Supratik Chakraborty
Madhavan Mukund (Eds.)

LNCS 7561

Automated Technology for Verification and Analysis

10th International Symposium, ATVA 2012
Thiruvananthapuram, India, October 2012
Proceedings

Springer

Lecture Notes in Computer Science 7561

Supratik Chakraborty
Madhavan Mukund (Eds.)

Automated Technology for Verification and Analysis

10th International Symposium, ATVA 2012
Thiruvananthapuram, India, October 3-6, 2012
Proceedings

Springer

Volume Editors

Supratik Chakraborty
Dept. of Computer Science and Engineering
Indian Institute of Technology Bombay, Powai
400076 Mumbai, Maharashtra, India
E-mail: supratik@cse.iitb.ac.in

Madhavan Mukund
Chennai Mathematical Institute
H1, SIPCOT IT Park, Kelambakkam
603103 Siruseri, Tamil Nadu, India
E-mail: madhavan@cmi.ac.in

ISSN 0302-9743
ISBN 978-3-642-33385-9
DOI 10.1007/978-3-642-33386-6

e-ISSN 1611-3349
e-ISBN 978-3-642-33386-6

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012946628

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Preface

This volume contains the invited and contributed papers presented at the 10^{th} International Symposium on Automated Technology for Verification and Analysis (ATVA 2012), held at Thiruvananthapuram (Trivandrum), India, during October 3–6, 2012. Over the last decade, ATVA has established itself as a premier venue for researchers and practitioners working on both theoretical and practical aspects of automated analysis, verification and synthesis of computing systems. The conference has also historically provided a forum for interaction between the regional and international research communities working in these areas. The rich legacy of ATVA continued this year as well, resulting in a very strong technical programme.

We received a total of 80 regular submissions and 9 tool submissions, excluding those that were incomplete, outside the scope of the conference, or concurrently submitted elsewhere. The submissions came from 20 different countries spanning 4 continents. Each submission was reviewed by at least 4, and in some cases even 6, reviewers. The Programme Committee sought the help of 145 external expert reviewers for detailed evaluations of the submissions. This was followed by two and a half weeks of extensive and spirited discussions among the members of the Programme Committee, based on which 25 regular papers and 4 tool papers were finally accepted.

The conference was privileged to have three distinguished computer scientists as invited speakers. Sharad Malik (Princeton University, USA), Andreas Podelski (University of Freiburg, Germany) and P.S. Thiagarajan (National University of Singapore, Singapore) readily agreed to give tutorials and invited talks at the conference. We thank all of them for enriching the conference with their participation.

It has been an absolute pleasure working with 35 distinguished colleagues from 16 countries as part of the Programme Committee. We thank all of them for helping spread the call for papers, providing detailed reviews for the submissions, participating in the online discussions with tremendous energy and enthusiasm, providing critical comments and useful suggestions whenever needed, and for everything else that helped shape the strong technical programme that we finally arrived at.

We thank the Steering Committee of ATVA for giving us the opportunity to host this conference in India for the first time. We also thank members of the Steering Committee for providing guidance on various aspects of planning of the conference.

The Organizing Committee of the conference put in several months of hard work to ensure that every aspect of the organization of the conference was attended to in detail. We thank all members of the Organizing Committee for their dedication to the success of the conference. Kumar Madhukar helped with several logistical aspects, including designing and maintaining the conference webpage, creating posters and handling email enquiries. We thank him for his contribution to the success of the conference. We also thank S. Ramesh, General Chair of the conference, for providing helpful guidance whenever it was needed.

From the time we drafted the proposal for hosting ATVA 2012 in India, Tata Consultancy Services has been unwavering in its support to the Indian Association for Research in Computing Science (IARCS) for making the conference a success. The helping hand lent by Tata Consultancy Services in every aspect of planning and organizing the conference deserves special mention. R. Venkatesh played a crucial role in our interactions with Tata Consultancy Services. We thank him for his special efforts in this regard.

A conference like ATVA cannot succeed without significant financial help from various agencies. We thank IARCS, Tata Consultancy Services, Microsoft Research India, Special Interest Group in Software Engineering of the Computer Society of India, and Corporate Research Technologies of Siemens Technology and Services Pvt. Ltd. for providing sponsorship to make the conference a success.

Finally, we thank the Easychair team for providing us with an excellent paper and review management system that made the entire process of reviewing and compiling the proceedings smooth. We also thank Springer for publishing the proceedings as a volume in the series Lecture Notes in Computer Science, and for all the editorial help rendered by them in compiling the proceedings.

October 2012

Supratik Chakraborty
Madhavan Mukund

Conference Committees

Steering Committee

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

General Chair

S. Ramesh	Global GM R&D

Programme Committee

Rajeev Alur	University of Pennsylvania
Christel Baier	Technical University of Dresden
Purandar Bhaduri	Indian Institute of Technology Guwahati
Jonathan Billington	University of South Australia
Gianpiero Cabodi	Politecnico di Torino
Supratik Chakraborty	Indian Institute of Technology Bombay
Deepak D'Souza	Indian Institute of Science, Bangalore
Pallab Dasgupta	Indian Institute of Technology Kharagpur
E. Allen Emerson	The University of Texas at Austin
Laurent Fribourg	LSV, ENS de Cachan and CNRS
Masahiro Fujita	University of Tokyo
Susanne Graf	Université Joseph Fourier/CNRS/VERIMAG
Teruo Higashino	Osaka University
Alan J. Hu	University of British Columbia
Franjo Ivančić	NEC Laboratories America
Joost-Pieter Katoen	RWTH Aachen University
Zurab Khasidashvili	Intel
Moonzoo Kim	KAIST
Padmanabhan Krishnan	Bond University
Orna Kupferman	Hebrew University
Insup Lee	University of Pennsylvania
Xuandong Li	Nanjing University
Madhavan Mukund	Chennai Mathematical Institute
K. Narayan Kumar	Chennai Mathematical Institute

Aditya Nori	Microsoft Research India
Jun Pang	University of Luxembourg
Doron A. Peled	Bar Ilan University
Sanjiva Prasad	Indian Institute of Technology Delhi
R. Venkatesh	Tata Consultancy Services
G. Ramalingam	Microsoft Research India
Anders P. Ravn	Aalborg University
Abhik Roychoudhury	National University of Singapore
Ashish Tiwari	SRI International
Mahesh Viswanathan	University of Illinois at Urbana-Champaign
Farn Wang	National Taiwan University
Hsu-Chun Yen	National Taiwan University
Wang Yi	Uppsala University

Organizing Committee

Supratik Chakraborty	Indian Institute of Technology Bombay
K. Kesavasamy	Tata Consultancy Services
Shrawan Kumar	Tata Consultancy Services
Madhavan Mukund	Chennai Mathematical Institute
S. Ramesh	Global GM R&D
R. Venkatesh	Tata Consultancy Services

Additional Reviewers

Abraham, Erika
Ahn, Jaemin
Albarghouthi, Aws
Amalio, Nuno
Ayoub, Anaheed
Balakrishnan, Gogul
Bayless, Sam
Berdine, Josh
Berwanger, Dietmar
Bollig, Benedikt
Brockschmidt, Marc
Bu, Lei
Chadha, Rohit
Chakraborty, Soumyodip
Chandra, Satish
Chang, Jian
Chattopadhyay, Sudipta
Chen, Xihui
Chittimalli, Pavan Kumar
Cyriac, Aiswarya
D'Silva, Vijay
Diciolla, Marco
Divakaran, Sumesh
Duggirala, Sridhar
Ekberg, Pontus
Esparza, Javier
Fraer, Ranan
Frehse, Goran
Fu, Hongfei
Gallasch, Guy
Gao, Sicun
Gastin, Paul
George, Benny
Ghafari, Naghmeh
Ghorbal, Khalil
Hansen, Rene Rydhof
Heam, Pierre-Cyrille
Heinen, Jonathan
Holik, Lukas
Hong, Shin
Huang, Chung-Hao
Huang, Geng-Dian
Hudak, Paul
Jagadeesan, Radha
Jansen, Christina
Jhala, Ranjit
Jiang, Jie-Hong
K.R., Raghavendra
Kahlon, Vineet
Kanade, Aditya
Kapoor, Hemangee
Kapoor, Kalpesh
Karandikar, Prateek
Karmarkar, Hrishikesh
Kesh, Deepanjan
Kiehn, Astrid
Kim, Baekgyu
Kim, Youngjoo
Kim, Yunho

King, Andrew
Klein, Joachim
Klueppelholz, Sascha
Komondoor, Raghavan
Korchemny, Dmitry
Krishna, K.V.
Kumar, Sandeep
Kumar, Shrawan
Kunz, César
Lal, Akash
Lampka, Kai
Lang, Frédéric
Leroux, Jerome
Linden, Jonatan
Liu, Jinzhuo
Liu, Lin
Madhavan, Ravichandhran
Madhukar, Kumar
Mardare, Radu
Metta, Ravindra
Meyer, Roland
Misra, Janardan
Mohan M., Raj
Muller, Tim
Mun, Seokhyeon
N., Raja
Noll, Thomas
Norman, Gethin
Nyman, Ulrik
Oliveira, Bruno
Olivo, Oswaldo
Pal, Debjit
Park, Junkil
Phan, Linh Thi Xuan
Picaronny, Claudine
Prabhakar, Pavithra
Prabhu, Santhosh
Qi, Dawei
Qiu, Xiaokang
Qu, Hongyang
Rajamani, Sriram
Ranise, Silvio
Reinkemeier, Philipp
Rinetzky, Noam
Roy, Pritam
Roy, Suman
Rümmer, Philipp
Saha, Diptikalyan
Samanta, Roopsha
Sampath, Prahladavaradan
Sankaranarayanan, Sriram
Sankur, Ocan
Sarangi, Smruti
Schnoebelen, Philippe
Seth, Anil
Shaikh, Siraj A.
Sharma, Arpit
Sproston, Jeremy
Sridhar, Nigamanth
Stigge, Martin
Stolz, Volker
Subotic, Pavle
Sureka, Ashish
Suresh, S.P.
Tabatabaeipour, Seyed Mojtaba
Teige, Tino
Ummels, Michael
Val, Celina Gomes Do
Van Breugel, Franck
Vaswani, Kapil
Vighio, Saleem
Von Essen, Christian
Vorobyov, Kostyantyn
Wang, Bow-Yaw
Wang, Linzhang
Wang, Shaohui
Wang, Shuling
West, Andrew
Wies, Thomas
Wisniewski, Rafael
Wu, Jung-Hsuan
Yang, Shun-Ching
Yu, Fang
Zeljić, Aleksandar
Zhang, Chenyi
Zhang, Miaomiao
Zhao, Jianhua

Table of Contents

Invited Papers

Automata Theory

Logics and Proofs

Model Checking

Software Verification

Synthesis

Verification and Parallelism

Probabilistic Verification

Constraint Solving and Applications

Probabilistic Systems

Verification of Computer Switching Networks: An Overview

Shuyuan Zhang[1], Sharad Malik[1], and Rick McGeer[2]

[1] Department of Electrical Engineering, Princeton University, Princeton, NJ
{shuyuanz,sharad}@princeton.edu
[2] HP Laboratory, Palo Alto, CA
rick.mcgeer@hp.com

Abstract. Formal verification has seen much success in several domains of hardware and software design. For example, in hardware verification there has been much work in the verification of microprocessors (e.g. [1]) and memory systems (e.g. [2]). Similarly, software verification has seen success in device-drivers (e.g. [3]) and concurrent software (e.g. [4]). The area of network verification, which consists of both hardware and software components, has received relatively less attention. Traditionally, the focus in this domain has been on performance and security, with less emphasis on functional correctness. However, increasing complexity is resulting in increasing functional failures and thus prompting interest in verification of key correctness properties. This paper reviews the formal verification techniques that have been used here thus far, with the goal of understanding the characteristics of the problem domain that are helpful for each of the techniques, as well as those that pose specific challenges. Finally, it highlights some interesting research challenges that need to be addressed in this important emerging domain.

1 Introduction

Today's computer networks have become extremely large and complicated. The increased scale is observed in datacenters, as well as enterprise networks which can have hundreds of thousands of networking devices. The increased complexity is due to multiple kinds of networking devices (routers, switches, Network Address Translators or NATs, firewalls) that need to work together to execute the diverse network functions such as routing, access control, encryption and network address translation. Some of these devices need to support multiple protocols to make the network safer and faster. Further, the implementations of the protocols and network devices differ across vendors. Thus, it is non-trivial to make the system work correctly and efficiently.

One of the difficulties in managing such a complex system is the correct configuration of the network devices. Misconfiguration of the network counts for more than half of network downtime [5]. The misconfiguration bugs result in different kinds of network errors, among which are reachability failures, forwarding loops, blackholes, access control failures, and isolation guarantee failures. These errors

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 1–16, 2012.

can violate the security requirements, fail the correct delivery of packets, and degrade the efficiency and performance of the network. Thus, it becomes critical that we can detect network errors and verify network properties as we run and maintain the network system.

Recent years have seen the application of formal verification techniques in network configuration verification. These attempts span a range of techniques, from graph-based analysis [6] to the use of various verification engines such as ternary symbolic simulation [7], model checking [8,9] and propositional logic property checking [10]. In this paper we provide a review of these techniques, including our recent work on using propositional property checking [11,12] with the goal of understanding the characteristics of the problem domain that are helpful for each of the techniques, as well as those that pose specific challenges. Finally, it highlights some interesting research challenges that need to be addressed in this important emerging domain.

This paper is organized as follows. Section 2 provides some general background for the problem domain. The next two sections provide an overview of the two main classes of approaches, based on finite-state machine verification and Boolean satisfiability. Finally, Section 5 discusses some directions for future work in this domain.

2 Network Systems and Properties: Background

2.1 Network System States

The network systems we consider are packet switching networks. The network components can be a variety of devices such as routers, switches, bridges, Network Address Translators (NAT), firewalls, and even OpenFlow switches [13]. These devices are connected by links. In this paper, we refer to these devices as middleboxes or simply switches. Figure 1 provides an illustrative sketch of a network comprising of switches connected by links. The switches process the packets and the links transfer them between switches. The processing can vary depending on the switch, e.g. a firewall decides which packets are allowed to go through and which are blocked based on a set of rules. A router decides which of its output ports (and thus the link connected to that port) an incoming packet should be routed to based on its routing table. The flow of packets is referred to as the network traffic. Each packet consists of a header and a payload. The header captures the information needed to process the packets, e.g. source/destination addresses and the time-to-live field which indicates how long the packet may continue to stay in the network. The header may be modified as a packet is processed by a switch. The payload contains the application data.

Historically, networking traffic is thought of as operating on two levels or *planes*. In general, these refer to classes of message exchanged between switches. In classic networking, switching/routing decisions are made by switches individually, on the basis of information captured in a number of on-switch data structures, known as the *forwarding information base* (FIB) and the *routing information base* (RIB). The FIB and the RIB are updated regularly by the switch

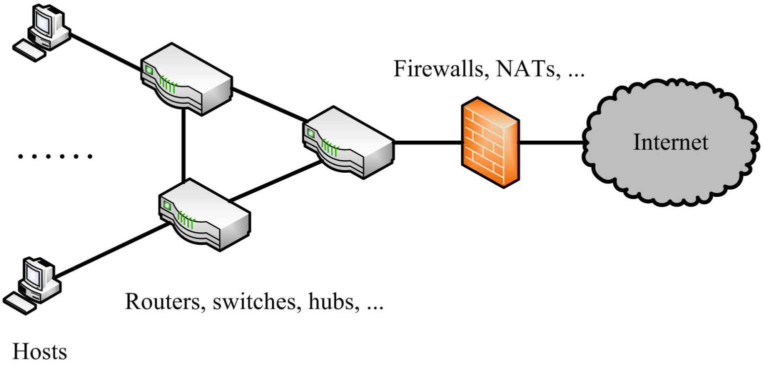


Fig. 1. Sketch of a Packet Switching Network

or router on the basis of messages received in the course of its operation. See, for example [14], among many others.

- The *Forwarding Data Plane* consists of application traffic, and is simply passed by the switch on the appropriate port as directed by the FIB and the RIB, with appropriate rewrites to the various header fields.
- The *Control Plane* consists of traffic used by the network to compute information in the RIB and the FIB. Classic examples of messages exchanged on the forwarding plane compute shortest paths to specific destinations, determine the location of devices with specific Media Access Control (MAC) or Internet Protocol (IP) addresses, and explicit control messages sent between devices on the network and from external control sources. Examples of the latter are *Border Gateway Protocol* [15] messages, which concern the handling of packets destined for locations outside the local area network and *Simple Network Management Protocol* [16]message.

The network states we are interested in here are the forwarding/blocking rules extracted from *Routing Information Base (RIB)* and *Forwarding Information Base (FIB)* in routers, *Access Control List (ACL)* in firewalls, and *Forwarding Table* in switches. We define the *switch state* as the collection of all the RIBs, FIBs, ACLs, forwarding tables, configuration policies stored in the switch at a single instance of time (see Figure 2). The *network state* is the collection of the switch states in all switches. These rules comprising the switch state usually have two fields, one matching field which specifies the packet header information for packets which should be processed using this rule, and one action field, which specifies what actions will be taken on the matching packets, i.e. how the packet is to be processed. This varies with the switch, e.g. a firewall rule will indicate if the matching packet should be dropped or allowed, a router will decide which output port (and connected link) a matching packet should be forwarded to. A payload of a packet is typically not considered in the rule matching as it does not determine the packet processing. The action field can be forwarding actions such as: blocking the packet, forwarding the packet to specific ports, flooding the packet (forwarding a copy to all but the incoming port), forwarding the packet

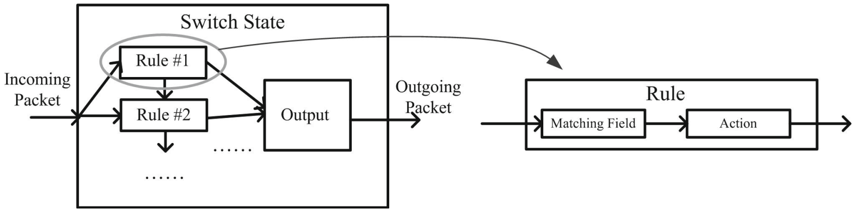


Fig. 2. The Switch State

to the incoming port, rewriting rules which rewrite some of the header fields of the packet (as in NATs), and packet encapsulation which includes an existing packet header in a new header (used for network security).

These network states we focus on here are completely static. They are snapshots of dynamic networks at a single instant in time and do not change during verification. Gude argued that changes in the network rules are on the order of tens of events per second for a network with thousands of hosts while packets arrive on the order of millions of arrivals per second for a 10Gbps link [17]. As network rule updates are much slower than the packet arrival rate, the network can be largely regarded as a static system. Consequently, we assume that the network system is stateless as it is completely fixed during verification and no packet can modify the network state. While some verification techniques consider dynamic system states such as the graph-based analysis by Xie *et al.* [6], *the focus of this paper is on formal verification techniques that consider a single snapshot of the network system state.* . We note that there has been recent work on safe update protocols for OpenFlow networks, which aim to ensure that verification certificates given by static techniques are not invalidated by network updates [18,19,20].

2.2 Network Properties

In this paper, the properties of interest are those related to functional correctness. We do not study properties related to speed metrics such as congestion, latency, and bandwidth. The following properties have been the subject of interest in the various verification efforts.

Reachability. *Reachability* is concerned with whether the network always successfully delivers packets to the intended end hosts. There are various flavors of reachability. A loosely defined reachability property can be that a packet P can get to the end host A. Although it specifies that the packet will reach A, it does not specify whether a copy of the packet may also get to other hosts. A stricter definition of reachability requires that P will always go to A and nowhere else.

Forwarding Loop. A network is said to have a *forwarding loop* if the same packet returns to a location that it has visited before. Again, there are several flavors of this property, e.g. returning to the same location with exactly the same header, or returning to the same location with a possibly different header. The former case indicates the presence of an infinite loop, since this packet will repeatedly return to this location. The latter case may also be undesirable since there is usually no reason for a packet to return to the same location. The classic defense against forwarding loops is the `Time-To-Live` field in the packet header, which sets the maximum hop count for a packet: each switch decrements the TTL field, and discards those whose TTL reaches zero. While TTL fields preserve network resources against infinite loops, TTL discards still represent a forwarding bug that a verification system must diagnose.

Packet Destination Control. Another class of properties specifies details of fine-grained packet handling. Examples include requiring that packets of a specific class be dropped -*blacklisting*, or requiring some packets to traverse specific devices or edges - *waypointing*. Blacklisting often arises in security applications, and waypointing in audit requirements.

Slice Isolation. Multi-tenant networks must be able to guarantee that mutually-untrusting users are guaranteed privacy. A virtual, isolated private network in a shared network is referred to as a *slice* of the network [21,7]. Slice isolation specifies that it is impossible for one slice to eavesdrop or interfere with another.

2.3 OpenFlow and Stateless Networking

The recent rise in interest in network verification [7,11] has been due to the introduction of the OpenFlow protocol [13,22,23,24,25,26,27,28]. OpenFlow centralizes the control plane into a centralized controller. In the OpenFlow protocol, the various control plane sensing and command messages become controller inputs and the various routing tables to the switches are controller outputs. An important implication of this change to centralized control is that while a distributed control plane may result in non-deterministic ordering of network state updates, a centralized controller will result in deterministic network state updates thus making it easier to reason about system states and enabling analysis of a single snapshot of the system state.

Verification of the controller, which in its most general sense is Turing-complete, remains undecidable. However, the *output* of the controller is another matter. It was observed in [11,12] that this could be transformed into a graph of state-free, combinational logic elements. Though the graph is cyclic, by a technique of unrolling the graph in time, an acyclic graph can be derived. An acyclic graph of state-free, combinational logic elements is a simple logic network; its verification is $\mathcal{NP}$-complete, and in fact [11,12] demonstrated that network verification problems could be transformed into satisfiability problems on logic networks.

The recognition that OpenFlow offered a logic abstraction of the network dovetailed with an emerging literature on declarative, state-free network specification [29,30,31]. FML [30] explicitly coupled a formal verification procedure to a formal declarative specification of network behavior. The class of specifications anticipated by [30] was relatively weak however; FML specifications were verifiable in poly-time, indicating either weak verification or limited specification.

3 Finite-State Machine Based Approaches

Three techniques borrowing heavily from the ideas of Finite-State Machine (FSM) verification were recently proposed: [7,8,9,19]. All three treat the packet as a finite state machine. The state is the tuple (h, p) where h is the packet header and p is its location. The transition function is encapsulated in the rules of the network devices. The initial state of the packet is an arbitrary function of the header bits and location, as required by the specific verification obligation; a completely unspecified initial state is possible. All possible evolutions of this state graph are enumerated by symbolically propagating the packet through the network, with the header bits set as necessary for each possible propagation.

The techniques vary in the structures and methods used to propagate packets, the domain of the transfer functions, and the representation of the verification obligation. [7] permits arbitrary functions of the header bits, and represents them as Boolean functions in sum-of-products form (disjunctive-normal form or DNF). [19] specifies network properties using Computation Tree Logic (CTL) [32], and uses model checking with NuSMV [33] to verify them. [8,9] use a similar formulation of network states, but explicitly build Binary Decision Diagrams (BDDs) for the network states and verify them using SMV [34,35].

When the set of all possible state vectors have been enumerated, i.e. a fixed point is reached, the iteration ceases and the set of reached states examined to determine if an undesirable state (bad combination of header bits and location) has been reached. The Header Space Analysis (HSA) approach [7] refers to the Boolean space of header bits as the *Header Space*; the *Network Space* is the space of the tuples (h, p). The packet location is an integer encoded as a Boolean in the usual fashion.

The behavior of each network device, including packet forwarding by routers and switches, packet header modification by NATs, and packet blockage by firewalls, is modeled as a transfer function over the Network Space. This function takes the current packet state as input, and outputs the possible new states, (h', p'), where h' is the new header and p' is the new location, exactly modeling the action of the network device. In general, the function is non-deterministic, even given a fully-specified network state, e.g., a multicast packet will have multiple new locations.

The Network Transfer Function is simply the disjunction of all the device transfer functions. Let (h, p) be the packet state. The Network Transfer Function Ψ is

$$\Psi(h,p) = \begin{cases} T_1(h,p) & \text{if } p \in switch_1 \\ ... & ... \\ T_n(h,p) & \text{if } p \in switch_n \end{cases}$$

If two devices are physically connected to each other, the outgoing packets from one device will directly reach the incoming port of the other. To represent this direct connection, the *Topology Transfer Function* Γ defined as

$$\Gamma(h,p) = \begin{cases} (h,p^*) & \text{if } p \text{ connected to } p^* \\ (h,NULL) & \text{if } p \text{ is not connected} \end{cases}$$

Here NULL refers to a dummy port.

The joint use of network transfer function and topology transfer function is used to emulate the propagation of packets through the network. Let function $\Phi = \Psi(\Gamma(.))$ and recursively applying function Φ to packet (h, p) for k times or $\Phi^k(h, p)$ simply gives us the packet headers and locations after the network processes the original packet (h, p) for k hops.

Both [8,9] and [19] use formulations equivalent to [7]. [8,9] represent the explicit transition relation $T(\mathbf{current}, \mathbf{next})$ as a BDD, using only a small subset of the header bits as packet state: source and destination IP address. Here **current** and **next** are the current and next values of the packet state respectively. Matching rules are implemented in strict priority: if rule C_i and rule C_j for $j > i$ both match packet header h, then C_i matches and C_j is discarded. The exact match function for C_i is then effectively $(\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{i-1} \wedge C_i)$. If C_i matches h then the packet state changes from (h, p) to the next state $(h^{'}, p^{'})$ based on the action of rule C_i. The transition relation is simply the disjunction of these selections over all the devices in the network. This formulation of the transition relation can be easily modified to accommodate more header bits in the matching rules, and non-strict priority among matching functions. [19] formulates the network transfer function as a NuSMV program in CTL, and uses NuSMV to represent the network states and perform the standard analyses.

Since [7] does not use an available verifier, this is the only system which explicitly employs a specific propagation and solution technique. As mentioned above, transition functions, reachable states, and verification properties are expressed as Boolean functions in DNF over the network space; the state reachability iteration is performed by propagating the reached states through the transfer function until a fixpoint is reached, a technique known as Ternary Symbolic Simulation [36].

3.1 Property Checking

Reachability Analysis. Network reachability for location a is obtained by setting the initial state to `dest_ip =` a, then performing the classic reachable-states iteration. Reachable states with a location set to an internal device are discarded, and the remaining reachable states must all have `location` $= a$.

To check the reachability between switch a and switch b, [7] first enumerates all the paths that connect a and b. $T(.)$ is iteratively applied to the packet for every switch along the path. Then reachability from a to b is defined as

$$R_{a \to b} = \bigcup_{a \to b \text{ paths}} \{T_n(\Gamma(T_{n-1}(...(\Gamma(T_1(h, p))...))\}$$

where h is the *header space*, p is a port of a, and $T_i(.)$ $(i \in [1, n])$ is the transfer function of switch S_i along the path $a \to S_1 \to ... \to S_{n-1} \to S_n \to b$. $R_{a \to b}$ are the final packets that will end up in b.

Forwarding Loop. Forwarding loops occur when a packet transits the same switch twice. Ternary symbolic simulation [7] checks for forwarding loops by manually injecting the whole *header space* into one of the ports and propagating through the network until every packet either 1) gets out of the network; 2) returns to a port already visited; or 3) returns to the port that the original *header space* is injected to. To incorporate the packet history, [7] extends the original packet state with the visit history of the switches a packet has passed through. If case 3) occurs, it has found a generic forwarding loop and it ignores case 2) to avoid finding the same loop.

[19] and [8,9] use explicit model checking, and formulate the forwarding loop condition as the following temporal logic formula for the underlying verifier: $(loc = a_1) \wedge \mathbf{EX}(\mathbf{EF}(loc = a_1))$. This captures the property that the same packet gets back to switch a_1 twice.

Packet Destination Control. Reitblatt proposed using CTL to check whether a packet can get out of the network, get dropped, go through certain switches, or never pass through certain links [19]. Al-Shaer's model can also verify those properties. All of these are easily verified through the reachable-states iteration; a packet being dropped is signified by simply assigning a new location for dropped packets.

Slice Isolation. A *slice* of a network or distributed system is an isolated virtual network; in the case of a network, it refers to the ability of two or more independent users to write non-conflicting rules for packet destinations. Though the two users share the network substrate, each directs its own packets individually. In HSA or the model checking based approaches, the slices are isolated if the rules in each slice are written over disjoint network spaces. If $T_1(x,y), T_2(x,y)$ are the transition relations for the two slices, the isolation condition is simply

$$(\exists y T_1(x,y)) \cap (\exists y) T_2(x,y)) = \emptyset$$

3.2 Discussion

When the notation and terminology of [7] is translated into conventional terms, it is apparent that the method is simply checking safety properties of finite-state machines, using state propagation based upon symbolic simulation of implicants. This is nothing more than the familiar reachable-states iteration:

$$R_n(x) = R_{n-1}(x) \cup \exists_y [T(y,x) \cap R_{n-1}(y)]$$

iterated to the fixpoint $R^* = R_n = R_{n-1}$, and the verification step is simply to apply the appropriate safety check:

$$R^*(x) \cap F(x) = \emptyset$$

where $F(x)$ represents the collection of bad states. The forwarding loop verification procedure detailed above could be done far more efficiently than is described in [7], using the insights gained by recognizing the isomorphism with finite-state machine traversal. The question of whether a forwarding loop exists is essentially a question of whether the location bits in the packet state are repeated. With the symbolic ternary simulation of HSA with a DNF representation, and writing the state vector as $x_H x_L$, where x_H are the values of the header bits and x_L the values of the location bits, the new implicants introduced into R_n are

$$\exists_y[T(y, x_H x_L) \cap R_{n-1}(y)]$$

The location bits are repeated if and only if the following condition holds:

$$(\exists_{x_H} R_{n-1}(x_H x_L)) \cap (\exists_{y x_H}[T(y, x_H x_L) \cap R_{n-1}(y)]) \neq \emptyset$$

This is clearly a polynomial computation (satisfiability and existential quantification for DNF is trivial, and it can be shown [37] that $\exists_{y x_H}[T(y, x_H x_L) \cap R_{n-1}(y)]$ is bounded above by the size of $T(y, x_H x_L)$, as is $R_{n-1}(x_H x_L)$). Of course, the size of $T(y, x_H x_L)$ may be exponential in the size of the network.

The isomorphism of the approach of [7] to the finite-state-machine verification method of [38] is striking: the same representation is used for the reachable state set, and the same algorithm is used for the iteration. [7] reports good experimental results on real networks; in contrast, researchers in verification have long since abandoned symbolic simulation of finite-state machines using DNF to represent state sets, because the size of the state sets explodes rapidly. The success of [7] suggests that the FSMs derived from networks are quite well-behaved under ternary symbolic simulation using DNF. A variety of theoretical and empirical observations support this view. In particular:

- **FSMs derived from networks are, in practice, shallow.** In [11], it was observed that a primary goal in network design is that packets traverse the network in as few hops as possible, and in particular do not traverse the same device twice. Bugs can occur – that is why a primary verification task is detecting forwarding loops – but there are a variety of safeguards built in to ensure this property. Chief among them is the *Time-To-Live* field in the packet header, which enforces a maximum hop count for a packet.
- **The network transition function is small compared to transition functions for computational FSMs.** Computational FSMs are typically expressed as the tensor product of component FSMs: because of the combinatorial properties of the tensor product, the resulting FSM transition function grows larger very rapidly. However, the network transition function in DNF is the *disjunction* of the transition functions of the component devices; disjunctions of sum of products forms grow the like the sum of the sizes of the component functions.
- **The network transition functions rely on relatively few header bits.** In principle, network devices may switch a packet on a large subset of the header bits. The Open Flow 1.1 specification identifies 15 separate fields as possibilities for switching. In practice, rules use a small subset of the header bits, for technological reasons. There is a high premium on making routing decisions using the "fast path" of network devices, which uses specialized hardware resources for matching. These resources generally offer only exact matches on specific field values, or matching on prefixes of field values, or single matches on hash functions. Ternary Content-Addressable Memories

(TCAMs) offer general matching, but these are expensive; so, typically, small TCAMs are used to buttress large, cheap, exact-match memories, which work off a few common fields: layer-2 and layer-3 addresses, source and destination port number, VLAN, protocol, type of service. Rule designers know this, and so they tend to craft rules which match on specific, fully-specified field values and prefixes of fields. It can be shown [37] that functions of this form grow slowly under various compositions, which offers a heuristic argument for the observed slow growth of network transition functions and reachable states.

- **The network transition relation is close to the network transition function in size.** [37] showed that the depth of an FSM was bounded above by the DNF size of its transition relation, as were the DNF sizes of its reachable states function R^*. The characteristic function of the transition relation of an FSM is given by $T(x, y) = 1$ iff $y = F(x)$, where $F(x)$ is the transition function. Usually, the transition relation is very large compared to the transition function, since equality across input and output fields results in a long list of enumerated implicants. However, if fields are either absent or fully specified – in other words, if implicants are all minterms over a suitably chosen subspace – then the transition relation is not much larger than the transition function, and the bounds given by [37] become relevant. That this property holds is suggested by the technological bias to minterm-heavy rulesets discussed above.

The methods of both [19] and [8,9] are isomorphic in formulation to the method of [7]; the only difference is the structures used to represent the state sets, the transition relations, and the verification obligations. [7] used a DNF representation; [8,9] used Binary Decision Diagrams; [19] relied on the internal structures of NuSMV. All three methods showed experimental success. In fact, each classic technique from the formal verification literature performed better on these examples than in verification of general hardware or software systems. We conjecture that this is due to the rulesets that network administrators write; as [37] observed, these rulesets have properties that lead to small BDDs and DNFs.

In sum, though [7] [19] [8,9] model the network verification problem as $\mathcal{P}$-Space complete FSM verification problems, as the network state is fixed, in practice it is an $\mathcal{NP}$-complete problem, with constraints that force the instances to be particularly easy to solve. In view of these, the superior results of weak techniques in this domain are explicable, though it is possible/likely that the use of stronger techniques would yield better results.

4 Boolean Satisfiability Based Approaches

Every form of verification discussed above expresses network properties as logic formulae, and uses the standard techniques of formal verification to solve them. One fundamental property of switching networks is bidirectionality: if there is a connection between port i and port j on a network device, data flows both from i to j and j to i. Dealing with cyclic graphs is therefore a core issue in network verification. In the methods of Section 3, this was done by modeling packet traversal of networks as the evolution of a finite state machine: transit of the packet from place to place (potentially with concomitant rewrite of header bits) was modeled as a change in state of the FSM. The negative aspect of this approach is that FSM verification is a $\mathcal{P}$-space complete problem.

When a finite state machine is known to be loop-free (evolution of the machine results in a fixed point of the state in a bounded number of iterations), then the FSM is isomorphic to a combinational logic network, polynomially related to the original FSM. Verifying combinational logic networks is far easier than verifying FSMs: combinational logic verification problems all reduce to satisfiability problems which are $\mathcal{NP}$-complete.

The technique which transforms a cyclic into an acyclic combinational network is *loop unrolling*, which in the case of a network is termed *unrolling in time* [11]. This technique, used in [11][10], assumes that verification issues concern the transit of a single packet [11] (this was also the fundamental implicit assumption underlying the various formulations in Section 3). In it, each network device at time t forwards a packet to a neighbor device at time $t + 1$. Here, "time" is in fact hop count, instead of real time. The resulting graph is acyclic. The transfer functions at each device are precisely the transfer functions given in the finite state machine formulations; however, in this technique, rather than modifying the state vector they simply are connected to the appropriate successor nodes in the next time slot. [11] makes the unrolling in time explicit; [10] implicitly does this by formulating verification properties as a two-dimensional matrix of functions, where the columns are logic functions at the switches, the rows are times or levels, and all connections go from level i to $i + 1$.

Mai's system, Anteater, has some similarities with Xie's static reachability analysis. Both [10] and [11] argue that the fundamental verification problem in networking is reachability; all other problems can be phrased as minor variants of reachability. Indeed, the principal difference is that while [6] represented properties as explicit sets of packets, [10][11] represent the set as characteristic functions. Verification of reachability, forwarding loop and packet destination control is isomorphic, once the appropriate translation is made between sets of packets and characteristic functions. [11] is essentially simply a thorough analysis of a method identical to [10], and heuristic arguments which suggest that the resulting satisfiability instances will be easy to solve. [12] provides additional detail including techniques for compact representation that minimize the unrolling needed and detailed property specification that permits for a rich set of properties. The methods of [10][11][12] are simply combinational versions of the stateful methods of Section 3.

In order to represent the transfer function of the network as a whole, the network function at each switch is realized as a logic network. There are two fundamental actions: transport of the input from port i to port j, and determining the values of the header bits on port j. The former is computed from the propagation rules of the switch; simultaneously, the output bits on port j as determined by the input bits on port i are computed. The propagation functions are then used to select the appropriate values of the input port.

4.1 Property Specification and Verification

Reachability. Reachability of switch s from a in $t + 1$ hops is determined by dynamic programming in the obvious way as the conjunction of the reachability of each neighbor u of s from a in t hops with the propagation function from u to s, and the disjunction of these functions over all neighbors. Endpoints b are simply terminal cases of this function. Formally, if $P(a, u, t)$ is the propagation function indicating that u is reachable from a in t hops, and $p(u, s)$ is the switch transfer function indicating that s is reachable from u, we have

$$P(a, s, t + 1) = \bigvee_{u} P(a, u, t) \wedge p(u, s)$$

Forwarding Loop. Two sorts of forwarding loops are of interest. In one restrictive case, a packet cannot traverse a switch twice at different times. This is encapsulated as $P(s, s, t) \equiv 0$ for each t, s. The second case incorporates packet header information: one cannot traverse the same switch with the same values. This adds additional constraints on the packet header values.

Packet Destination Control. Packet destination control is written simply as a function of the reachability propagation functions, e.g.packet blacklisting checks that a packet cannot exit the network, i.e. it must be dropped by the network.

Slice Isolation. Slice isolation is worth a separate discussion, because it is time-invariant and does not rely on network unrolling. At each switch, the propagation functions written by each user must be disjoint; since this is time-invariant, it is easily checked for each switch in isolation.

4.2 Discussion

Much of the discussion of finite-state machine based approaches apply here as well. Fundamentally, FSM based approaches iteratively construct a logic function by composition, and test the resulting logic function for satisfiability; different verification obligations lead to different functions. When an FSM is isomorphic to a combinational logic function, as the global transition functions on switching networks generally are, the FSM-based approaches essentially build up the same logic network as would be found directly by calculation. As a result, in practice the FSM and combinational logic based approaches will wind up doing the same calculation over mathematically-equivalent objects; all that differs is the data structures used to represent these objects and the calculation techniques.

A thorough analysis and discussion of this single underlying object can be found in [11]. The analysis of the underlying functions in the Boolean space strongly suggested that the resulting underlying functions are amenable to the standard techniques of logic verfication, specifically ternary symbolic simulation, model checking, and propositional logic verification using SAT.

The approach in [11] dispenses with much of the complexity of other approaches, which maintain additional structures and information for various calculations. For example, [10] maintains a switch history list for each packet, and uses this to check for forwarding loops. This is something which is calculated directly by the pure-functional approach. There may be advantages in either practical computational efficiency or in computing side propositions in the construction and maintenance of side data structures: for example, keeping track of the actual path traversed by a packet. This can be done by the pure-functional approach, but it is awkward.

This pure-functional approach is under development and test; if succesful and practical, it opens up a rich new domain for future directions in verification and switching research,

5 Conclusions and Future Directions

This paper makes a case that FSM and SAT based techniques can be successfully deployed for the verification of static snapshots of computer switching networks. The techniques that we have covered have been shown to be useful for practical sized networks. Specifically, the HSA method has been deployed for the verification of the Stanford backbone network [7]. The success of the FSM method is, in part, due to the small size of the packet state and the small depth of its state space. It is unclear if state instrumentation to encode more complex properties will retain these favorable characteristics. Increasing the number of state-bits will push the capacity limits of both BDD and SAT based model checkers. With the SAT-based propositional logic verification approach, as long as the properties can be expressed as propositional logic formulas, the size of the property formula is relatively small compared to the network state formula. Thus, its scalability is less dependent on the type of property being checked.

