic predicates.

It is also noted that our approach on random-guided input generation is similar to that on the well-known technique of QuickCheck [25]. However, when developing heuristic rules, we make possibility to embed logic in higher levels into program flow analysis, through which the semantic of program functions and modules can be captured and reasoned to generate more meaningful testsuits, in an automatic manner.

6 Conclusion

This paper introduces a tool known as COMBINE which combines two popular formal methods, theorem proving and model checking, for verifying imperative programs in a bindingly manner. We have tried to make full use of the advantages and partially overcome the disadvantages of the adopted methods. Our COMBINE tool has been developed as a Web-based system used to teach programming to students which gains some promising initial results.

Acknowledgment

This work is part of the Higher Education Project 2 project (supported by World Bank and Hochiminh – Vietnam National University). We also would like to express our appreciation to Anh The Do, Linh Viet Huynh, Nam Thanh Doan, Thien Duc Nguyen and Duc Nghia Le Hoang for their contributions to this work. Our gratitude also goes to anonymous reviewers for their constructive comments that significantly help to improve the paper quality.

References

[1] Clarke, E.M., Wing, J.M., et al.: Formal methods: State of the art and future directions. ACM Survey 28, 626–643 (1996)

[2] Duffy, D.A.: Principles of Automated Theorem Proving. John Wiley & Sons, Chichester (1991)

[3] Flanagan, C., Qadeer, S.: Predicate Abstraction for Software Verification. In: Conference Record for of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 191–202 (2002)

[4] Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. Logic of Programs, 52–71 (1981)

[5] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, preliminary design (version 1.4) (October 2008)

[6] Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: International Conference on Compiler Construction, pp. 213–228 (2002)
[7] Quan, T.T., Hoang, D.L.N., Nguyen, B.T., Nguyen, A.N., Tran, Q.D., Nguyen, P.H., Bui, T.H., et al.: MAFSE: A model-based framework for software verification. In: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement, Singapore (2010)
[8] Quan, T.T., Hoang, D.L.N., Nguyen, V.H., Nguyen, P.H.: Model–based Generation of Structured Error–Flows in Imperative Programs. In: Proceedings of International Conference on Advanced Computing and Applications (ACOMP 2010), Vietnam (2010)
[9] Z3: An Efficient SMT Solver, `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
[10] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A Theorem Prover for Program Checking, Tehcnical report, `http://www.hpl.hp.com/techreports/2003/HPL-2003-148.html`
[11] The Alt-Ergo Theorem Prover, `http://alt-ergo.lri.fr/`
[12] Frama-C, `http://frama-c.com/`
[13] Jessie plugin, `http://frama-c.cea.fr/jessie.html`
[14] Spin – Formal Verification, `http://spinroot.com/`
[15] The Coq Proof Assistant, `http://www.lix.polytechnique.fr/coq/`
[16] Isabelle, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
[17] Redlog, `http://redlog.dolzmann.de/`
[18] Why: a software verification platform, `http://why.lri.fr/`
[19] HIP Overview, `http://loris-7.ddns.comp.nus.edu.sg/~project/hip/index.html.`
[20] Java PathFinder, `http://babelfish.arc.nasa.gov/trac/jpf`
[21] NuSMV – a new symbolic model checker, `http://nusmv.fbk.eu/`
[22] Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
[23] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002: Principles of Programming Languages, pp. 58–70 (2002)
[24] Flanagan, C.: Hybrid type checking. In: POPL 2006: Principles of Programming Languages, pp. 245–256 (2006)
[25] Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming, pp. 268–279 (2000)

Rbminer: A Tool for Discovering Petri Nets from Transition Systems

Marc Solé[1] and Josep Carmona[2]

[1] Computer Architecture Department, UPC
msole@ac.upc.edu
[2] Software Department, UPC
jcarmona@lsi.upc.edu

Abstract. The theory of regions was introduced in the nineties to enable the transformation of an automata into a Petri net. From very restricting initial requirements, the theory has evolved in several dimensions in the last two decades, widening the scope of application to more general scenarios. In contrast, few tools have appeared to support these new theories, thus relegating the potential of the area only to the academic domain. This paper introduces `rbminer`, a tool that combines the theory of regions with linear algebra to compute a basis of state regions. Due to its light space requirements, this approach may contribute to bridge the gap between the theory of regions and its industrial application.