These initial results are based on modest sized systems. However, overall, both FSM and SAT based approaches will need to be tested for larger scale systems, e.g. entire datacenters or large scale enterprise networks. This will likely need development of new ideas in their solutions, or at the least adaptation of scaling techniques used in other domains. For example, large datacenters are likely to have symmetry in their structure. This may enable the use of parametric model checking techniques [39], or symmetry reduction in model-checking [40] and SAT-based techniques [41]. Their application will open up new challenges.

A further, more ambitious, goal is to use the verification techniques as a core for network synthesis. Logic synthesis techniques have been used successfully for synthesis of the rules for a single switch [42]. The basic technique for checking the equivalence of two firewalls has been used to formulate the problem of the synthesis of optimal firewalls [12]. Synthesis, of course, is of greater complexity than verification, and the optimal firewall synthesis problem is formulated as a Quantifed Boolean Formula (QBF) Satisfiability problem. Besides the increased theoretical complexity of QBF, practical QBF solvers have only shown limited success. Thus, the practical success of such synthesis formulations is not clear. However, high-quality synthesis can have tremendous benefits, both in improving network performance, as well as reducing design complexity.

Finally, this paper has focused on stateless switching networks. While this focus is well justified, including the network state transitions, for example through the specification of an OpenFlow controller that sets the switch rules, will allow for full system verification. This will involve interesting modeling of the allowable network state transitions. Critical to this is careful choice of models for the specification of OpenFlow controllers. Many current controller implementation methods (e.g. Nox[17], Floodlight[43]) are Turing-complete: while these models offer computational power, verification certificates that can be offered are very weak. Computational power and verifiability are competing qualities. As a result, in any field of computation, one wants the *weakest* computational model that will accomplish the task. Some controller specification methods are written over weak computational models (e.g., [30]); others such as [44] offer some possibility of strong verification, by restricting the domain of discourse of potentially powerful computational elements.

In conclusion, this paper highlights an important emerging verification domain that can benefit from the advances in formal verification, discusses initial successes of both FSM and SAT based approaches and highlights important areas of work for extending the impact of formal verification in this domain.

References

1. Burch, J.R., Dill, D.L.: Automatic Verification of Pipelined Microprocessor Control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
2. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the futurebus+ cache coherence protocol. Formal Methods in System Design 6, 217–232 (1995), doi:10.1007/BF01383968
3. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
4. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation OSDI 2008, pp. 267–280. USENIX Association, Berkeley (2008)
5. Alimi, R., Wang, Y., Yang, Y.R.: Shadow configuration as a network management primitive. SIGCOMM Comput. Commun. Rev. 38(4), 111–122 (2008)
6. Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On static reachability analysis of ip networks. In: INFOCOM Comput. Commun. Societ. Preceedings IEEE, vol. 3 (2005)
7. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI 2012, pp. 9–9. USENIX Association, Berkeley (2012)
8. Al-Shaer, E., Marrero, W., El-Atawy, A., ElBadawi, K.: Network configuration in a box: towards end-to-end verification of network reachability and security. In: 17th IEEE International Conference on Network Protocols, ICNP 2009, pp. 123–132 (October 2009)
9. Al-Shaer, E., Al-Haj, S.: Flowchecker: configuration analysis and verification of federated openflow infrastructures. In: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig 2010, pp. 37–44. ACM, New York (2010)

10. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM 2011, pp. 290–301. ACM, New York (2011)
11. McGeer, R.: Verification of switching network properties using satisfiability. In: ICC Workshop on Software-Defined Networks (June 2012)
12. Zhang, S.: Model checking/boolean satisfiability in switch network verification and synthesis. Princeton University, Department of Electrical Engineering, Ph.D. Research Seminar Examination Report (May 2012)
13. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38(2), 69–74 (2008)
14. Moy, J.: RFC 2328: OSPF Version 2. Technical report, IETF (1998)
15. Hares, S., Rekhter, Y., Li, T., Addresses, E.: A Border Gateway Protocol 4 (BGP-4). Technical Report 4271, RFC Editor, Fremont, CA, USA (January 2006)
16. Harrington, D., Presuhn, R., Wijnen, B.: An architecture for describing simple network management protocol (snmp) management frameworks. Technical report, RFC Editor, United States (2002)
17. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: Nox: towards an operating system for networks. SIGCOMM Comput. Commun. Rev. 38(3), 105–110 (2008)
18. Reitblatt, M., Foster, N., Rexford, J., Walker, D.: Software updates in openflow networks: Change you can believe in. In: Proceedings of HotNets (2011)
19. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., David, W.: Abstractions for network update. SIGCOMM Comput. Commun. Rev. (August 2012)
20. McGeer, R.: A safe, efficient update protocol for openflow networks. In: Proceedings of Hot SDN (2012)
21. Sherwood, R., Gibb, G., Yap, K.K., Casado, M., Appenzeller, G., McKeown, N., Parulkar, G.: Can the production network be the testbed. In: OSDI (2010)
22. Foundation, T.O.N.: The openflow switch specification, http://OpenFlowSwitch.org
23. Casado, M., McKeown, N.: The virtual network system. In: ACM SIGCSE (2005)
24. Casado, M., Garfinkel, T., Akella, A., Freedman, M., Boneh, D., McKeown, N., Shenker, S.: Sane: A protection architecture for enterprise networks. In: Usenix Security (2006)
25. Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S.: Ethane: Taking control of the enterprise. In: Proceedings of ACM SIGCOMM (August 2007)
26. Casado, M., Koponen, T., Moon, D., Shenker, S.: Rethinking packet forwarding hardware. In: Proc. Seventh ACM SIGCOMM HotNets Workshop (2008)
27. Casado, M., Freedman, M.J., Pettit, J., Luo, J., Gude, N., McKeown, N., Shenker, S.: Rethinking enterprise network control. Transactions on Networking (ToN) 17(4), 1270–1283 (2009)
28. Casado, M., Koponen, T., Ramanathan, R., Shenker, S.: Virtualizing the network forwarding plane. In: PRESTO (2010)
29. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. CACM 52(11), 87–95 (2009)
30. Hinrichs, T., Gude, N., Casado, M., Mitchell, J., Shenker, S.: Practical declarative network management. In: Proceedings of ACM SIGCOMM Workshop: Research on Enterprise Networking, WREN (2009)

31. Voellmy, A., Hudak, P.: Nettle: Taking the Sting Out of Programming Network Routers. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 235–249. Springer, Heidelberg (2011)
32. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. Journal of Computer and System Sciences 30(1), 1–24 (1985)
33. Cimatti, A., Clarke, E., 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)
34. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 1020 states and beyond. Information and Computation 98(2), 142–170 (1992)
35. McMillan, K.L.: Symbolic Model Checking, 1st edn. Kluwer Academic Publishers (1993)
36. Bryant, R., Seger, C.-J.: Formal Verification of Digital Circuits Using Symbolic Ternary System Models. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 33–43. Springer, Heidelberg (1991)
37. McGeer, R.: New results on bdd sizes and implications for verification. In: Proceedings of the International Workshop on Logic Synthesis (June 2012)
38. Devadas, S., Ma, H.K.T., Newton, A.R.: On the verification of sequential machines at differing levels of abstraction. IEEE Trans. on CAD of Integrated Circuits and Systems 7(6), 713–722 (1988)
39. Emerson, E., Namjoshi, K.: On model checking for non-deterministic infinite-state systems. In: Proceedings of Thirteenth Annual IEEE Symposium on Logic in Computer Science, pp. 70–80 (June 1998)
40. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design 9, 105–131 (1996), doi:10.1007/BF00625970
41. Aloul, F., Sakallah, K., Markov, I.: Efficient symmetry breaking for boolean satisfiability. IEEE Transactions on Computers 55(5), 549–558 (2006)
42. McGeer, R., Yalagandula, P.: Minimizing rulesets for tcam implementation. In: Proceedings IEEE Infocom (2009)
43. The floodlight openflow controller, http://floodlight.openflowhub.org/
44. Foster, N., Harrison, R., Meola, M.L., Freedman, M.J., Rexford, J., Walke, D.: Frenetic: A high-level language for openflow networks. In: ACM PRESTO 2010 (2010)

Dynamic Bayesian Networks: A Factored Model of Probabilistic Dynamics

Sucheendra K. Palaniappan and P.S. Thiagarajan

School of Computing, National University of Singapore,
{suchee,thiagu}@comp.nus.edu.sg

1 Introduction

The modeling and analysis of probabilistic dynamical systems is becoming a central topic in the formal methods community. Usually, Markov chains of various kinds serve as the core mathematical formalism in these studies. However, in many of these settings, the probabilistic graphical model called dynamic Bayesian networks (DBNs) [4] can be a more appropriate model to work with. This is so since a DBN is often a factored and succinct representation of an underlying Markov chain. Our goal here is to describe DBNs from this standpoint. After introducing the basic formalism, we discuss inferencing algorithms for DBNs. We then consider a simple probabilistic temporal logic and the associated model checking problem for DBNs with a finite time horizon. Finally, we describe how DBNs can be used to study the behavior of biochemical networks.

2 Dynamic Bayesian Networks

There are many variants of DBNs with differing modeling capabilities. We shall begin with the simplest version in order to highlight the main features of a DBN.

A DBN $\mathcal{D}$ has an associated set of system variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ each taking values from a finite domain V. It also has a discrete time domain $\mathcal{T} = \{0, 1, \ldots\}$ associated with it. Often $\mathcal{T}$ will be a finite set. The structure of $\mathcal{D}$ consists of an acyclic directed graph $G_{\mathcal{D}} = (N, E)$ with $N = \mathcal{X} \times \mathcal{T}$. Thus there will be one node of the form X_i^t for each $t \in \mathcal{T}$ and each $i \in \{1, 2, \ldots, n\}$. The node X_i^t is to be viewed as a random variable that records the value assumed by the variable X_i at time t. The edge relation is derived by fixing the parenthood relation $PA : \mathcal{X} \rightarrow 2^{\mathcal{X}}$ over the system variables. Intuitively, $PA(X_i)$ is the set of system variables whose values at time t -probabilistically- influence the value assumed by X_i at time $t + 1$. This crucial structural information is to be obtained from the application at hand and will often be readily available.

The map PA will in turn induce the map $Pa : N \rightarrow 2^N$ given by: $Pa(X_i^t) = 0$ if $t = 0$. For $t > 0$, $X_j^{t'} \in Pa(X_i^t)$ iff $t' = t - 1$ and $X_j \in PA(X_i)$. The edge relation E is then given by: $(X_j^{t'}, X_i^t) \in E$ iff $X_j^{t'} \in Pa(X_i^t)$.

The dynamics of $\mathcal{D}$ is specified locally by assigning a *conditional probability table* (CPT) C_i^t to each node X_i^t. Suppose $Pa(X_i^t) = \{X_{j_1}^{t-1}, X_{j_2}^{t-1}, \ldots, X_{j_m}^{t-1}\}$. Then an

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 17–25, 2012.

entry in C_i^t will be of the form: $C_i^t(x_i \mid x_{j_1}, x_{j_2}, x_{j_m}) = p$. It says if the system is in a state at $t-1$ in which $X_{j_l} = x_{j_l}$ for $1 \leq l \leq m$, then the probability of $X_i = x_i$ being the case at t is p. As might be expected, it is required that $\sum_{x_i \in V} C_i^t(x_i \mid x_{j_1}, x_{j_2}, x_{j_m}) = 1$ for each $(x_{j_1}, x_{j_2}, \ldots, x_{j_m}) \in V^m$.

For the simple version of a DBN that we are considering, the CPTs of a system variable are assumed to be time invariant. In other words, for each i and each $t, t' \in \mathcal{T}$ we will have $C_i^t(x_i \mid x_{j_1}, x_{j_2}, x_{j_m}) = C_i^{t'}(x_i \mid x_{j_1}, x_{j_2}, x_{j_m})$. Consequently one can specify both the structure and the local dynamics in terms of two adjacent slices. This so called "2-Time slice Bayesian network" ("2-TBN") representation of DBNs is quite standard [4]. An example of such a DBN but which have "unrolled" for illustration is shown in Fig.1.

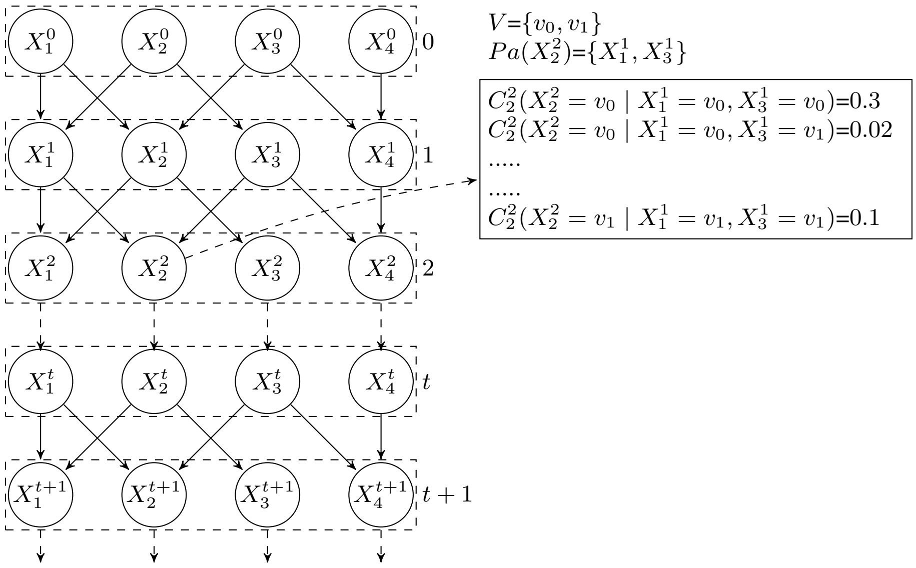

V={v_0, v_1}
$Pa(X_2^2)$={X_1^1, X_3^1}
$C_2^2(X_2^2 = v_0 \mid X_1^1 = v_0, X_3^1 = v_0)$=0.3
$C_2^2(X_2^2 = v_0 \mid X_1^1 = v_0, X_3^1 = v_1)$=0.02
.....
.....
$C_2^2(X_2^2 = v_1 \mid X_1^1 = v_1, X_3^1 = v_1)$=0.1
X_1^0 X_2^0 X_3^0 X_4^0 0
X_1^1 X_2^1 X_3^1 X_4^1 1
X_1^2 X_2^2 X_3^2 X_4^2 2
X_1^t X_2^t X_3^t X_4^t t
X_1^{t+1} X_2^{t+1} X_3^{t+1} X_4^{t+1} $t+1$

Fig. 1. Example of a DBN

Suppose $\mathcal{T} = \mathbb{N}$, the set of non-negative integers. Then the global dynamics of $\mathcal{D}$ can be described by a Markov chain. To bring this out, we first define $\hat{i} = \{j \mid X_j \in PA(X_i)\}$ to capture PA in terms of the corresponding indices. We let $\mathbf{s}_I$ will denote a vector of values over the index set $I \subseteq \{1, 2, \ldots, n\}$. It will be viewed as a map $\mathbf{s}_I : I \to V$. We will denote $\mathbf{s}_I(i)$ (with $i \in I$) as $\mathbf{s}_{I,i}$ or just $\mathbf{s}_i$ if I is clear from the context. If I is the full index set $\{1, 2, \ldots, n\}$, we will simply write $\mathbf{s}$.

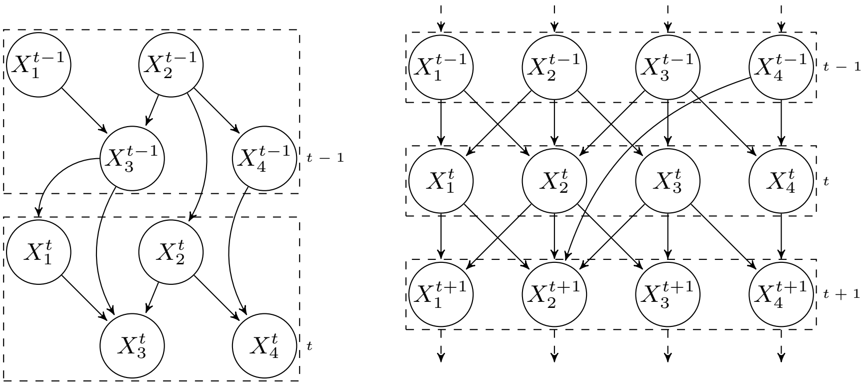
The Markov chain $M_{\mathcal{D}}$ induced by $\mathcal{D}$ can now be defined via:
$M_{\mathcal{D}} : V^n \times V^n \to [0, 1]$ where for each $\mathbf{s}, \mathbf{s}' \in V^n$ we have $M_{\mathcal{D}}(\mathbf{s}, \mathbf{s}') = \prod p_i$ where $p_i = C_i(\mathbf{s}'_i \mid \mathbf{s}_{\hat{i}})$ for each i. It is easy to check that $M_{\mathcal{D}}$ is indeed a Markov chain.

We note that $M_{\mathcal{D}}$ has potentially K^n states and K^{2n} transitions where $K = |V|$. In contrast, the size of $\mathcal{D}$ is essentially determined by the CPTs and their description will be at most of size $n \cdot K^d$ where $d = max\{|PA(X_i)|\}_{1 \leq i \leq n}$. Often d will be much smaller than n. In this sense the DBN is a succinct and factored representation of a large underlying Markov chain. Equally important, in many applications the transition probabilities of the chain may be obscure and inaccessible while the parenthood relationship between the variables as well as the CPT entries are easily identifiable.

In case $\mathcal{T} = \{0, 1, \ldots, T\}$ is a finite set, the global dynamics of $\mathcal{D}$ can still be be specified as a Markov chain $M_{\mathcal{D}}$. Its set of states will be $V^n \times \mathcal{T}$ and its transition relation given by:
(i) $M_{\mathcal{D}}((\mathbf{s}, t), (\mathbf{s}', t')) = 0$ if $t' \neq t{+}1$ or $t = t' = T$ and $\mathbf{s} \neq \mathbf{s}'$ (ii) $M_{\mathcal{D}}((\mathbf{s}, T), (\mathbf{s}, T)) = 1$ for every $\mathbf{s} \in V^n$. (iii) $M_{\mathcal{D}}((\mathbf{s}, t), (\mathbf{s}', t+1)) = \prod p_i$ if $t < T$ with $p_i = C_i(\mathbf{s}'_i \mid \mathbf{s}_{\hat{i}})$ for each i.

Thus every state of the form $(\mathbf{s}, t)$ with $t < T$ will be a transient state while every state of the form $(\mathbf{s}, T)$ will be an absorbing state.

There many variants of DBNs. Firstly, there could be dependencies between nodes belonging to the same time slice as shown in Fig 2(a). Secondly there could be dependencies across non-adjacent time-slices as shown in Fig 2(b). Some of the variables may be continuously valued. Yet another variation is where the CPTs for a system variable are time variant. In this case one must explicitly specify the CPT for each node instead of using a 2-TBN presentation. Last but not least, many of the state variable may be unobservable. The state of the system at any time point will be recorded as the values of the observable nodes. Apart from the CPTs specifying the local dynamics over the system variables, there will also be a probabilistic description of how the value of an observable variable is determined at a time point in terms of the values of its parent unobservable nodes. In fact it is this variant of a DBN that is typically used in AI applications [4]. We will however avoid this complication here. In what follows we will assume our DBNs to be as described above except that we will allow the CPTs corresponding each system variable to be time variant while the time domain is assumed to be finite.



(a) (b)

Fig. 2. Variants of DBNs

3 Probabilistic Inferencing Methods

The probability distribution $P(X_1^t, X_2^t, \ldots, X_n^t)$ describes the possible states of the system at time t. In other words, $P(\boldsymbol{X}^t = \boldsymbol{x})$ is the probability that the system will reach the state $\boldsymbol{x}$ at t. Starting from $P(\boldsymbol{X}^0)$ at time 0, given by $P(\boldsymbol{X}^0 = \boldsymbol{x}) = \prod_i C_i^0(\boldsymbol{x}_i)$, one would like to compute $P(X_1^t, \ldots, X_n^t)$ for a given t. As before, we will use $\hat{i} = \{j \mid X_j \in PA(X_i)\}$ to capture the set of indices of the parents of X_i and $V_{\hat{i}}$ will denote the tuple of values defined by $\hat{i}$.

We can use the CPTs to inductively compute this:

$$P(\boldsymbol{X}^t = \boldsymbol{x}) = \sum_{\boldsymbol{u}} P(\boldsymbol{X}^{t-1} = \boldsymbol{u})\Big(\prod_i C_i^t(\boldsymbol{x}_i \mid \boldsymbol{u}_{\hat{i}})\Big) \tag{1}$$

with $\boldsymbol{u}$ ranging over V^n.

Since $|V| = K$, the number of possible states at t is K^n. Hence explicitly computing and maintaining the probability distributions is feasible only if n is small or if the underlying graph of the DBN falls apart into many disjoint components. Neither restriction is realistic and hence one needs approximate ways to maintain $P(\boldsymbol{X}^t)$ compactly and compute it efficiently.

In fact, in most applications what one is interested in is the computation of the marginal distribution M_i^t of variable X_i. We shall view M_i^t to be a map $M_i^t : V \to [0, 1]$ satisfying $\sum_{v \in V} M_i^t(v) = 1$. Intuitively, $M_i^t(v)$ is the probability of X_i assuming the value v at time t. It is given by: $M_i^t(v) = \sum_{\mathbf{x}, \mathbf{x}(i)=v} \mathrm{P}(X_j^t = \mathbf{x}(j) \mid 1 \leq j \leq n)$.

Two interesting algorithms have been proposed for computing and maintaining these marginal probability distributions approximately [5,6]. Such approximate distributions are usually called *belief states* and denoted by B, B^t etc. In Boyen-Koller algorithm (BK, for short) [5], a belief state is maintained compactly as a product of the probability distributions of independent clusters of variables. This belief state is then propagated *exactly* at each step through the CPTs. Then the new belief state is compacted again into a product of the probability distributions of the clusters. Consequently the exact propagation step of the algorithm can become infeasible for large clusters. To get around this, the factored frontier algorithm (FF, for short) maintains a belief state as a product of the marginal distributions of the individual variables.

FF computes inductively a sequence B^t of marginals as:

- $B_i^0(v) = C_i^0(v)$,
- $B_i^t(v) = \sum_{\boldsymbol{u} \in V_{\hat{i}}} [\prod_{j \in \hat{i}} B_j^{t-1}(\boldsymbol{u}_j)] C_i^t(v \mid \boldsymbol{u})$.

Both BK and FF algorithm can sometimes incur significant errors due to way they maintain and propagate the global probability distributions. To handle this the hybrid factored frontier algorithm (HFF, for short) was proposed in [7]. HFF attempts to reduce the error made by FF by maintaining the current belief state as a hybrid entity; for a small number of global states called *spikes*, their current probabilities are maintained. The probability distribution over the remaining states is represented, as in FF, as a product of the marginal probability distributions. HFF has been shown [7] to be scalable and efficient with reduced errors. It may be viewed as a parametrized extension to FF where σ, the number of spikes to be maintained is the parameter. When $\sigma = 0$ we get FF whereas $\sigma =\mid V^n \mid$ corresponds to the exact inferencing algorithm. It turns out that that additional complexity of HFF (over that of FF) is only quadratic in σ and the accuracy increases monotonically as σ increases.

4 Probabilistic Model Checking on the DBNs

DBNs can be subjected to probabilistic model checking [8,9,10]. To illustrate this, we shall describe a simple probabilistic temporal logic and construct an approximate model checking procedure based on FF.

In our temporal logic, the atomic propositions will be of the form $(i, v)\#r$ with $\# \in \{\leq, \geq\}$ and r is a rational number in $[0, 1]$. The proposition $(i, v) \geq r$, if asserted at time point t, says that $M_i^t(v) \geq r$; similarly for $(i, v) \leq r$.

The formulas of our logic termed PBL (probabilistic bounded LTL) is then given by: (i) Every atomic proposition is a formula. (ii) If φ and φ' are formulas then so are $\sim \varphi$ and $\varphi \vee \varphi'$. (iii) If φ and φ' are formulas then so are $\boldsymbol{O}(\varphi)$ and $\varphi \boldsymbol{U} \varphi'$. Thus probability enters the logic solely at the level of atomic propositions.

The derived propositional connectives such as $\wedge, \supset, \equiv$ etc. and the temporal connectives $\boldsymbol{F}$ ("sometime from now") and $\boldsymbol{G}$ ("always from now") are defined in the usual way.

We assume we are dealing with a DBN $\mathcal{D}$ whose time domain is $\{0, 1, \ldots, T\}$ and whose CPTs for each system variable can vary over time. The formulas are interpreted over the sequence of marginal probability distribution vectors $\sigma = \mathbf{s}_0\mathbf{s}_1 \ldots \mathbf{s}_T$ generated by $\mathcal{D}$. In other words, for $0 \leq t \leq T$, $\mathbf{s}_t = (M_1^t, M_2^t, \ldots, M_n^t)$. Consequently $\mathbf{s}_t(i) = M_i^t$ for $1 \leq i \leq n$. We also let $\sigma(t) = \mathbf{s}_t$ for $0 \leq t \leq T$. We now define the notion of $\sigma(t) \models \varphi$ inductively:

- $\sigma(t) \models (i, v) \geq r$ iff $M_i^t(v) \geq r$. Similarly
  $\sigma(t) \models (i, v) \leq r$ iff $M_i^t(v) \leq r$.
- The propositional connectives $\sim$ and $\vee$ are interpreted in the usual way.
- $\sigma(t) \models \boldsymbol{O}(\varphi)$ iff $\sigma(t+1) \models \varphi$.
- $\sigma(t) \models \varphi \boldsymbol{U} \varphi'$ iff there exists $t \leq t' \leq T$ such that $\sigma(t') \models \varphi'$ and for every t'' with $t \leq t'' < t'$, $\sigma(t'') \models \varphi$.

We say that the DBN $\mathcal{D}$ meets the specification φ and this is denoted as $\mathcal{D} \models \varphi$ iff $\sigma(0) \models \varphi$. The model checking problem is, given $\mathcal{D}$ and φ, to determine whether or not $\mathcal{D} \models \varphi$. To solve this problem, we begin by letting $SF(\varphi)$ denote the set of subformulas of φ. Since φ will remain fixed we will write below SF instead of $SF(\varphi)$.

The main step is to construct a labeling function st which assigns to each formula $\varphi' \in SF$ a subset of $\{\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_T\}$ denoted $st(\varphi')$. After the labeling process is complete, we declare $\mathcal{D} \models \varphi$ just in case $\mathbf{s}_0 \in st(\varphi)$. Starting with the atomic propositions, the labeling algorithm goes through members of SF in ascending order in terms of their structural complexity. Thus φ' will be treated before $\sim \varphi'$ is treated and both φ' and φ'' will be treated before $\varphi' \boldsymbol{U} \varphi''$ is treated and so on.

Let $\varphi' \in SF(\varphi)$. Then:

- If $\varphi' = A$ -where A is an atomic proposition- then $\mathbf{s}_t \in st(A)$ iff $\sigma(t) \models A$. We run the DBN inference algorithm (say, FF) to determine this. In other words, $\mathbf{s}_t \in st(A)$ iff $B_i^t(v) \geq r$ where $A = (i, v) \geq r$ and B_i^t is the marginal distribution of X_i^t computed by the inference algorithm. Similarly $\mathbf{s}_t \in st(A)$ iff $B_i^t(v) \leq r$ in case $A = (i, v) \leq r$.
- If $\varphi' = \sim \varphi''$ then $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_t \notin st(\varphi'')$.

- If $\varphi' = \varphi_1 \vee \varphi_2$ then $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_t \in st(\varphi_1)$ or $\mathbf{s}_t \in st(\varphi_2)$.
- Suppose $\varphi' = \boldsymbol{O}(\varphi'')$. Then $\mathbf{s}_T \notin st(\varphi')$. Further, for $0 \leq t < T$, $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_{t+1} \in st(\varphi'')$.
- Suppose $\varphi' = \varphi_1 \boldsymbol{U}\, \varphi_2$. Then we decide whether or not $\mathbf{s}_t \in st(\varphi')$ by starting with $t = T$ and then treating decreasing values of t. Firstly $\mathbf{s}_T \in st(\varphi')$ iff $\mathbf{s}_T \in st(\varphi_2)$. Next suppose $t < T$ and we have already decided whether or not $\mathbf{s}_{t'} \in st(\varphi')$ for $t < t' \leq T$. Then $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_t \in st(\varphi_2)$ or $\mathbf{s}_t \in st(\varphi_1)$ and $\mathbf{s}_{t+1} \in st(\varphi')$.

$\varphi' = \boldsymbol{F}(\varphi'')$ and $\varphi' = \boldsymbol{G}(\varphi'')$ can be handled directly. As in the case of $\boldsymbol{U}$, we start with $t = T$ and consider decreasing values of t:

- Suppose $\varphi' = \boldsymbol{F}(\varphi'')$. Then $\mathbf{s}_T \in st(\varphi')$ iff $\mathbf{s}_T \in st(\varphi'')$. For $t < T$, $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_t \in st(\varphi'')$ or $\mathbf{s}_{t+1} \in st(\varphi')$.
- Suppose $\varphi' = \boldsymbol{G}(\varphi'')$. Then $\mathbf{s}_T \in st(\varphi')$ iff $\mathbf{s}_T \in st(\varphi'')$. For $t < T$, $\mathbf{s}_t \in st(\varphi')$ iff $\mathbf{s}_t \in st(\varphi'')$ and $\mathbf{s}_{t+1} \in st(\varphi')$.

Due to the fact the model checking procedure just needs to treat one model which is a finite sequence, it is a very simple procedure. Its time complexity is linear in the size of the formula φ.

5 An Application: DBN Approximations of the ODEs Based Dynamics of Biopathways

Biological pathways are usually described by a network of biochemical reactions. The dynamics of these reaction networks can be modeled and analyzed as a set of Ordinary Differential Equations (ODEs); one equation of the form $\frac{dy}{dt} = f(\mathbf{y}, \mathbf{r})$ for each molecular species y, with f describing the kinetics of the reactions that produce and consume y, while $\mathbf{y}$ is the set (vector) of molecular species taking part in these reactions and $\mathbf{r}$ are the rate constants associated with these reactions. These ODEs will be nonlinear due to the kinetic laws governing the reactions and high-dimensional due to the large number of molecular agents involved in the pathway. Hence closed form solutions will not be obtainable. Further many of the rate constants appearing in the ODEs will be unknown. As a result, one must resort to large scale numerical simulations to perform analysis tasks such as parameter estimation and sensitivity analysis. In addition, only a limited amount of noisy data of limited precision will be available for carrying out model calibration (i.e. parameter estimation) and model validation. Guided by these considerations a method for approximating the ODE dynamics of biological pathways as a DBN was developed in [11].

The first step in this approximation procedure is to discretize the time domain. For biopathways, experimental data will be available only for a few time points with the value measured at the final time point typically signifying the steady state value. Hence we assume the dynamics is of interest only for discrete time points and, furthermore, only up to a maximal time point. We denote these time points as $\{0, 1, \ldots, T\}$. Next we assume that the values of the variables can be observed with only finite precision and accordingly partition the range of each variable y_i of the ODE into a set of intervals $\mathbf{I}_i$ ($\mathbf{I}_j$). The initial values as well as the parameters of the ODE system are assumed to be

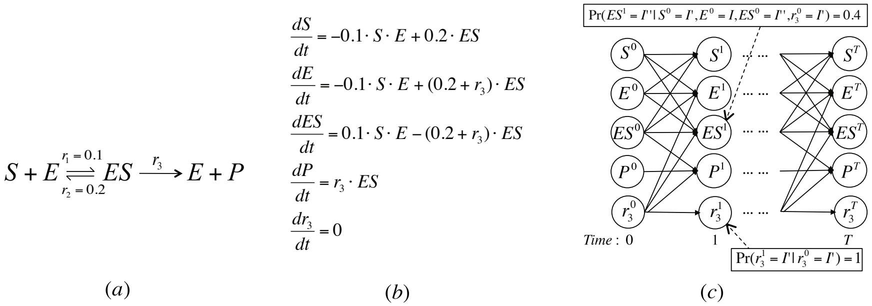

$S + E \underset{r_2 = 0.2}{\overset{r_1 = 0.1}{\rightleftharpoons}} ES \xrightarrow{r_3} E + P$
$\frac{dS}{dt} = -0.1 \cdot S \cdot E + 0.2 \cdot ES$
$\frac{dE}{dt} = -0.1 \cdot S \cdot E + (0.2 + r_3) \cdot ES$
$\frac{dES}{dt} = 0.1 \cdot S \cdot E - (0.2 + r_3) \cdot ES$
$\frac{dP}{dt} = r_3 \cdot ES$
$\frac{dr_3}{dt} = 0$
$\Pr(ES^1 = I' \mid S^0 = I', E^0 = I, ES^0 = I'', r_3^0 = I') = 0.4$
S^0 E^0 ES^0 P^0 r_3^0
S^1 E^1 ES^1 P^1 r_3^1
S^T E^T ES^T P^T r_3^T
Time : 0
1
T
$\Pr(r_3^1 = I' \mid r_3^0 = I') = 1$
(a)
(b)
(c)

Fig. 3. (a) The enzyme catalytic reaction network. (b) The ODE model. (c) The DBN approximation.

distributions (usually uniform) over certain intervals. We then sample the initial states of the system many times [11] and generate a trajectory by numerical integration for each sampled initial state. The resulting set of trajectories is then treated as an approximation of the dynamics of ODE system.

Unknown rate constants are handled as additional variables. We assume that the minimum and maximum values of these unknown rate constant variables are known. We then partition these ranges of values also into a finite numbers of intervals, and fix a uniform distribution over *all* the intervals. For each such variable r_j we add the equation $\frac{dr_j}{dt} = 0$ to the system of ODEs. This will reflect the fact that once the initial value of a rate constant has been sampled, this value will not change during the process of generating a trajectory. Naturally, different trajectories can have different initial values.

A key idea is to compactly store the generated set of trajectories as a DBN. This is achieved by exploiting the network structure and by simple counting. First we specify one random variable Y_i for each variable y_i of the ODE model

For each unknown rate constant r_j , we add one random variable R_j. The node Y_k^{t-1} will be in $Pa(Y_i^t)$ iff $k = i$ or y_k appears in the equation for y_i. Further, the node R_j^{t-1} will be in $Pa(Y_i^t)$ iff r_j appears in the equation for y_i. On the other hand R_j^{t-1} will be the only parent of the node R_j^t.

Suppose $Pa(Y_i^t) = \{Z_1^{t-1}, Z_2^{t-1}, \ldots, Z_k^{t-1}\}$. Then a CPT entry of the form $C_i^t(I \mid I_1, I_2, \ldots, I_k) = p$ says that p is the probability of the value of y_i falling in the interval I at time t, given that the value of Z_j was in I_j for $1 \leq j \leq k$. The probability p is calculated through simple counting. Suppose N is the number of generated trajectories. We record, for how many of these trajectories, the value of Z_j falls in the interval I_j simultaneously for each $j \in \{1, 2, \ldots, k\}$. Suppose this number is J. We then determine for how many of these J trajectories, the value of Y_i falls in the interval I at time t. If this number is J' then p is set to be $\frac{J'}{J}$.

If r_j is an unknown rate constant, in the CPT of R_j^t we will have $P(R_j^t = I \mid R_j^{t-1} = I') = 1$ if $I = I'$ and $P(R_j^t = I \mid R_j^{t-1} = I') = 0$ otherwise. This is because the sampled initial value of r_j does not change during numerical integration. Suppose r_j appears on the right hand side of the equation for y_i and $Pa(Y_i^t) = \{Z_1^{t-1}, Z_2^{t-1}, \ldots, Z_\ell^{t-1}\}$ with $Z_\ell^{t-1} = R_j^{t-1}$. Then for each choice of interval values for nodes other than R_j^{t-1} in $Pa(Y_i^t)$ and for each choice of interval value $\widehat{I}$ for r_j

there will be an entry in the CPT of Y_i^t of the form $P(y_i^t = I \mid Z_1^{t-1} = I_1, Z_2^{t-1} = I_2, \ldots, R_j^{t-1} = \widehat{I}) = p$. This is so since we will sample for all possible initial interval values for r_j. In this sense the CPTs record the approximated dynamics for all possible combinations of interval values for the unknown rate constants. The main ideas of this construction are illustrated in Fig.3. In this example we have assumed that r_3 is the only unknown rate constant.

After building this DBN, we use a Bayesian inference based technique to perform parameter estimation to complete the construction of the model (the details can be found in [11]). This will result in a calibrated DBN in which each unknown rate constant will have a specific interval value assigned to it. The one time cost of constructing the DBN can be easily recovered through the substantial gains obtained in doing parameter estimation and sensitivity analysis [11]. This method can cope with large biochemical networks with many unknown parameters. It has been used to uncover new biological facts about the complement system [12] where the starting point was a ODE based model with 71 unknown parameters. We have shown in [8] how a GPU based implementation of the approximation procedure can considerably extend the scalability of this approach. We have also shown how interesting properties of the dynamics of biological systems can be verified via probabilistic model checking along the lines sketched in the previous section (using FF).

6 Conclusion

Dynamic Bayesian networks are a formalism for representing complex stochastic dynamical systems in a compact and natural way. Their exact analysis is difficult. However efficient approximate inferencing algorithms are available using which one can also construct -approximate- probabilistic model checking procedures. We feel that they can play an important role in the modeling and analysis of biochemical networks. Our own work in which we have used DBNs to approximate the ODEs based dynamics of signaling paths supports this. A new avenue to explore in this connection will be to extract DBN models of biochemical networks from descriptions based on rule based languages such as kappa [13] and Bionetgen [14] as well as the chemical master equation [15]. In these formalism the underlying dynamic model is a Continuous Time Markov chain (CTMC). However the structure and locality of the interactions in the biochemical network under study should readily lend itself to a semantics based on DBNs. Finally, verification methods for DBNs based on Bayesian statistical model checking [16] are worth exploring further.

Acknowledgements. The research reported here has been partially supported by the Singapore Ministry of Education grant MOE2011-T2-2-12.

References

1. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6, 512–535 (1994)
2. Baier, C., Katoen, J.: Principles of model checking. The MIT Press (2008)

3. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
4. Koller, D., Friedman, N.: Probabilistic Graphical Models - Principles and Techniques. MIT Press (2009)
5. Boyen, X., Koller, D.: Tractable Inference for Complex Stochastic Processes. In: Proc. 14th Int. Conf. Uncertainty in Artificial Intelligence (UAI 1998), pp. 33–42 (1998)
6. Murphy, K.P., Weiss, Y.: The Factored Frontier Algorithm for Approximate Inference in DBNs. In: Proc. 17th Int. Conf. Uncertainty in Artificial Intelligence (UAI 2001), pp. 378–385 (2001)
7. Palaniappan, S.K., Akshay, S., Genest, B., Thiagarajan, P.S.: A Hybrid Factored Frontier Algorithm for Dynamic Bayesian Network Models of Biopathways. In: Proc. 9th Int. Conf. on Computational Methods in Systems Biology (CMSB 2011), pp. 35–44 (2011)
8. Liu, B., Hagiescu, A., Palaniappan, S.K., Chattopadhyay, B., Cui, Z., Wong, W., Thiagarajan, P.S.: Approximate probabilistic analysis of biopathway dynamics. Bioinformatics 28(11), 1508–1516 (2012)
9. Langmead, C., Jha, S., Clarke, E.: Temporal Logics as Query Languages for Dynamic Bayesian Networks: Application to D. Melanogaster Embryo Development. Technical report, Carnegie Mellon University (2006)
10. Langmead, C.J.: Generalized queries and bayesian statistical model checking in dynamic bayesian networks: Application to personalized medicine. In: Proc. 8th Ann. Intnl Conf. on Comput. Sys. Bioinf (CSB), pp. 201–212 (2009)
11. Liu, B., Hsu, D., Thiagarajan, P.S.: Probabilistic Approximations of ODEs based Bio-Pathway Dynamics. Theor. Comput. Sci. 412, 2188–2206 (2011)
12. Liu, B., Zhang, J., Tan, P.Y., Hsu, D., Blom, A.M., Leong, B., Sethi, S., Ho, B., Ding, J.L., Thiagarajan, P.S.: A Computational and Experimental Study of the Regulatory Mechanisms of the Complement System. PLoS Comput. Biol. 7(1), e1001059 (2011)
13. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-Based Modelling of Cellular Signalling. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 17–41. Springer, Heidelberg (2007)
14. Faeder, J.R., Blinov, M.L., Goldstein, B., William, H.S.: Rule-based modeling of biochemical networks. Complexity 10, 22–41 (2005)
15. Henzinger, T.A., Mikeev, L., Mateescu, M., Wolf, V.: Hybrid numerical solution of the chemical master equation. In: Proceedings of the 8th International Conference on Computational Methods in Systems Biology, CMSB 2010, pp. 55–65. ACM, New York (2010)
16. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian Approach to Model Checking Biological Systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)

Interpolant Automata
(Invited Talk)

Andreas Podelski

University of Freiburg, Germany

Abstract. We will cover the principles underlying a recent approach to the verification of different classes of correctness properties (safety, termination) for different classes of programs (sequential, recursive, concurrent). The approach is based on the notion of interpolant automata. A Floyd-Hoare correctness proof of the correctness of a trace (i.e., a sequence of statements) consists of a sequence of assertions, the *interpolants* of the trace. The sequence can be constructed, e.g., by static analysis or by an SMT solver with interpolant generation. We use the interpolants as the states of an automaton which has a transition $\varphi \xrightarrow{a} \varphi'$ if the Hoare triple $\{\varphi\}a\{\varphi'\}$ is valid. The resulting *interpolant automaton* recognizes a language over the alphabet of statements. The language is a set of *correct traces*, i.e., traces that obey the given correctness specification of a given program. The program is proven correct if the regular language of its program traces is contained in a union of interpolant automata. The new proof method consists of accumulating interpolant automata until the inclusion holds. Checking the inclusion is comparable to finite-model checking (the finite model defines the set of program traces, the property defines the set of correct traces). Interpolant automata are a modular, succinct, and program-independent presentation of a correctness argument.