1 Introduction

In this paper we present `rbminer`, a tool that transforms a state-based representation of a system, a transition system (TS) [1], into an event-based model such as a Petri net (PN) [2]. This tool is based on an extension of the *theory of regions*, which was introduced by Ehrenfeucht and Rozenberg in the seminal paper [3]. The underlying theory of the paper can be found in a recent publication [4].

The derivation of a PN from a TS has many applications, ranging from hardware synthesis [5] to Process mining [6]. In Process mining, the problem is to derive a PN from a set of *logs* (traces of a real system). These logs can be easily converted into a TS where `rbminer` can be applied. Another application is visualization: large TSs exhibiting a high degree of concurrency may be succinctly summarized in the form of a PN.

Related work [7,8] includes approaches based on regions of languages [9], for which some comparison can be found in [4], and tools based on the same underlying region theory such as `petrify` [5] and `genet` [10]. The former can only be used for the synthesis of safe PNs, while the latter can both *synthesize* (bisimilarity is guaranteed) and *mine* (only language inclusion is guaranteed) k-bounded PNs.

2 Overview of the Theory Behind `rbminer`

Intuitively, a region is a set of states in a transition system that has a homogeneous relation with respect to the events, *i.e.* every event either enters this set,

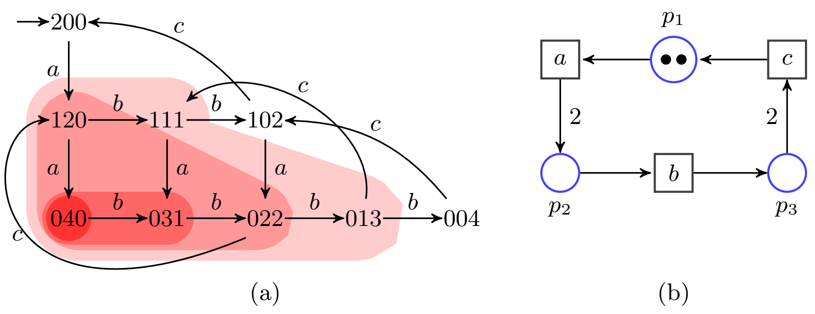


Fig. 1. (a) A TS (b) Mined PN

or exits it, or never crosses its boundary. A region corresponds to a place in the derived Petri net. In `rbminer`, the notion of *general region* is used, where the corresponding place can have up to k tokens.

Let us use the example of Fig. 1 to illustrate the theory. The method assumes that a k is initially given for the search of a k-bounded Petri net. The basic idea is that regions are represented by multisets (*i.e.* a state might have multiplicity greater than one). Fig. 1(a) depicts a TS with 9 states and 3 events. After synthesis, the Petri net in (b) is obtained. For illustration purposes, we label each state with a 3-digit label that corresponds to the marking of places p_1, p_2 and p_3 in the reachability graph of the PN. The shadowed states represent the general region that characterizes place p_2. Each red tone represents a different multiplicity of the state (4 for the darkest and 1 for the lightest). Each event has a *constant gradient* with respect to this region (+2 for a, -1 for b and 0 for c). The gradient indicates how the event changes the multiplicity of the state after firing, and is usually denoted as a vector, *e.g.* (2,-1,0) for the region of place p_2. Gradients are used to weight the arcs between places and transitions in the derived PN: the gradient +2 for event a in the region shown corresponds to the arc with weight 2 between the transition a and the place p_2, meaning that every time the transition a fires, it puts two tokens into p_2.

What differentiates `rbminer` from other tools based on the same theory is the use of a state region basis: any region can be expressed as a linear combination of a small subset of regions[1], called *basis*. Formally, any region r can be expressed as $\sum_i c_i \cdot r_i$, where r_i are regions in the basis and $c_i \in \mathbb{Z}$. Our tool computes such basis, and then derives new regions by producing combinations of the regions in the basis. The algorithm for exploring the region space via combination of regions in the basis is a crucial element in our tool. Although this algorithm is not described in this paper, its basic parameters are described in the next section. In the example of Fig. 1 the basis has only two regions, with gradients $r_1 = (-2, 1, 0)$ and $r_2 = (-1, 0, 1)$, which can produce, by linear combination, the gradients of the three places in the net, *e.g.* $p_1 = r_2$, $p_2 = -r_1$ and $p_3 = r_1 - 2r_2$. Refer to [4] for further details on the theory.

[1] The number of regions in the basis is bounded by the number of events [4].

$$\begin{tikzcd}
\text{Log} \arrow[drr] & \text{FSM} \arrow[dr] & \text{State graph} \arrow[d] & \text{CSP} \arrow[dl] & \text{CCS} \arrow[dll] & \mathsf{PN} \arrow[dlll] \\
 & & \mathsf{TS} \arrow[d, dash] & & & \\
 & & {} \arrow[rrr, "\textit{Mining}"] & & & \mathsf{PN}\ N,\ \mathcal{L}(\mathsf{TS}) \subseteq \mathcal{L}(N)
\end{tikzcd}$$

Fig. 2. Flow for PN derivation

3 Features of the Tool

The typical workflow of the tool can be seen in Fig. 2. Any object whose behavior might be mapped into a TS can be considered. Several tools exist that are able to perform this kind of conversions. Additionally, together with `rbminer` we provide some helper applications, like `log2ts`, which converts a log into a TS.

`rbminer` is implemented in C++, and uses the STL library for data structures and input parsing. The tool has the following features:

Explicit compact representation. Regions are represented as vectors of integers, in which the multiplicity of one state corresponds to one vector position. Although this imposes a limit on the size of the TS that can be mined, it turns out that this strategy is usually more effective than using a symbolic representation, as the experiments show.

Mining of k-bounded PNs. The use of general regions enables the derivation of k-bounded PNs [2], where the parameter k is selected by the user.

Parameterized search. The tool contains three parameters that control the exploration of the region basis: the aggregation factor (agg), and the allowed values for the combination coefficients ($minval$ and $maxval$). Since any region r can be expressed as $\sum_i c_i \cdot r_i$, the algorithm will only explore combinations in which $minval \leq c_i \leq maxval$ and the number of non-zero c_i coefficients is, at most, agg.

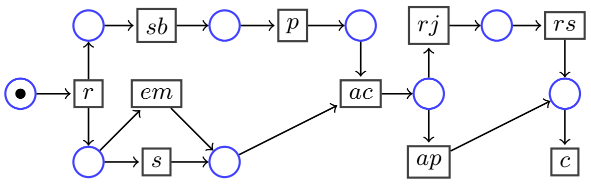
Removal of redundant regions. Regions whose removal does not change the behavior of the derived PN are removed. These regions correspond to redundant places in the PN, and therefore the structure of the net is simplified.

Efficient conversion of logs into TSs. The `log2ts` tool has three conversion modes available, namely *sequential*, *multiset* and *CFM*. The latter method is able to dramatically reduce the size of the obtained TS at the cost of producing more overapproximated nets in some cases [4].

Causality heuristic search. Analyzing the causality between the events in the TS, important relations between events can be detected and later be used to speed up the exploration of the region space. This is a new feature with respect to the original implementation of [4].