The talk is based on joint work with Matthias Heizmann and Jochen Hoenicke and on joint work with Azadeh Farzan and Zachary Kincaid.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, p. 26, 2012.

Approximating Deterministic Lattice Automata

Shulamit Halamish[*] and Orna Kupferman

Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
{lamit,orna}@cs.huji.ac.il

Abstract. Traditional automata accept or reject their input, and are therefore Boolean. *Lattice automata* generalize the traditional setting and map words to values taken from a lattice. In particular, in a fully-ordered lattice, the elements are $0, 1, \ldots, n-1$, ordered by the standard $\leq$ order. Lattice automata, and in particular lattice automata defined with respect to fully-ordered lattices, have interesting theoretical properties as well as applications in formal methods. *Minimal deterministic automata* capture the combinatorial nature and complexity of a formal language. Deterministic automata have many applications in practice.

In [13], we studied minimization of deterministic lattice automata. We proved that the problem is in general NP-complete, yet can be solved in polynomial time in the case the lattices are fully-ordered. The multi-valued setting makes it possible to combine reasoning about lattice automata with *approximation*. An approximating automaton may map a word to a range of values that are close enough, under some pre-defined distance metric, to its exact value. We study the problem of finding minimal approximating deterministic lattice automata defined with respect to fully-ordered lattices. We consider approximation by absolute distance, where an exact value x can be mapped to values in the range $[x-t, x+t]$, for an approximation factor t, as well as approximation by separation, where values are mapped into t classes. We prove that in both cases the problem is in general NP-complete, but point to special cases that can be solved in polynomial time.

1 Introduction

Novel applications of automata theory are based on *weighted automata*, which map an input word to a value from a semi-ring over a large domain [22,8]. The semi-ring used in the automata is often a *finite distributive lattice*. A lattice $\langle A, \leq \rangle$ is a partially ordered set in which every two elements $a, b \in A$ have a least upper bound (a join b) and a greatest lower bound (a meet b). In particular, in a *fully-ordered* lattice (a.k.a. *linearly-* or *totally-ordered* lattice: one in which $a \leq b$ or $b \leq a$ for all elements a and b in the lattice), join and meet correspond to max and min, respectively.

For example (see Figure 1), in the abstraction application, researchers use the lattice $\mathcal{L}_3$ of three fully-ordered values [3,23], as well as its generalization to $\mathcal{L}_n$ [7]. In query checking [6], the lattice elements are sets of formulas, ordered by the inclusion order, as in $\mathcal{L}_{2^{\{a,b,c\}}}$ [4]. When reasoning about inconsistent viewpoints, each viewpoint is Boolean, and their composition gives rise to products of the Boolean lattice, as in $\mathcal{L}_{2,2}$ [9,14]. In addition, when specifying prioritized properties of systems, one uses lattices

[*] Supported in part by the Israeli Ministry of Science and Technology.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 27–41, 2012.

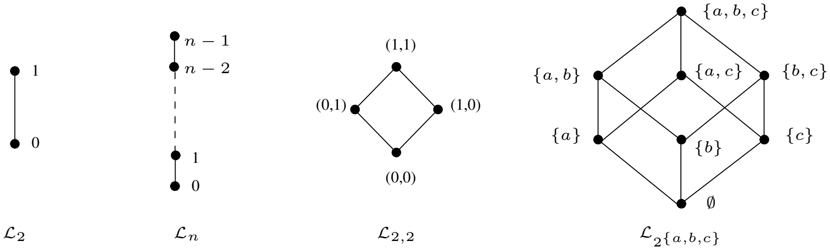


Fig. 1. Some lattices

in order to specify the priorities [1]. Finally, LTL has been extended to *latticed LTL* (LLTL, for short), where the atomic propositions can take lattice values. The semantics of LLTL is defined with respect to multi-valued Kripke structures and it can specify their quantitative properties [7,16].

In a *nondeterministic lattice automaton* on finite words (LNFA, for short) [16], each transition is associated with a *transition value*, which is a lattice element (intuitively indicating the truth of the statement “the transition exists”), and each state is associated with an *initial value* and an *acceptance value*, indicating the truth of the statements “the state is initial/accepting”, respectively. Each run r of an LNFA $\mathcal{A}$ has a value, which is the *meet* of the values of all the components of r: the initial value of the first state, the transition value of all the transitions taken along r, and the acceptance value of the last state. The value of a word w is then the *join* of the values of all the runs of $\mathcal{A}$ on w. Accordingly, an LNFA $\mathcal{A}$ over an alphabet Σ and lattice $\mathcal{L}$ induces an *$\mathcal{L}$-language* $L(\mathcal{A}) : \Sigma^* \to \mathcal{L}$. Note that traditional finite automata (NFAs) can be viewed as a special case of LNFAs over the lattice $\mathcal{L}_2$. In a *deterministic* lattice automaton on finite words (LDFA, for short), at most one state has an initial value that is not $\bot$ (the least lattice element), and for every state q and letter σ, at most one state q' is such that the value of the transition from q on σ to q' is not $\bot$. Thus, an LDFA $\mathcal{A}$ has at most one run whose value is not $\bot$ on each input word. The value of this run is the value of the word in the language of $\mathcal{A}$.

For example, the LDFA $\mathcal{A}$ in Figure 2 is over the alphabet $\Sigma = \{a, b, c, \#\}$ and the lattice $\mathcal{L} = \langle\{0, 1, 2, 3\}, \leq\rangle$. All states have acceptance value 3, and this is also the initial value of the single initial state. The $\mathcal{L}$-language of $\mathcal{A}$ is $L : \Sigma^* \to \mathcal{L}$ such that $L(\epsilon) = 3$, $L(a) = 3$, $L(a \cdot \#) = 1$, $L(b) = 1$, $L(b \cdot \#) = 1$, $L(c) = 3$, $L(c \cdot \#) = 2$, and $L(w) = 0$ for all other $w \in \Sigma^*$.

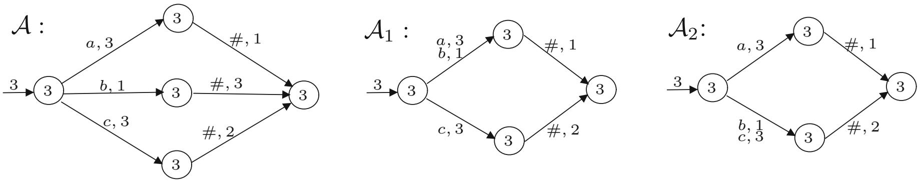


Fig. 2. A fully-ordered LDFA with two different minimal LDFAs

Minimal deterministic automata capture the combinatorial nature and complexity of formal languages. Beyond this theoretical importance, deterministic automata have many applications in practice. They are used in run-time monitoring, pattern recognition, and modeling systems. Thus, the minimization problem for deterministic automata is of great interest, both theoretically and in practice. For traditional Boolean automata on finite words, a minimization algorithm, based on the Myhill-Nerode right congruence on the set of words, generates in polynomial time a canonical minimal deterministic automaton [19,20]. A polynomial algorithm based on a right congruence is known also for deterministic weighted automata over the tropical semi-ring [18].

In [13], we studied the minimization problem for LDFAs. We showed that it is impossible to define a right congruence in the context of latticed languages, and that no canonical minimal LDFA exists. The difficulty is demonstrated in the LDFA $\mathcal{A}$ in Figure 2. It is not hard to see that both $\mathcal{A}_1$ and $\mathcal{A}_2$, which are not isomorphic, are minimal LDFAs equivalent to $\mathcal{A}$. The lack of a right congruence makes the minimization problem much more complicated than in the Boolean setting, and in fact also than in the setting of the tropical semi-ring. It is shown in [13] that the minimization problem for LDFAs is NP-complete in general. As good news, it is also shown in [13] that even though it is impossible to define a right congruence even when the LDFA is defined with respect to a fully-ordered lattice (indeed, the LDFA in Figure 2 is defined with respect to $\mathcal{L}_4$), it is possible to minimize such LDFAs in polynomial time.

The multi-valued setting makes it possible to combine reasoning about weighted and lattice automata with *approximation*. In this context, an approximating LDFA may map a word to a range of values that are close enough, under some pre-defined distance metric, to its exact value.

Approximations are widely used in computer science in cases where finding an exact solution is impossible or too complex. In the context of automata, approximation is used already in the Boolean setting: DFAs are used in order to approximate regular [24] and non-regular [10] languages. Applications of approximating DFAs include network security pattern matching, where one-sided errors are acceptable, and abstraction-refinement methods in formal methods, where DFAs that over- and under-approximate the concrete language of the system or the specification are used [17]. In the weighted setting, researchers used approximating automata in order to cope with the fact that not all weighted automata can be determinized [2,5]. For example, [2] introduces t-determinization: given a weighted automaton $\mathcal{A}$ and an approximation factor $t > 1$, constructs a deterministic weighted automaton $\mathcal{A}'$ such that for all words $w \in \Sigma^*$, it holds that $L(A)(w) \leq L(A')(w) \leq t \cdot L(A)(w)$. Approximation not only makes determinization possible but also leads to automata that are significantly smaller [2,5].

In this paper we study the approximation of lattice automata and the problem of finding minimal approximating LDFAs. We focus on LDFAs defined with respect to fully-ordered lattices. While it is possible to define distance metrics also in the setting of partially-ordered lattices, for example by using lattice chains, we find the notion of approximation cleaner in the setting of fully-ordered lattices. Also, since the minimization problem for LDFAs over partially ordered lattices is NP-hard already without approximations, we cannot expect the problem of finding a minimal approximating LDFA to be easier. Finally, as we discuss below, applications of approximated minimization exist already for fully-ordered lattices.

In fully-ordered lattices, there is a natural way to define the distance between two lattice elements: finite fully-ordered lattices are all isomorphic to $\mathcal{L}_n$, for some $n \geq 1$, and we define the distance between two elements a and b in the lattice to be $|a-b|$.[1] We refer to approximations with respect to this metric as *distance approximations*: Consider an integer $n \geq 1$, an approximation factor $0 \leq t \leq n-1$, and two LDFAs $\mathcal{A}$ and $\mathcal{A}'$ over $\mathcal{L}_n$. We say that $\mathcal{A}'$ *t-approximates* $\mathcal{A}$ iff for all words $w \in \Sigma^*$ we have that $|L(\mathcal{A}')(w) - L(\mathcal{A})(w)| \leq t$. That is, $\mathcal{A}'$ t-approximates $\mathcal{A}$ if it maps every word w to a value whose distance from $L(\mathcal{A})(w)$ is at most t. We first show that the problem of deciding whether $\mathcal{A}'$ t-approximates $\mathcal{A}$ is NLOGSPACE-complete. We then turn to the problem of finding a minimal LDFA that t-approximates a given LDFA. We study the corresponding decision problem, and then conclude about the search problem. It follows from [13] that the special case of finding a minimal 0-approximating LDFA can be solved in polynomial time. We show that when $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$, the problem is NP-complete, and in fact is even inapproximable. On the other hand, for $\lfloor \frac{n}{2} \rfloor \leq t \leq n-1$, the problem can be solved in constant time (simply since a t-approximating LDFA of size one always exists).

A different natural way to define approximations in a fully-ordered lattice involves the introduction of *separators*. Formally, a *t-separation* of $\mathcal{L}_n$, for $1 \leq t \leq n$, is a partition $\mathcal{P} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{t-1}\}$ of $\{0, 1, \ldots, n-1\}$ such that all the sets in the partition are not empty and contain only successive elements. An approximation by t-separation maps a set of successive values to a single value. Formally, consider an integer $n \geq 1$, an approxomation factor $1 \leq t \leq n$, an LDFA $\mathcal{A}$ over $\mathcal{L}_n$ and an LDFA $\mathcal{A}'$ over $\mathcal{L}_t$. We say that $\mathcal{A}'$ *t-separates* $\mathcal{A}$ iff there is a t-separation $\mathcal{P}$ of $\mathcal{L}_n$ such that for all words $w \in \Sigma^*$ we have that $L(\mathcal{A}')(w) = i$ iff $L(\mathcal{A})(w) \in \mathcal{P}_i$. For example, when $n = 10$ and $\mathcal{P} = \{\{0\}, \{1\}, \{2, \ldots, 9\}\}$, the LDFA $\mathcal{A}'$ may agree with $\mathcal{A}$ on all words that are mapped to 0 and 1, and map to 2 all words that are mapped by $\mathcal{A}$ to $\{2, \ldots, 9\}$. We first show that the problem of deciding whether $\mathcal{A}'$ t-separates $\mathcal{A}$ according to a given t-separation is NLOGSPACE-complete. We then turn to the problem of finding a minimal LDFA that t-separates a given LDFA. We show that the problem can be solved in polynomial time for a fixed t, and is NP-complete when t is a parameter to the problem.

Beyond the theoretical motivation for studying minimization of approximating LDFAs with respect to the two distance metrics, both metrics are useful in practice. Recall the use of fully-ordered lattices in the specification of prioritized properties of systems [1]. Consider an LDFA over $\mathcal{L}_n$ that corresponds to the specification. Assume that the (Boolean) language of all words that are assigned some value i has a large Myhill-Nerod index, thus a DFA for it, and hence also the LDFA, is big. It is often possible to approximate the assignment of priorities so that "problematic" values are no longer problematic and an LDFA for the specification is much smaller. Using the distance metric, the approximation may map a value x to values in $[x-t, x+t]$. Such a metric is

[1] Another popular isomorphic lattice uses the range $0, \ldots, 1$, with the elements being $0, \frac{1}{n}, \frac{2}{n}, \ldots$. We find it simpler to work with $\mathcal{L}_n$. Clearly, our results can be easily adjusted to all fully-order lattices.

useful when we tolerate a change of order between the prioritized properties but want to limit the range in which priorities are changed. Using separators, the approximation is not restricted to a certain range, but bounds the number of classes to which successive priorities can be mapped. Such a metric is useful when we care about the order of the prioritized properties and do not care about their values. As another example, recall the application of the three-value lattice $\mathcal{L}_3$ in abstraction. A 2-separation of $\mathcal{L}_3$ corresponds to either an under-approximation of the concrete system, in case the "unknown" value is grouped with "false", or to an over-approximation, in case "unknown" value is grouped with "true". Finally, both types of approximation may be useful when using LDFAs for the specification of quantitative properties.

Due to lack of space, some proofs are omitted in this version and can be found in the full version, in the authors' home pages.

2 Fully-Ordered Lattices and Lattice Automata

Let $\langle A, \leq \rangle$ be a partially ordered set, and let P be a subset of A. An element $a \in A$ is an *upper bound* on P if $a \geq b$ for all $b \in P$. Dually, a is a *lower bound* on P if $a \leq b$ for all $b \in P$. An element $a \in A$ is the *least element of* P if $a \in P$ and a is a lower bound on P. Dually, $a \in A$ is the *greatest element of* P if $a \in P$ and a is an upper bound on P. A partially ordered set $\langle A, \leq \rangle$ is a *lattice* if for every two elements $a, b \in A$ both the least upper bound and the greatest lower bound of $\{a, b\}$ exist, in which case they are denoted $a \vee b$ (a *join* b) and $a \wedge b$ (a *meet* b), respectively. For an integer $n \geq 1$, let $[n] = \{0, 1, \ldots, n-1\}$. The fully-ordered lattice (a.k.a. linearly- or totally-ordered lattice) with n elements is $\mathcal{L}_n = \langle [n], \leq \rangle$, where $\leq$ is the usual "less than" relation, thus $0 \leq 1 \leq \cdots \leq n-1$. For a set $P \subseteq [n]$ of elements, the least upper bound of P, namely the join of the elements in P, is $\max P$. The greatest lower bound of P, namely the meet of the elements in P, is $\min P$. In particular, the meet and join of $[n]$ are 0 and $n-1$, also referred to as $\bot$ and $\top$, respectively.

Consider a lattice $\mathcal{L} = \langle A, \leq \rangle$. For a set $\mathcal{X}$ of elements, an *$\mathcal{L}$-set over $\mathcal{X}$* is a function $S : \mathcal{X} \to A$ assigning to each element of $\mathcal{X}$ a value in A. It is convenient to think about $S(x)$ as the truth value of the statement "x is in S". We say that an $\mathcal{L}$-set S is *Boolean* if $S(x) \in \{\top, \bot\}$ for all $x \in \mathcal{X}$. In particular, all $\mathcal{L}_2$-sets are Boolean.

Consider a lattice $\mathcal{L} = \langle A, \leq \rangle$ and an alphabet Σ. An *$\mathcal{L}$-language* is an $\mathcal{L}$-set over Σ^*. Thus, an $\mathcal{L}$-language $L : \Sigma^* \to A$ assigns a value in A to each word over Σ.

A *deterministic lattice automaton* on finite words (LDFA) is $\mathcal{A} = \langle \mathcal{L}, \Sigma, Q, Q_0, \delta, F \rangle$, where $\mathcal{L}$ is a lattice, Σ is an alphabet, Q is a finite set of states, $Q_0 \in \mathcal{L}^Q$ is an $\mathcal{L}$-set of initial states, $\delta \in \mathcal{L}^{Q \times \Sigma \times Q}$ is an $\mathcal{L}$-transition-relation, and $F \in \mathcal{L}^Q$ is an $\mathcal{L}$-set of accepting states. The fact that $\mathcal{A}$ is deterministic is reflected in two conditions on Q_0 and δ. First, there is exactly one state $q \in Q$, called the *initial state* of $\mathcal{A}$, such that $Q_0(q) \neq \bot$. In addition, for every state $q \in Q$ and letter $\sigma \in \Sigma$, there is at most one state $q' \in Q$, called the *σ-destination* of q, such that $\delta(q, \sigma, q') \neq \bot$. The *run* of an LDFA on a word $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_l$ is a sequence $r = q_0, \ldots, q_l$ of $l+1$ states, where q_0 is the initial state of $\mathcal{A}$, and for all $1 \leq i \leq l$ it holds that q_i is the *σ_i-destination* of q_{i-1}. The *value* of w is $val(w) = Q_0(q_0) \wedge \bigwedge_{i=1}^{l} \delta(q_{i-1}, \sigma_i, q_i) \wedge F(q_l)$. The $\mathcal{L}$-language of $\mathcal{A}$, denoted $L(\mathcal{A})$, maps each word w to the value of its run in $\mathcal{A}$. Note that since $\mathcal{A}$ is deterministic, it has at most one run on w whose value is not $\bot$. In the special case of a lattice automaton over $\mathcal{L}_n$, the value of the run is simply the minimal value appearing in the run. An LDFA is *simple* if Q_0 and δ are Boolean. An example of an LDFA over a fully-ordered lattice can be found in Figure 2.

Note that traditional deterministic automata over finite words (DFA, for short) correspond to LDFA over the lattice $\mathcal{L}_2$. Indeed, over $\mathcal{L}_2$, a word is mapped by $L(\mathcal{A})$ to the value $\top$ iff the run on it uses only transitions with value $\top$ and its final state has acceptance value $\top$.

Analyzing the size of $\mathcal{A}$, one can refer to $|\mathcal{L}|$, $|Q|$, and $|\delta|$. Since the emphasize in this paper is on the size of the state space, we use $|\mathcal{A}|$ to refer to the size of its state space.

3 Approximation by Distances

In this section we study the problem of approximating LDFAs over fully-ordered lattices using the natural distance metric, in which the distance between two elements a and b is $|a - b|$. We first formally define approximation with respect to this metric.

Definition 1. *Consider an integer $n \geq 1$, an approximation factor $0 \leq t \leq n - 1$, and two LDFAs $\mathcal{A}$ and $\mathcal{A}'$ over $\mathcal{L}_n$. We say that $\mathcal{A}'$ t-*approximates *$\mathcal{A}$ iff for all words $w \in \Sigma^*$ we have that $|L(\mathcal{A}')(w) - L(\mathcal{A})(w)| \leq t$.*

Example 1. Figure 3 depicts an LDFA $\mathcal{A}$ over the lattice $\mathcal{L}_7$ and the alphabet $\Sigma = \{a_1, \ldots, a_6, \#\}$. It also depicts an LDFA $\mathcal{A}'$ that 1-approximates $\mathcal{A}$. One can see that $\mathcal{A}'$ agrees with $\mathcal{A}$ on the values of the words $\epsilon, a_1, \ldots, a_6, a_2\#, a_5\#$, while the values of the words $a_1\#, a_3\#, a_4\#, a_6\#$ differs by 1. All other words are mapped by both $\mathcal{A}$ and $\mathcal{A}'$ to 0. The approximation enables $\mathcal{A}'$ to merge the upper and lower three states that are reachable in one transition in $\mathcal{A}$.

Theorem 1. *Let $n \geq 1$ and $t \geq 0$. Given two LDFAs $\mathcal{A}$ and $\mathcal{A}'$ over $\mathcal{L}_n$, deciding whether $\mathcal{A}'$ t-approximates $\mathcal{A}$ is NLOGSPACE-complete.*

Proof. We start with the upper bound and rely on the fact that co-NLOGSPACE = NLOGSPACE [15]. To decide whether $\mathcal{A}'$ does not t-approximate $\mathcal{A}$, it is enough to find a word $w \in \Sigma^*$ such that $|L(A')(w) - L(A)(w)| > t$. In the full version we show that if such a word exists, then there also exists a word of size at most $n^2|\mathcal{A}'||\mathcal{A}|$ satisfying this

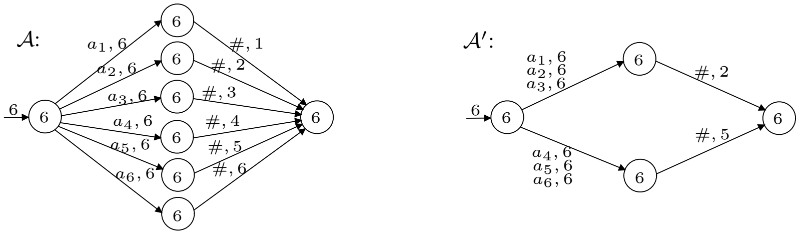


Fig. 3. An LDFA with a 1-approximation of it

condition[2]. Consequently, one can guess a word of size at most $n^2|\mathcal{A}'||\mathcal{A}|$, compute its value on both $\mathcal{A}$ and $\mathcal{A}'$ in logarithmic space, and return that $\mathcal{A}'$ does not t-approximate $\mathcal{A}$ iff $|L(A')(w) - L(A)(w)| > t$. Finally, the lower bound is by an easy reduction from the reachability problem in directed graphs. □

Approximation can lead to a significant reduction in the size of the automata. Formally, for all $t \geq 1$ there is a family of LDFAs for which t-approximation allows for an exponential reduction in the state space whereas $(t-1)$-approximation allows no reduction. To see this, note that applying 1-approximation on languages over $\mathcal{L}_2$ results in an LDFA with one state, no matter how complicated the original automaton was. This trivial example can be naturally extended to all $t \geq 1$ by considering lattices over $[n]$ and $\mathcal{L}_n$-languages that map the accepted and non-accepted words to two lattice values l and l', respectively, such that $|l - l'| = t$.

Motivated by the importance of generating small automata, our goal is to find an LDFA $\mathcal{A}'$ with a minimal number of states that t-approximates a given LDFA $\mathcal{A}$. We show that the problem is trivial when t is large enough. For the non-trivial interesting case where t is small with respect to the lattice, the problem becomes much more complicated, and we prove that it is NP-complete. However, for $t = 0$, where the problem coincides with minimization, it gets back to the easy side and can be solved in polynomial time [13].

Consider the corresponding decision problem: APRXLDFA=$\{\langle \mathcal{A}, n, t, k\rangle : \mathcal{A}$ is an LDFA over $\mathcal{L}_n$ with a t-approximating $\mathcal{A}'$ over $\mathcal{L}_n$ such that $|\mathcal{A}'| \leq k\}$. As we shall see, the complexity of APRXLDFA depends on the relation between n and t. We therefore study also the family of problems (n,t)-APRXLDFA, in which n and t are not parameters and rather are fixed. That is, (n,t)-APRXLDFA=$\{\langle \mathcal{A}, k\rangle : \mathcal{A}$ is an LDFA over $\mathcal{L}_n$ with a t-approximating $\mathcal{A}'$ over $\mathcal{L}_n$ such that $|\mathcal{A}'| \leq k\}$.

Theorem 2. *Let $n \geq 1$. The problem (n,t)-APRXLDFA:*

- *[13] Can be solved in polynomial time for $t = 0$.*
- *Is NP-complete for $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$.*
- *Can be solved in constant time for $t \geq \lfloor \frac{n}{2} \rfloor$.*

Proof. We start with the second case. For the upper bound, given $\mathcal{A}$ and k, a witness for their membership in (n,t)-APRXLDFA is an LDFA $\mathcal{A}'$ as required. Assuming $k \leq |\mathcal{A}|$ (otherwise, $\langle A, k\rangle$ clearly belongs to (n,t)-APRXLDFA), the size of $\mathcal{A}'$ is linear in the input. By Theorem 1, we can verify that $\mathcal{A}'$ t-approximates $\mathcal{A}$ in polynomial time.

For the lower bound, we show a polynomial-time reduction from the Minimal Automaton Identification problem (MAI, for short), proved to be NP-complete in [12]. The MAI problem refers to the minimal DFA whose language agrees with a set of observations.[3] Formally, let Σ be an alphabet of size two. A *data* is a set $D = \{(w_1, y_1), \ldots, (w_n, y_n)\}$, where for all $1 \leq i \leq n$, we have that $w_i \in \Sigma^*$ and $y_i \in \{0,1\}$, and $w_i \neq w_j$ for all $1 \leq i \neq j \leq n$. A DFA $\mathcal{A}$ *agrees* with D iff $L(\mathcal{A})(w_i) = y_i$ for all

[2] In fact, one can prove that there exists such a word of size at most $|\mathcal{A}'||\mathcal{A}|$, but this is not required for our purpose.

[3] The result in [12] is stated by means of Mealy machines. It can, however, be easily adapted to DFAs.

$1 \leq i \leq n$. Then, MAI $= \{\langle D, k\rangle : D$ is data, and there is a DFA $\mathcal{A}$ with $|\mathcal{A}| \leq k$ that agrees with $D\}$.

We now turn to describe the reduction, starting with the case n is even and $t = \frac{n}{2} - 1$. Note that for t to be at least 1, it must be that $n \geq 4$. Given an input $\langle D, k\rangle$ for MAI, let Σ be the alphabet and let $D = \{(w_1, y_1), \ldots, (w_m, y_m)\}$. We construct an LDFA $\mathcal{A} = \langle \mathcal{L}_n, Q, \Sigma, \delta, Q_0, F\rangle$ as described below.

- The states Q are defined as follows. Let P be the set of all prefixes of the words $w_1, \ldots, w_m$. Each prefix $w \in P$ induces a state q_w. Note that prefixes that are common to more than one word contribute one element to P, and therefore induce one state in Q. In addition, there is a state q_{sink}.
- For all $w \in P$ and $\sigma \in \Sigma$, if $w\sigma \in P$, then $\delta(q_w, \sigma, q_{w\sigma}) = \top$; Otherwise, $\delta(q_w, \sigma, q_{sink}) = \top$. In addition, $\delta(q_{sink}, \sigma, q_{sink}) = \top$ for all $\sigma \in \Sigma$.
- $Q_0(q_\epsilon) = \top$, and $Q_0(q) = \bot$ for all other states $q \in Q$.
- The acceptance values are defined as follows. Consider first the states $\{q_{w_1}, \ldots, q_{w_m}\}$ corresponding to the words appearing in D. For all $1 \leq i \leq m$, if $y_i = 1$, then $F(q_{w_i}) = \top$; Otherwise, $F(q_{w_i}) = \bot$. For all other $q \in Q$, we define $F(q) = \frac{n}{2}$.

Note that $\mathcal{A}$ is deterministic. Also, since the components of $\mathcal{A}$ are all of size polynomial in the data D, the reduction is polynomial.

Example 2. Let $n = 4$. Figure 4 depicts the LDFA $\mathcal{A}_D$ corresponding to the data $D = \{(aa, 1), (aab, 0), (ab, 0), (bba, 1), (bbb, 0)\}$ over $\Sigma = \{a, b\}$. All transitions described in the figure have the value $\top$. The states with acceptance value $\top$ correspond to words $w \in \{a, b\}^*$ such that $(w, 1) \in D$, and symmetrically, the states with acceptance value $\bot$ correspond to words $w \in \{a, b\}^*$ such that $(w, 0) \in D$. The states with acceptance value of 2 on the left correspond to the strict prefixes of the words in D. Finally, the rightmost state is q_{sink}, and its acceptance value is also 2.

Intuitively, the goal of $\mathcal{A}$ is to keep the information stored in D in a way that would enable to restore it from an LDFA that t-approximates $\mathcal{A}$. By mapping to 0 and $n-1$ and defining t to be strictly smaller than $\frac{n}{2}$, we ensure that the t-approximating LDFAs do not mix up words that are mapped in D to different values. This way, we can associate a t-approximation $\mathcal{A}'$ of $\mathcal{A}$ with a DFA $\mathcal{B}$ that agrees with D and vice versa.

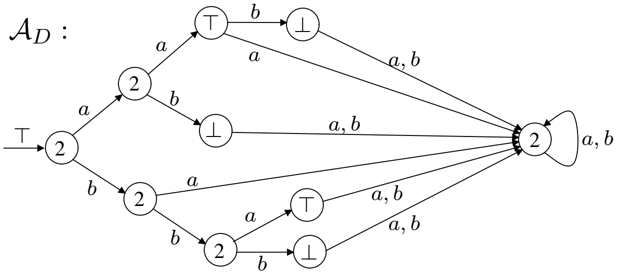


Fig. 4. The LDFA $\mathcal{A}_D$ induced by the data D

In the full version we prove formally that $\langle D, k\rangle \in$ MAI iff $\langle A, k\rangle \in (n,t)$-APRXLDFA. That is, there exists a DFA $\mathcal{B}$ with $|\mathcal{B}| \leq k$ that agrees with D iff there exists a t-approximating $\mathcal{A}'$ for $\mathcal{A}$ with $|\mathcal{A}'| \leq k$.

Now, recall that the reduction above assumes that n is even and $t = \frac{n}{2} - 1$. Hence, we still have to show that (n,t)-APRXLDFA is NP-hard for all $n \geq 1$ and $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$.

Let $n \geq 1$ and $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$. Since $2t+2$ is even and $t = \frac{2t+2}{2} - 1$, the reduction above shows that $(2t+2,t)$-APRXLDFA is NP-hard. In the full version, we show a polynomial-time reduction from $(2t+2,t)$-APRXLDFA to (n,t)-APRXLDFA, and conclude that (n,t)-APRXLDFA is NP-hard for all $n \geq 1$ and $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$.

We now turn to the third part of the theorem. Let $t \geq \lfloor \frac{n}{2} \rfloor$. A constant time algorithm that is given $\langle \mathcal{A}, k\rangle$ and decides whether $\langle \mathcal{A}, k\rangle \in (n,t)$-APRXLDFA can simply return "yes" for all inputs. In the full version we prove correctness by showing that every LDFA $\mathcal{A}$ over $\mathcal{L}_n$ has a t-approximating $\mathcal{A}'$ of size one. □

Going back to the problem APRXLDFA, we can now reduce (n,t)-APRXLDFA to APRXLDFA for some fixed $n \geq 1$ and $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$. By the second part of Theorem 2, we conclude with the following.

Theorem 3. *APRXLDFA is NP-complete.*

Remark 1. By Theorem 3, it is unlikely that there can ever be an efficient exact algorithm for APRXLDFA. A natural question to ask is whether the problem can be efficiently approximated. In [21], the authors show that the MAI problem, from which we have reduced, cannot be approximated within any polynomial. That is, assuming P$\neq$ NP, there does not exist a polynomial time algorithm that on a data input D can always return a DFA of size at most polynomially larger than *opt*, where *opt* is the smallest DFA that agrees with D. Therefore, the reduction described in the proof of Theorem 2 in fact gives a stronger result: APRXLDFA is inapproximable.

Theorems 2 and 3 study the complexity of deciding whether there is a t-approximating LDFA of size at most k. Below we discuss the corresponding search problem, of constructing a minimal LDFA. Consider the three cases stated in Theorem 2. For $t = 0$, the approximation problem coincides with minimization, and there is an algorithm generating a minimal LDFA in polynomial time [13]. For $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$, we can first perform a binary search to find the minimal k, and then verify a guessed LDFA of size k. Therefore, the problem of constructing a minimal LDFA belongs to the class FNP (of problems where the goal is to return a witness in an NP decision problem). Finally, for $\lfloor \frac{n}{2} \rfloor \leq t$, we have seen that a t-approximating LDFA can map all words to the value $\lfloor \frac{n}{2} \rfloor$, and a minimal one can do it with a single state.

4 Approximation by Separation

Approximation by distance poses a uniform requirement on the elements of the lattice. In this section we introduce and study another natural metric, based on lattice separation.

Definition 2. *Let $n \geq 1$ and $1 \leq t \leq n$. A t-separation of $\mathcal{L}_n$ is a partition $\mathcal{P} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{t-1}\}$ of $[n]$ into t non-empty sets such that each set contains only successive elements.*

Definition 3. *Consider an integer $n \geq 1$, an approximation factor $1 \leq t \leq n$, an LDFA $\mathcal{A}$ over $\mathcal{L}_n$ and an LDFA $\mathcal{A}'$ over $\mathcal{L}_t$. We say that $\mathcal{A}'$ t-separates $\mathcal{A}$ iff there is a t-separation $\mathcal{P}$ of $\mathcal{L}_n$ such that for all words $w \in \Sigma^*$ we have that $L(\mathcal{A}')(w) = i$ iff $L(\mathcal{A})(w) \in \mathcal{P}_i$.*

Intuitively, an approximation by t-separation maps a set of successive values to a single value. One can see that the quality of the approximation improves as t grows. Indeed, for $t = 1$ we have only one set containing all elements, allowing us to map all words to the same value. On the other hand, when $t = n$, each element constitutes a singleton set, and $L(\mathcal{A}') = L(\mathcal{A})$.

Example 3. Figure 5 depicts an LDFA $\mathcal{A}$ over the lattice $\mathcal{L}_7$ and the alphabet $\Sigma = \{a_1, \ldots, a_6, \#\}$. The LDFA $\mathcal{A}'$ over $\mathcal{L}_3$ 3-separates $\mathcal{A}$ with respect to the 3-separation $\mathcal{P}_0 = \{0, 1, 2, 3, 4\}$, $\mathcal{P}_1 = \{5\}$, and $\mathcal{P}_2 = \{6\}$. One can see that the words $\epsilon, a_1, \ldots, a_6$, and $a_6\#$ are mapped by $\mathcal{A}$ to 6 and by $\mathcal{A}'$ to 2, and that the word $a_5\#$ is mapped by $\mathcal{A}$ to 5 and by $\mathcal{A}'$ to 1. All other words are mapped by $\mathcal{A}$ to values taken from the set $\{0, 1, 2, 3, 4\}$, and by $\mathcal{A}'$ to 0.

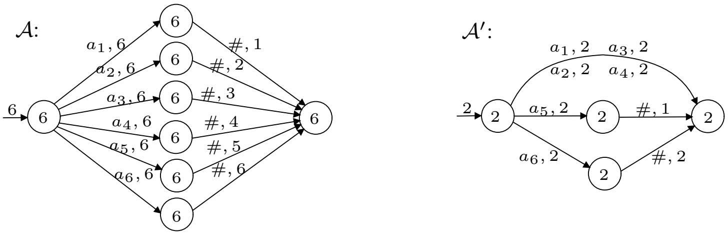


Fig. 5. An LDFA $\mathcal{A}$ with a 3-separating LDFA $\mathcal{A}'$

Theorem 4. *Let $n \geq 1$ and $1 \leq t \leq n$. Given an LDFA $\mathcal{A}$ over $\mathcal{L}_n$, an LDFA $\mathcal{A}'$ over $\mathcal{L}_t$, and a t-separation $\mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_{t-1}\}$ of $\mathcal{L}_n$, deciding whether $\mathcal{A}'$ t-separates $\mathcal{A}$ with respect to $\mathcal{P}$ is NLOGSPACE-complete.*

The proof of Theorem 4 uses the same considerations as in Theorem 1, and can be found in the full version.

We now turn to consider the problem of finding a minimal t-separating LDFA. As we have seen in Section 1, there are practical situations in which the input includes a specific t-separation of $\mathcal{L}_n$, and the goal is to find a minimal $\mathcal{A}'$ that t-separates $\mathcal{A}$ with respect to that separation. We show below that in such a case the problem can be solved in polynomial time.

Theorem 5. *Let $n \geq 1$ and $1 \leq t \leq n$. Given an LDFA $\mathcal{A}$ over $\mathcal{L}_n$ and a t-separation $\mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_{t-1}\}$ of $\mathcal{L}_n$, constructing a minimal LDFA $\mathcal{A}'$ over $\mathcal{L}_t$ that t-separates $\mathcal{A}$ with respect to $\mathcal{P}$ can be done in polynomial time.*

Proof. Let $\mathcal{B}$ be the LDFA over $\mathcal{L}_t$ obtained from $\mathcal{A}$ by replacing each value $j \in [n]$ appearing in $\mathcal{A}$ by the value $i \in [t]$ for which $j \in \mathcal{P}_i$. Let $\mathcal{A}'$ be a minimal LDFA equivalent to $\mathcal{B}$. By [13], $\mathcal{A}'$ can be constructed in time polynomial in $\mathcal{B}$, which can clearly be constructed in time polynomial in $\mathcal{A}$. We claim that $\mathcal{A}'$ is a minimal LDFA that t-separates $\mathcal{A}$ with respect to $\mathcal{P}$.

We first show that for all $w \in \Sigma^*$, we have that $L(\mathcal{A}')(w) = i$ iff $L(\mathcal{A})(w) \in \mathcal{P}_i$. Since $L(\mathcal{A}') = L(\mathcal{B})$, it is enough to show that for all $w \in \Sigma^*$ we have $L(\mathcal{B})(w) = i$ iff $L(\mathcal{A})(w) \in \mathcal{P}_i$. Let $w \in \Sigma^*$. It holds that $L(\mathcal{A})(w) \in \mathcal{P}_i$ iff at least one of the values read along the run of $\mathcal{A}$ on w belong to $\mathcal{P}_i$, and the other values belong to $\mathcal{P}_i, \ldots, \mathcal{P}_{t-1}$. This holds iff at least one of the values read along the run of $\mathcal{B}$ on w equals to i, and the other values are in $\{i, \ldots, t-1\}$. Finally, this holds iff $L(\mathcal{B})(w) = i$.