1	r, s, sb, p, ac, ap, c
2	r, sb, em, p, ac, ap, c
3	r, sb, p, em, ac, rj, rs, c
4	r, em, sb, p, ac, ap, c
5	r, sb, s, p, ac, rj, rs, c
6	r, sb,p, s, ac, ap, c
7	r, sb, p, em, ac, ap, c

(a)



(b)

Fig. 3. Process mining: (a) log, (b) Mined Petri net

4 Examples

Process Mining

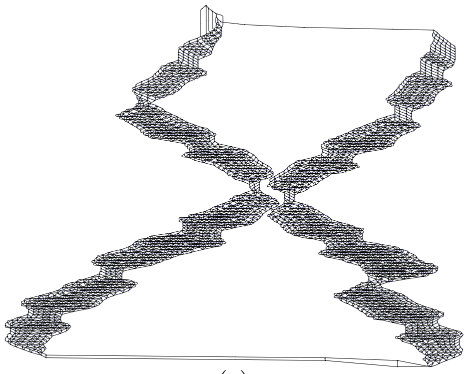
Imagine that we have a set of traces and want a PN that covers all these traces: this is one of the big challenges of Process Mining [6]. Fig. 3(b) shows a small log describing the process of handling customer orders. The log contains seven traces with the following activities: *r=register*, *s=ship*, *sb=send_bill*, *p=payment*, *ac=accounting*, *ap=approved*, *c=close*, *em=express_mail*, *rj=rejected*, and *rs= resolve*. By using the helper application `log2ts`, a TS can be obtained (not shown) for which `rbminer` derives the PN in Fig. 3(b). The region basis for the example contained only 8 regions.

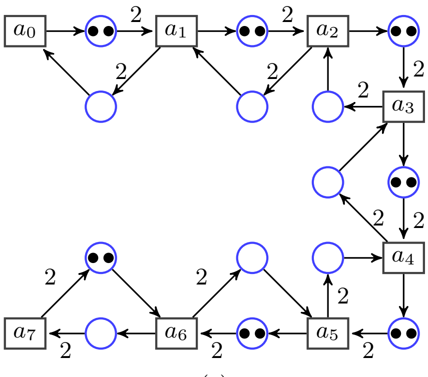
Synthesis of Concurrent Systems

As Figure 2 shows, several models for concurrent systems can be mapped to a state-based representation like the transition system. However, due to the well known *state explosion problem*, the TS representation of these systems is typically large and not good for visualization. In contrast, a PN is often a very good model for concurrent systems, given that concurrency is represented explicitly. Fig. 4(a) shows a complex TS describing a concurrent system. The region basis found by `rbminer` contains only 7 regions, and the gradients for each region in the basis are shown in (b) (order $(a_0, a_1, \ldots, a_7)$). Finally, by exploring this region basis, `rbminer` is able to derive the 2-bounded PN in (c) that synthesizes the TS. Clearly, the succinctness of the PN model in representing large and concurrent systems is very nicely illustrated in this example.

5 Experiments

We compare the performance and the quality of `rbminer` with respect to two other tools for similar purposes: the `Parikh` miner in the ProM tool and `genet`. The `Parikh` miner [11] uses the language-based theory of regions combined with ILP, `genet` implements the classical TS-based approach with a symbolic representation of the TSs, and the `rbminer` tool implements two methodologies, the


(a)



(c)

(b) { (-1,2,0,0,0,0,0,0), (-1,0,4,0,0,0,0,0), (-1,0,0,8,0,0,0,0), (-1,0,0,0,16,0,0,0), (-1,0,0,0,0,32,0,0), (-1,0,0,0,0,0,64,0), (-1,0,0,0,0,0,0,128) }

Fig. 4. (a) A TS with 8 events and 2187 states (b) The gradients of the 7 regions in the basis (c) Mined PN

standard approach `rbminer-std` of [4], and a novel causality heuristic approach named `rbminer-causal` which was developed recently.

The first set of experiments uses some logs from [11]. Since `genet` and `rbminer` need a TS as an input, we have used the `log2ts` application to build the TSs. For each method we provide the number of places (column P) of the mined PN, the time in seconds to compute it, and its *appropriateness* [12]. This metric quantifies to which extent the model describes the log and the simplicity of the net using a number between 0 (poor) and 1 (excellent). We limited the amount of memory and time available to 1Gb and 10000 seconds, respectively.