The fact that $\mathcal{A}'$ has a minimal number of states follows from the correctness of the minimization algorithm, and from the fact that all LDFAs that t-separate $\mathcal{A}$ with respect to a specific t-separation have the same language. □

We now turn to consider the case where the user does not provide a specific t-separation. That is, we are given an LDFA $\mathcal{A}$ over $\mathcal{L}_n$ and $t \geq 1$, and we seek an LDFA $\mathcal{A}'$ with a minimal number of states that t-separates $\mathcal{A}$. For example, as discussed in Section 1, when the user does not care about the way priorities are grouped, or, in the case of abstraction, when the user hesitates between working with an over- or an under-approximation, the t-separation is not given. Consider the corresponding decision problem SEPLDFA=$\{\langle \mathcal{A}, n, t, k\rangle : \mathcal{A}$ is an LDFA over $\mathcal{L}_n$ with a t-separating $\mathcal{A}'$ over $\mathcal{L}_t$ such that $|\mathcal{A}'| \leq k\}$. As in Section 3, we study also the family of problems (n,t)-SEPLDFA, in which n and t are not parameters and rather are fixed. That is, (n,t)-SEPLDFA=$\{\langle \mathcal{A}, k\rangle : \mathcal{A}$ is an LDFA over $\mathcal{L}_n$ with a t-separating $\mathcal{A}'$ over $\mathcal{L}_t$ such that $|\mathcal{A}'| \leq k\}$.

Theorem 6. *For all $n \geq 1$ and $t \geq 1$, the problem (n,t)-SEPLDFA can be solved in polynomial time.*

Proof. For fixed $n \geq 1$ and $t \geq 1$, there is a fixed number of possible t-separations of $\mathcal{L}_n$. Therefore, one can go over all t-separations, construct for each the corresponding minimal LDFA and return“yes” iff one of them has at most k states. By Theorem 5, each check, and therefore also the whole procedure, can be done in polynomial time. □

It is not hard to see that the algorithm above shows that the problem stays solvable in polynomial time also when n is not fixed and is a parameter to the problem. We now turn to study the problem SEPLDFA, in which n and t are parameters, and show that this problem is NP-complete. In Section 3, the NP-hardness of APRXLDFA follows directly from the hardness of (n,t)-APRXLDFA for $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$. Here, however, the problem (n,t)-SEPLDFA can be solved in polynomial time for all $n \geq 1$ and $t \geq 1$, so the fact that n and t are parameters is crucial.

Theorem 7. *The problem SEPLDFA is NP-complete.*

Proof. As in the case of (n,t)-APRXLDFA, membership in NP follows directly from Theorem 4.

For the lower bound, we show a polynomial time reduction from the NP-complete Maximum-Bisection problem on regular graphs (MBRG, for short) [11]. The Maximum Bisection of a graph $G = \langle V, E \rangle$, for V of an even size, is a partition of V into two equally sized sets that maximizes the number of edges between those sets. For regular graphs, in which all vertices have the same degree, the problem coincides with the problem of finding $T \subseteq V$ such that $|T| = \frac{|V|}{2}$ and $e(T)$ is minimal, where $e(T)$ is the number of edges among the vertices of T. Formally, $e(T) = |E \cap (T \times T)|$. The corresponding decision problem can therefore be formulated as MBRG = $\{\langle G, k \rangle :$ $G = \langle V, E \rangle$ is an undirected regular graph with an even number of vertices, such that there is a set $T \subseteq V$ with $|T| = \frac{|V|}{2}$ and $e(T) \leq k\}$.

For technical convenience, instead of reducing the MBRG problem to SEPLDFA directly, we go through the following variant of the problem: MBRG$' = \{\langle G, v, k \rangle :$ $G = \langle V, E \rangle$ is an undirected graph with an odd number of vertices, the vertex v touches all other vertices, and there is a set $T \subseteq V$ with $|T| = \frac{|V|-1}{2}$, $v \notin T$, and $e(T) \leq k\}$. In the full version, we describe an easy polynomial-time reduction from MBRG to MBRG$'$, proving that it is NP-hard.

We now turn to describe the reduction from MBRG$'$ to SEPLDFA. Let $\langle G, v, k \rangle$ be an input to MBRG$'$, where $G = \langle V, E \rangle$ is such that $V = \{v_1, \ldots, v_n, v\}$ and $E = \{e_1, \ldots, e_m\}$. Note that $n = |V| - 1$ and $m = |E|$. We construct an LDFA $\mathcal{A} = \langle \mathcal{L}_{n+1}, \Sigma, Q, \delta, Q_0, F \rangle$, where:

- $\Sigma = \{a_1, \ldots, a_m, b_1, \ldots, b_n, c_1, \ldots, c_n\}$. Thus, each edge $e_i \in E$ induces a letter a_i, and each vertex $v_i \in V \setminus \{v\}$ induces two letters, b_i and c_i.
- $Q = \{q_1, \ldots, q_m, q_1^1, q_1^2, q_2^1, q_2^2, \ldots, q_n^1, q_n^2, q_{init}, q_{fin}\}$. Thus, each edge $e_i \in E$ induces a state q_i, and each vertex $v_i \in V \setminus \{v\}$ induces two states q_i^1 and q_i^2. In addition there are two states q_{init} and q_{fin}.
- The transition relation is defined as follows.
  - For all $1 \leq i \leq m$, we have $\delta(q_{init}, a_i, q_i) = \top$.
  - For all $1 \leq i \leq m$ and $1 \leq j \leq n$, if e_i touches v_j, then $\delta(q_i, b_j, q_j^1) = \top$, otherwise $\delta(q_i, b_j, q_j^2) = \top$.
  - For all $1 \leq j \leq n$, we have $\delta(q_j^2, b_j, q_j^1) = \delta(q_j^1, b_j, q_j^1) = \top$.
  - For all $1 \leq j \leq n$, we have $\delta(q_j^1, c_j, q_{fin}) = \delta(q_j^2, c_j, q_{fin}) = \top$.
  - For all other $q, q' \in Q$ and $\sigma \in \Sigma$, we have $\delta(q, \sigma, q') = \bot$.
- $Q_0(q_{init}) = \top$, and $Q_0(q) = \bot$ for all other $q \in Q$.
- For all $1 \leq j \leq n$, we have $F(q_j^1) = j$ and $F(q_j^2) = j - 1$. For all other $q \in Q$, we have $F(q) = \top$.

Note that $\mathcal{A}$ is indeed deterministic, and has $m + 2n + 2$ states. Also, since the components of $\mathcal{A}$ are all of size polynomial in the input graph, the reduction is polynomial. We refer to the states q_i as “the left column” and to the states q_j^1 and q_j^2 as “the right column” (see Figure 6).

Example 4. Figure 6 depicts a graph G and its induced LDFA $\mathcal{A}_G$. All transitions described in the figure have the value $\top$. Due to space and clarity considerations, acceptance values have been omitted, as they are $\top$ for all states except for the states on the right column, where $F(q_j^1) = j$ and $F(q_j^2) = j - 1$ for all $1 \leq j \leq 4$. Also, we do not draw all edges in the middle, but a symbolic sample that demonstrates the idea.

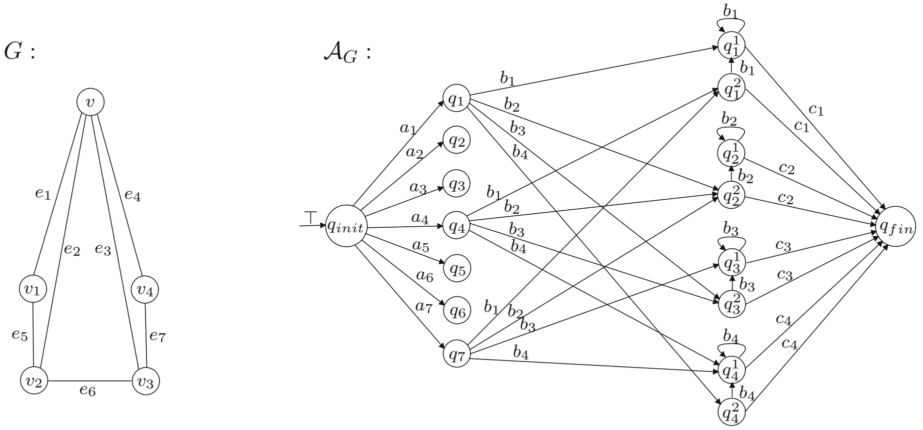


Fig. 6. A graph G and its induced LDFA $\mathcal{A}_G$

The idea behind the reduction is as follows. Let $t = \frac{n}{2} + 1$. A subset $T \subseteq V$ with $|T| = \frac{|V|-1}{2}$ induces a t-separation $\mathcal{P}$ of $\mathcal{L}_{n+1}$ in which $j-1$ and j are not in the same set iff $v_j \in T$. That is, $\mathcal{P}$ separates the lattice on the indices of the vertices of T. The LDFA $\mathcal{A}$ is constructed so that the size of the minimal LDFA $\mathcal{A}'$ that t-separates $\mathcal{A}$ with respect to $\mathcal{P}$ depends on $e(T)$. In order to see the dependency, consider the equivalence relation $\sim_T \subseteq E \times E$, where $e_1 \sim_T e_2$ iff e_1 and e_2 agree on touching the vertices of T. That is, for all $v \in T$ both e_1 and e_2 touch v or none of them does. We argue that the number of states in the left column of $\mathcal{A}'$ depends on $e(T)$, and the number of all other states does not depend on $e(T)$. To see this, consider first the right column of $\mathcal{A}'$. It is possible to merge the states q_j^1 and q_j^2 of $\mathcal{A}$ into one state $q_j^{1,2}$ in $\mathcal{A}'$ iff $j-1$ and j are not separated in $\mathcal{P}$, iff $v_j \notin T$. Therefore, the number of states in that column in $\mathcal{A}'$ depends only on $|T|$, and does not depend on $e(T)$. As for the left column, it is possible to merge states associated with edges in the same equivalence class of $\sim_T$. Indeed, since equivalent edges agree on touching the vertices of T, the states associated with them in $\mathcal{A}$ agree on out-going transitions: for j such that $v_j \in T$, transitions leaving with the letter b_j lead to the same state q_j^1 or q_j^2 already in $\mathcal{A}$. For j such that $v_j \notin T$, the two possible destination states q_j^1 and q_j^2 have already been merged in $\mathcal{A}'$, so transitions on b_j can all reach the same merged state $q_j^{1,2}$. Hence, the number of states on the left column in $\mathcal{A}'$ is equal to the number of equivalence classes. Finally, since each edge that touches two vertices of T induces an equivalence class of its own, the number of states in the left column depends on $e(T)$.

Formalizing the above intuitive explanation requires much care. For example, we should justify the fact that $\mathcal{A}'$ indeed maintains the left/right column structure. In the full version, we formally prove that there is a set $T \subseteq V$ such that $|T| = \frac{n}{2}$, $v \notin T$, and $e(T) \leq k$ iff there exists an LDFA $\mathcal{A}'$ over $\mathcal{L}_t$ with at most $k + 2n + 3$ states that t-separates $\mathcal{A}$, for $t = \frac{n}{2} + 1$. □

We note that although the SEPLDFA problem is generally NP-hard when n and t are given as parameters, there are still cases of parameters for which the problem can be solved in polynomial time. For example, consider the family of pairs $\langle n, t \rangle$ such that

$t = n - c$, for a fixed $c \geq 0$. The number of possible t-separations in these cases is fixed, so one can apply the same considerations as in Theorem 6 and solve the problem in polynomial time. Also, as in the case of t-approximation, the problem of returning the minimal LDFA that t-separates a given LDFA, for parameters t and n is in FNP. Finally, comparing Theorems 5 and 7 we get that the computational bottleneck of SEPLDFA is the need to find a good t-separation. Once such a separation is given, finding a minimal LDFA can be done in polynomial time.

5 Discussion

We studied the problem of finding a minimal LDFA that approximates a given LDFA defined with respect to a fully-ordered lattice. We showed that the complexity of the problem depends on the relation between the lattice size and the approximation factor and also depends on whether we view them as fixed.

Our complexity results may remind the reader of classic NP-complete problems, like vertex cover, where the goal is to decide the existence of some object ("the witness") of a certain size k. Typically, the existence of the required object can be decided in polynomial time for a fixed k, while the problem is NP-complete when k is a parameter to the problem. Despite of this resemblance, our setting here is very different, and the NP-hardness proofs are quite challenging. To see the difference, note that the factors we fix in (n, t)-APRXLDFA and (n, t)-SEPLDFA do not include the size of the witness! The latter is k, which is part of the input. Another difficulty we face follows from the fact that, unlike in classic combinatorial problems, where, say, the vertices in the graph are not ordered, here we have no symmetry between the elements. For example, when an LDFA reads a lattice value that is greater than the values read already, the accumulated value is not affected. On the other hand, reading a value that is smaller affects the accumulated value. Coping with non-symmetry involves the design of languages that take into an account the order induced by the lattice, making our reductions complicated. The complication is reflected also in the fact that when $t = 0$, it is possible to use this non-symmetry and come up with a polynomial algorithm. Finally, note that our "fixed-parameter" variants fix both n and t, and still (n, t)-APRXLDFA is NP-hard when $1 \leq t \leq \lfloor \frac{n}{2} \rfloor - 1$. It is not hard to see that our bounds and proofs stay valid also for the t-APRXLDFA and t-SEPLDFA variants, when only t is fixed. In particular, t-SEPLDFA can be solved in polynomial time.

As discussed in Section 1, distance metrics can be defined also for partially-ordered lattices. In our future work we plan to study minimization of approximating LDFAs defined with respect to such lattices. Working with partially-ordered lattices, more factors are added to the pictures. For example, we may look for possible linearizations of the partial order for approximation of fully-ordered lattices by partially-ordered ones.

References

1. Alur, R., Kanade, A., Weiss, G.: Ranking Automata and Games for Prioritized Requirements. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 240–253. Springer, Heidelberg (2008)

2. Aminof, B., Kupferman, O., Lampert, R.: Formal Analysis of Online Algorithms. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 213–227. Springer, Heidelberg (2011)
3. Bruns, G., Godefroid, P.: Model Checking Partial State Spaces with 3-Valued Temporal Logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
4. Bruns, G., Godefroid, P.: Temporal logic query checking. In: Proc.16th LICS, pp. 409–420 (2001)
5. Buchsbaum, A.L., Giancarlo, R., Westbrook, J.: An approximate determinization algorithm for weighted finite-state automata. Algorithmica 30(4), 503–526 (2001)
6. Chan, W.: Temporal-logic Queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
7. Chechik, M., Devereux, B., Gurfinkel, A.: Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 16–36. Springer, Heidelberg (2001)
8. Droste, M., Kuich, W., Vogler, H. (eds.): Handbook of Weighted Automata. Springer (2009)
9. Easterbrook, S., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: Proc. 23rd CSE, pp. 411–420 (2001)
10. Eisman, G., Ravikumar, B.: Approximate Recognition of Non-regular Languages by Finite Automata. In: Kanchanasut, K., Levy, J.-J. (eds.) ACSC 1995. LNCS, vol. 1023, pp. 219–228. Springer, Heidelberg (1995)
11. Feige, U., Karpinski, M., Langberg, M.: A note on approximating max-bisection on regular graphs. ECCC 7(43) (2000)
12. Mark Gold, E.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
13. Halamish, S., Kupferman, O.: Minimizing Deterministic Lattice Automata. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 199–213. Springer, Heidelberg (2011)
14. Hussain, A., Huth, M.: On model checking multiple hybrid views. Technical Report TR-2004-6, University of Cyprus (2004)
15. Immerman, N.: Nondeterministic space is closed under complement. Information and Computation 17, 935–938 (1988)
16. Kupferman, O., Lustig, Y.: Lattice Automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
17. Larsen, K.G., Thomsen, G.B.: A modal process logic. In: Proc. 3rd LICS (1988)
18. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)
19. Myhill, J.: Finite automata and the representation of events. Technical Report WADD TR-57-624, pp. 112–137. Wright Patterson AFB, Ohio (1957)
20. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
21. Pitt, L., Warmuth, M.K.: The minimum consistent DFA problem cannot be approximated within any polynomial. Journal of the ACM 40, 95–142 (1993)
22. ESF Network programme. Automata: from mathematics to applications (AutoMathA) (2010), http://www.esf.org/index.php?id=1789
23. 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)
24. Watson, B.W., Kourie, D.G., Strauss, T., Ngassam, E.K., Cleophas, L.G.: Efficient automata constructions and approximate automata. Int. J. Found. Comput. Sci. 19(1), 185–193 (2008)

Tight Bounds for the Determinisation and Complementation of Generalised Büchi Automata[*]

Sven Schewe and Thomas Varghese

University of Liverpool

Abstract. Generalised Büchi automata are Büchi automata with multiple accepting sets. They form a class of automata that naturally occurs, e.g., in the translation from LTL to ω-automata. In this paper, we extend current determinisation techniques for Büchi automata to generalised Büchi automata and prove that our determinisation is optimal. We show how our optimal determinisation technique can be used as a foundation for complementation and establish that the resulting complementation is tight. Moreover, we show how this connects the optimal determinisation and complementation techniques for ordinary Büchi automata.

1 Introduction

While finite automata over infinite words were first introduced in Büchi's decidability proof for the monadic second-order logic of one successor (S1S) [1], they are most widely used in model checking, realisability checking, and synthesis procedures for linear time temporal logic (LTL) [11].

Büchi automata are an adaptation of finite automata to languages over infinite words. They differ from finite automata only with respect to their acceptance condition: while finite runs of finite automata are accepting if a final state is visited at the end of the run, an infinite run of a Büchi automaton is accepting if a final state is visited (or a final transition is taken) infinitely many times. Although this might seem to suggest that automata manipulations for Büchi automata are equally simple as those for finite automata, this is unfortunately not the case. In particular, Büchi automata are not closed under determinisation. While a simple subset construction suffices to efficiently determinise finite automata [13], deterministic Büchi automata are strictly less expressive than nondeterministic Büchi automata. For example, deterministic Büchi (or generalised Büchi) automata cannot recognise the simple ω-regular language that consists of all infinite words that contain only finitely many *a*'s.

Determinisation therefore requires automata with more involved acceptance mechanisms [14,9,10,4,17], such as Muller's subset condition [8], Rabin's *pairs* condition [12] or its dual, the Streett condition [19], or the parity condition. Also, an $n^{\Omega(n)}$ lower bound for the determinisation of Büchi automata has been established [20] even if we allow for Muller objectives, which implies that a simple subset construction cannot suffice.

Rabin's extension of the correspondence between automata and monadic logic to the case of trees [12] built on McNaughton's doubly exponential determinisation construction [8], and Muller and Schupp's [9] efficient nondeterminisation technique

[*] This work was supported by the Engineering and Physical Science Research Council grant EP/H046623/1 'Synthesis and Verification in Markov Game Structures'.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 42–56, 2012.

for alternating tree automata is closely linked to the determinisation of nondeterministic word automata. Safra was the first to introduce singly-exponential determinisation constructions for Büchi [14] and Streett [15] automata and current determinisation techniques [10,17] build on Safra's work. For instance, Schewe's determinisation technique for Büchi automata [17] builds on Safra's [14] and Piterman's [10] determinisation procedures using a separation of concern, where the main acceptance mechanism, represented by *history trees*, is separated from the formal acceptance condition, e.g., a Rabin or parity condition. History trees can be seen as a simplification of Safra trees [14]. In a nutshell, they represent a family of breakpoint constructions; sufficiently many to identify an accepting run, and sufficiently few to be concise.

The standard translation from LTL to ω-automata [3] goes to *generalised* Büchi automata, which have multiple accepting sets and require that a final state from each accepting set is visited infinitely many time. There are several ways to determinise a generalised Büchi automaton with n states and k accepting sets. One could start with translating the resulting generalised Büchi automaton first to an ordinary nondeterministic Büchi automaton with nk states and a single accepting set, resulting in a determinisation complexity of roughly $(nk)^{O(nk)}$ states, or one could treat it as a Streett automaton, which is equally expensive and has a more complex determinisation construction.

Schewe's determinisation procedure [17] proves to be an easy target for generalisation, because it separates the acceptance mechanism from the accepting condition. To extend this technique from ordinary to generalised Büchi, it suffices to apply a round-robin approach to all breakpoints under consideration. That is, each subset is enriched by a natural number identifying the accepting set, for which we currently seek for the following breakpoint. Each time a breakpoint is reached, we turn to the next accepting set. Note that this algorithm is a generalisation in the narrower sense: in case that there is exactly one accepting set, it behaves exactly as the determinisation procedure for Büchi automata in [17]. An algorithm to determinise generalised Büchi automata to deterministic parity automata using this method was used in [5], similarly extending Piterman's construction [10,7].

Using the current techniques and bounds for the determinisation of these automata [10,17,7], we find that for a nondeterministic generalised Büchi automaton with n states and k accepting sets, we get a deterministic Rabin automaton with $\mathsf{ghist}_k(n)$ states and $2^n - 1$ Rabin pairs. The function $\mathsf{ghist}_k(n)$ is approximately $(1.65\,n)^n$ for $k = 1$, $(3.13\,n)^n$ for $k = 2$, and $(4.62n)^n$ for $k = 3$, and converges against $(1.47kn)^n$ for large k. These bounds can also be used to establish smaller maximal sizes of minimal models, which is useful for Safraless determinisation procedures [6,18,5]. It would be simple to extend the transformation to deterministic parity automata from [17] to obtain an automaton with $O(n!^2\,2^k)$ states and $2n+1$ priorities. In this sense, the difference between determinising Büchi and generalised Büchi is negligible if k is small compared to n.

Colcombet and Zdanowski [2] showed that Schewe's determinisation procedure for Büchi automata is optimal. Our extension of this lower bound to generalised Büchi automata generalises their techniques, showing that the determinisation is optimal.

We also discuss a bridge between optimal determinisation and tight complementation. We show how the nondeterministic power of an automaton can be exploited by using a more concise data structure compared to determinisation (flat trees instead of general ones). This bridge again results in a generalisation of the Büchi complementation procedure discussed in [16] in the narrower sense: for one accepting set, the resulting automata coincide. We also provide a matching lower bound: we show for alphabets L_n^k that the size of a generalised Büchi automaton that recognises the complement of a full generalised Büchi automaton with n states and k accepting sets must be larger than $|L_n^k|$, while the ordinary Büchi automaton we construct is smaller than $|L_{n+1}^{k+1}|$. For large k – that is, if k is not small compared to n – $|L_n^k|$ is approximately $\left(\frac{kn}{e}\right)^n$. This improves significantly over the $\big(\Omega(nk)\big)^n$ bound established by Yan [20].

2 Nondeterministic and Deterministic Automata

Nondeterministic Rabin automata are used to represent ω-regular languages $L \subseteq \Sigma^\omega = \omega \to \Sigma$ over a finite alphabet Σ. In this paper, we use automata with trace based acceptance mechanisms. We denote $\{1,2,\ldots,k\}$ by $[k]$.

A nondeterministic Rabin automaton with k accepting pairs is a quintuple $\mathcal{A} = (\Sigma, Q, I, T, \{(A_i,R_i) \mid i \in [k]\})$, consisting of a finite alphabet Σ, a finite set Q of states with a non-empty subset $I \subseteq Q$ of initial states, a set $T \subseteq Q \times \Sigma \times Q$ of transitions from states through input letters to successor states, and a finite family $\{(A_i,R_i) \in 2^T \times 2^T \mid i \in [k]\}$ of Rabin pairs.

Nondeterministic Rabin automata are interpreted over infinite sequences $\alpha : \omega \to \Sigma$ of input letters. An infinite sequence $\rho : \omega \to Q$ of states of $\mathcal{A}$ is called a *run* of $\mathcal{A}$ on an input word α if the first letter $\rho(0) \in I$ of ρ is an initial state, and if, for all $i \in \omega$, $\big(\rho(i), \alpha(i), \rho(i+1)\big) \in T$ is a transition. For a run ρ of a word α, we denote with $\overline{\rho} : i \mapsto \big(\rho(i), \alpha(i), \rho(i+1)\big)$ the *transitions of* ρ.

A run $\rho : \omega \to Q$ is *accepting* if, for some index $i \in [k]$, some transition $t \in A_i$ in the accepting set A_i of the Rabin pair (A_i,R_i), but no transition $t' \in R_i$ from the rejecting set R_i of this Rabin pair appears infinitely often in the transitions of ρ, $\overline{\rho}$. ($\exists i \in [k].\ \mathit{inf}(\overline{\rho}) \cap A_i \neq \emptyset \wedge \mathit{inf}(\overline{\rho}) \cap R_i = \emptyset$ for $\mathit{inf}(\overline{\rho}) = \{t \in T \mid \forall i \in \omega\ \exists j > i$ such that $\overline{\rho}(j) = t\}$). A word $\alpha : \omega \to \Sigma$ is *accepted* by $\mathcal{A}$ if $\mathcal{A}$ has an accepting run on α, and the set $\mathcal{L}(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \alpha$ is accepted by $\mathcal{A}\}$ of words accepted by $\mathcal{A}$ is called its *language*.

For technical convenience we also allow for finite runs $q_0 q_1 q_2 \ldots q_n$ with $T \cap \{q_n\} \times \{\alpha(n)\} \times Q = \emptyset$. Naturally, no finite run satisfies the Rabin condition. Finite runs are rejecting, and have no influence on the language of an automaton.

Two particularly simple types of Rabin automata are of special interest: generalised Büchi, and Büchi automata. Büchi automata are Rabin automata with a single accepting pair $(F, \emptyset)$ with an empty set of rejecting transitions. The transitions in F are traditionally called final, and Büchi automata are denoted $\mathcal{A} = (\Sigma, Q, I, T, F)$.

Generalised Büchi automata, denoted $\mathcal{A} = (\Sigma, Q, I, T, \{F_i \mid i \in [k]\})$, have a family of accepting (or: final) sets. A run ρ of a generalised Büchi automaton is accepting if it contains infinitely many transitions from all final sets ($\forall i \in [k].\ \mathit{inf}(\overline{\rho}) \cap F_i \neq \emptyset$).

Let $\delta : (q,\sigma) \mapsto \{q' \in Q \mid (q,\sigma,q') \in T\}$ denote the successor function for a set of transitions. A Rabin, Büchi, or generalised Büchi automaton is called *deterministic* if δ is deterministic ($\forall (q,\sigma) \in Q \times \Sigma.\ |\delta(q,\sigma)| \leq 1$) and $I = \{q_0\}$ is singleton. We denote deterministic automata as $(\Sigma, Q, q_0, \delta', \Gamma)$, where Γ is the acceptance condition, and $\delta' : Q \times \Sigma \to Q$ is the partial function that is undefined in the preimage of $\emptyset$ for δ and otherwise satisfies $\delta(q,\sigma) = \{\delta'(g,\sigma)\}$.

3 Determinisation

The determinisation construction described in this section is a generalisation of Schewe's construction for nondeterministic Büchi automata [17], which in turn is a variation of Safra's [14]. We first define the structure that captures the acceptance mechanism of our deterministic Rabin automaton.

Ordered trees. We call a tree $\mathcal{T} \subseteq \mathbb{N}^*$ an *ordered tree* if it satisfies the following constraints, and use the following terms:

- every element $v \in \mathcal{T}$ is called a *node*,
- if a node $v = n_1 \dots n_j n_{j+1} \in \mathcal{T}$ is in $\mathcal{T}$, then $v' = n_1 \dots n_j$ is also in $\mathcal{T}$, v' is called the *predecessor* of v, denoted $\mathsf{pred}(v)$ ($\mathsf{pred}(\varepsilon)$ is undefined),
- the empty sequence $\varepsilon \in \mathcal{T}$, called the *root*, is in $\mathcal{T}$, and
- if a node $v = n_1 \dots n_j$ is in $\mathcal{T}$, then $v' = n_1 \dots n_{j-1} i$ is also in $\mathcal{T}$ for $0 < i < j$; we call v' an *older sibling* of v (and v a *younger sibling* of v'), and denote the set of older siblings of v by $\mathsf{os}(v)$.

Thus, ordered trees are non-empty and closed under predecessors and older siblings.

Generalised history tree. A generalised history tree $\mathcal{G}$ over Q for k accepting sets is a triple $\mathcal{G} = (\mathcal{T}, l, h)$ such that:

- $\mathcal{T}$ is an ordered tree,
- $l : \mathcal{T} \to 2^Q \smallsetminus \{\emptyset\}$ is a labelling function such that
  - $l(v) \subsetneq l(\mathsf{pred}(v))$ holds for all $\varepsilon \neq v \in \mathcal{T}$,
  - the intersection of the labels of two siblings is disjoint ($\forall v, v' \in \mathcal{T}.\ v \neq v' \wedge \mathsf{pred}(v) = \mathsf{pred}(v') \Rightarrow l(v) \cap l(v') = \emptyset$), and
  - the union of the labels of all siblings is *strictly* contained in the label of their predecessor ($\forall v \in \mathcal{T}\ \exists q \in l(v)\ \forall v' \in \mathcal{T}.\ v = \mathsf{pred}(v') \Rightarrow q \notin l(v')$), and
- $h : \mathcal{T} \to [k]$ is a function that labels every node with a natural number from $[k]$. We call $F_{h(v)}$ the *active* accepting set of v.

For a generalised history tree $\mathcal{G} = (\mathcal{T}, l, h)$, $(\mathcal{T}, l)$ is the history tree introduced in [17]. Generalised history trees are enriched by the second labelling function, h, that is used to relate nodes with a particular accepting set.

3.1 Determinisation Construction

Let $\mathcal{A} = (\Sigma, Q, I, T, \{F_i \mid i \in [k]\})$ be a generalised Büchi automaton with $|Q| = n$ states and k accepting sets. We will construct an equivalent deterministic Rabin automaton $\mathcal{D} = (\Sigma, D, d_0, \delta, \{(A_i, R_i) \mid i \in J\})$.

[17] separates the transition mechanism from the acceptance condition. We follow the same procedure and describe the transition mechanism below.

- D is the set of generalised history trees over Q.
- d_0 is the generalised history tree $(\{\varepsilon\}, l : \varepsilon \mapsto I, h : \varepsilon \mapsto 1)$.
- J is the set of nodes that occur in some ordered tree of size n.

- For every tree $d \in D$ and letter $\sigma \in \Sigma$, the transition $d' = \delta(d,\sigma)$ is the result of the following sequence of transformations:
  1. *Raw update of l.* We update l to the function l_1 by assigning, for all $v \in \mathcal{T}$, $l_1 : v \mapsto \{q \in Q \mid \exists q' \in l(v).\ (q',\sigma,q) \in T\}$, i.e., to the σ successors of $l(v)$.
  2. *Sprouting new children.* For every node $v \in d$ with c children, we sprout a new child vc. Let $\mathcal{T}_n$ be the tree of new children. Then we define, for all v in T_n, $l_1 : v \mapsto \big\{q \in Q \mid \exists q' \in l\big(\mathsf{pred}(v)\big).\ (q',\sigma,q) \in f_{h(\mathsf{pred}(v))}\big\}$, i.e., to the σ successors of the active accepting sets of their parents, and extend h to $T_n' = \mathcal{T} \cup \mathcal{T}_n$ by $h : v \mapsto 1$ for all $v \in \mathcal{T}_n$
  3. *Stealing of labels.* We obtain a function l_2 from l_1 by removing, for every node v with label $l(v) = Q'$ and all states $q \in Q'$, q from the labels of all younger siblings of v and all of their descendants.
  4. *Accepting and removing.* We denote with $\mathcal{T}_r \subseteq T_n'$ the set of all nodes v whose label $l_2(v)$ is now equal to the union of the labels of its children. We obtain $\mathcal{T}'$ from $\mathcal{T}_n'$ by removing all descendants of nodes in $\mathcal{T}_r$, and restrict the domain of l_2 and h accordingly. (The resulting tree $\mathcal{T}'$ may no longer be ordered.)
     Nodes in $\mathcal{T}' \cap \mathcal{T}_r$ are called *accepting*. We obtain h_1 from h by choosing $h : v \mapsto h(v)+1$ for accepting nodes v with $h(k) \neq k$, $h_1 : v \mapsto 1$ for accepting nodes v with $h(k) = k$, and $h_1 : v \mapsto h(v)$ for all non-accepting nodes.
     The transition is in A_v iff v is accepting.
  5. *Renaming and rejecting.* To repair the orderedness, we call $\|v\| = |\mathsf{os}(v) \cap \mathcal{T}'|$ the number of (still existing) older siblings of v, and map $v = n_1 \dots n_j$ to $v' = \|n_1\|\ \|n_1n_2\|\ \|n_1n_2n_3\| \dots \|v\|$, denoted $\mathsf{rename}(v)$.
     We update a triple $(\mathcal{T}', l_2, h_1)$ from the previous step to $d' = \big(\mathsf{rename}(\mathcal{T}'), l', h'\big)$ with $l' : \mathsf{rename}(v) \mapsto l_2(v)$ and $h' : \mathsf{rename}(v) \mapsto h_1(v)$.
     We call a node $v \in \mathcal{T}' \cap \mathcal{T}$ *stable* if $v = \mathsf{rename}(v)$, and we call all nodes in J *rejecting* if they are not stable. The transition is in R_v iff v is rejecting.

3.2 Correctness

In order to establish the correctness of our determinisation construction, we need to prove that $L(\mathcal{A}) = L(\mathcal{D})$, that is, we need to ascertain that the ω-language accepted by the nondeterministic generalised Büchi automaton is equivalent to the ω-language accepted by the deterministic Rabin automaton.

Theorem 1 **($L(\mathcal{D}) \subseteq L(\mathcal{A})$).** *Given that there is a node $v \in d$ (where d is a generalised history tree) which is eventually always stable and always eventually accepting for an ω-word α, then there is an accepting run of $\mathcal{A}$ on α.*

Notation. For an ω-word α and $j \geq i$, we denote with $\alpha[i,j[$ the word $\alpha(i)\alpha(i+1)\alpha(i+2)\dots\alpha(j-1)$. We denote with $Q_1 \rightarrow^{\alpha} Q_2$ for a finite word $\alpha = \alpha_1 \dots \alpha_{j-1}$ that there is, for all $q_j \in Q_2$ a sequence $q_1 \dots q_j$ with $q_1 \in Q_1$ and $(q_i, \alpha_i, q_{i+1}) \in T$ for all $1 \leq i < j$. If, for all $q_j \in Q_2$, there is such a sequence that contains a transition in F_a, we write $Q_1 \Rightarrow_a^{\alpha} Q_2$.

Proof. Let $\alpha \in L(\mathcal{D})$. Then there is a v that is eventually always stable and always eventually accepting in the run $\rho_{\mathcal{D}}$ of $\mathcal{D}$ on α. We pick such a v.

Let $i_0 < i_1 < i_2 < \dots$ be an infinite ascending chain of indices such that

- v is stable for all transitions $(d_j, \alpha(j), d_{j+1})$ with $j \geq i_0$, and
- the chain $i_0 < i_1 < i_2 < \ldots$ contains exactly those indices $i \geq i_0$ such that $(d_{i-1}, \alpha(i-1), d_i)$ is accepting; this implies that h is updated exactly at these indices.

Let $d_i = (\mathcal{T}_i, l_i, h_i)$ for all $i \in \omega$. By construction, we have

- $I \rightarrow^{\alpha[0,i_0[} l_{i_0}(v)$, and
- $l_{i_j}(v) \Rightarrow_{h_{i_j}}^{\alpha[i_j,i_{j+1}[} l_{i_{j+1}}(v)$.

Exploiting König's lemma, this provides us with the existence of a run of $\mathcal{A}$ on α that visits all accepting sets F_i of $\mathcal{A}$ infinitely many times. (Note that the value of h is circulating in the successive sequences of the run.) This run is accepting, and α therefore in the language of $\mathcal{A}$. □

Theorem 2 **($L(\mathcal{A}) \subseteq L(\mathcal{D})$).** *Given that there is an accepting run of $\mathcal{A}$, there is a node which is eventually always stable and always eventually accepting.*

Notation. For a state q of $\mathcal{A}$ and a generalised history tree $d = (\mathcal{T}, l, h)$, we call a node a *host node of q*, denoted $\mathsf{host}(q,d)$, if $q \in l(v)$ but not in $l(vc)$ for any child vc of v.

Proof. We fix an accepting run $\rho = q_0 q_1 \ldots$ of $\mathcal{A}$ on an input word α, and let $\rho_{\mathcal{D}} = d_0 d_1 \ldots$ be the run of $\mathcal{D}$ on α. We then define the related sequence of host nodes $\vartheta = v_0 v_1 v_2 \ldots = \mathsf{host}(q_0,d_0)\mathsf{host}(q_1,d_1)\mathsf{host}(q_2,d_2)\ldots$. Let l be the shortest length $|v_i|$ of these nodes that occurs infinitely many times.

We follow the run and see that the initial sequence of length l of the nodes in ϑ eventually stabilises. Let $i_0 < i_1 < i_2 < \ldots$ be an infinite ascending chain of indices such that the length $|v_j| \geq l$ of the j-th node is not smaller than l for all $j \geq i_0$, and equal to $l = |v_i|$ for all indices $i \in \{i_0, i_1, i_2, \ldots\}$ in this chain. This implies that $v_{i_0}, v_{i_1}, v_{i_2}, \ldots$ is a descending chain when the single nodes v_i are compared by lexicographic order. As the domain is finite, almost all elements of the descending chain are equal, say $v_i := \pi$. In particular, π is eventually always stable.

Let us assume for the sake of contradiction, that this stable prefix π is accepting only finitely many times. We choose an index i from the chain $i_0 < i_1 < i_2 < \ldots$ such that π is stable for all $j \geq i$. (Note that π is the host of q_i for d_i, and $q_j \in l_j(\pi)$ holds for all $j \geq i$.)

As ρ is accepting, there is a smallest index $j > i$ such that $(q_{j-1}, \alpha(j-1), q_j) \in F_{h_i(\pi)}$. Now, as π is not accepting, q_i must henceforth be in the label of a child of π, which contradicts the assumption that infinitely many nodes in ϑ have length $|\pi|$.

Thus, π is eventually always stable and always eventually accepting. □

Corollary 1 **($L(\mathcal{A}) = L(\mathcal{D})$).** *The deterministic Rabin automaton generated by our determinisation procedure is language equivalent to the original generalised Büchi automaton.*

4 Complementation

In this section we connect determinisation and complementation. In order to construct a concise data structure for complementation, we first show that we can cut acceptance into two phases: a finite phase where we track only the reachable states, and an infinite phase where we also track acceptance. We then use this simple observation to devise an abstract complementation procedure, and then suggest a succinct data structure for it.

We first argue that acceptance of a word $\alpha \cdot \alpha'$ with $\alpha \in \Sigma^*$ and $\alpha' \in \Sigma^\omega$ depends only on α' and the states reachable through α.

Lemma 1. *If $I \rightarrow^\alpha Q \Leftrightarrow I \rightarrow^\beta Q$ then $\alpha \cdot \alpha' \in L(\mathcal{A}) \Leftrightarrow \beta \cdot \alpha' \in L(\mathcal{A})$.*

Proof. It is easy to see how an accepting run of $\mathcal{A}$ on $\alpha \cdot \alpha'$ can be turned into an accepting run on $\beta \cdot \alpha'$, and vice versa. □

This provides us with the following abstract description of a nondeterministic acceptance mechanism for the complement language of $\mathcal{A}$.

1. When reading an ω-word α, we first keep track of the reachable states R for a finite amount of time. (subset construction)
2. Eventually, we swap to a tree that consists only of nodes that are henceforth stable, and that are the only nodes that are henceforth stable, such that none of these nodes is henceforth accepting.
3. We verify the property described in (2).

Lemma 2. *The abstract decision procedure accepts an input word iff it is rejected by the deterministic Rabin automaton $\mathcal{D}$.*

Proof. The 'only if' direction follows directly from the previous lemma.

For the 'if' direction, we can guess a point i in the run of $\mathcal{D}$ on α where all eventually stable nodes are introduced and stable, and none of them is henceforth accepting. We claim that we can simply guess this point of time, but instead of going to the respective generalised tree $d_i = (\mathcal{T}, l, h)$, we go to $d_i' = (\mathcal{T}', l', h')$, where $\mathcal{T}'$ is the restriction of $\mathcal{T}$ to the henceforth stable states, and l' and h' are the restrictions of l and h to $\mathcal{T}'$. (Note that the subtree of henceforth stable nodes is always ordered.)

Clearly, all nodes in $\mathcal{T}'$ are stable, and none of them are accepting in the future. It remains to show that none of their descendants is stable. Assume one of the children a node $v \in \mathcal{T}'$ spawns eventually is stable. We now consider a part of the 'run' of our mechanism starting at i, $d_i' d_{i+1}' d_{i+2}' \ldots$. Invoking König's Lemma, we get a run $\rho = q_0 q_1 \ldots$ such that, for some $j > i$ and for all $m > j$, some $v_j = \mathsf{host}(q_j, d_j')$, which is a true descendant of v, is the host of q_j. Using a simple inductive argument that exploits that v is henceforth stable but not accepting, this implies for the run $\rho = d_0 d_1 \ldots$ that, for the same $j > i$ and for all $m > j$, some $v_j' = \mathsf{host}(q_j, d_j')$, which is a true descendant of v, is a host of q_j. This implies in turn that some descendant of v is eventually stable and thus leads to a contradiction. □

We call an ordered tree *flat* if it contains only nodes of length ≤ 1.

Lemma 3. *We can restrict the choice in (2) to flat trees.*

Proof. If we rearrange the nodes in $\mathcal{T}$ following the "stealing and hosting order", that is, mapping a node v with length ≥ 1 to a smaller node v' with length ≥ 1 if either v' is an ancestor of v or an initial sequence of v is an older sibling of an initial sequence of v'. This describes a unique bijection $b : \mathcal{T} \to \mathcal{F}$, where $\mathcal{F}$ is the flat tree with $|\mathcal{T}| = |\mathcal{F}|$, and we choose $d_i' = (\mathcal{F}, l' : b(v) \mapsto l(\mathsf{pred}(b(v))) \cup l(v) \setminus l(\mathsf{pred}(v)), h' : b(v) \mapsto h(v))$ instead of $d_i = (\mathcal{T}, l, h)$. (The complicated looking $l' : b(v) \mapsto l(\mathsf{pred}(b(v))) \cup l(v) \setminus l(\mathsf{pred}(v))$ just means $v = \mathsf{host}(q, d_i) \Leftrightarrow b(v) = \mathsf{host}(q, d_i')$, that is, the hosts are moved, not the full label.)

It is easy to see that, if we compare two runs starting in d_i and d_i' on any word, they keep this relation. □

To obtain a succinct data structure for the second phase of the run, we do not follow the precise development of the individual new children of the henceforth stable nodes, but rather follow simple subset constructions. One subset that is kept for all stable nodes is the *union* of the nodes of its children. Note that, to keep track of this union, it is not necessary to keep track of the distribution of these sets to the different children (let alone their descendants).

To check that all children spawned at a particular point j in a run will eventually be deleted, one can keep track of an additional subset: the union of all labels of nodes of children that already existed in j. If this subset runs empty, then all of these children have been removed. Vice versa, if all of these children are removed, then this subset runs empty.

Note that these subsets are, in contrast to the nodes in the flat generalised history tree, not numbered. For efficiency, note that it suffices to use the second subset for only one of the nodes in the flat trees at a time, changing this node in a round robin fashion.

Theorem 3. *The algorithm outlined above describes a nondeterministic Büchi automaton that recognises the complement of the language of $\mathcal{A}$.*

The trees and sets we need can be encoded using the following data structure. We first enrich the set of states by a fresh marker m, used to mark the extra subset for new children in the stable node under consideration, to $Q_m = Q \cup \{m\}$. We then add the normal subsets that capture the label of all children as a child of each stable state as a single child of this state. For a single stable state that we currently track, we add a (possibly second and then younger) child for new children. The labelling is as described above, except that m is added to the label for new children (which otherwise might be empty) and its ancestors.

We can now choose $h : v \to k+1$ for all non-stable nodes v in this tree. As this naming convention clearly identifies these nodes, we can represent the tree as a flat tree.

4.1 Complexity of Complementing Generalised Büchi Automata

In this section, we establish lower bounds for the complementation of generalised Büchi automata and show that the construction we outlined tightly meets these lower bounds. Our lower bound proof builds on *full* automata. A generalised Büchi automaton $\mathcal{B}_n^k = (\Sigma, Q, I, T, \{F_i \mid i \in [k]\})$ is called *full* if

- $\Sigma_n^k = 2^{Q \times [k+1] \times Q}$, $|Q| = n$, and $I = Q$,
- $T = \{(q, \sigma, q') \mid \exists i \in [k+1].\ (q, i, q') \in \sigma\}$, and

- $F_i = \{(q,\sigma,q') \mid (q,i,q') \in \sigma\}$.

As each generalised Büchi automaton with n states and k accepting sets can be viewed as a language restriction (by alphabet projection) of a full automaton, full automata are useful tools in establishing lower bounds. We show that, for each $\mathcal{B}_n^k$, there is a family of $L_n^k \subseteq \Sigma_n^k$ such that a^ω is not in the language of $\mathcal{B}_n^k$ for any $a \in L_n^k$, and each nondeterministic generalised Büchi automaton that recognises the complement language of $\mathcal{B}_n^k$ must have at least $|L_n^k|$ states. The size of this alphabet is such that the size of $\mathcal{B}_n^k$ is between $|L_n^k|$ and $|L_{n+1}^{k+1}|$, which provides us with tight bounds for the complementation of generalised Büchi automata.

Let us first define the letters in L_n^k. We call a function $f : Q \to \mathbb{N}$ *full* if its domain is $[n]$ for some n. Let f be a full function with domain $[n]$ then we call a function $f_\# : [n] \to [k]$ a *k-numbering* of f. We denote by $\mathsf{enc}(f, f_\#)$ the letter encoding a function f with k-numbering $f_\#$ as the letter that satisfies

- $(p, \mathsf{enc}(f, f_\#), q) \in T$ iff $f(q) \leq f(p)$, and
- $(p, \mathsf{enc}(f, f_\#), q) \in F_b$ iff *either* $f(q) < f(p)$ *or* $(f(p) = f(q)$ and $f_\#\big(f(p)\big) \neq b)$.

Obviously, if two full functions f, g with respective k-numberings $f_\#, g_\#$ encode the same letter $\mathsf{enc}(f, f_\#) = \mathsf{enc}(g, g_\#)$, then they are equal. First, we note that the word a^ω is not in the language of $\mathcal{B}_n^k$.

Lemma 4. *Let f be a full function, $f_\#$ be a k-numbering of f, and let a be the letter encoded by f and $f_\#$. Then a^ω is rejected by $\mathcal{B}_n^k$.*

Proof. Assume, for contradiction, that there is an accepting run ρ of $\mathcal{B}_n^k$ on a^ω. By the definition of an encoding, the sequence $f_i = f(\rho_i)$ is monotonously decreasing. It will therefore stabilise eventually, say at j. (I.e., $\forall l \geq j.\ f(\rho_l) = f(\rho_j)$.) By the definition of accepting transitions for encoded letters there will henceforth be no more transition from the final transitions $F_{f_\#(f_j)}$. ↯ □

We now define $L_n^k = \{\mathsf{enc}(f, f_\#) \mid f$ is full and $f_\#$ is a k-numbering of $f\}$.

Theorem 4. *A generalised Büchi automaton $\mathcal{C}_n^k$ that recognises the complement language of $\mathcal{B}_n^k$ has at least $|L_n^k|$ states.*

Proof. The previous lemma establishes that a^ω is accepted by $\mathcal{C}_n^k$. We choose accepting runs ρ_a with infinity set I_a for each letter $a \in L_n^k$, and show by contradiction that I_a and I_b are disjoint for two different letters $a, b \in L_n^k$.

Assume that this is not the case for two different letters a and b. It is then simple to infer from their accepting runs ρ_a and ρ_b natural numbers $l, m, n, a \in \mathbb{N}$ such that $\rho = \rho_a[0,l](\rho_b[a, a+m]\rho_a[l, l+n])^\omega$ is accepting. Then $w = a^l(b^m a^n)^\omega$ is accepted by $\mathcal{C}_n^k$, as ρ is a run of w. We lead this to a contradiction by showing that w is in the language of $\mathcal{B}_n^k$.

We have $a = \mathsf{enc}(f, f_\#) \neq b = \mathsf{enc}(g, g_\#)$. Let us first assume $f = g$. Then $f_\# \neq g_\#$, and we can first choose an i with $f_\#(i) \neq g_\#(i)$ and then a q with $f(q) = i$. It is now simple to construct an accepting run with trace q^ω for $\mathcal{B}_n^k$ for w. ↯

Let us now assume $f \neq g$. We then set i to the minimal number such that f and g differ in i $(f^{-1}(i) \neq g^{-1}(d)$, where $^{-1}$ denotes the preimage of i. W.l.o.g., we assume

$f^{-1}(i) \setminus g^{-1}(i) \neq \emptyset$. We choose a $q \in f^{-1}(i) \setminus g^{-1}(i)$. It is now again simple to construct an accepting run with trace q^ω for $\mathcal{B}_n^k$ for w. ↯

This closes the case distinction and provides the main contradiction. □

The only thing that remains to be shown is tightness. But there is obviously an injection from the flat trees described at the end of the complementation (plus the subsets) of an automaton with n states and k accepting pairs into L_{n+1}^{k+1}. This provides:

Theorem 5. $|L_{n+1}^{k+1}| > |\mathcal{B}_n^k|$.

This provides bounds which are tight in k and n with a negligible margin of 1. For large k, the size $|L_n^k|$ can be approximated by $\left(\frac{kn}{e}\right)^n$: It is not hard to show that the size of L_n^k is dominated by encodings that refer to functions from $[n]$ onto $[n]$, and the number of these encodings is $n!k^n$. (E.g., $|L_n^n| < (e-1)n!n^n$.)

Our conjecture is that the construction is tight at least in n. The reason for this assumption is that the increment in n stems from the round robin construction that keeps track of the stable node under consideration, while the alphabet L_n^k refers to the far more restricted case that stable states never spawn new children, rather than merely requiring that none of the children spawned is henceforth stable.

5 Optimality of Our Determinisation Construction

Colcombet and Zdanowski have shown that the determinisation technique we use (and used in [17]) is optimal for the case of determinising Büchi automata. Their proof is by reducing the resulting deterministic Rabin automaton to a game and citing the memory required for the winner of the game to establish a lower bound on the size of the deterministic Rabin automaton [2]. In this section, we extend their result to the case of generalised Büchi automata. We show that the function $\mathsf{ghist}_k(n)$, that maps n to the number of generalised history trees for k accepting sets—and hence to the number of states of the resulting deterministic Rabin automaton obtained by our determinisation construction—is also a lower bound for the number of states needed for language equivalent deterministic Rabin automata.

5.1 Games

We follow the conventions defined in [2]. A *two player* game is a tuple $G = (V, E, L, W)$, where V is a set of states of the game, which is partitioned into V_0 and V_1, states for the two players, Player 0 and Player 1 respectively, $E \in V \times L \times V$ is the transition relation and $W \in L^\omega$ is the winning condition. We require that, for every $v \in V$, there exists some state $v' \in V$ such that $(v, a, v') \in E$ is a transition from the location v to v'. We say that (v, a, v') produces the letter a.

A *play* of the game G is a maximal sequence of locations $p = (v_0, a_0, v_1, a_1 \ldots)$ such that, for all $i \geq 0$, we have $(v_i, a_i, v_{i+1}) \in E$. Let $p_L = (a_0 a_1 \ldots)$ be the sequence of letters generated by the play p. Player 0 wins the play p if $p_L \in W$. Player 1 wins otherwise.

A strategy for player X is a function that specifies how the player should play depending on the history of transitions. A *strategy* for player X is a function σ mapping finite sequences $(v_0, a_0, \dots, a_{n-1}, v_n)$ into E. A play p is *compatible* with a strategy σ for player 0 if, for all prefixes $(v_0, a_0, \dots, v_{n-1}, a_{n-1}, v_n)$ of p, $\sigma(v_0, \dots v_{n-1}) = (v_{n-1}, a_{n-1}, v_n)$ if $v_{n-1} \in V_0$. A strategy σ for player 0 is a *winning strategy* if player 0 wins every play compatible with σ. Strategies for player 1 are defined similarly, with V_1 instead of V_0.

A strategy with *memory of size m* for player 0 is a tuple $(M, update, choice, init)$, where M is a set of size m, called the memory, *update* is a mapping from $M \times E$ to M, *choice* is a mapping from $V_0 \times M$ to E, and $init \in M$. Player X wins a game with memory of size m if she has a winning strategy with memory of size m. When we have a strategy with memory of size 1, we call it a *positional strategy*.

A game G is called a *Rabin game* if the winning condition W is described by a Rabin condition over the transitions of this game. For every *Rabin game*, player 0 can win using a positional strategy [21].

Finite games of infinite duration can effectively be reduced to deterministic automata and vice-versa. Given a deterministic Rabin automaton $\mathcal{D}$ with n states that accepts the language W of a generalised Büchi automaton $\mathcal{A}$, and a game with winning condition W, one can construct the product of the automaton $\mathcal{D}$ with the game, and derive a game with the Rabin condition as the winning condition of the game. It is obvious that such a game (with its Rabin condition) admits positional strategies and that the deterministic Rabin automaton maintains the memory for a strategy in the original game with the generalised Büchi winning condition.

Lemma 5. *[2] If player 0 wins a game with the winning condition W (the generalised Büchi condition) while requiring memory of size n, then every deterministic Rabin automaton that is language equivalent to W has at least n states.*

5.2 Proving a Lower Bound

Lemma 5 provides a viable argument to prove a lower bound on the determinisation of generalised Büchi automata. For this, we use the full automata introduced in the previous section.

Closely following Colcombet & Zdanowski's proof [2], we use $\mathcal{B}_n^k = (\Sigma, Q, I, T, \{F_i \mid i \in [k]\})$. To prove our lower bound, we first restrict ourselves to a constant set of reachable states. We then proceed by proving a (tight) bound within this restricted scenario. Finally, we extend this lower bound to the general case.

Restricting the set of reachable states. We define the set of states reachable by a word u, Reach(u) by induction. Obviously, Reach$(\varepsilon) = Q$. For $v \in \Sigma^*$ and $a \in \Sigma$, we define Reach(va) as follows: for all $q \in Q$ and $q' \in Q$, $q' \in$ Reach(va) iff $q \in$ Reach(v) and there is a transition $(q, i, q') \in a$. Let Σ_S be the set of letters $a \in \Sigma$ such that Reach$(a) = S$ and Reach$(v) = S$ implies Reach$(va) = S$. Let $\mathcal{L}(\mathcal{B}_S^k)$ be $\mathcal{L}(\mathcal{B}_n^k) \cap \Sigma_S{}^\omega$.

Each generalised history tree $d \in D$ maintains the set of states reachable by the generalised Büchi automaton at the current position of the input word in the label $l(\varepsilon)$ of its root. Thus, if we restrict ourselves to a set of states $S \subseteq Q$, it is enough if we consider the set of generalised history trees D_S, which contain the states $S \subseteq Q$ in the labels of their root ε. The runs of the deterministic Rabin automaton $\mathcal{D}_n^k$, which we get when determinising $\mathcal{B}_n^k$, on an ω-word $\alpha \in \Sigma_S{}^{\omega}$, is therefore a sequence in $D_S{}^{\omega}$.

A game to prove tightness. In this restricted context, we define a game G such that player 0 wins G, requiring at least memory $|D_S|$, which will in turn establish that any deterministic Rabin automaton accepting $\mathcal{L}(\mathcal{B}_S^k)$ has at least $|D_S|$ states.

We define the game G with the winning condition $\mathcal{L}(\mathcal{B}_S^k)$. Formally, $G = (V, E, L, W)$ where V is the set of states of the game partitioned into V_0 and V_1, the states for players 0 and 1 respectively. V_0 is a singleton set $\{v_0\}$ and V_1 consists of the initial state of the game v_{in} and one position v_d for each generalised history tree $d \in D_S$. The labelling function L is the alphabet Σ_S^+. The winning condition W is $\mathcal{L}(\mathcal{B}_S^k)$. The game transitions E are as follows:

- $(v_{in}, u, v_0) \in E$ for all $u \in \Sigma_S^+$,
- $(v_0, id_S, v_d) \in E$ for each generalised history tree d, where $id_S = \{(q, k+1, q) \mid q \in S\}$ is the input letter that maintains all trees in D_S,
- $(v_d, u, v_0) \in E$ for each generalised history tree $d \in D_S$ and word $u \in \Sigma_S^+$ if u is *profitable* for d. We call u profitable if there is a v in the ordered tree of d such that
  - v is stable throughout the sequence of a run of $\mathcal{D}_n^k$ starting in d when u is read, and
  - v is accepting for some transition in the sequence.

Player 0 has a simple winning strategy in this game. It suffices if player 0 keeps track of the current state of $\mathcal{D}_n^k$ when following the ω-word produced in this game. When it is her turn and the run is in state d, then she plays to v_d. This way, player 1 is forced to play only profitable words, which leads to a minimum node that is always eventually profitable, and hence to acceptance [2].

Lemma 6. *Player 0 has a winning strategy in the game G.*

Modified game to prove memory lower bound. We define a modified game G_{mod} by removing one of player 1's game states v_d, thereby denying player 0 the corresponding move. This is the only difference to the game G. This lemma is the technical core of the proof requiring an analysis of the differences between two trees.

Lemma 7. *Let $d \neq d'$ be generalised history trees in D_S. There exists a word u such that*

- *$(v_{d'}, u, v_0)$ is a move in G_{mod},*
- *$d \to^u d$,*
- *u is not profitable for d.*

Proof. We distinguish two cases. First, we assume that $d = (\mathcal{T}, l, h)$ and $d' = (\mathcal{T}, l, h')$. This is the easy part: we can simply use a node $v \in \mathcal{T}$ such that $h(v) \neq h'(v)$, but this does not hold for any descendant of v. We then choose $Q' = \{q \in S \mid v = \mathsf{host}(q, d)\}$ to be the set of nodes hosted by v. (Note that these are the same for d and d'.)

In this case, we can simply play the one letter word $\alpha = id_S \cup \{(q, h'(v), q) \mid q \in Q'\}$, which satisfies all the properties from above: clearly the transition is profitable for $v_{d'}$ (as v is accepting in the respective transition $d' \to^{\alpha} d''$) whereas $d \to^{\alpha} d$ holds while none of the nodes of $\mathcal{T}$ is accepting.

Now we assume that $d = (\mathcal{T}, l, h)$ and $d' = (\mathcal{T}', l', h')$ with $(\mathcal{T}', l') \neq (\mathcal{T}, l)$. But for this case, we can almost use the same strategy for choosing a finite word u as for ordinary history trees [2]. The extra challenge is that, when reconstructing d, it is not enough to spawn a new child, we also have to update h, which can be done using a sequence of letters like the letter α from above after reconstructing a node v. □

With the help of this lemma, we can prove the following.

Lemma 8. *For every $d \in D_S$, player 1 has a winning strategy in G_{mod}.*

A winning strategy for player 1 is as follows. From v_{in}, player 1 plays a word u such that $d_0 \to^{u} d$, where d_0 is the initial state of the deterministic Rabin automaton. A good response from player 0 would be to move to v_d. But v_d has been removed and player 0 has to move to a different game state, say $v_{d'}$ for some $d' \neq d$. Player 1 responds according to Lemma 7. Using this strategy, player 1 ensures that, when the play gets infinite, there is no node that is always eventually accepting and the deterministic Rabin automaton $\mathcal{D}$ does not accept this word. Hence, $u \notin \mathcal{L}(\mathcal{B}_S^k)$ and player 1 wins.

Corollary 2. *Player 0 has no winning strategy with memory $|D_S| - 1$ in the game G.*

If player 0 had a winning strategy with memory $|D_S| - 1$, then there would be a game state v_d which is never visited by this strategy. But this would also mean that player 0 can win G_{mod} with this strategy, which contradicts Lemma 8. Using Lemma 5, we now obtain:

Theorem 6. *Every deterministic Rabin automaton that accepts $\mathcal{L}(\mathcal{B}_S^k)$ has size at least $|D_S|$.*

Extending the lower bound to the unrestricted case. We have proven a lower bound for the case where we restricted the set of reachable states. We now extend our result to the general case. We do this by decomposing the deterministic Rabin automaton accepting $L(\mathcal{B}_n^k)$ into disjoint sets of states. Such a decomposition is feasible due to the following lemma.

Lemma 9. *[2] Let $\mathcal{D}$ be the deterministic Rabin automaton accepting $\mathcal{L}(\mathcal{B}_n^k)$ with transition function δ and initial state d_0. If $\delta(d_0, u) = \delta(d_0, v)$, then $Reach(u)$ =$Reach(v)$.*

The above lemma describes the scenario where we restricted the set of reachable states, considering only the set of generalised history trees D_S. The automaton $\mathcal{D}$ restricted to D_S can be seen as a Rabin automaton which accepts $\mathcal{L}(\mathcal{B}_n^k)$ with restriction to letters in Σ_S. Because the sets D_S are disjoint, we have

$$\mathsf{ghist}_k(n) = |D| = \sum_{S \subseteq Q} |D_S|.$$

Theorem 7. *Every deterministic Rabin automaton accepting $\mathcal{L}(\mathcal{B}_n^k)$ has size at least* $\mathsf{ghist}_k(n)$.

ghist_k can be estimated in a similar way as the number of history trees for the determinisation of Büchi automata [17], as generalised history trees are, just like history trees, ordered trees with further functions on the set of nodes of the tree. Using the functions from [17] $\mathsf{ghist}_k(n) \in \sup_{x>0} O\big(m(x) \cdot k^{\beta(x)} \cdot 4^{\beta(x)}\big)$, providing

$(1.65n)^n$, for $k=1$, $(3.13n)^n$ for $k=2$, and $(4.62n)^n$ for $k=3$.

This value converges against

$$(1.47kn)^n$$

for large k.

Note that, when the generalised Büchi automaton we start with has exactly one accepting set, our bound construction coincides with [17].

References

1. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Berkeley, California, USA, pp. 1–11. Stanford University Press (1960, 1962)
2. Colcombet, T., Zdanowski, K.: A Tight Lower Bound for Determinization of Transition Labeled Büchi Automata. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 151–162. Springer, Heidelberg (2009)
3. Gerth, R., Dolech, D., Peled, D., Vardi, M.Y., Wolper, P., Liege, U.D.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification Testing and Verification, pp. 3–18. Chapman & Hall (1995)
4. Kähler, D., Wilke, T.: Complementation, Disambiguation, and Determinization of Büchi Automata Unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008)
5. 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)
6. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005), Pittsburgh, PA, USA, October 23–25, pp. 531–540 (2005)
7. Liu, W., Wang, J.: A tighter analysis of Piterman's Büchi determinization. Information Processing Letters 109, 941–945 (2009)
8. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9(5), 521–530 (1966)
9. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoretical Computer Science 141(1-2), 69–107 (1995)
10. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Journal of Logical Methods in Computer Science 3(3:5) (2007)
11. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), Providence, Rhode Island, USA, October 31-November 1, pp. 46–57. IEEE Computer Society Press (1977)

12. Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Transaction of the American Mathematical Society 141, 1–35 (1969)
13. Rabin, M.O., Scott, D.S.: Finite automata and their decision problems. IBM Journal of Research and Development 3, 115–125 (1959)
14. Safra, S.: On the complexity of ω-automata. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS 1988), White Plains, New York, USA, October 24-26, pp. 319–327. IEEE Computer Society Press (1988)
15. Safra, S.: Exponential determinization for omega-automata with strong-fairness acceptance condition (extended abstract). In: STOC, pp. 275–282 (1992)
16. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J.-Y. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009). Leibniz International Proceedings in Informatics (LIPIcs), vol. 3, pp. 661–672. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2009)
17. Schewe, S.: Tighter Bounds for the Determinisation of Büchi Automata. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009)
18. 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)
19. Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. Information and Control 54(1/2), 121–141 (1982)
20. Yan, Q.: Lower bounds for complementation of *omega*-automata via the full automata technique. Journal of Logical Methods in Computer Science 4(1:5) (2008)
21. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200(1-2), 135–183 (1998)

A Succinct Canonical Register Automaton Model for Data Domains with Binary Relations*

Sofia Cassel[1], Bengt Jonsson[1], Falk Howar[2], and Bernhard Steffen[2]

[1] Dept. of Information Technology, Uppsala University, Sweden
{sofia.cassel,bengt.jonsson}@it.uu.se
[2] Chair for Programming Systems, Technical University Dortmund, Germany
{falk.howar,steffen}@cs.tu-dortmund.de

Abstract. We present a novel canonical automaton model for languages over infinite data domains, that is suitable for specifying the behavior of services, protocol components, interfaces, etc. The model is based on register automata. A major contribution is a construction of succinct canonical register automata, which is parameterized on the set of relations by which elements in the data domain can be compared. We also present a Myhill Nerode-like theorem, from which minimal canonical automata can be constructed. This canonical form is as expressive as general deterministic register automata, but much better suited for modeling in practice since we lift many of the restrictions on the way variables can be accesed and stored: this allows our automata to be significantly more succinct than previously proposed canonical forms. Key to the canonical form is a symbolic treatment of data languages, which allows us to construct minimal representations whenever the set of relations can be equipped with a so-called branching framework.

1 Introduction

Our aim is to develop automata formalisms that can be used for systems specification, verification, testing, and modeling. It is crucial to be able to model not only control, but also data aspects of a system's behavior, and express relations between data values and how they affect control flow. For example, we may want to express that a password entered matches a previously registered one, that a sequence number is in some interval, or that a user identity can be found in some specific group.

There are many kinds of automata augmented with data, for example timed automata [1], counter automata, data-independent transition systems [15], and different kinds of register automata. Many of these types of automata have long been used for specification, verification, and testing (e.g., [18]). In our context, *register automata* is a very interesting formalism [3,13,7]. A register automaton has a finite set of registers (or state variables) and processes input symbols using a predefined set of operations (tests and updates) over input data and registers.

* Supported in part by the European FP7 project CONNECT (IST 231167).

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 57–71, 2012.

Modeling and reasoning about systems becomes significantly easier if automata can be transformed into a canonical form: this is exploited in equivalence and refinement checking, e.g., through (bi)simulation based criteria [14,17], and in automata learning (aka regular inference) [2,9,19]. There are standard algorithms for minimization of finite automata, based on the Myhill-Nerode theorem [11,16], but it has proven difficult to carry over such constructions to automata models over infinite alphabets, including timed automata [21].

More recently, canonical automata based on extensions of the Myhill-Nerode theorem have been proposed for languages in which data values can be compared for equality [8,3,6], and also for inequality when the data domain has a total order (in [3,6]). In these works, canonicity is obtained at the price of rather strict restrictions on how data is stored in variables and which guards may be used in transitions: two variables may not store the same data value, and in the ordered case each state enforces a fixed ordering between its variables. These restrictions often cause a blow-up in the number of states, since they require testing and encoding accidental as well as essential[1] relations between data values in a word. For instance, a cross-product of two independent automata, representing, e.g., the interleaving of two independent languages, will result in a blow-up due to the recording of accidental relations between data values of the two languages.

In [7], we presented a succinct canonical automaton model, based on a Myhill-Nerode characterization, for languages where data is compared for equality. Our model does not require different variables to store different values, and allows representing only essential relations between data values. Our approach results in register automata that are minimal in a certain class, and that can be exponentially more succinct than similar, previously proposed automata formalisms [8,3,6]. We have also exploited our model for active learning of data languages [12].

In this paper, we extend our canonical automaton model of [7] to data domains where data values can be compared using an arbitrary set of relations. We consider data languages that are able to distinguish words by comparing data values using only the relations in this set, and propose a form of RA that accept such languages. To achieve succinctness, our construction must be able to filter out unnecessary tests between data values, and also produce the weakest possible guards that still make the necessary distinctions between data words. It is a challenge to achieve such succinctness while maintaining canonicity. We approach it by using a symbolic representation of data languages in the form of decision-tree-like structures, called *constraint decision trees*. Constraint decision trees have superficial similarities with decision diagrams or BDDs, but since relations on the data domain typically impose asymmetries in the tree, we cannot use the minimization techniques for BDDs. Instead, we introduce a signature-specific *branching framework* that may restrict the allowable guards, and also allows us to compare branches in the tree in order to filter out unnecessary guards. Under some conditions on the branching framework, we obtain the nontrivial result that our decision trees are minimal.

[1] By *essential* relation, we mean a test which is necessary for recognizing the language.

As an illustration, if our data domain is equipped with tests for equality and (ordered) inequality, then after processing three data values (say, d_1, d_2, d_3), it may be that the only essential test between a fourth value d_4 and these three is whether $d_4 \leq d_1$ or not (i.e., all other comparisons not essential for determining whether the data word is accepted). In previous automaton proposals, this would typically result in 7 different cases, representing all possible outcomes of testing d_4 against the three previous values. In our proposal, however, we take into account whether comparisons are essential or not, resulting in only 2 cases.

Related work. An early work on generalizing regular languages to infinite alphabets is due to Kaminski and Francez [13], who introduced finite memory automata (FMA) that recognize languages with infinite input alphabets. Since then, a number of formalisms have been suggested (pebble automata, data automata, . . .) that accept different flavors of data languages (see [20,5,4] for an overview). Many of these formalisms recognize data languages that are invariant under permutations on the data domain, corresponding to the ability to test for equality on the data domain. Much of the work focuses on non-deterministic automata and are concerned with closedness properties and expressiveness results of data languages. A model that represents relations between data values without using registers or variables is proposed by Grumberg et al. [10].

Our interest lies in canonical deterministic RAs that can be used to model the behavior of protocols or (restricted) programs. Kaminski and Francez [8], Benedikt et al. [3], and Bojanczyk et al. [6] all present Myhill-Nerode theorems for data languages with equality tests. Canonicity is achieved by restricting how state variables are stored, which prompted us to propose a more succinct construction in [7].

There are a few extensions of Myhill-Nerode theorems to more general sets of relations between data values. Benedikt et al. [3] and Bojanczyk et al. [6] consider the case where the data domain is equipped with a total order. They present canonical automata, in which stored variables must be known to obey a total order, and in which guards must be as tight as possible (we term such automata *complete*): such restrictions may lead to unintuitive and significant blow-ups in the number of control locations.

Organization. In the next section we provide a motivating example, and introduce the register automaton model as a basis for representing data languages. In Section 3, we introduce a succinct representation of data languages, which suppresses non-essential tests, in the form of a decision tree-like structure called *constraint decision trees* (CDTs). Based on this representation, in Section 4 we define a Nerode congruence, and show that it characterizes minimal canonical forms of deterministic RAs. We also discuss the effects of restricting the RA in different ways. Conclusions are provided in Section 5.

2 Data Languages and Register Automata

In this section, we introduce *data languages*. Data languages can be seen as languages over finite alphabets augmented with data. A data symbol is of the form $\alpha(d)$ where each α is an action and each d is a data value from some (possibly infinite) domain. A data word is a sequence of data symbols, and a data language is a set of data words.

We will consider data languages that can be recognized by comparing data values using relations from a given set $\mathcal{R}$ of binary relations. For example, if the set $\mathcal{R}$ of binary relations includes only the equality relation, this means that data languages will be closed under permutations on the data domain.

A *register automaton* (RA) is an automaton model capable of recognizing a data language. It reads data values as input and has registers (or variables) for storing them. When reading a data value, a register automaton can compare it to one or more variables in order to determine, e.g., its next transition. In the following sections, we will describe a register automaton model that recognizes data languages parameterized on a set of binary relations.

Example. Let $\mathcal{R} = \{<, =\}$ and let data values be rational numbers. Consider the data language L_2, consisting of data words where the last data value is the second-largest one in the entire data word. (Whenever the largest data value occurs several times, we call a data value second-largest if it is equal to the largest data value.) This language contains, for example, the data words $\alpha(\mathtt{3})$ $\alpha(\mathtt{4})$ $\alpha(\mathtt{4})$ and $\alpha(\mathtt{9})$ $\alpha(\mathtt{1})$ $\alpha(\mathtt{4})$ $\alpha(\mathtt{2})$ $\alpha(\mathtt{8})$. In the first case, the last data value is equal to the largest data value in the word. In the second case, the last data value is smaller than the largest data value in the word.

A register automaton ($\mathcal{A}_2$) that recognizes L_2 is shown in Figure 1. (For brevity, we have omitted the actions in the figure.) The $\mathcal{A}_2$ automaton has three locations, each with a set of associated variables. Accepting locations are denoted by two concentric circles, and the initial location is marked by an arrow. Arcs represent transitions, and they are labeled with guards and variable assignments. Informally, at each transition, a new data value, represented by the formal parameter p, is read by the automaton and compared to the existing location variables (using the guards). Depending on the outcome of the comparison, variables may be assigned new values, either the current data value or the value of another variable.

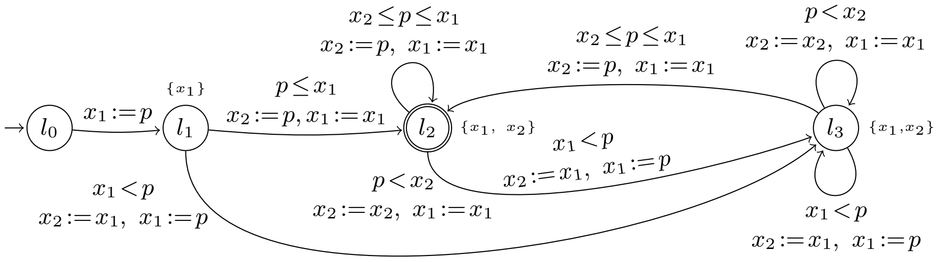


Fig. 1. Running example: the A_2 automaton

The second-largest data value seen so far is always kept in the variable x_2 and the largest data value seen so far in the variable x_1. Thus after having read the first two data values in any data word, the automaton will have reached a 'steady-state' where both variables x_1 and x_2 have stored data values, and any new data values are compared to these. The automaton will then alternate between locations l_2 and l_3 until the end of the data word is reached. This is because whenever a new data value is read by the automaton, it needs only distinguish between three cases: p is smaller than x_2, p is larger than x_1, or $x_2 \leq p \leq x_1$ in order to determine whether to transition to the accepting location l_2 or the rejecting location l_3. □

2.1 Data Languages

Assume an unbounded domain $\mathcal{D}$ of data values, and a set $\mathcal{R}$ of binary relations on $\mathcal{D}$. Assume a set of *actions*, each with an *arity* that determines how many parameters it takes from the domain $\mathcal{D}$. In this paper, we assume that all actions have arity 1; it is straightforward to extend the results to the general case.

A *data symbol* is a term of form $\alpha(d)$, where α is an action and d is a data value from the domain $\mathcal{D}$. A *data word* is a sequence of data symbols. Two data words $w_d = \alpha_1(d_1)\ldots\alpha_n(d_n)$ and $w'_d = \alpha_1(c_1)\ldots\alpha_n(c_n)$, are *equivalent*, denoted $w_d \approx_{\mathcal{R}} w'_d$, if $d_i \; R \; d_{i'} \leftrightarrow c_i \; R \; c_{i'}$ whenever $R \in \mathcal{R}$, for $1 \leq i, i' \leq n$ and $1 \leq i \leq n_j, 1 \leq i' \leq n_{j'}$. Intuitively, w_d and w'_d are equivalent if they have the same sequences of actions and they cannot be distinguished by the relations in $\mathcal{R}$. A *data language* is a set $\mathcal{L}$ of data words, which respects $\mathcal{R}$ in the sense that $w_d \approx_{\mathcal{R}} w'_d$ implies $w_d \in \mathcal{L} \leftrightarrow w'_d \in \mathcal{L}$. We will often represent a data language as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for accept and $-$ for reject.

2.2 Register Automata

Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, and a finite set of *variables* (or registers), ranged over by $x_1, x_2, \ldots$.

A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p is a formal parameter. A *parameterized word* is a sequence of parameterized symbols in which all formal parameters are distinct, i.e., we assume a (re)naming scheme that avoids clashes. A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) between formal parameters or variables.

Definition 1 (RA). *A* register automaton *(RA) is a tuple $\mathcal{A} = (L, l_0, X, T, \lambda)$, where*

- *L is a finite set of* locations*,*
- *$l_0 \in L$ is the* initial location*,*
- *X maps each location $l \in L$ to a finite set $X(l)$ of variables, where $X(l_0)$ is the empty set,*
- *T is a finite set of* transitions*, each of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$, where*

  - *l is a source location,*
  - *l' is a target location,*
  - *$\alpha(\overline{p})$ is a parameterized symbol,*
  - *g is a guard over $\overline{p}$ and $X(l)$, and*
  - *π (the* assignment*) is a mapping from $X(l')$ to $X(l) \cup \overline{p}$ (intuitively, the variable $x \in X(l')$ is assigned the value of $\pi(x)$), and*
- *$\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject),*

such that for any location l and action α, the disjunction of all guards g in transitions $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in T$ is equivalent to true *i.e., $\mathcal{A}$ is* completely specified*.* □

Semantics of a register automaton. A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. A standard way to describe how this is done is to define a state of $\mathcal{A}$ as consisting of a location and an assignment to the variables of that location. Then, one can describe how $\mathcal{A}$ processes a data word symbol by symbol: on each data symbol, $\mathcal{A}$ finds a transition with a guard that is satisfied by the parameters of the symbol and the current assignment to variables; this transition determines a next location and an assignment to the variables of the new location. When the last symbol has been processed, the word is accepted if an accepting location has been reached, otherwise the word is rejected. We omit a more formal account.

An RA is *determinate* (called a DRA) if no data word can be processed in two different ways to reach both accepting and rejecting locations. A data word is *accepted* (*rejected*) by a DRA $\mathcal{A}$ if processing the word reaches an accepting (rejecting) location. We define $\mathcal{A}(\mathtt{w}_d)$ to be $+$ $(-)$ if the data word $\mathtt{w}_d$ is accepted (rejected) by $\mathcal{A}$. The language recognized by $\mathcal{A}$ is the set of data words that it accepts.

3 Symbolic Representation of Data Languages

A given data language may be accepted by many different DRAs. In order to obtain a succinct, canonical form of DRAs, we will in this section define a canonical representation of data languages; in the next section we will describe how to derive canonical DRAs from this representation.

We first introduce a symbolic representation for sets of data words, called *constrained words*. These can also be regarded as representing runs of a register automaton. We can then use sets of constrained words, together with a classification of these words as "accepted" or "rejected", as a representation of data languages. Such classified sets will be called *constraint decision trees*. We establish, as a central result (in Theorem 1), that any data language can be represented by a *minimal* set of constrained words, corresponding to a minimal constraint decision tree. This minimal set will correspond to the set of runs of our canonical automaton, and will serve several purposes during automata construction:

- it will allow us to keep only the essential relations between data values and filter out inessential (accidental) relations between data values,

- from it, we can derive the parameters an automaton must store in variables after processing a data word, and
- we can transform parts of it directly into transitions when constructing a canonical DRA.

Constrained words. A *parameterized word* $w = \alpha_1(p_1) \cdots \alpha_k(p_k)$ is a data word where concrete data values are replaced by formal parameters. We let parameters be indexed $1 \cdots |w|$, where $|w|$ is the number of formal parameters in w, i.e., the sequence of parameters is $p_1 \cdots p_{|w|}$. A *literal* is of the form $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$, where p_i and p_j are formal parameters and $R \in \mathcal{R}$. A *constraint* ϕ is a conjunction of literals. We say that a constraint ϕ is *weaker than* a constraint ϕ', and that ϕ' is *stronger than* ϕ if ϕ' implies ϕ. A *constrained word* is a pair $\langle w, \phi \rangle$ consisting of a parameterized word w and a constraint ϕ over the formal parameters of w. If l is a literal of form $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$, then the *level of* l *in* w is the maximum of i and j. A constraint ϕ is k*-level in* w if it contains only literals of level k in w, and it is $\leq k$*-level in* w if it contains only literals of level $\leq k$ in w. Let $\phi^w|_k$ denote the conjunction of all k-level literals in $\langle w, \phi \rangle$. Similarly, define $\phi^w|_{<k}$ as the conjunction of all literals of ϕ of level smaller than k in w. Define $\phi^w|_{\leq k}$, $\phi^w|_{>k}$, and $\phi^w|_{\geq k}$ analogously.

A constraint ϕ is a k*-atom* if all its literals are of level at most k, and ϕ implies either $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$ for any $R \in \mathcal{R}$ whenever $i, j \leq k$. Intuitively, an atom is a maximal consistent constraint, i.e., it cannot be more specified without becoming inconsistent. A constrained word $\langle w, \phi \rangle$ is an *atom* if ϕ is a $|w|$-atom.