The benefits of using basis of regions are twofold. First, the memory consumption is very low: in all cases the maximum amount of memory used by `rbminer` was 10Mb. This is a clear advantage over other approaches, notably `genet`, which is very memory demanding. Second, the running times are, in general, much lower, while the quality is quite similar across all tools.

In addition to these experiments on logs, we have also mined cyclic TSs that represent: a system with shared resources (SR), a producer-consumer environment (PC), and a pipeline of n processes (BP).

Table 1. Mining of large logs, with parameters $agg = 4$, $minval = -1$, $maxval = 1$ and $k = 1$ for `rbminer`

	`genet`			`Parikh`			`rbminer-std`			`rbminer-causal`		
Log	P	Time	App.	P	Time	App.	P	Time	App.	P	Time	App.
a32f0n00_5	31	1	0.95	31	112	0.93	31	2	0.95	31	1	0.95
t32f0n00_5	memout			30	9208	0.99	30	5	0.92	30	4	0.92
a42f0n00_5	timeout			44	1557	1.0	46	33	1.0	46	26	1.0

Table [2] compares the mining capabilities of `genet` and `rbminer`, since `Parikh` cannot handle cyclic TSs. In both cases, no extra behavior was included in the PN, but the nets were very different in terms of compactness. In all cases `rbminer` could reconstruct the original PNs from which the TSs were derived. The aggregation factor was set in each case to the value where synthesis was achieved. For some benchmarks these values are quite low, showing that in some cases a very shallow exploration of the region space might suffice. This contributes to a better performance of `rbminer`, since the methods underlying are dominated by the aggregation factor and the number of regions in the basis.

Table 2. Mining of cyclic TSs

	genet		rbminer-std			rbminer-causal		
Bench.	P	Time	Agg	P	Time	Agg	P	Time
PC(9,6)	62	332	10	20	1	10	20	1
SR(7,5)	241	1190	7	29	1565	7	29	52
BP(10)	timeout		2	20	0.1	2	20	1

6 Conclusions, Tool Availability and Acknowledgements

This paper presents `rbminer`, a tool to support the discovery of PNs from TSs. The tool has reached a mature state after incorporating recent enhancements to the theory with respect to [4]. There is a web page for the tool:

`http://www.lsi.upc.edu/{$\sim$}jcarmona/rbminer/rbminer.html`

where related papers, a tutorial and the Linux binaries can be obtained. This work has been supported by projects TIN2007-66523 and TIN2007-63927.

References

1. Arnold, A.: Finite Transition Systems. Prentice-Hall, Englewood Cliffs (1994)
2. Murata, T.: Petri Nets: Properties, analysis and applications. Proceedings of the IEEE, 541–580 (April 1989)
3. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Part I, II. Acta Informatica 27, 315–368 (1990)
4. Solé, M., Carmona, J.: Process mining from a basis of state regions. In: Proc. 31st Int. Conference on Applications and Theory of Petri Nets (June 2010)
5. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Trans. on Information and Systems E80-D(3), 315–325 (1997)
6. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE TKDE 16(9), 1128–1142 (2004)
7. van Dongen, B., Alves de Medeiros, A., Wen, L.: Process mining: Overview and outlook of petri net discovery algorithms. In: ToPNoC II, pp. 225–242 (2009)

8. Lorenz, R., Mauser, S., Juhás, G.: How to synthesize nets from languages: a survey. In: WSC 2007: Proc. of the 39th Conference on Winter Simulation, pp. 637–647 (2007)
9. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Proc. Int. Conf. on Business Process Management, pp. 375–383 (2007)
10. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: A tool for the synthesis and mining of Petri nets. In: ACSD, pp. 181–185 (2009)
11. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: Petri Nets, pp. 368–387 (2008)
12. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1), 64–95 (2008)

Author Index