A data word $\mathtt{w}_d$ *satisfies* a constrained word $\langle w, \phi \rangle$, denoted $\mathtt{w}_d \models \langle w, \phi \rangle$, if w and $\mathtt{w}_d$ have the same sequence of actions, and the data values in $\mathtt{w}_d$ satisfy ϕ in the obvious way.

Example. Let $w = \alpha_1(p_1) \; \alpha_2(p_2) \; \alpha_3(p_3)$ be a parameterized word, and let $\phi = p_1 \leq p_3 \leq p_2$. Then $\langle w, \phi \rangle$ is a constrained word. Let $w_d = \alpha_1(3) \; \alpha_2(7) \; \alpha_3(4)$ be a data word. Then $w_d \models \langle w, \phi \rangle$, and that $\langle w, \phi \rangle$ is an atom. □

3.1 Constraint Decision Trees

We will now introduce constraint decision trees, and how they recognize data languages. Let a k*-branching* be a set of k-level constraints whose disjunction is equivalent to *true*. Let ϕ be a $\leq (k-1)$-level constraint. A k-level constraint ψ is ϕ*-admissible* (or admissible after ϕ) if ϕ implies $(\phi \wedge \exists p_k \; \psi)$, i.e., if ψ does not add any additional constraint between the parameters of ϕ. A k-branching Ψ is ϕ*-admissible* if each k-level constraint in Ψ is ϕ-admissible.

A set Φ of constrained words is *prefix-closed* if $\langle wv, \phi \rangle \in \Phi$ implies $\langle w, \phi^{wv}|_{\leq |w|} \rangle \in \Phi$. A set Φ is *extension-closed* if for any $\langle w, \phi \rangle \in \Phi$ and any action α, the set of $(|w|+1)$-level constraints ψ such that $\langle w\alpha(p_{|w|+1}), \phi \wedge \psi \rangle \in \Phi$ forms a ϕ-admissible $(|w|+1)$-branching.

Definition 2 (CDT). *A* constraint decision tree *(CDT)* $\mathcal{T}$ *is a pair* $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}} \rangle$ *where* $Dom(\mathcal{T})$ *is a non-empty prefix-closed and extension-closed set of constrained words, and* $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto \{+, -\}$ *is a mapping from* $Dom(\mathcal{T})$ *to* $\{+, -\}$. □

A CDT $\mathcal{T}$ is *determinate* (called a DCDT) if $\lambda_{\mathcal{T}}(\langle w, \phi\rangle) = \lambda_{\mathcal{T}}(\langle w, \phi'\rangle)$ whenever $\mathtt{w}_d \models \langle w, \phi\rangle$ and $\mathtt{w}_d \models \langle w, \phi'\rangle$ for some data word $\mathtt{w}_d$. It is *complete* if all constrained words in $Dom(\mathcal{T})$ are atoms.

A DCDT defines a language $\lambda_{\mathcal{T}}$ defined by $\lambda_{\mathcal{T}}(\mathtt{w}_d) = \lambda_{\mathcal{T}}(\langle w, \phi\rangle)$ whenever $\mathtt{w}_d \models \langle w, \phi\rangle$. Intuitively, a CDT can be thought of as a set of runs of a register automaton. Each constrained word $\langle w, \phi\rangle$ represents a path through the automaton: the parameterized word w is the sequence of actions and formal parameters, and ϕ is the conjunction of all guards that are tested along the path. A design constraint for our canonical automaton model is that all essential tests concerning the relationship between a parameter p_i and previously received parameters (i.e., parameters p_j with $j < i$) should be performed when p_i is processed. This is reflected in the property of admissibility, which intuitively means that a guard should not retroactively constrain the relation between previously received parameters. The property of extension-closed implies that the CDT is completely specified in the sense that it can classify any data word as accepted or rejected.

In the following, we will show that for each language $\mathcal{L}$, we can construct a canonical CDT that faithfully represents $\mathcal{L}$, and which is also minimal under some restrictions. We will first try to provide some intuition for our construction.

Constructing a canonical DCDT. A first attempt at constructing a canonical DCDT $\mathcal{T}$ could be to simply include all atoms in the domain $Dom(\mathcal{T})$, thus resulting in a complete DCDT. This DCDT will surely be able to correctly classify a data language, but it will typically be prohibitively large. We need to find ways to reduce the size of the CDT while still rendering it capable to correctly classify the language it represents.

Example. Let $\mathcal{D}$ be the set of rational numbers, and let $\mathcal{R} = \{<, =\}$. Assume that $\mathcal{T}$ is a complete DCDT. The constrained words in $Dom(\mathcal{T})$ of the form $\langle a(p_1)b(p_2)c(p_3), \phi\rangle$ would then be such that ϕ specifies some total order between p_1, p_2, p_3. The question is then whether we actually need a total order between the parameters in order to correctly classify the data language represented by $\mathcal{T}$. Perhaps this language is insensitive to the ordering between p_2 and p_3, or it simply does not distinguish the case $p_2 < p_3$ from $p_2 = p_3$. We would like the CDT to reflect this by replacing atoms by weaker constrained words. A constrained word is weaker than an atom if the atom implies the constrained word. This means that a constrained word can be used to represent several atoms, i.e., we can 'merge' the atoms.

Two atoms can be represented by the same constraint if any constraint that is admissible after the one atom is also admissible after the second atom (and vice versa), and their classifications (accept/reject) match. However, sometimes we want to merge atoms that do not fulfill these conditions. Consider the atoms $p_1 < p_2 \wedge p_2 < p_3$ and $p_1 < p_2 \wedge p_2 = p_3$. We can *not* add the same set of constraints after both atoms; for instance, the 4-level constraint $p_2 < p_4 \wedge p_4 < p_3$ is admissible after the atom $p_1 < p_2 \wedge p_2 < p_3$, but not after $p_1 < p_2 \wedge p_2 = p_3$.

We can solve this problem by introducing an ordering $\sqsubseteq_\phi$ between extensions of an atom. We then try to use the $\sqsubseteq_\phi$-smaller extension to classify the larger extension. This can be done if the classifications match and the resulting constraints are admissible. In the above example, $p_2 < p_3$ and $p_2 = p_3$ extend the atom $p_1 < p_2$. If we order them as $p_2 < p_3 \sqsubseteq_\phi p_2 = p_3$, we can check if the classification of the extensions of $p_1 < p_2 \wedge p_2 = p_3$ matches the classification of the extensions of $p_1 < p_2 \wedge p_2 < p_3$. If they do, we can merge the atoms into $p_1 < p_2 \wedge p_2 \leq p_3$. □

3.2 Branching Frameworks

Let us now describe the structure that must be predefined in order to define a canonical DCDT. We assume that the set $\mathcal{R}$ of binary relations on $\mathcal{D}$ is fixed.

Let ϕ be a $k-1$-constraint. A *guard hierarchy for* ϕ is a set $\mathcal{P}$ of ϕ-admissible k-level constraints, in which the set of maximally strong (wrp. to implication) constraints ψ (called *atomic branches* of $\mathcal{P}$) are such that $\phi \wedge \psi$ is an atom, and such that the set of atomic branches of $\mathcal{P}$ forms a ϕ-admissible k-branching.

Definition 3 (Branching framework). *A* branching framework *is a mapping $\mathcal{M}$ which to each k–1-atom ϕ assigns a pair $\langle \mathcal{P}, \sqsubseteq_\phi \rangle$, where $\mathcal{P}$ is a guard hierarchy for ϕ, and $\sqsubseteq_\phi$ is a partial order on the atomic branches of $\mathcal{P}$.* □

Intuitively, the elements of $\mathcal{P}$ are the possible k-level constraints that test the next parameter that follows after ϕ in a CDT. The atomic branches represent the "most constrained" k-level constraints that completely characterize how the next parameter p_k is related to previous parameters $p_1, \ldots, p_{k-1}$. Note that different guards or atomic branches need not be mutually exclusive.

For any k-level constraint g in $\mathcal{P}$, define the *support of g after ϕ*, denoted *supp*$^\phi(g)$, as the set of atomic branches $\psi \in \mathcal{P}$ such that $(\phi \wedge \psi)$ implies $(\phi \wedge g)$. Since the set of atomic branches forms a k-branching, it follows that any element g in $\mathcal{P}$ represents the set of atomic branches in *supp*$^\phi(g)$ in the sense that

$$(\phi \wedge g) \leftrightarrow (\phi \wedge \bigvee_{\psi \in supp^\phi(g)} \psi) \ .$$

Definition 4. *A branching framework $\mathcal{M}$ is* adequate *if whenever $\mathcal{M}(\phi) = \langle \mathcal{P}, \sqsubseteq_\phi \rangle$, then*

- *for each constraint g in $\mathcal{P}$, the set supp$^\phi(g)$ contains a unique minimal (wrp. to $\sqsubseteq_\phi$) atomic branch, called the* principal atomic branch *of g, and*
- *Whenever two constraints g, g' in $\mathcal{P}$ have the same principal atomic branch, then $g \vee g'$ is in $\mathcal{P}$.* □

The concept of principal atomic branch can be extended from k-constraints to arbitrary constrained words as follows. For a CDT $\mathcal{T}$, an adequate branching framework $\mathcal{M}$, and any constrained word $\langle w, \phi \rangle$, define the *principal atom* of ϕ inductively, as follows:

- The principal atom of the empty constraint *true* over the empty sequence of parameters is *true*

- If ϕ is a constraint over $p_1, \ldots, p_k$, let ϕ' be the principal atom of $\phi^w|_{\leq k-1}$, let ψ be $\phi^w|_k$, and let $\mathcal{M}(\phi')$ be $\langle \mathcal{P}, \sqsubseteq_\phi \rangle$. Then the principal atom of ϕ over $p_1, \ldots, p_k$ is the atom $\phi' \wedge \psi'$, where ψ' is the principal atomic branch of ψ.

The branching framework determines what kinds of constraints we allow in the CDT. For example, if we are dealing with data values that are rational numbers and subject to some order $<$, we might allow constraints to specify intervals (such as $p_2 < p_4 < p_1$). Whenever we can extend a constraint ϕ by some other constraint g, we must also be able to extend the principal atom of ϕ by g, i.e., the constraints ϕ and g must be compatible.

Example. Consider the case where $\mathcal{R}$ is $\{=\}$. We can obtain the model in our previous work [7], by a branching framework which assigns to a $\leq (k-1)$-atom ϕ stating that the parameters $p_1, \ldots, p_{k-1}$ are all different, the pair $\langle \mathcal{P}, \sqsubseteq_\phi \rangle$, where

- $\mathcal{P}$ consists of all conjunctions of subsets of the literals $p_1 \neq p_k, \cdots, p_{k-1} \neq p_k$ (of these, only the maximal one is an atom), as well as the $k-1$ constraints of form $p_i = p_k$ for $i = 1, \ldots, k-1$, and
- $(p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k) \sqsubseteq_\phi p_i = p_k$ for $i = 1, \ldots, k-1$, but $p_i = p_k$ and $p_j = p_k$ are not ordered for $i \neq j$.

Consequently, each k-level constraint with nontrivial support will have $p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k$ as principal atomic branch. □

Example. Let us next consider the case where $\mathcal{D}$ is the set of rational numbers, and $\mathcal{R}$ is $\{<, =\}$ (it is important not to let $\mathcal{D}$ be some set of integers, since that case is more complicated). An atom ϕ over $p_1, \ldots, p_{k-1}$ will specify some order between $p_1, \ldots, p_{k-1}$, for example $p_1 < \cdots < p_{k-1}$. A suitable branching framework will assign to a $\leq (k-1)$-atom ϕ the pair $\langle \mathcal{P}, \sqsubseteq_\phi \rangle$, where

- Ψ contains all possible non-empty intervals between to parameters p_i, p_j with $1 \leq i \leq j \leq k$, e.g., $p_2 \leq p_k < p_5$, or (some degenerate cases) $p_k = p_i$, or $p_k < p_1$, or $p_{k-1} < p_k$. Among these, only the minimal intervals are atoms.
- The relation $\sqsubseteq_\phi$ will then be the smallest transitive relation that fulfills the following conditions:
  - $p_i < p_k < p_{i+1} \sqsubseteq_\phi p_{i-1} < p_k < p_i$ (for $i = 2, \ldots, k-2$), and
$p_1 < p_k < p_2 \sqsubseteq_\phi p_k < p_1$, and
$p_{k-1} < p_k \sqsubseteq_\phi p_{k-2} < p_k < p_{k-1}$,
  - $p_{i-1} < p_k < p_i \sqsubseteq_\phi p_k = p_i$ (for $i = 2, \ldots, k-1$), and
$p_k < p_1 \sqsubseteq_\phi p_k = p_1$.

□

Definition 5. Let $\mathcal{T}$ be a CDT, and let $\mathcal{M}$ be an adequate branching framework. Then $\mathcal{T}$ is an *$\mathcal{M}$-CDT* if for any $\langle w, \phi \rangle \in Dom(\mathcal{T})$ with ϕ' being the principal atom of ϕ, it is the case that whenever ψ is a $|w|+1$-level constraint with $\langle w\alpha(p_{|w|+1}), \phi \wedge \psi \rangle \in Dom(\mathcal{T})$, then $\psi \in \mathcal{P}$, where $\langle \mathcal{P}, \sqsubseteq_\phi \rangle = \mathcal{M}(\phi')$ □

3.3 Minimal Constraint Decision Trees

We will now show how to obtain a minimal constraint decision tree for a data language.

Theorem 1 (Minimal DCDT). Let $\mathcal{M}$ be an adequate branching framework. Then for any data language $\mathcal{L}$, there is a unique minimal $\mathcal{M}$-DCDT $\mathcal{T}$ such that $\mathcal{L} = \lambda_{\mathcal{T}}$. □

By minimal, we mean that if $\mathcal{T}'$ is any other $\mathcal{M}$-DCDT with $\mathcal{L} = \lambda_{\mathcal{T}'}$, then any constrained word in $Dom(\mathcal{T}')$ is contained in a constrained word in $Dom(\mathcal{T})$. Intuitively, this means that $Dom(\mathcal{T})$ uses the weakest possible constraints that are necessary in order to be able to correctly recognize the language $\mathcal{L}$. We will sometimes use the term *$\mathcal{L}$-essential* (constrained) words (or just $\mathcal{L}$-essential words) for members of $Dom(\mathcal{T})$ where $\mathcal{T}$ is the minimal DCDT with $\mathcal{L} = \lambda_{\mathcal{T}}$.

Proof. (Sketch.) We prove Theorem 1 by construction. The minimal DCDT for $\mathcal{L}$ is constructed starting from the $\sqsubseteq_{\phi}$-minimal atoms that serve as the leaf nodes. Atoms are merged to form guards in a bottom-up fashion. This is only possible if we assume a bounded length of words that are classified by $\mathcal{L}$. We will therefore assume a maximal length n of data words and construct a minimal "truncated" DCDT $\mathcal{T}_n$, which correctly classifies data words of length at most n. We can then show that $\mathcal{T}_n$ "grows monotonically" with increasing n, so that $\mathcal{T}$ can be taken as a limit of the trees $\mathcal{T}_n$.

Example. We will now describe how to construct the canonical CDT for our running example L_2. Recall that a branching framework assigns a set of partially ordered branches to each atom. The partial order determines in what order we will add branches as guards in the minimal CDT, and also which branches can be merged to form guards. We will consider atoms of increasing k-level. At each level, we will construct the subtree (set of constraints) of each atomϕ in increasing $\sqsubseteq_{\phi}$-order. We will then check if any of the atoms can be merged. At the leaf level in the tree, atoms can be merged if they have the same classification. For reasons of brevity, we will here only consider words of length 3 at most.

In the L_2 example, the 1-level atom is *true*. We generate the set of 2-level atomic branches, ordered as $p_1 < p_2 \sqsubseteq_{\emptyset} p_2 < p_1 \sqsubseteq_{\emptyset} p_1 = p_2$.

We apply the branching framework to the smallest branch $p_1 < p_2$, and obtain the set of 3-level atomic branches after $p_1 < p_2$, ordered as $\{p_2 < p_3 \sqsubseteq_{(p_1 < p_2)} p_1 < p_3 < p_2 \sqsubseteq_{(p_1 < p_2)} p_3 = p_2, \;\; p_1 < p_3 < p_2 \sqsubseteq_{(p_1 < p_2)} p_3 < p_1 \sqsubseteq_{(p_1 < p_2)} p_3 = p_1\}$. The smallest branch is $p_2 < p_3$, which is a rejecting leaf. We construct the first 3-level L_2-essential constrained word as $p_2 < p_3$.

The next smallest branch is $p_1 < p_3 < p_2$ which is accepted. It can be merged with the larger branches $p_1 = p_3$ and $p_2 = p_3$, since they are both also accepting. (Because we are at the leaf level in the tree, this is sufficient to determine whether branches can be merged.) We thus construct the second 3-level L_2-essential constraint as $p_1 \leq p_3 \leq p_2$. The only branch left is now $p_3 < p_1$, which will become the third L_2-essential constraint.

There are no more 3-level branches to classify, so we go back to level 2. Since we have already constructed the subtree after $p_1 < p_2$, we proceed to construct the subtree after the next smallest 2-level branch, $p_2 < p_1$ in the same manner. This results in the L_2-essential constraints $p_2 \leq p_3 \leq p_1$, $p_3 < p_2$ and $p_1 < p_3$.

The last remaining 2-level branch is $p_1 = p_2$. We construct the subtree, resulting in the constraints $p_1 = p_2 = p_3$, $p_3 < p_1 = p_2$ and $p_1 = p_2 < p_3$. It is possible to use the subtree after $p_2 < p_1$ to classify the constraints after $p_1 = p_2$, so we can merge these atoms into $p_2 \leq p_1$.

We obtain the set of L_2-essential constrained words of length 3 as
$\{\langle w, p_3 < p_2 \leq p_1 \rangle, \langle w, p_2 \leq p_3 \leq p_1 \rangle, \langle w, p_2 \leq p_1 < p_3 \rangle, \langle w, p_3 < p_1 < p_2 \rangle,$
$\langle w, p_1 \leq p_3 \leq p_2 \rangle, \langle w, p_1 < p_2 < p_3 \rangle\}$ where $w = \alpha(p_1)\alpha(p_2)\alpha(p_3)$.

The L_2-essential constrained words directly correspond to paths in the $\mathcal{A}_2$ automaton in Figure 1. For example, the path $l_0 \rightarrow l_1 \rightarrow l_3 \rightarrow l_3$ corresponds to the constrained word $\langle w, p_3 < p_1 < p_2 \rangle$. Since $\mathcal{A}_2$ always stores the largest data value seen so far in the variable x_1 and the second-largest seen so far in x_2, this path requires variables to be reassigned. The variable x_1 will first store the first data value. Then, because the second data value is larger than the first, x_1 will be re-assigned the second data value (and the first data value will be moved to x_2).

4 Nerode Congruence

We will now define a Nerode-like congruence on constrained words. As in the classical Nerode congruence for regular languages, we will define constrained words to be equivalent if their suffixes are equivalent.

In order to describe how to fold a constraint decision tree into a register automaton, we need to decide which parameters to store as variables in the automaton. We associate a set of *memorable* parameters with each constrained word. These are the parameters that occur in the word and are needed in some guard in some of its suffixes. When the CDT is folded into a register automaton, the memorable parameters of a node will become location variables at the location that corresponds to that particular node.

Let us first define how a constrained word can be split into a prefix and a suffix. Consider a constrained word $\langle w, \phi \rangle$, where the parameterized word w is a concatenation uv, and u has k parameters. We can make a corresponding split of ϕ as $\phi^w|_{\leq k} \wedge \phi^w|_{>k}$. Then $\langle u, \phi^w|_{\leq k} \rangle$ (the prefix) is a constrained word, but $\langle v, \phi^w|_{>k} \rangle$ (the suffix) is in general not, since $\phi^w|_{>k}$ refers to parameters that are not in v. We therefore define a $\langle u, \phi \rangle$*-suffix* as a tuple $\langle v, \psi \rangle$, where ψ is a constraint over parameters of u and v, in which each literal contains at least one parameter from v, and such that $\langle uv, \phi \wedge \psi \rangle$ (which we often denote $\langle u, \phi \rangle; \langle v, \psi \rangle$) is a constrained word.

Definition 6 (Memorable). *Let $\mathcal{L}$ be a data language, let $\mathcal{M}$ be an adequate branching framework, and let $\mathcal{T}$ be the minimal $\mathcal{M}$-DCDT recognizing $\mathcal{L}$. The $\mathcal{L}$-*memorable parameters *of a constrained word $\langle w, \phi \rangle \in Dom(\mathcal{T})$, denoted $mem_{\mathcal{L}}(\langle w, \phi \rangle)$, is the set of parameters in w that occur in some $\langle w, \phi \rangle$-suffix $\langle v, \psi \rangle$ such that $\langle w, \phi \rangle; \langle v, \psi \rangle \in Dom(\mathcal{T})$.* □

In order for our canonical form to capture exactly the causal relations between parameters, we will allow memorable parameters to be permuted. when comparing words. Two words will be considered equivalent if they require equivalent parameters to be stored, independent of their ordering or their names.

Definition 7 (Nerode congruence). *Let $\mathcal{L}$ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing $\mathcal{L}$. We define the equivalence $\equiv_{\mathcal{L}}$ on constrained words by $\langle w, \phi\rangle \equiv_{\mathcal{L}} \langle w', \phi'\rangle$ if there is a bijection $\gamma : mem_{\mathcal{L}}(\langle w, \phi\rangle) \mapsto mem_{\mathcal{L}}(\langle w', \phi'\rangle)$ such that*

- *$\langle v, \psi\rangle$ is a $\langle w, \phi\rangle$-suffix such that $\langle wv, \phi \wedge \psi\rangle \in Dom(\mathcal{T})$ iff $\langle v, \gamma(\psi)\rangle$ is a $\langle w', \phi'\rangle$-suffix such that $\langle w'v, \phi' \wedge \gamma(\psi)\rangle \in Dom(\mathcal{T})$, and then*
- *$\mathcal{L}(\langle wv, \phi \wedge \psi\rangle) = \mathcal{L}(\langle w'v, \phi' \wedge \gamma(\psi)\rangle)$,*

where $\gamma(\psi)$ is obtained from ψ by replacing all parameters in $mem_{\mathcal{L}}(\langle w, \phi\rangle)$ by their image under γ.

Intuitively, two constrained words are equivalent if they induce the same residual languages modulo a remapping of their memorable parameters. The equivalence $\equiv_{\mathcal{L}}$ is also a congruence in the following sense. If $\langle w, \phi\rangle \equiv_{\mathcal{L}} \langle w', \phi'\rangle$ is established by the bijection $\gamma : \mathrm{mem}_{\mathcal{L}}(\langle w, \phi\rangle) \mapsto \mathrm{mem}_{\mathcal{L}}(\langle w', \phi'\rangle)$, then for any $\mathrm{mem}_{\mathcal{L}}(\langle w, \phi\rangle)$-suffix $\langle v, \psi\rangle$ we have $\langle w, \phi\rangle; \langle v, \psi\rangle \equiv_{\mathcal{L}} \langle w', \phi'\rangle; \langle v, \gamma(\psi)\rangle$.

By using the Nerode congruence, we can 'fold' a constraint decision tree into a register automaton, mapping constrained words that are equivalent by this congruence to the same location.

We are now able to relate our Nerode congruence to DRAs.

Theorem 2 (Myhill-Nerode). A data language $\mathcal{L}$ is recognizable by a DRA iff the equivalence $\equiv_{\mathcal{L}}$ on $\mathcal{L}$-essential words has finite index.

Proof. (Sketch.) We will not detail the extension of the Myhill-Nerode theorem here, but refer to the version stated in [7]For the if-direction, we construct a DRA from a congruence. The locations of the resulting DRA will be given by the finitely many equivalence classes of the Nerode relation on essential words. Transitions will be extracted from the representative words in each equivalence class. Location variables will be given by the memorable parameters of the representative words. For the only-if direction, we assume any DRA that accepts $\mathcal{L}$. The proof idea then is to show that two $\mathcal{L}$-essential constrained words corresponding to sequences of transitions that lead to the same location are also equivalent w.r.t. $\equiv_{\mathcal{L}}$, i.e., that one location of a DRA cannot represent more than one class of $\equiv_{\mathcal{L}}$. This can be shown using congruence properties. □

Let us now reconsider our running example. In general, many different register automata can be constructed that accept some given data language – for example, one that simply stores each new data value in a location variable, regardless of whether or not it will be referenced later. This leads to an unnecessary blowup in the number of location variables. Another automaton might only store data values that will be referenced later, but perform all possible tests on newly received data values. Such an automaton would distinguish between the two cases

$x_2 \leq p < x_1$ and $x_2 = p < x_1$ in the self-loop at location l_2 in the automaton of Figure 1. In general, canonical models are easier to define if strong restrictions are imposed on their form. In previous proposals, typical such restrictions include to maintain a priori relations between stored variables (e.g., that $x_2 < x_1$), or that guards always be as strong as possible (thus duplicating the self-loop at location l_2). In this paper, we have lifted several such restrictions, while still producing canonical automata.

Example. Consider the $\mathcal{A}_2$ automaton in Figure 1. Here, the locations l_2 and l_3 both have two location variables x_1 and x_2. The variable x_1 always stores the largest data value seen so far, and the variable x_2 stores the second-largest data value seen so far. There is no uniqueness restriction on the variables, so it may well be that x_1 and x_2 store equal data values. Imposing the uniqueness restriction on the location variables of this automaton will lead to l_2 and l_3 being duplicated. The automaton will thus have two accepting states and two rejecting, depending on whether the two largest data values seen so far are equal or not. (The 'initial' state l_0 has no location variables; l_1 only has one location variable, and they will both stay that way.) □

5 Conclusions and Future Work

In this paper, we have presented a succinct canonical register automaton model for data languages, in which data values can be compared by an arbitrary given set of relations. This construction consistently and significantly generalizes our previous work [7], which considered only the equality relation. Our construction gives rise to automata that are often considerably more succinct than those of previous proposals.

The main technical contribution of the paper is the symbolic treatment of data languages using branching frameworks to organizate relations on the data domain canonically. This allows us to extend our ideas from [7] resulting in a Myhill Nerode-like theorem for this larger class of data languages.

Our immediate plans are to use these results to derive canonical models of realistic protocols, services, and interfaces, as well as generalizing Angluin-style active learning to this class of systems.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
3. Benedikt, M., Ley, C., Puppis, G.: What you must remember when processing data words. In: Proc. 4th Alberto Mendelzon Int. Workshop on Foundations of Data Management, Buenos Aires, Argentina. CEUR Workshop Proceedings, vol. 619 (2010)

4. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theoretical Computer Science 411, 702–715 (2010)
5. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Log. 12(4), 27 (2011)
6. Bojanczyk, M., Klin, B., Lasota, S.: Automata with group actions. In: LICS, pp. 355–364. IEEE Computer Society (2011)
7. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A Succinct Canonical Register Automaton Model. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 366–380. Springer, Heidelberg (2011)
8. Francez, N., Kaminski, M.: An algebraic characterization of deterministic regular languages over infinite alphabets. Theoretical Computer Science 306(1-3), 155–175 (2003)
9. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
10. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable Automata over Infinite Alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010)
11. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
12. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring Canonical Register Automata. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 251–266. Springer, Heidelberg (2012)
13. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329–363 (1994)
14. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation 86(1), 43–68 (1990)
15. Lazić, R.S., Nowak, D.: A Unifying Approach to Data-Independence. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 581–595. Springer, Heidelberg (2000)
16. Nerode, A.: Linear Automaton Transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
17. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal of Computing 16(6), 973–989 (1987)
18. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Trans. on Software Engineering 30(1), 29–42 (2004)
19. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation 103(2), 299–347 (1993)
20. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
21. Wilke, T.: Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 694–715. Springer, Heidelberg (1994)

Rabinizer: Small Deterministic Automata for LTL(F,G)

Andreas Gaiser[1,⋆], Jan Křetínský[1,2,⋆⋆], and Javier Esparza[1]

[1] Institut für Informatik, Technische Universität München, Germany
{gaiser,jan.kretinsky,esparza}@model.in.tum.de
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. We present Rabinizer, a tool for translating formulae of the fragment of linear temporal logic with the operators **F** (eventually) and **G** (globally) into deterministic Rabin automata. Contrary to tools like ltl2dstar, which translate the formula into a Büchi automaton and apply Safra's determinization procedure, Rabinizer uses a direct construction based on the logical structure of the formulae. We describe a number of optimizations of the basic procedure, crucial for the good performance of Rabinizer, and present an experimental comparison.

1 Introduction

The automata-theoretic approach to model checking is one of the most important successes of theoretical computer science in the last decades. It has led to many tools of industrial strength, like Holzmann's SPIN. In its linear-time version, the approach translates the negation of a specification, formalized as a formula of Linear Time Temporal logic (LTL), into a non-deterministic ω-automaton accepting the possible behaviours of the system that violate the specification. Then the product of the automaton with the state space of the system is constructed, and the resulting ω-automaton is checked for emptiness. Since the state space can be very large (medium-size systems can easily have tens of millions of states) and the size of a product of automata is equal to the product of their sizes, it is crucial to transform the formula into a small ω-automaton: saving one state in the ω-automaton may amount to saving tens of millions of states in the product. For this reason, cutting down the number of states has been studied in large depth, and very efficient tools like LTL2BA [4] have been developed.

In recent years the theory of the automata-theoretic approach has been successfully extended to probabilistic systems and to synthesis problems. However, these applications pose a new challenge: they require to translate formulas into *deterministic* ω-automata (loosely speaking, the applications require a game-theoretical setting with $1\frac{1}{2}$ or 2 players, whose arenas are only closed under product with deterministic automata) [3]. For this, deterministic Rabin, Streett, or parity automata can be used. The standard approach is to first transform LTL

[⋆] The author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

[⋆⋆] The author is supported by the Czech Science Foundation, grant No. P202/12/G061.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 72–76, 2012.

formulae into non-deterministic Büchi automata and then translate these into deterministic Rabin automata by means of Safra's construction [10]. (The determinization procedure of Muller-Schupp is known to produce larger automata [1].) In particular, this is the procedure followed by `ltl2dstar` [5], the tool used in PRISM [8], the leading probabilistic model checker. However, the approach has two disadvantages: first, since Safra's construction is not tailored for Büchi automata derived from LTL formulae, it often produces (much) larger automata than necessary; second, there are no efficient ways to minimize Rabin automata.

We have recently presented a procedure to directly transform LTL formulae into deterministic automata [7]. The procedure, currently applicable to the (**F**,**G**)-fragment of the logic, heavily exploits formula structure to yield much smaller automata for important formulae, in particular for formulae describing fairness. For instance, a conjunction of three fairness constraints requiring more than one million states with `ltl2dstar` only required 1560 states in [7].

While the experiments of [7] are promising, they were conducted using a primitive implementation. In this paper we report on subsequent work that has transformed the prototype of [7] into `Rabinizer`, a tool incorporating several non-trivial optimizations, and mature enough to be offered to the community. For example, for the formula above, `Rabinizer` returns an automaton with only 462 states.

2 Rabinizer and Optimizations

We assume the reader is familiar with LTL and ω-automata. The idea of the construction of [7] is the following. The states of the Rabin automaton consist of two components. The first component is the LTL formula that, loosely speaking, remains to be satisfied. For example, if a state has $\varphi = \mathbf{F}a \wedge \mathbf{G}b$ as first component, then reading the label $\{a, b\}$ leads to another state with $\mathbf{G}b$ as first component. The second component remembers the last label read (one-step history). To see why this is necessary, observe that for $\varphi = \mathbf{GF}a$ the first component of all states is the same. The second component allows one to check whether a is read infinitely often. Finally, while the Rabin acceptance condition is a disjunction of Rabin pairs, the construction yields a disjunction of *conjunctions* of Rabin pairs, cf. e.g. $\varphi = \mathbf{GF}a \wedge \mathbf{GF}b$, and so in a final step the "generalized Rabin" automaton is expanded into a Rabin automaton.

The only optimizations of implementation used for the experiments of [7] are the following. Firstly, only the reachable state space is constructed. Secondly, the one-step history only records letters appearing in the first component of each state. Thirdly, a simple subsumption of generalized Rabin conditions is considered and the stronger (redundant) conditions are removed. Nevertheless, no algorithm to do this has been presented and manual computation had to be done to obtain the optimized results.

`Rabinizer` is a mature implementation of the procedure of [7] with several additional non-trivial optimizations. `Rabinizer` is written in Java, and uses BDDs to construct the state space of the automata and generate Rabin pairs. While for "easy" formulas `ltl2dstar` would often generate slightly smaller automata than the implementation of [7], `Rabinizer` only generates a larger automaton in 1 out of 27 benchmarks. Moreover, for "difficult" formulas, `Rabinizer` considerably outperforms the previous implementation. We list the most important optimizations performed.

- The evolution of the first component containing the formula to be satisfied has been altered. In the original approach, in order to obtain the acceptance condition easily, not all known information has been reflected immediately in the state space thus resulting in redundant "intermediate" states.
- The generalized Rabin condition is now subject to several optimizations. Firstly, conjunctions of "compatible" Rabin pairs are merged into single pairs thus reducing the blowup from generalized Rabin to Rabin automaton. Secondly, some subformulae, such as outer **F** subformulae, are no more considered in the acceptance condition generation.
- The one-step history now does not contain full information about the letters, but only equivalence classes of letters. The quotienting is done in the coarsest way to still reflect the acceptance condition. A simple example is a formula $\varphi = \mathbf{GF}(a \vee b)$ where we only distinguish between reading any of $\{\{a\}, \{b\}, \{a, b\}\}$ and reading $\emptyset$.
- The blow-up of the generalized Rabin automaton into a Rabin automaton has been improved. Namely, the copies of the original automaton are now quotiented one by one according to the criterion above, but only the conjuncts corresponding to a particular copy are taken into account. Thus we obtain smaller (and different) copies.
  Further, linking of the copies is now made more efficient. Namely, the final states in all but one copy have been removed completely.
- No special state is dedicated to be initial without any other use. Although this results only in a decrease by one, it plays a role in tiny automata.

For further details, correctness proofs, and a detailed input/output description, see `Rabinizer`'s web page `http://www.model.in.tum.de/tools/rabinizer/`.

3 Experimental Results

The following table shows the results on formulae from BEEM (BEnchmarks for Explicit Model checkers)[9] and formulae from [11] on which `ltl2dstar` was originally tested [6]. In both cases, we only take formulae of the (**F**,**G**)-fragment. In the first case this is 11 out of 21, in the second 12 out of 28. There is a slight overlap between the two sets. Further, we add conjunctions of strong fairness conditions and a few other formulae.

For each formula φ, we give the size of the Rabin automaton generated by `ltl2dstar` (using the recommended configuration with `LTL2BA`), the prototype of [7], and `Rabinizer`. For reader's convenience, we also include the size of *non-deterministic* Büchi automata generated by `LTL2BA` [4] and its recent improvement `LTL3BA` [2] whenever they differ. The last two columns state the number of Rabin pairs for automata generated by `ltl2dstar` and `Rabinizer`, respectively.

In all the cases but one, `Rabinizer` generates automata of the same size as `ltl2dstar` or often considerably smaller. Further, while in some cases `Rabinizer` generates one additional pair, it generates less pairs when the number of pairs is high. Runtimes have not been included, as `Rabinizer` transforms all formulae within a second except for the conjunction of three fairness constraints. This one took 13 seconds on an Intel i7 with 8 GB RAM, whereas `ltl2dstar` crashes here and needs more than one day on a machine with 64 GB of RAM.

Formula	`ltl2dstar`	[7]	`Rabinizer`	`LTL2(3)BA`	$*$-pairs	R-pairs
$\mathbf{G}(a \vee \mathbf{F}b)$	4	5	4	2	**1**	2
$\mathbf{FG}a \vee \mathbf{FG}b \vee \mathbf{GF}c$	8	9	8	6	3	3
$\mathbf{F}(a \vee b)$	2	4	2	2	1	1
$\mathbf{GF}(a \vee b)$	2	3	2	2	1	1
$\mathbf{G}(a \vee \mathbf{F}a)$	4	3	**2**	2	**1**	2
$\mathbf{G}(a \vee b \vee c)$	3	4	**2**	1	1	1
$\mathbf{G}(a \vee \mathbf{F}(b \vee c))$	4	5	4	2	**1**	2
$\mathbf{F}a \vee \mathbf{G}b$	4	7	**3**	4	2	2
$\mathbf{G}(a \vee \mathbf{F}(b \wedge c))$	4	5	4	5 (2)	**1**	2
$\mathbf{FG}a \vee \mathbf{GF}b$	4	5	4	5	2	2
$\mathbf{GF}(a \vee b) \wedge \mathbf{GF}(b \vee c)$	7	10	**3**	3	2	**1**
$(\mathbf{FF}a \wedge \mathbf{G}\neg a) \vee (\mathbf{GG}\neg a \wedge \mathbf{F}a)$[1]	1	4	1	1	0	0
$\mathbf{GF}a \wedge \mathbf{FG}b$	3	5	3	3	1	1
$(\mathbf{GF}a \wedge \mathbf{FG}b) \vee (\mathbf{FG}\neg a \wedge \mathbf{GF}\neg b)$	5	5	**4**	7	2	2
$\mathbf{FG}a \wedge \mathbf{GF}a$	2	3	2	2 (3)	1	1
$\mathbf{G}(\mathbf{F}a \wedge \mathbf{F}b)$	5	10	**3**	3	1	1
$\mathbf{F}a \wedge \mathbf{F}\neg a$	4	8	4	4	1	1
$(\mathbf{G}(b \vee \mathbf{GF}a) \wedge \mathbf{G}(c \vee \mathbf{GF}\neg a)) \vee \mathbf{G}b \vee \mathbf{G}c$	**13**	36	18	11	**3**	4
$(\mathbf{G}(b \vee \mathbf{FG}a) \wedge \mathbf{G}(c \vee \mathbf{FG}\neg a)) \vee \mathbf{G}b \vee \mathbf{G}c$	14	18	**6**	12 (8)	4	**3**
$(\mathbf{F}(b \wedge \mathbf{FG}a) \vee \mathbf{F}(c \wedge \mathbf{FG}\neg a)) \wedge \mathbf{F}b \wedge \mathbf{F}c$	7	18	**5**	15 (10)	**1**	2
$(\mathbf{F}(b \wedge \mathbf{GF}a) \vee \mathbf{F}(c \wedge \mathbf{GF}\neg a)) \wedge \mathbf{F}b \wedge \mathbf{F}c$	7	18	**5**	13 (10)	2	2
$(\mathbf{GF}a \rightarrow \mathbf{GF}b)$	4	5	4	5	2	2
$(\mathbf{GF}a \rightarrow \mathbf{GF}b) \wedge (\mathbf{GF}c \rightarrow \mathbf{GF}d)$	11324	34	**18**	14	8	**4**
$\bigwedge_{i=1}^{3}(\mathbf{GF}a_i \rightarrow \mathbf{GF}b_i)$	1 304 707	1 560	**462**	40	10	**8**
$\mathbf{GF}(\mathbf{F}a \vee \mathbf{GF}b \vee \mathbf{FG}(a \vee b))$	14	5	**4**	25 (6)	4	**3**
$\mathbf{FG}(\mathbf{F}a \vee \mathbf{GF}b \vee \mathbf{FG}(a \vee b))$	145	5	**4**	24 (6)	9	**3**
$\mathbf{FG}(\mathbf{F}a \vee \mathbf{GF}b \vee \mathbf{FG}(a \vee b) \vee \mathbf{FG}b)$	181	5	**4**	24 (6)	9	**3**

References

1. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. Theor. Comput. Sci. 363(2), 224–233 (2006)
2. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
3. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
4. 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), `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/`
5. Klein, J.: ltl2dstar - LTL to deterministic Streett and Rabin automata, `http://www.ltl2dstar.de/`
6. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theor. Comput. Sci. 363(2), 182–195 (2006)
7. Křetínský, J., Esparza, J.: Deterministic Automata for the (F,G)-Fragment of LTL. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 7–22. Springer, Heidelberg (2012)

8. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
9. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
10. Safra, S.: On the complexity of ω-automata. In: FOCS, pp. 319–327 (1988)
11. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)

The Unary Fragments of Metric Interval Temporal Logic: Bounded versus Lower Bound Constraints

Paritosh K. Pandya and Simoni S. Shah

Tata Institute of Fundamental Research, Colaba, Mumbai 400005, India

Abstract. We study two unary fragments of the well-known metric interval temporal logic *MITL*$[\mathsf{U}_I, \mathsf{S}_I]$ that was originally proposed by Alur and Henzinger, and we pin down their expressiveness as well as satisfaction complexities. We show that *MITL*$[\mathsf{F}_\infty, \mathsf{P}_\infty]$ which has unary modalities with only lower-bound constraints is (surprisingly) expressively complete for Partially Ordered 2-Way Deterministic Timed Automata (*po2DTA*) and the reduction from logic to automaton gives us its NP-complete satisfiability. We also show that the fragment *MITL*$[\mathsf{F}_b, \mathsf{P}_b]$ having unary modalities with only bounded intervals has *NEXPTIME*-complete satisfiability. But strangely, *MITL*$[\mathsf{F}_b, \mathsf{P}_b]$ is strictly less expressive than *MITL*$[\mathsf{F}_\infty, \mathsf{P}_\infty]$. We provide a comprehensive picture of the decidability and expressiveness of various unary fragments of *MITL*.

1 Introduction

Metric Temporal Logic *MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ is a well established logic for specifying quantitative properties of timed behaviors in real-time. In this logic, the temporal modalities U_I and S_I are time constrained by a time interval I. A formula $\phi \mathsf{U}_I \psi$ holds at a position i provided there exists a strictly later position j where ψ holds and ϕ must hold for all in between positions. Moreover the "time distance" between j and i must be in the interval I. Interval $I = \langle l, u\rangle$ has integer valued endpoints and it can be open, closed, half open, or singular (i.e. $[c,c]$). It can even be unbounded, i.e. of the form $\langle l, \infty)$. Unary modalities $\mathsf{F}_I\phi$ and $\mathsf{P}_I\phi$ can be defined as $(true)\mathsf{U}_I\phi$ and $(true)\mathsf{S}_I\phi$, respectively. Unfortunately, satisfiability of *MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ formulae and their model checking (against timed automata) are both undecidable in general [AH93, Hen91].

In their seminal paper [AFH96], the authors proposed the sub logic *MITL*$[\mathsf{U}_I, \mathsf{S}_I]$ having only non-punctual (or non-singular) intervals. Alur and Henzinger [AFH96, AH92] showed that the logic *MITL*$[\mathsf{U}_I, \mathsf{S}_I]$ has *EXPSPACE*-complete satisfiability[1]. In another significant paper [BMOW08], Bouyer *et al* showed that sublogic of *MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ with only bounded intervals, denoted *MTL*$[\mathsf{U}_b, \mathsf{S}_b]$, also has *EXPSPACE*-complete satisfiability. These results are practically significant since many real time properties can be stated with bounded or non-punctual interval constraints.

In quest for more efficiently decidable timed logics, Alur and Henzinger considered the fragment *MITL*$[\mathsf{U}_{0,\infty}, \mathsf{S}_{0,\infty}]$ consisting only of "one-sided" intervals, and showed that it has *PSPACE*-complete satisfiability. Here, allowed intervals are of the form $[0, u\rangle$ or

[1] This assumes that the time constants occurring in the formula are written in binary. We follow the same convention throughout this paper.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 77–91, 2012.

$\langle l, \infty)$ thereby enforcing either an upper bound or a lower bound time constraint in each modality.

Several real-time properties of systems may be specified by using the unary *future* and *past* modalities alone. In the untimed case of finite words, the unary fragment of logic $LTL[\mathsf{U},\mathsf{S}]$ has a special position: the unary temporal logic $LTL[\mathsf{F},\mathsf{P}]$ has NP-complete satisfiability [EVW02] and it expresses exactly the unambiguous star-free languages which are characterized by Partially ordered 2-Way Deterministic Finite Automata (*po2dfa*) [STV01].

Inspired by the above, in this paper, we investigate several "unary" fragments of $MITL[\mathsf{U}_I,\mathsf{S}_I]$ and we pin down their exact decision complexities as well as expressive powers. *In this paper, we confine ourselves to point-wise MITL with finite strictly monotonic time*, i.e. the models are finite timed words where no two letters have the same time stamp.

As our main results, we identify two fragments of unary logic $MITL[\mathsf{F}_I,\mathsf{P}_I]$ for which a remarkable drop in complexity of checking satisfiability is observed, and we study their automata as well as expressive powers. These fragments are as follows.

- Logic $MITL[\mathsf{F}_\infty,\mathsf{P}_\infty]$ embodying only unary "lower-bound" interval constraints of the form $F_{\langle l,\infty)}$ and $P_{\langle l,\infty)}$. We show that satisfiability of this logic is *NP*-complete.
- Logic $MITL[\mathsf{F}_b,\mathsf{P}_b]$ having only unary modalities $F_{\langle l,u\rangle}$ and $P_{\langle l,u\rangle}$ with bounded and non-singular interval constraints where ($u \neq \infty$). We show that satisfiability of this logic is *NEXPTIME*-complete.

In both cases, an automata theoretic decision procedure is given as a language preserving reduction from the logic to Partially Ordered 2-Way Deterministic Timed Automata (*po2DTA*). These automata are a subclass of the 2Way Deterministic Timed Automata *2DTA* of Alur and Henzinger [AH92] and they incorporate the notion of partial-ordering of states. They define a subclass of timed regular languages called unambiguous timed regular languages (*TUL*) (see [PS10]). *po2DTA* have several attractive features: they are boolean closed (with linear blowup only) and their non-emptiness checking is *NP*-complete. The properties of *po2DTA* together with our reductions give the requisite decision procedures for satisfiability checking of logics $MITL[\mathsf{F}_\infty,\mathsf{P}_\infty]$ and $MITL[\mathsf{F}_b,\mathsf{P}_b]$.

The reduction from $MITL[\mathsf{F}_\infty,\mathsf{P}_\infty]$ to *po2DTA* uses a nice optimization which becomes possible in this sublogic: truth of a formula at any point can be determined as a simple condition between times of first and last occurrences of its modal subformulas and current time. A much more sophisticated but related optimization is required for the logic $MITL[\mathsf{F}_b,\mathsf{P}_b]$ with both upper and lower bound constraints: truth of a formula at any point in a unit interval can be related to the times of first and last occurrences of its immediate modal subformulas in some "related" unit intervals. The result is an inductive bottom up evaluation of the first and last occurrences of subformulas which is carried out in successive passes of the two way deterministic timed automaton.

For both the logics, we show that our decision procedures are optimal. We also verify that the logic $MITL[\mathsf{F}_I]$ consisting only of the unary future fragment of $MITL[\mathsf{U}_I,\mathsf{S}_I]$ already exhibits *EXPSPACE*-complete satisfiability. Moreover, the unary future fragment $MITL[\mathsf{F}_0]$ with only upper bound constraints has *PSPACE*-complete satisfiability, whereas $MITL[\mathsf{F}_\infty,\mathsf{P}_\infty]$ with only lower bound constraints has *NP*-complete satisfiability. A comprehensive picture of decision complexities of fragments of $MITL[\mathsf{F}_I, \mathsf{P}_I]$ is obtained and summarized in Figure 1.

We also study the expressive powers of logics $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ and $MITL[\mathsf{F}_b, \mathsf{P}_b]$. We establish that $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ is expressively complete for *po2DTA*, and hence it can define all unambiguous timed regular languages (*TUL*). This is quite surprising as *po2DTA* include guards with simultaneous upper and lower bound constraints as well as punctual constraints, albeit only occurring deterministically. Expressing these in $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$, which has only lower bound constraints, is tricky. We remark that $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty] \equiv po2DTA$ is a rare instance of a precise logic automaton connection within the $MTL[\mathsf{U}_I, \mathsf{S}_I]$ family of timed logics.

We also establish that $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ is strictly more expressive than the bounded unary logic $MITL[\mathsf{F}_b, \mathsf{P}_b]$. Combining these results with decision complexities, we conclude that $MITL[\mathsf{F}_b, \mathsf{P}_b]$, although less expressive, is exponentially more succinct as compared to the logic $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$. Completing the picture, we show that, for expressiveness, $MITL[\mathsf{F}_b, \mathsf{P}_b] \subsetneq MITL[\mathsf{F}_\infty, \mathsf{P}_\infty] \subsetneq MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}] \subsetneq MITL[\mathsf{F}_I, \mathsf{P}_I]$. For each logic, we give a sample property that cannot be expressed in the contained logic (see Figure 2). The inexpressibility of these properties in lower logics are proved using an EF theorem for *MTL* formulated earlier [PS11].

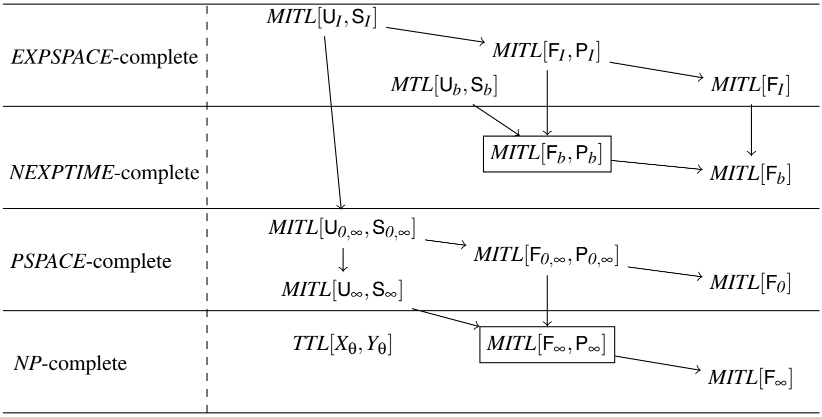
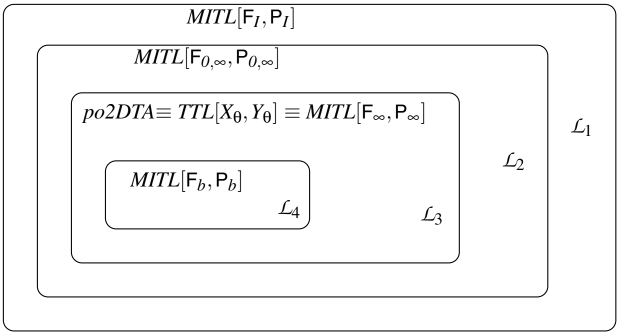

EXPSPACE-complete
NEXPTIME-complete
PSPACE-complete
NP-complete
$MITL[\mathsf{U}_I, \mathsf{S}_I]$
$MITL[\mathsf{F}_I, \mathsf{P}_I]$
$MTL[\mathsf{U}_b, \mathsf{S}_b]$
$MITL[\mathsf{F}_I]$
$MITL[\mathsf{F}_b, \mathsf{P}_b]$
$MITL[\mathsf{F}_b]$
$MITL[\mathsf{U}_{0,\infty}, \mathsf{S}_{0,\infty}]$
$MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$
$MITL[\mathsf{U}_\infty, \mathsf{S}_\infty]$
$MITL[\mathsf{F}_0]$
$TTL[X_\theta, Y_\theta]$
$MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$
$MITL[\mathsf{F}_\infty]$

Fig. 1. Unary MITL: fragments with satisfiability complexities. Arrows indicate syntactic inclusion. The boxed logics are the two main fragments studied in this paper.

2 Preliminaries

Let $\mathbb{R}$ and $\mathbb{N}$ be the set of real and natural numbers and let $\mathbb{R}_0$ be the set of non-negative reals. An *interval* is a convex subset of $\mathbb{R}$, bounded by non-negative integer constants or ∞. The left and right ends of an interval may be open ("(" or ")") or closed ("[" or "]"). We denote by $\langle x, y \rangle$ a generic interval whose ends may be open or closed. An interval is said to be *bounded* if it doesn't extend to infinity. It is said to be *singular* if it is of the form $[c, c]$ for some constant c, and non-singular otherwise.



$\mathcal{L}_1 = \mathsf{F}_{(0,\infty)}[a \wedge \mathsf{F}_{(1,2)}c]$

$\mathcal{L}_2 = \mathsf{F}_{(0,\infty)}[a \wedge \mathsf{F}_{[0,2]}c]$

$\mathcal{L}_3 = \mathsf{F}_{(0,\infty)}[a \wedge \mathsf{F}_{(2,\infty)}c]$

$\mathcal{L}_4 = \mathsf{F}_{(0,1)}[a \wedge \mathsf{F}_{(1,2)}c]$

Fig. 2. Expressiveness of Unary *MITL* fragments

Let Σ be a finite alphabet. A finite timed word (*TW*) over Σ is a finite sequence $\rho = (\sigma_1, \tau_1), \cdots (\sigma_n, \tau_n)$ of event-timestamp pairs, such that the sequence of real timestamps is strictly increasing: $\forall i < n \,.\, \tau_i < \tau_{i+1}$. This gives strictly monotonic timed words. The length of a *TW* ρ, is denoted by $\#\rho$, and $dom(\rho) = \{1, ...\#\rho\}$. For convenience, we assume that $\tau_1 = 0$. Let $T\Sigma^*$ be the set of strictly monotonic timed words over the alphabet Σ.

2.1 Unary *MITL* and Its Fragments

$MITL[\mathsf{U}_I, \mathsf{S}_I]$ is the non-punctual fragment of Metric Temporal logic, in which timing constraints are expressed as non-singular intervals with integral bounds. Let $a \in \Sigma$ and I be a non-singular interval with non-negative integer bounds or ∞. The syntax of $MITL[\mathsf{U}_I, \mathsf{S}_I]$ formulas is given by

$$\phi \;:=\; a \;\mid\; \phi \mathsf{U}_I \phi \;\mid\; \phi \mathsf{S}_I \phi \;\mid\; \neg\phi \;\mid\; \phi \vee \phi$$

We shall consider the models to be finite timed words with strictly monotonic time (point-wise semantics). Consider a timed word $\rho = (\sigma_1, \tau_1), ...(\sigma_{\#\rho}, \tau_{\#\rho})$. The boolean operations have their usual meaning. $\rho, i \models a$ iff $\sigma_i = a$. The semantics of modal formulas is defined inductively by the following rules:

$\rho, i \models \phi_1 \mathsf{U}_I \phi_2$ iff $\exists j > i \,.\, \rho, j \models \phi_2$ and $\tau_j - \tau_i \in I$
and $\forall i < k < j$, $\rho, k \models \phi_1$

$\rho, i \models \phi_1 \mathsf{S}_I \phi_2$ iff $\exists j < i \,.\, \rho, j \models \phi_2$ and $\tau_i - \tau_j \in I$
and $\forall j < k < i$, $\rho, k \models \phi_1$

The unary modalities may be derived as (future) $\mathsf{F}\phi_1 := \top \mathsf{U} \phi_1$ and (past) $\mathsf{P}\phi_1 := \top \mathsf{S} \phi_1$.

Unary sublogics The logic $MITL[\mathsf{F}_I, \mathsf{P}_I]$ is the unary sublogic of $MITL[\mathsf{U}_I, \mathsf{S}_I]$, which is confined to the unary *future* and *past* modalities alone. Some fragments of $MITL[\mathsf{F}_I, \mathsf{P}_I]$ that we shall consider in this paper are as follows. See Figure 2 for examples.

- $MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$allows only interval constraints of the form $[0, u\rangle$ or $\langle l, \infty)$. Thus, each modality enforces either an upper bound or a lower bound constraint.
- $MITL[\mathsf{F}_b, \mathsf{P}_b]$ is $MITL[\mathsf{F}_I, \mathsf{P}_I]$ with the added restriction that all interval constraints are bounded intervals of the form $\langle l, u\rangle$ with $u \neq \infty$.

- *MITL*$[\mathsf{F}_\infty, \mathsf{P}_\infty]$ is the fragment of *MITL*$[\mathsf{F}_I, \mathsf{P}_I]$ where all interval constraints are "lower bound" constraints of the form $\langle l, \infty)$.
- *MITL*$[\mathsf{F}_0, \mathsf{P}_0]$ is the fragment in which all interval constraints (whether bounded or unbounded) are "upper bound" constraints of the form $[0, u\rangle$.
- *MITL*$[\mathsf{F}_I]$, *MITL*$[\mathsf{F}_{0,\infty}]$, *MITL*$[\mathsf{F}_b]$, *MITL*$[\mathsf{F}_\infty]$ and *MITL*$[\mathsf{F}_0]$ are the corresponding *future*-only fragments.

Size of MITL$[F_I, P_I]$ *formulas* Consider any *MITL*$[\mathsf{F}_I, \mathsf{P}_I]$ formula ϕ, represented as a DAG. Let n be the number of modal operators in the DAG of ϕ. Let k be the product of all constants that occur in ϕ. Then, the modal-DAG size l of ϕ whose constants are presented in some logarithmic encoding (e.g., binary) is within constant factors of $(n + logk)$.

Definition 1. *(Normal Form for MITL*$[F_I, P_I]$*) Let* $\mathscr{B}(\{\psi_i\})$ *denote a* boolean *combination of formulas from finite set* $\{\psi_i\}$. *Then a normal form formula* ϕ *is given by*

$$\phi \ := \ \bigvee_{a \in \Sigma} (a \wedge \mathscr{B}(\{\psi_i\}))$$

where each ψ_i *is a modal formula of the form:*

$$\psi \ := \ F_I(\phi) \ | \ P_I(\phi)$$

where each ϕ *is also in normal form.*

A subformula ϕ in normal form is said to be an F-type *modal argument (or modarg in brief)* if it occurs within an F-modality (as $\mathsf{F}_I(\phi)$). It is a P-type modarg if it occurs as $\mathsf{P}_I(\phi)$. Each ψ_i is said to be a *modal sub formula*.

Proposition 1. *Every MITL*$[F_I, P_I]$ *formula* ζ *may be expressed as an equivalent normal form formula* ϕ *of modal-DAG size linear in the modal-DAG size of* ζ.

Proof. Given ζ, consider the equivalent formula $\zeta \wedge \bigvee_{a \in \Sigma} a$. Transform this formula in disjunctive normal form treating modal subformulas as atomic. Now apply reductions such as $a \wedge b \wedge \mathscr{B}(\psi_i) \ \equiv \ \bot$ (if $a \neq b$) and $a \wedge \mathscr{B}(\psi_i)$ otherwise. The resulting formula is equivalent to ζ. Note that DNF representation does not increase the modal-DAG size of the formula. Apply the same reduction to modargs recursively. □

2.2 *po2DTA*

Alur and Henzinger [AH92] introduced 2-way deterministic timed automata (*2DTA*) over timed words. These timed automata have a head which may move in either direction and are equipped with a finite set of clocks (or registers). In each transition a subset of these clocks can be reset to the value of the time stamp at which the transition is taken (current timestamp). The clocks retain their value till reset (thus they are more like registers).

In [PS10], we defined a special class of *2DTA* called Partially-ordered 2-way Deterministic Timed Automata (*po2DTA*). The only loops allowed in the transition graph of these automata are self-loops. This condition naturally defines a partial order on the set of states (hence the name). Another restriction is that clock resets may occur only on progress edges. *po2DTA* are formally defined below.

Let C be a finite set of clocks. A *guard* g is a timing constraint on the clock values and has the form:

$$g \;:=\; g_1 \wedge g_2 \;\mid\; x - T \approx c \;\mid\; T - x \approx c \text{ where } \approx \in \{<, \leq, >, \geq, =\} \text{ and } c \in \mathbb{N}.[2]$$

Here, T denotes the current time value. Let G_C be the set of all guards over C. A clock valuation is a function which assigns to each clock a non-negative real number. Let $\nu, \tau \models g$ denote that a valuation ν satisfies the guard g when T is assigned a real value τ. If ν is a clock valuation and $x \in C$, let $\nu' = \nu \otimes (x \to \tau)$ denote a valuation such that $\forall y \in C \,.\, y \neq x \Rightarrow \nu'(y) = \nu(y)$ and $\nu'(x) = \tau$. Two guards g_1 and g_2 are said to be disjoint if for all valuations ν and all reals r, we have $\nu, r \models \neg(g_1 \wedge g_2)$. A special valuation ν_{init} maps all clocks to 0.

Two-way automata "detect" the ends of a word, by appending the word with special *end-markers* on either side. Hence, if $\rho = (\sigma_1, \tau_1)...(\sigma_n, \tau_n)$ then the run of a *po2DTA* is defined on a timed word $\rho' = (\rhd, 0)(\sigma_1, \tau_1)...(\sigma_n, \tau_n), (\lhd, \tau_n)$.

Definition 2 (Syntax of *po2DTA*). *Fix an alphabet Σ and let $\Sigma' = \Sigma \cup \{\rhd, \lhd\}$. Let C be a finite set of clocks. A po2DTA over alphabet Σ is a tuple $M = (Q, \leq, \delta, s, t, r, C)$ where $(Q, \leq)$ is a partially ordered and finite set of states such that r,t are the only minimal elements and s is the only maximal element. Here, s is the initial state, t the accept state and r the reject state. The set $Q \setminus \{t, r\}$ is partitioned into Q_l and Q_r (making the head move resp. left and the right on transitions leading into them) with $s \in Q_r$. The progress transition function is a partial function $\delta : ((Q_l \cup Q_r) \times \Sigma' \times G_C) \to Q \times 2^C)$ which specifies the progress transitions of the automaton, such that if $\delta(q, a, g) = (q', R)$ then $q' < q$ and $R \in 2^C$ is the subset of clocks that is reset to the current time stamp. Every state q in $Q \setminus \{t, r\}$ has a default "else" self-loop transition which is taken in all such configurations for which no progress transition is enabled. Hence, the automaton continues to loop in a given state q and scan the timed word in a single direction (depending on whether $q \in Q_l$ or Q_r), until one of the progress transitions is taken. Note that there are no transitions from the terminal states (r and t).*

The transition function satisfies the following conditions. Let $\delta(q, a, g) = (q', X)$.

- *If $a = \lhd$ then $q' \in Q_l$ and if $a = \rhd$ then $q \in Q_r$. This prevents the head from falling off the end-markers.*
- *(Determinism) For all $q \in Q$ and $a \in \Sigma'$, if there exist distinct transitions $\delta(q, a, g_1) = (q_1, X_1)$ and $\delta(q, a, g_2) = (q_2, X_2)$, then g_1 and g_2 are disjoint (as defined below).*

Definition 3 (Run). *Let $\rho = (\sigma_1, \tau_1), (\sigma_2, \tau_2)...(\sigma_m, \tau_m)$ be a given timed word. The configuration of a po2DTA at any instant is given by (q, ν, l) where q is the current state, the current value of the clocks is given by the valuation function ν and the current head position is $l \in dom(\rho')$. In this configuration, the head reads the letter σ_l and the time stamp τ_l.*

The run of a po2DTA on the timed word ρ with and starting head position $k \in dom(\rho')$ and starting valuation ν is the (unique) sequence of configurations $(q_1, \nu_1, l_1) \cdots (q_n, \nu_n, l_n)$ such that

[2] Note that the guards $x - T \approx c$ and $T - x \approx c$ implicitly include the conditions $x - T \geq 0$ and $T - x \geq 0$ respectively.

- *Initialization:* $q_1 = s$, $l_1 = k$ *and* $\nu_1 = \nu$. *The automaton always starts in the initial state s.*
- *If the automaton is in a configuration* (q_i, ν_i, l_i) *and there exists a (unique) transition* $\delta(q_i, a, g) = (p, X)$ *such that* $\sigma_{l_i} = a$ *and* $\nu_i, \tau_{l_i} \models g$. *Then,*
  - $q_{i+1} = p$
  - $\nu_{i+1}(x) = \tau_{l_i}$ *for all clocks* $x \in X$, *and* $\nu_{i+1}(x) = \nu_i(x)$ *otherwise.*
  - $l_{i+1} = l_i - 1$ *if* $p \in Q_l$, $l_{i+1} = l_i + 1$ *if* $p \in Q_r$ *and* $l_{i+1} = l_i$ *if* $p \in \{t, r\}$
- *If the automaton is in a configuration* (q_i, ν_i, l_i) *and there does not exist a transition* $\delta(q_i, a, g)$ *such that* $\sigma_{l_i} = a$ *and* $\nu_i, \tau_{l_i} \models g$. *Then,*
  - $q_{i+1} = q_i$
  - $\nu_{i+1}(x) = \nu_i(x)$ *for all clocks* $x \in C$ *and*
  - $l_{i+1} = l_i - 1$ *if* $p \in Q_l$ *and* $l_{i+1} = l_i + 1$ *if* $p \in Q_r$
- *Termination:* $q_n \in \{t, r\}$. *The run is accepting if* $q_n = t$ *and rejecting if* $q_n = r$.

Let $\mathcal{F}_\mathcal{A}$ *be a function such that* $\mathcal{F}_\mathcal{A}(\rho, \nu, i)$ *gives the final configuration* (q_n, ν_n, l_n) *of the unique run of* $\mathcal{A}$ *on* ρ *starting with the configuration* (s, ν, i). *The language accepted by an automaton* $\mathcal{A}$ *is given by* $\mathcal{L}(\mathcal{A}) = \{\rho \mid \mathcal{F}_\mathcal{A}(\rho, \nu_{init}, 1) = (t, \nu', i)$*,for some* $i, \nu'\}$.

Example 1. Figure 3 shows an example po2DTA. This automaton accepts timed words with the following property: There is b in the interval $[1,2]$ and c occurs before it. Moreover, if j is the position of the first b in the interval $[1,2]$ then there is a c exactly at the timestamp $\tau_j - 1$.

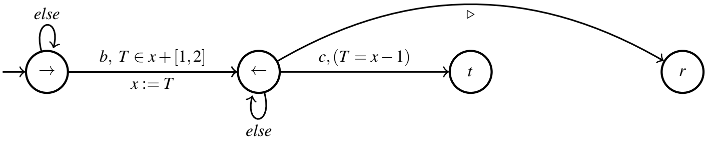


Fig. 3. Example of po2DTA

Properties of po2DTA

Since the automaton is deterministic and two way, complementation and other boolean operations on these automata may be achieved with only a linear blow up in the size of the automaton. (Intersection and union are achieved by sequentially running the two component automata one after the other with a backward scan to the start position in-between.) Given *po2DTA* with n states and k_{max} being the largest constant in the guards, it can be shown that the size (length) of its *smallest word (model)* is polynomial in n, the largest timestamp in the model is linear in $(n-1)(k_{max}+1)$ and each timestamp in the model has a fractional part which is a multiple of $1/n$. Hence, assuming binary encoding of timestamps, the size of this small model is polynomial in the size of the automaton. Hence, non-emptiness of a *po2DTA* may be checked with *NP*-complete complexity. (See [PS10]).

3 From *MITL*$[\mathsf{F}_I,\mathsf{P}_I]$-Fragments to *po2DTA*

In their seminal paper [AH92], Alur and Henzinger reduced the checking of satisfiability of *MITL*$[\mathsf{U}_I,\mathsf{S}_I]$ to non-emptiness of Reversal-bounded 2-way deterministic timed automata (*RB2DTA*). They asserted *EXPSPACE* decision complexity for *MITL*$[\mathsf{U}_I,\mathsf{S}_I]$ by this method. In this section, we explore reductions from some fragments of Unary *MITL* to *po2DTA*. A powerful optimization becomes possible when dealing with the unary sublogics *MITL*$[\mathsf{F}_b,\mathsf{P}_b]$ and *MITL*$[\mathsf{F}_\infty,\mathsf{P}_\infty]$. The truth of a modal formula $M_I\phi$ for a time point τ_i in an interval I can be reduced to a simple condition involving time differences between τ_i and the times of first and last occurrence of ϕ within some related intervals. We introduce some notation below.

Marking timed words with first and last ϕ*-positions.* Consider a formula ϕ in normal form, a timed word $\rho \in T\Sigma^*$ and an interval I. Let $Idx_I^\phi(\rho) = \{i \in dom(\rho) \mid \rho, i \models \phi \wedge \tau_i \in I\}$. Given set S of positions in ρ let $min(S)$ and $max(S)$ denote the smallest and largest positions in S, with the convention that $min(\emptyset) = \#\rho$ and $max(\emptyset) = 1$. Let $\mathcal{F}_I^\phi(\rho) = \tau_{min(Idx_I^\phi(\rho))}$ and $\mathcal{L}_I^\phi(\rho) = \tau_{max(Idx_I^\phi(\rho))}$ denote the times of first and last occurrence of ϕ within interval I in word ρ. If the subscript I is omitted, it is assumed to be the default interval $[0,\infty)$.

3.1 From *MITL*$[\mathsf{F}_\infty,\mathsf{P}_\infty]$ to *po2DTA*

Fix an *MITL*$[\mathsf{F}_\infty,\mathsf{P}_\infty]$ formula Φ in normal form. We shall construct a language- equivalent *po2DTA* $\mathcal{A}_\Phi$ by an inductive bottom-up construction. But first we assert an important property on which our automaton construction is based. Its proof is straightforward, and given in the full version of the paper.

Lemma 1. *Given a timed word* ρ *and* $i \in dom(\rho)$*,*

1. $\rho, i \models F_{[l,\infty)}\phi$ *iff* $\tau_i \leq (\mathcal{L}^\phi(\rho) - l) \wedge \tau_i < \mathcal{L}^\phi(\rho)$
2. $\rho, i \models F_{(l,\infty)}\phi$ *iff* $\tau_i < (\mathcal{L}^\phi(\rho) - l)$
3. $\rho, i \models P_{[l,\infty)}\phi$ *iff* $\tau_i \geq (\mathcal{F}^\phi(\rho) + l) \wedge \tau_i > \mathcal{F}^\phi(\rho)$
4. $\rho, i \models P_{(l,\infty)}\phi$ *iff* $\tau_i > (\mathcal{F}^\phi(\rho) + l)$

The above lemma shows that truth of $\mathsf{F}_{\langle l,\infty)}\phi$ ($\mathsf{P}_{\langle l,\infty)}\phi$) at a position can be determined by knowing the value of $\mathcal{L}^\phi(\rho)$ (respectively, $\mathcal{F}^\phi(\rho)$). Hence, for each F-type modarg ϕ of Φ, we introduce a clock y^ϕ to freeze the value $\mathcal{L}^\phi(\rho)$ and P-type modarg ϕ of Φ, we introduce a clock x^ϕ to freeze the value $\mathcal{F}^\phi(\rho)$.

Now we give the inductive step of automaton construction. Consider an F-type modarg ϕ. The automaton $\mathcal{A}(\phi)$ is as shown in Figure 4. If $\phi = \vee_{a\in\Sigma}\ (a \wedge \mathcal{B}_a(\{\psi_i\}))$, then for the clock y^ϕ, and $a \in \Sigma$, we derive the guard $\mathcal{G}(y^\phi, a)$ which is the guard on the transition labelled by a in $\mathcal{A}(\phi)$, and which resets y^ϕ. This is given by $\mathcal{G}(c,a) = \mathcal{B}_a(cond(\psi_i))$. To define $cond(\psi_i)$, let variable T denote the time stamp of current position. Then, the condition for checking truth of a modal subformula ψ is a direct encoding of the conditions in lemma 1 and is given in the table in figure 4. It is now straightforward to see that $\mathcal{A}(\phi)$ clocks exactly the last position in the word, where ϕ

ψ	$cond(\psi)$
$\mathsf{F}_{[l,\infty)}\phi$	$T \leq (y^{\phi} - l) \wedge T < y^{\phi}$
$\mathsf{F}_{(l,\infty)}\phi$	$T < (y^{\phi} - l)$
$\mathsf{P}_{[l,\infty)}\phi$	$T \geq (x^{\phi} + l) \wedge T > x^{\phi}$
$\mathsf{P}_{(l,\infty)}\phi$	$T > (x^{\phi} + l)$

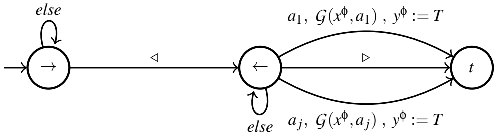

else
$\rightarrow$
$\triangleleft$
$\leftarrow$
else
$a_1,\ \mathcal{G}(x^{\phi},a_1)\ ,\ y^{\phi} := T$
$\triangleright$
$a_j,\ \mathcal{G}(x^{\phi},a_j)\ ,\ y^{\phi} := T$
t

Fig. 4. Table for $cond(\psi)$ and automaton $\mathcal{A}(\phi)$ for an F-type ϕ

holds. A symmetrical construction can be given for P-type modarg ϕ. The following lemma states its key property which is obvious from the construction. Hence we omit its proof.

Lemma 2. *Given a modarg ϕ and any timed word ρ, let ν_0 be a valuation where $\nu_0(x^{\delta}) = \mathcal{F}^{\delta}(\rho)$ and $\nu_0(y^{\delta}) = \mathcal{L}^{\delta}(\rho)$ for each modarg subformula δ of ϕ, and $\nu_0(x^{\phi}) = \tau_{\#\rho}$ and $\nu_0(y^{\phi}) = 0$. If ν is the clock valuation at the end of the run of $\mathcal{A}(\phi)$ starting with ν_0, then $\nu(x^{\delta}) = \nu_0(x^{\delta})$, $\nu(y^{\delta}) = \nu_0(y^{\delta})$ for each δ, and additionally,*

- *if ϕ is F modarg then $\nu(y^{\phi}) = \mathcal{L}^{\phi}(\rho)$.*
- *if ϕ is P modarg then $\nu(x^{\phi}) = \mathcal{F}^{\phi}(\rho)$.*

Theorem 1. *For any MITL$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ formula Φ, there is a language-equivalent po2DTA $\mathcal{A}(\Phi)$ whose size is linear in the modal-DAG size of the formula. Hence, satisfiability of MITL$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ is in NP.* □

Proof. Firstly, assume that Φ is in the normal form as described in Definition 1. Note that reduction to normal form results in a linear blow-up in the modal-DAG size of the formula (Proposition 1). The construction of the complete automaton $\mathcal{A}(\Phi)$ is as follows. In an initial pass, all the x^{ϕ} clocks are set to $\tau_{\#\rho}$. Then, the component automata $\mathcal{A}(\phi)$ for clocking modargs (ϕ) are composed in sequence with innermost modargs being evaluated first. This bottom-up construction, gives us the initial-valuation conditions at every level of induction, as required in Lemma 2. Finally, the validity of Φ at the first position may be checked.

This construction, gives a language-equivalent *po2DTA* whose number of states is linear in the number of nodes in the DAG of Φ and the largest constant in the guards of $\mathcal{A}(\Phi)$ is equal to the largest constant in the interval constraints of Φ. Hence we can conclude that satisfiability of *MITL*$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ formulas is decidable in *NP*-time. □

3.2 From *po2DTA* to *MITL*$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$

Theorem 2. *Given a po2DTA $\mathcal{A}$, we may derive an equivalent MITL$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ formula $\phi_{\mathcal{A}}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi_{\mathcal{A}})$*

Here, we shall illustrate the reduction of *po2DTA* to *MITL*$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ by giving a language equivalent *MITL*$[\mathsf{F}_{\infty}, \mathsf{P}_{\infty}]$ formula for the *po2DTA* in Example 1. This *po2DTA* first scans in the forward direction and clocks the first b in the time interval $[1,2]$ (this is a bounded constraint), and then checks if there is a c exactly 1 time unit to its past by a backward scan (this is a punctual constraint). The automaton contains guards with both upper and lower bound constraints as well as a punctual constraints. It is critical for our reduction that the progress transitions are satisfied at unique positions in the word. The detailed translation is given in the full version of the paper.

Consider the following $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ formulas. Define $Atfirst := \neg\mathsf{P}\top$ as the formula which holds only at the first position in the word.

$$\begin{aligned}
&\phi_1 := b \ \wedge \mathsf{P}_{[1,\infty)} Atfirst \ \wedge \neg\mathsf{P}_{(2,\infty)} Atfirst \\
&\phi_2 := \phi_1 \wedge \neg\mathsf{P}_{(0,\infty)}\phi_1 \\
&\Phi := \mathsf{F}_{[0,\infty)}[\phi_2 \wedge \mathsf{P}_{[1,\infty)}(c \wedge \neg\mathsf{F}_{(1,\infty)}\phi_2)\ ]
\end{aligned}$$

The formula ϕ_1 holds at any b within the time interval $[1,2]$. The formula ϕ_2 holds at the *unique* first b in $[1,2]$. The formula Φ holds at the initial position in a word iff the first b in $[1,2]$ has a c exactly 1 time unit behind it. Note that the correctness of Φ relies on the *uniqueness* of the position where ϕ_2 holds. It is now easy to verify that the $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ formula Φ exactly accepts the timed words that are accepted by the *po2DTA* in example 1.

3.3 Embedding $MITL[\mathsf{F}_b, \mathsf{P}_b]$ into *po2DTA*

We show a language-preserving conversion of an $MITL[\mathsf{F}_b, \mathsf{P}_b]$ formula to a language-equivalent *po2DTA*.

Consider an $MITL[\mathsf{F}_b, \mathsf{P}_b]$ formula Φ in the normal form. We can inductively relate the truth of a subformula $\psi = \mathsf{F}_{\langle l,l+1\rangle}\phi$ or $\mathsf{P}_{\langle l,l+1\rangle}\phi$ within a unit interval $[r, r+1)$ to the values $\mathcal{F}_I^\phi(\rho)$ and $\mathcal{L}_I^\phi(\rho)$ of its sub-formula ϕ for suitable unit-length intervals I, by the following lemma[3].

Lemma 3. *Given a timed word* ρ *and integers* r, l *and* $i \in dom(\rho)$ *with* $\tau_i \in [r, r+1)$ *we have:*

- $\rho, i \models F_{\langle_a l, l+1\rangle_b}\phi$ *iff*
  - *(1a)* $\tau_i < \mathcal{L}^\phi_{[r+l,r+l+1)}(\rho) \ \wedge \ \tau_i \in [r, (\mathcal{L}^\phi_{[r+l,r+l+1)}(\rho) - l)\rangle_a$ *OR*
  - *(1b)* $\tau_i < \mathcal{L}^\phi_{[r+l+1,r+l+2)}(\rho) \ \wedge \ \tau_i \in \langle_b(\mathcal{F}^\phi_{[r+l+1,r+l+2)}(\rho) - (l+1)), (r+1))$
- $\rho, i \models P_{\langle_a l, l+1\rangle_b}\phi$ *iff*
  - *(2a)* $\tau_i > \mathcal{F}^\phi_{[r-l-1,r-l)}(\rho) \ \wedge \ \tau_i \in \ [r, (\mathcal{L}^\phi_{[r-l-1,r-l)}(\rho) + l + 1)\rangle_b$ *OR*
  - *(2b)* $\tau_i > \mathcal{F}^\phi_{[r-l,r-l+1)}(\rho) \ \wedge \ \tau_i \in \ \langle_a(\mathcal{F}^\phi_{[r-l,r-l+1)}(\rho) + l), (r+1))$

Proof. This lemma may be verified using the figure 5. We give proof for $\mathsf{F}_{\langle_a l, l+1\rangle_b}\phi$ (let $\psi = \mathsf{F}_{\langle_a l, l+1\rangle_b}\phi$) omitting the symmetric case of $\mathsf{P}_{\langle_a l, l+1\rangle_b}\phi$. Fix a timed word ρ.

Case 1: (1a) holds. (We must show that $\rho, i \models \psi$). Since conjunct 1 holds, clearly $Idx^\phi_{[r+l,r+1+1)} \neq \emptyset$ and it has max element j such that $\tau_j = \mathcal{L}^\phi_{[r+l,r+l+1)}(\rho)$ and $\rho, j \models \phi$

[3] We shall use convention $\langle_a l, u\rangle_b$ to denote generic interval which can be open, closed or half open. Moreover, we use subscripts a, b fix the openness/closedness and give generic conditions such as $\langle_a 2, 3\rangle_b + 2 = \langle_a 4, 5\rangle_b$. This instantiates to $(2,3) + 2 = (4,5)$ and $(2,3] + 2 = (4,5]$ and $[2,3) + 2 = [4,5)$ and $[2,3] + 2 = [4,5]$. Interval $[r,r)$ is empty.

and $i < j$. Also, by second conjunct of (1a) $\tau_i \in [r, \tau_j - l\rangle_a$. Hence, by examination of Figure 5, $\tau_i \in \tau_j - \langle_a l, l+1\rangle_b$ and hence $\rho, i \models \psi$.

Case 2: (1b) holds. (We must show that $\rho, i \models \psi$). Since conjunct 1 holds, clearly $Idx^{\phi}_{[r+l+1,r+1+2)} \neq \emptyset$ and it has min element j such that $\tau_j = \mathcal{F}^{\phi}_{[r+l+1,r+l+2)}(\rho)$ and $\rho, j \models \phi$. Also, by second conjunct of (2a) $\tau_i \in \langle_b \tau_j - (l+1), (r+1))$. Hence, $i < j$ and by examination of Figure 5,$\tau_i \in \tau_j - \langle_a l, l+1\rangle_b$ and hence $\rho, i \models \psi$.

Case 3: $\rho, i \models \psi$ and first conjunct of (1b) does not hold. (We must show that (1a) holds.) Since $\tau_i \not< \mathcal{L}^{\phi}_{[r+l+1,r+l+2)}(\rho)$ we have $Idx^{\phi}_{[r+l+1,r+l+2)}(\rho) = \emptyset$.

Since $\rho, i \models \psi$, for some $\tau_i \in [r, r+1)$ there is $\tau_j > \tau_i$ s.t. $\rho, j \models \phi$ and $r + l \leq \tau_j < r + l + 1$ as well as $\tau_j \in \tau_i + \langle_a l, l+1\rangle_b$, and $\tau_j \leq \mathcal{L}^{\phi}_{[l+r,l+r+1)}(\rho)$. Hence, we have $\tau_i \in \mathcal{L}^{\phi}_{[l+r,l+r+1)}(\rho) - \langle_a l, l+1\rangle_b$ which from Figure 5 give us that $\tau_i \in [r, \mathcal{L}^{\phi}_{[l+r,l+r+1)}(\rho) - l\rangle_a$. Also, $\tau_i < \mathcal{L}^{\phi}_{[l+r,l+r+1)}(\rho)$. Hence (1a) holds.

Case 4: $\rho, i \models \psi$ and first conjunct of (1b) holds but the second conjunct of (1b) does not hold. (We must show that (1a) holds.) As $\rho, i \models \psi$, for some $\tau_i \in [r, r+1)$ there is $\tau_j > \tau_i$ s.t. $\tau_j \in \tau_i + \langle_a l, l+1\rangle_b$ and $\rho, j \models \phi$.
Since $\tau_i < \mathcal{L}^{\phi}_{[r+l+1,r+l+2)}(\rho)$ we have $Idx^{\phi}_{[r+l+1,r+l+2)}(\rho) \neq \emptyset$. However, second conjunct of (1b) does not hold. Hence, $\tau_i <_b \mathcal{F}^{\phi}_{[r+l+1,r+l+2)}(\rho) - (l+1)$. By examination of Figure 5, we conclude that $Idx^{\phi}_{[r+l,r+1+1)} \neq \emptyset$ and $\tau_j \leq \mathcal{L}^{\phi}_{[r+l,r+l+1)}(\rho)$. Hence, $\tau_j - l \leq \mathcal{L}^{\phi}_{[r+l,r+l+1)}(\rho) - l$. This gives us that $\tau_i \in [r, \tau_j - l\rangle_a$ (see Figure 5). Thus, (1a) holds. □

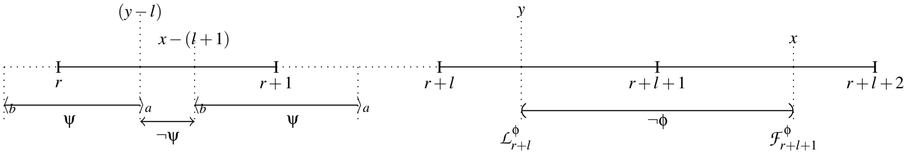


Fig. 5. Case of $\psi := \mathsf{F}_{\langle_a l, l+1\rangle_b}\phi$

From Lemma 3, we can see that in order to determine the truth of a formula of the form $\psi = \mathsf{F}_{\langle l,l+1\rangle}\phi$ at any time stamp in $[r, r+1)$, it is sufficient to clock the first and last occurrences of ϕ in the intervals $[r+l, r+l+1)$ and $[r+l+1, r+l+2)$. Similarly, in order to determine the truth of a formula of the form $\psi = \mathsf{P}_{\langle l,l+1\rangle}\phi$ at any time stamp in $[r, r+1)$, it is sufficient to clock the first and last occurrences of ϕ in the intervals $[r-l, r-l+1)$ and $[r-l-1, r-l)$.

The automaton $\mathcal{A}(\Phi)$ is constructed in an inductive, bottom-up manner as follows. For every modarg ϕ of Φ, we first inductively evaluate the set of unit intervals within which its truth must be evaluated. Each such requirement is denoted by a tuple $(\phi, [r, r+1))$. This is formalized as a closure set of a subformula wrt an interval. For an interval I, let $spl(I)$ denote a partition set of I, into unit length intervals. For example, if $I = (3,6]$ then $spl(I) = \{(3,4), [4,5), [5,6]\}$. The closure set may be built using the following rules.

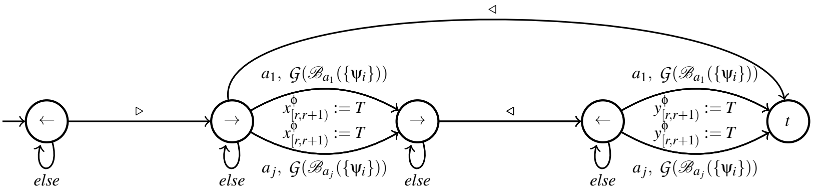

$\triangleleft$
$\triangleright$
$\triangleleft$
$\leftarrow$
$\rightarrow$
$\rightarrow$
$\leftarrow$
t
$a_1,\ \mathcal{G}(\mathcal{B}_{a_1}(\{\psi_i\}))$
$x^{\phi}_{[r,r+1)} := T$
$x^{\phi}_{[r,r+1)} := T$
$a_j,\ \mathcal{G}(\mathcal{B}_{a_j}(\{\psi_i\}))$
$a_1,\ \mathcal{G}(\mathcal{B}_{a_1}(\{\psi_i\}))$
$y^{\phi}_{[r,r+1)} := T$
$y^{\phi}_{[r,r+1)} := T$
$a_j,\ \mathcal{G}(\mathcal{B}_{a_j}(\{\psi_i\}))$
else
else
else
else

Fig. 6. Automaton $\mathcal{A}(\phi,[r,r+1))$

- $Cl(\phi,[r,r+1)) = \{(\phi,[r,r+1))\} \cup_j Cl(\psi_j,[r,r+1))$,
  where $\{\psi_j\}$ is the set of immediate modal subformulas of ϕ.
- $Cl(\mathsf{F}_I\phi,[r,r+1)) = \cup_{\langle l,l+1\rangle \in spl(I)} Cl(\mathsf{F}_{\langle l,l+1\rangle},[r,r+1))$
- $Cl(\mathsf{P}_I\phi,[r,r+1)) = \cup_{\langle l,l+1\rangle \in spl(I)} Cl(\mathsf{P}_{\langle l,l+1\rangle},[r,r+1))$
- $Cl(\mathsf{F}_{\langle l,l+1\rangle}\phi,[r,r+1)) = Cl(\phi,[r+l,r+l+1)) \cup Cl(\phi,[r+l+1,r+l+2))$
- $Cl(\mathsf{P}_{\langle l,l+1\rangle}\phi,[r,r+1)) = Cl(\phi,[r-l-1,r-l)) \cup Cl(\phi,[r-l,r-l+1))$

Define strict closure $SCl(\phi,[r,r+1)) = Cl(\phi,[r,r+1)) \setminus \{(\phi,[r,r+1))\}$. The following lemma states the key property of $\mathcal{A}(\phi,[r,r+1))$.

Table 1.

ψ	$cond(\psi,[r,r+1))$
$\mathsf{F}_{\langle_a l,l+1\rangle_b}\delta$	$T < y^{\delta}_{[r+l,r+l+1)} \ \wedge\ T \in [r,(y^{\delta}_{[r+l,r+l+1)} - l)\rangle_b\ \vee$ $T < y^{\delta}_{[r+l+1,r+l+2)} \ \wedge\ T \in \langle_a(x^{\delta}_{[r+l+1,r+l+2)} - (l+1)),(r+1))$
$\mathsf{P}_{\langle_a l,l+1\rangle_b}\delta$	$T > x^{\delta}_{[r-l-1,r-l)} \ \wedge\ T \in\ [r,(y^{\delta}_{[r-l-1,r-l)} + l + 1)\rangle_a\ \vee$ $T > x^{\delta}_{[r-l,r-l+1)} \ \wedge\ T \in\ \langle_b(x^{\delta}_{[r-l,r-l+1)} + l),(r+1))$
$\mathsf{F}_I(\delta)$	$\bigvee_{\langle l,l+1\rangle \in spl(I)}(cond(\mathsf{F}_{\langle l,l+1\rangle}\delta,[r,r+1)))$
$\mathsf{P}_I(\delta)$	$\bigvee_{\langle l,l+1\rangle \in spl(I)}(cond(\mathsf{P}_{\langle l,l+1\rangle}\delta,[r,r+1)))$

Lemma 4. *For any modarg ϕ in normal form, timed word ρ and integer r, we construct an automaton $\mathcal{A}(\phi,[r,r+1))$ such that, if the initial clock valuation is ν_0, with $\nu_0(x^{\delta}_I) = \mathcal{F}^{\delta}_I(\rho)$ and $\nu_0(y^{\delta}_I) = \mathcal{L}^{\delta}_I(\rho)$ for all $(\delta,I) \in SCl(\phi,[r,r+1))$ and $\nu_0(x^{\phi}_{[r,r+1)}) = \#\rho$ and $\nu_0(y^{\phi}_{[r,r+1)}) = 0$, then the automaton $\mathcal{A}(\phi,[r,r+1))$ will accept ρ and terminate with a valuation ν such that*

- $\nu(x^{\phi}_{[r,r+1)}) = \mathcal{F}^{\phi}_{[r,r+1)}(\rho)$
- $\nu(y^{\phi}_{[r,r+1)}) = \mathcal{L}^{\phi}_{[r,r+1)}(\rho)$ *and*
- $\nu(c) = \nu_0(c)$*, for all other clocks.*

Proof (sketch). The automaton $\mathcal{A}(\phi,[r,r+1))$ is given in Figure 6. For each $(\delta,I) \in SCl(\phi,[r,r+1))$, $\mathcal{A}(\phi,[r,r+1)$ uses the clock values of x^{δ}_I and y^{δ}_I in its guards, and it resets the clocks $x^{\phi}_{[r,r+1)}$ and $y^{\phi}_{[r,r+1)}$.

For every ψ, which is an immediate modal subformula of ϕ, we derive $cond(\psi,[r,r+1))$ as given in Table 1. The first two rows in Table 1 are directly adapted from Lemma 3. The last two rows, may be easily inferred from the semantics of $MITL[\mathsf{F}_b,\mathsf{P}_b]$. Hence, we may infer that $\forall i \in dom(\rho)$ if $\tau_i \in [r,r+1)$ then $\nu_0,\tau_i \models cond(\psi,r)$ iff $\rho,i \models \psi$. Now, if $\phi = \bigvee_{a\in\Sigma}(a \wedge \mathscr{B}_a(\psi_i))$, then the guard on the transitions labelled by a, which reset $x^{\phi}_{[r,r+1)}$ and $y^{\phi}_{[r,r+1)}$ (as in figure 6) is given by $\mathcal{G}(\phi,[r,r+1),a) = \mathscr{B}_a(cond(\psi_i,[r,r+1)))$. It is straightforward to see that $\forall i \in dom(\rho)$ if $\tau_i \in [r,r+1)$ and $\sigma_i = a$ then $\nu_0,\tau_i \models \mathcal{G}(\phi,[r,r+1),a)$ iff $\rho,i \models \phi$. By observing the *po2DTA* in figure 6, we can infer that it clocks the first and last ϕ-positions in the unit interval $[r,r+1)$, and respectively assigns it to $x^{\phi}_{[r,r+1)}$ and $y^{\phi}_{[r,r+1)}$.

□

Theorem 3. *Given any MITL$[F_b,P_b]$ formula Φ, we may construct a po2DTA which is language-equivalent to Φ. Satisfiability of MITL$[F_b,P_b]$ formulas is decidable in NEXPTIME-time.*

Proof. Firstly, Φ is reduced to the normal form, as described in section 2.1. The automaton is given by $\mathcal{A}_\Phi = \mathcal{A}_{reset};\mathcal{A}_{induct};\mathcal{A}_{check}$[4]. While $\mathcal{A}_{reset}$ makes a pass to the end of the word and resets all x^{δ}_I (for all $(\delta,I) \in Cl(\Phi,[0,1))$) to the value $\tau_{\#\rho}$, $\mathcal{A}_{induct}$ sequentially composes $\mathcal{A}(\delta,I)$ in a bottom-up sequence. This ensures that the conditions for the initial valuation of each of the component automata, as required in lemma 4, are satisfied. Finally, $\mathcal{A}_{check}$ checks if the clock value $x^{\Phi}_{[0,1)} = 0$, thereby checking the validity of Φ at the first position in the word.

Complexity: Assuming DAG representation of the formula, reduction to normal form only gives a linear blow up in size of the DAG. Observe that the $Cl(\psi,[r,r+1))$ for $\psi = \mathsf{F}_I(\phi)$ or $\mathsf{P}_I(\phi)$ contains $m+1$ number of elements of the form $\phi,[k,k+1)$, where m is the length of the interval I. Hence, if interval constraints are encoded in binary, it is easy to see that the size of $Cl(\Phi,[0,1))$ is $O(2^l)$, where l is the modal DAG-size of Φ. Since each $\mathcal{A}(\phi,[r,r+1))$ has a constant number of states, we may infer that the number of states in $\mathcal{A}(\Phi)$ is $O(2^l)$. Since the non-emptiness of a *po2dfa* may be decided with NP-complete complexity, we conclude that satisfiability of a $MITL[\mathsf{F}_b,\mathsf{P}_b]$ formula is decidable with NEXPTIME complexity. □

4 Expressiveness and Decision Complexities

Figure 1 depicts the satisfaction complexities of various unary sublogics of *MITL* that are studied in this paper.

Theorem 4. *[Lower Bounds]*

- *Satisfiability of MITL$[F_I]$ (and hence MITL$[F_I,P_I]$) is EXPSPACE-hard.*
- *Satisfiability of MITL$[F_b]$ (and henceMITL$[F_b,P_b]$) is NEXPTIME-hard.*
- *Satisfiability of MITL$[F_0]$ (and hence also MITL$[F_{0,\infty}]$ and MITL$[F_{0,\infty},P_{0,\infty}]$) is PSPACE-hard.*

[4] The operator “;” denotes sequential composition of *po2DTA*.

Proof. We use tiling problems ([Boa97], [Für83]) of suitable complexities and reduce their instances to formulas of the corresponding unary *MITL* fragment, such that solvability of the instance reduces to satisfiability of the formula. The detailed reductions are quite routine and omitted. They may be found in the full version.

The relative expressiveness of the fragments of Unary $MITL[\mathsf{F}_I, \mathsf{P}_I]$ is given in Theorem 5 and is depicted in Figure 2. The figure also indicates the languages considered to separate the logics expressively.

Theorem 5. *[Expressiveness]*
$MITL[F_b, P_b] \subsetneq MITL[F_\infty, P_\infty] \subsetneq MITL[F_{0,\infty}, P_{0,\infty}] \subsetneq MITL[F_I, P_I]$

Proof. The formal proofs of separations between $MITL[\mathsf{F}_b, \mathsf{P}_b]$ and $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$, and $MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$ and $MITL[\mathsf{F}_I, \mathsf{P}_I]$ are given in the full version. They use the technique of EF-games for MTL, which was introduced in [PS11].

We shall now prove that $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty] \subsetneq MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$. Logic $MITL[\mathsf{F}_\infty, \mathsf{P}_\infty]$ is a syntactic fragment of $MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$. We shall now show that it is strictly less expressive than $MITL[\mathsf{F}_{0,\infty}, \mathsf{P}_{0,\infty}]$ by showing that there is no *po2DTA* which accepts the language given by the formula $\phi := \mathsf{F}_{[0,\infty)}(a \wedge \mathsf{F}_{[0,2)}c)$. The proof relies on the idea that since a *po2DTA* may be normalized to one that has a bounded number of clocks (bounded by the number of progress edges), and every edge may reset a clock at most once on a given run, the *po2DTA* cannot "check" every a for its matching c in a timed word which has sufficiently many ac pairs.

Assuming to contrary, let $\mathcal{A}$ be a *po2DTA* with m number of progress edges , such that $\mathcal{L}(\phi) = \mathcal{L}(\mathcal{A})$. Now consider the word ρ consisting of the event sequence $(ac)^{4m+1}$ where the x^{th} ac pair gives the timed subword $(a, 3x)(c, 3x + 2.5)$. Thus, each c is 2.5 units away from the preceding a. Hence, $\rho \notin \mathcal{L}(\phi)$. Consider the run of $\mathcal{A}$ over ρ. There are a maximum number of m clocks in $\mathcal{A}$ that are reset, in the run over ρ.

By a counting argument, there are at least $m+1$ (possibly overlapping but distinct) subwords of ρ of the form $acacac$, none of whose elements have been "clocked" by $\mathcal{A}$. Call each such subword a group. Enumerate the groups sequentially. Let v_j be a word identical to ρ except that the j^{th} group is altered, such that its middle c is shifted by 0.7 t.u. to the left, so that v_j satisfies the property required in ϕ. Note that there are at least $m+1$ such distinct v_j's and for all j, $v_j \in \mathcal{L}(\phi)$.

Claim: Given a v_j, if there exists a progress edge e of $\mathcal{A}$ such that in the run of $\mathcal{A}$ on v_j, e is enabled on the altered c, then for all $k \neq j$, e is not enabled on the altered c of v_k. (This is because, due to determinism, the altered c in v_j must satisfy a guard which neither of its two surrounding c's in its group can satisfy).
From the above claim, we know that the m clocks in $\mathcal{A}$, may be clocked on at most m of the altered words v_j. However, the family $\{v_j\}$ has at least $m+1$ members. Hence, there exists a k such that the altered c of v_k, (and the k^{th} group) is not reachable by ψ in ρ or any of the $\{v_j\}$. Hence $w \models \psi$ iff $v_k \models \psi$. But this is a contradiction as $\rho \notin L(\phi)$ and $v_k \in L(\phi)$ with $L(\phi) = L(\psi)$.

Therefore, there is no *po2DTA* which can express the language $\mathcal{L}(\phi)$. □

References

[AFH96] Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM 43(1), 116–146 (1996)

[AH92] Alur, R., Henzinger, T.A.: Back to the future: Towards a theory of timed regular languages. In: FOCS, pp. 177–186 (1992)

[AH93] Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Inf. Comput. 104(1), 35–77 (1993)

[BMOW08] Bouyer, P., Markey, N., Ouaknine, J., Worrell, J.B.: On Expressiveness and Complexity in Real-Time Model Checking. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 124–135. Springer, Heidelberg (2008)

[Boa97] Van Emde Boas, P.: The convenience of tilings. In: Complexity, Logic, and Recursion Theory, pp. 331–363. Marcel Dekker Inc. (1997)

[EVW02] Etessami, K., Vardi, M.Y., Wilke, T.: First-order logic with two variables and unary temporal logic. Inf. Comput. 179(2), 279–295 (2002)

[Für83] Fürer, M.: The Computational Complexity of the Unconstrained Limited Domino Problem (with Implications for Logical Decision Problems). In: Börger, E., Rödding, D., Hasenjaeger, G. (eds.) Rekursive Kombinatorik 1983. LNCS, vol. 171, pp. 312–319. Springer, Heidelberg (1984)

[Hen91] Henzinger, T.A.: The Temporal Specification and Verification of Real-Time Systems. PhD thesis, Stanford University (1991)

[PS10] Pandya, P.K., Shah, S.S.: Unambiguity in Timed Regular Languages: Automata and Logics. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 168–182. Springer, Heidelberg (2010)

[PS11] Pandya, P.K., Shah, S.S.: On Expressive Powers of Timed Logics: Comparing Boundedness, Non-punctuality, and Deterministic Freezing. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 60–75. Springer, Heidelberg (2011)

[STV01] Schwentick, T., Thérien, D., Vollmer, H.: Partially-ordered two-way automata: A new characterization of DA. In: Kuich, W., Rozenberg, G., Salomaa, A. (eds.) DLT 2001. LNCS, vol. 2295, pp. 239–250. Springer, Heidelberg (2002)

On Temporal Logic and Signal Processing

Alexandre Donzé[1], Oded Maler[2], Ezio Bartocci[4], Dejan Nickovic[3],
Radu Grosu[4], and Scott Smolka[5]

[1] EECS Department, University of California, Berkeley, USA
donze@imag.fr
[2] Verimag, Université Joseph Fourier/CNRS, Gières, France
[3] Austrian Institute of Technology, Vienna, Austria
[4] Department of Computer Engineering, Vienna, University of Technology, Austria
[5] Department of Computer Science, Stony Brook University, USA

Abstract. We present *Time-Frequency Logic* (TFL), a new specification formalism for real-valued signals that combines temporal logic properties in the time domain with frequency-domain properties. We provide a property checking framework for this formalism and illustrate its expressive power in defining and recognizing properties of musical pieces. Like hybrid automata and their analysis techniques, the TFL formalism is a contribution to a unified systems theory for hybrid systems.

1 Introduction

The exportation of Temporal Logic (TL) from philosophy [31,32] to systems design [28,29] is considered a turning point in formal verification [33], putting the focus on the *ongoing input-output behavior* of a *reactive* system [14] rather than on the final output of a complex program. While reactive systems might be a concept worth distinguishing in Computer Science, in other domains such as Control, Signal Processing and Circuit Design, being reactive is the rule, not the exception. Such systems are viewed by designers as networks of transducers (block diagrams) communicating continuously via *signals*: functions from Time to some domain, such as the Reals. This is the world view underlying data-flow languages and frameworks such as Lustre [2], Signal [1], Simulink[1] and Ptolemy [27], as well as transistor-level circuit simulators.

TL provides a convenient framework for writing in a compact and formal way specifications that the system under design should satisfy. It was initially intended to evaluate clean and well-defined *sequences* of states and events as found in digital systems. In the last couple of years, it has been extended to the specification of properties of *real-valued* signals defined over *dense time* [19,21] and applied to diverse domains ranging from analog circuits [17] to biochemical reactions [6]. The logic STL (*signal temporal logic*) allows designers to speak of properties related to the *order* of discrete events and the *temporal distance* between them, where "events" correspond to changes in the satisfaction of some

[1] http://www.mathworks.com/tagteam/43815_9320v06_Simulink7_v7.pdf

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 92–106, 2012.

predicate (e.g., threshold crossing) over the real variables. Traditional performance measures used to evaluate signals are more continuous and "event-free", for example, averaged/discounted integrals of some variables over time, and the added expressivity is an important contribution to the emergence of a hybrid *Systems Theory*.

One of the technical and cultural impediments to the adoption of STL, especially in analog circuits (its original motivating domain), was its purely *time-domain* nature; i.e., it did not lend itself to *frequency-domain* analysis. This kind of analysis is based on the Fourier spectrum of the signal which in many engineering applications is more important than the properties of the signal itself; i.e., its properties in the time domain, which STL is intended to express. One reason for this bias is that real-life analog signals are accompanied by omnipresent *noise*, i.e., random perturbations of the desired signal. This is conveniently dealt with in the frequency domain via filtering. Typically, the noise component of a signal populates a range of frequencies different from the range of the signal of interest. If we keep only the latter, then the amplitude of the noise is strongly reduced. More generally, an analog signal is usually a composition of multiple sources, and the first purpose of signal processing is the *separation* of these sources. In the frequency domain, this is done by simple operations such as thresholding or filtering the range of frequencies of interest, assuming that each source has a range clearly distinct from the ranges of others.

Source separation, noise removal and signal filtering are fundamental operations which are *time-invariant*. They affect the signal in the same way from time $t = -\infty$ to $t = \infty$. Since the Fourier transform (FT) is defined over this unbounded time-interval, it is appropriate for these operations. However, when it comes to characterize bounded or local-time intervals, the FT becomes cumbersome in practice, as its definition aggregates for each frequency of interest all values along the duration of the signal. This observation naturally led to the search for proper *time-frequency* analysis techniques, beginning from straightforward extensions of FT (such as the Short Time Fourier Transform (STFT) [10]) and culminating with the sophisticated and versatile family of Wavelet transforms [22]. Time-frequency analysis as a branch of Signal Processing have seen contributions of tremendous importance. To name only one, modern compression algorithms at the root of Jpeg, Mpeg, etc, file formats are all based on wavelet theory.

In this paper, we complement this evolution toward a fusion of time-domain and frequency-domain analysis by proposing a unified logical formalism for expressing hybrid (time-frequency) properties of signals. Some preliminary work in this direction is reported in [3] in which the author describes "envelope" predicates over the Fourier coefficients in the context of hybrid systems verification. However they use the standard Fourier transform, thus not treating the tighter coupling of time and frequency domains that we investigate here. Attempts have been done in the past to apply time-frequency analysis as a design methodology to different application domains. In the context of analog circuits, time-frequency analysis was used to study dynamic current supplies, oscillators, leapfrog and state variable filters etc. In the bio-medical domain time-frequency analysis was applied to detect anomalous ECG signals. We observed that the common disadvantage of these techniques is the lack of a formalism to express complex temporal patterns of frequency responses and a suitable machinery to automatically detect them. Our formalism, which we call *Time-Frequency Logic* (TFL), is implemented in a generic monitoring framework, part of the tool Breach [5].

A somewhat related work is *Functional reactive programming* (FRP) [16], a paradigm for high-level declarative programming of hybrid systems, instantiated with the Yampa language which was used to program industrial-strength mobile robots [26] but also to synthesize music [11]. Although it shares similar concepts with our work (signal transformers), FRP as a *programming* paradigm remains complementary to TFL, which is a *specification* formalism supporting, in particular, acausality and non-determinism.

The rest of the paper is organized as follows. Section 2 gives a short introduction to STL. Section 3 is a crash course on Fourier analysis.[2] Section 4 explains time-frequency analysis and how it is integrated in the TFL monitoring framework, while Section 5 demonstrates TFL's applicability to Music.

2 Signal Temporal Logic

We present STL in a manner that makes it closer to the world of Control and Signal Processing. Rather than taking the logic as a primary object and put all the numerical predicates and functions as part of domain-specific *signature* (as in [21]), we take a general framework of a data-flow network (as has been done in [4]) and add the temporal operators (*until* and *since*) as a special type of signal transducers. This approach is also close to [24], where AMS-LTL is extended with *auxiliary functions* that allow to embed arbitrary signal transducers in the body of property specifications.

This style of presentation is facilitated by a somewhat non-orthodox way of defining the semantics of temporal logic using *temporal testers* [18,30,23]. This approach has already been applied to STL monitoring [19,21] and to translating the MITL logic into timed automata [20]. An STL formula φ is viewed as a network of signal operators (transducers), starting with the raw signals x (sequences of atomic propositions in the discrete case), and culminating in a top-level signal φ,[3] whose value at t represents the satisfaction of the top-level formula at time t: $\varphi[t] = 1$ iff $(x, t) \models \varphi$. Each sub-formula of the form $\varphi = f(\varphi_1, \varphi_2)$ is thus associated with a signal transducer realizing f, whose inputs are the satisfaction signals of φ_1 and φ_2. The whole apparatus for monitoring the satisfaction of a formula by a signal can thus be viewed as a network of operators working on signals of two major types: *numerical* (raw signals and those obtained by numerical operations on them) and *Boolean* (satisfaction signals of sub-formulae).

[2] We recommend [15] as a first reading for computer scientists.

[3] We make a compromise between the conventions of Logic and those of other less formal domains by writing abusively φ for both the formula and its satisfaction signal and will do the same for variables x and their associated raw signals.

We assume signals defined as functions from Time $\mathbb{T}$ to some domain D. The range $\mathbb{T}$ of the signal can be finite $[0, r]$, infinite $[0, \infty]$ or bi-infinite $[-\infty, \infty]$ and we will make distinctions only when needed.

Definition 1 (Signal Operator). *A* signal operator *is a mapping* $f : (\mathbb{T} \to D_1) \to (\mathbb{T} \to D_2)$*, where* D_1 *and* D_2 *are, respectively, the domains of the input and output signals.*

The domains D_1 and D_2 define the *type* of the operators, and we always assume that the arguments of these operators match the type. We assume that all operators are (approximately) computable so that given some representation of a signal x, it is possible to produce a representation $y = f(x)$.

Definition 2 (Operator Classification). *Let* f *be a signal operator and let* $y = f(x)$*. We say that* f *is*

- Pointwise (memoryless) *if it is a lifting to signals of a function* $f : D_1 \to D_2$*, that is,* $\forall t \; y[t] = f(x[t])$*;*
- Causal *if, for every* t*,* $y[t]$ *is computed based on at most* $x[0], \ldots, x[t]$*;*
- Acausal *(otherwise)*

The causal operators are, for example, *past* temporal operators, back-shifts, or integrals over temporal windows that extend backwards. The advantage of such operators is that they can be naturally monitored *online*, which is particularly important for monitoring *real* systems rather than simulated *models*. The acausal operators are the future temporal operators and other operators that depend on values in temporal windows that extend beyond the current point in time.

Definition 3 (Temporal Operators). *Let* φ_1 *and* φ_2 *be two Boolean signals and let* a, b *be two positive numbers satisfying* $a \leq b$*. Then* $\psi_1 = \varphi_1 \, \mathcal{U}_{[a,b]} \varphi_2$ *(*φ_1 until φ_2*) are the signals satisfying*[4]

$$\begin{array}{l} (x,t) \models \varphi_1 \, \mathcal{U}_{[a,b]} \varphi_2 \;\; \textit{if} \\ \quad \exists t' \in t \oplus [a,b] \;\; (x,t') \models \varphi_2 \;\; \wedge \;\; \forall t'' \in [t,t'] \;\; (x,t'') \models \varphi_1[t'']) \end{array} \tag{1}$$

Recalling that over the Booleans, $\wedge$ and $\vee$ coincide with min and max, (1) is equivalent to:

$$\psi_1[t] = \max_{t' \in t \oplus [a,b]} \min(\varphi_2[t'], \; \min_{t'' \in [t,t']} \varphi_1[t'']) \tag{2}$$

The derived operator $\Diamond_{[a,b]} \varphi = \mathit{true} \, \mathcal{U}_{[a,b]} \varphi$ (*eventually* φ) is true at t when $\varphi[t']$ holds in *some* $t' \in t \oplus [a,b]$, while $\Box_{[a,b]} \varphi = \neg \Diamond_{[a,b]} \neg \varphi$ (*always* φ) requires φ to hold throughout the interval $t \oplus [a,b]$. The untimed *until*, $\mathcal{U} = \mathcal{U}_{[0,\infty]}$ does not pose any metric constraints on the timing of the future occurrence of φ_2.

Let us assume a set $x_1, \ldots, x_m$ of variables and a family F of signal operators including

[4] Expression $t \oplus [a,b]$ denotes the interval $[t + a, t + b]$.

- Pointwise operators that realize standard arithmetical and logical operators such as $+$, $\cdot$, min, max, $\wedge$, $\neg$, $=$ and $<$;
- Other useful operators such as integral, convolution, etc.
- As many instances as needed of $\mathcal{U}_{[a,b]}$.

Definition 4 (STL Syntax). *The syntax of an STL formula is defined inductively as*

- *An atomic formula is any variable x_i or any rational constant c;*
- *If φ is a formula, so is any $f(\varphi)$ for any operator $f \in F$ compatible with the type of φ.*
- *If φ_1 and φ_2 are formulae, so is any $f(\varphi_1, \varphi_2)$ for any operator $f \in F$ compatible with the types of φ_1 and φ_2.*

The semantics of an STL formula φ relative to a raw signal $x = (x_1, \ldots, x_m)$ is immediate; that is, the semantics of x is the signal x and the semantics of a constant is the constant signal c. Then the semantics of $f(\varphi)$ or $f(\varphi_1, \varphi_2)$ is obtained by applying the operator f to the semantics of φ or φ_1 and φ_2.

The work in [19,21] shows that given an STL formula φ and a signal x, there is an algorithm that can check whether x satisfies φ by computing the satisfaction signals of all sub-formulae. The algorithm works on the parse tree of φ, scanning the raw signals and propagating values upwards in the tree as well as backwards and forward in time until the satisfaction of all sub-formulae are computed, including $\varphi[0]$. The tool AMT [25] realizing this algorithm solves various practical problems that we do not discuss here such as the interpretation of STL over signals of finite duration, bridging the gap between ideal mathematical signals defined over $\mathbb{R}$ and actual signals given by a finite sequence of sampled values, combining online and offline monitoring, etc. A quantitative semantics for STL has been presented in [7], which returns as output positive or negative numbers indicating how robustly the property is satisfied or violated. These numbers are propagated naturally using a real-valued version of (2). The Breach tool [5] implements monitoring for this semantics.

3 Frequency Analysis in a Nutshell

The essence of frequency analysis [8] is that a signal can be transformed into an alternative representation consisting of a weighted sum of basic elementary signals, namely, sinusoids of various frequencies and phases. E.g., the signal x of Fig. 1 can be written as $x = x_1 + x_2 + x_3$ with $x_i[t] = b_i \sin 2\pi\omega_i t$, where b_i is the amplitude/coefficient of the sinusoid of frequency ω_i. Thus, the signal is transformed from a time-domain representation $x : \mathbb{T} \to D$ to a function $\hat{x}$ mapping frequencies to their coefficients. Many standard signal-processing operations are best defined as manipulating these coefficients. E.g., if we nullify b_3, we remove the high-frequency component of x to obtain signal $\tilde{x}$, which can be viewed as removing noise from x (Fig. 1).

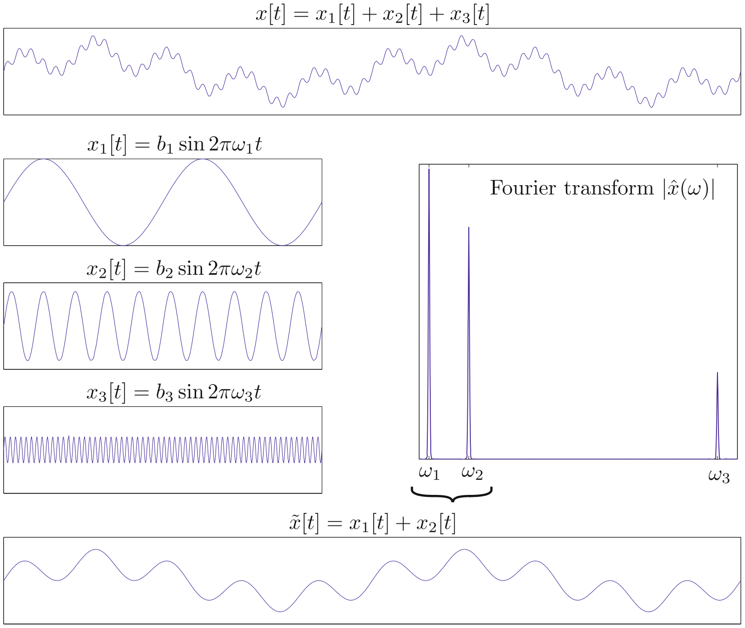


Fig. 1. Fourier transform of a sum of sinusoids and filtering the highest frequency

More formally, on any interval of size $T_0 = 1/\omega_0$, a signal x can be decomposed into a *Fourier series*:

$$x[t] = \frac{a_0}{2} + \sum_{k=0}^{+\infty} a_k \cos(2\pi\omega_k t) + b_k \sin(2\pi\omega_k t) \text{ with } \omega_k = k\omega_0.$$

This can be written more concisely using Euler's formula $e^{\mathbf{i}x} = \cos(x) + \mathbf{i}\sin(x)$:

$$x[t] = \sum_{k=-\infty}^{+\infty} c_k e^{2\mathbf{i}\pi k\omega_0 t} \tag{3}$$

Coefficients $\{c_k, k \in \mathbb{Z}\}$ provide a discrete spectrum for x on $[0, T_0]$. The *Fourier transform* maps x on the *whole time domain* to a continuous spectrum $\{c_\omega,\ \omega \in \mathbb{R}\}$ containing all real frequencies. The *inverse Fourier transform* (IFT), which recovers x from its spectrum, can be written as:

$$x[t] = \int_{-\infty}^{+\infty} c_\omega e^{2\pi\mathbf{i}\omega t} d\omega, \text{ where } c_\omega = \hat{x}(\omega) = \int_{-\infty}^{+\infty} x[t] e^{-2\mathbf{i}\pi\omega t} dt. \tag{4}$$

which can be seen as a generalization of the Fourier series (3). In practice, the coefficients c_ω are computed for a finite discrete set of frequencies using the FFT algorithm (Fast Fourier Transform [9]), but in the following, we keep the presentation in the continuous domain. This is in the spirit of STL, whose semantics is defined relative to dense-time signals, leaving to the monitoring algorithm the burden of dealing with time-discretization, interpolation, etc.

As mentioned previously, a convenient interpretation of the FT and its inverse is that of a decomposition of x into a sum of sinusoidal components, taking the family of functions $\phi_\omega : t \to e^{2\pi \mathbf{i} \omega t}$ as elementary "blocks" for the decomposition. With this notation, the IFT (4) becomes $x[t] = \int_{-\infty}^{+\infty} c_\omega \; \phi_\omega[t] d\omega$. However, the use of ϕ_ω as an elementary analysis block has the drawback that its definition involves the values of x for all times t. Thus, finding a subset of frequencies or some transformation of $\hat{x}$ that affects a precise time interval for x is not trivial. This motivated the search for other transforms using analysis functions that have a localization both in frequency and in time.

4 Combining Time and Frequency Properties

The Short-Time Fourier Transform (STFT). In the theory of signal processing, the extension of classical frequency analysis to a combined time-frequency analysis is realized by replacing the analysis function $\phi_\omega[t] = e^{2\pi \mathbf{i} \omega t}$ used in the FT with some other analysis function $\phi_{\omega,\tau}$ such that the new transform involves the values of x not only around a specific frequency ω but also around a given time τ (see, e.g., Chapter 4 of [22]). The *short-time* or *windowed* Fourier transform (STFT), first introduced by Gabor [10], uses a straightforward definition of such an analysis function. It consists of the product of ϕ_ω and a *window function* $g[t-\tau]$ whose purpose is simply to filter the values of x outside a neighborhood of τ by forcing them to be 0. It can be as simple as the *rectangular* function of length $L > 0$:

$$g_L[t] = \begin{cases} 1/L & \text{if } t \in [-\frac{L}{2}, \frac{L}{2}], \\ 0 & \text{else.} \end{cases} \tag{5}$$

but other functions with better properties such as the Hanning or Gaussian window functions are usually preferred [22]. They satisfy the normalization property $\int_{-\infty}^{+\infty} g[t]dt = 1$. In the following we assume that g has the form (5) and that the only parameter varying is its length L.

Having chosen a window function g_L, the new analysis function for the STFT is defined as the product $\phi_{\omega,\tau}[t] = \phi_\omega[t] g_L[t-\tau]$ of ϕ_ω and the translation of g_L around τ. Consequently, the STFT of x in (ω, τ), denoted $\hat{x}_L(\omega, \tau)$, defines a *two-dimensional* spectrum $\{c_{\omega,\tau} : (\omega, \tau) \in \mathbb{R}^2\}$. As with the IFT (4), x can be recovered from its spectrum and the analysis function $\phi_{\omega,\tau}$ using the inverse form of the STFT :

$$x[t] = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} c_{\omega,\tau} \phi_{\omega,\tau}[t] d\omega d\tau \tag{6}$$

For a given pair (ω, τ) of frequency and time, the coefficient $c_{\omega,\tau}$, i.e., the STFT of x in (ω, τ), is given by

$$c_{\omega,\tau} = \hat{x}_L(\omega, \tau) = \int_{-\infty}^{+\infty} x[t] g_L(t-\tau) e^{-2\mathbf{i}\pi\omega t} dt. \tag{7}$$

In practice, it can be computed with a straightforward extension of the FFT algorithm, using a sliding window and multiplying x by the g_L window function. The STFT can be visualized as a *spectrogram*, which plots the norms (or more commonly the squared norm) of the coefficients $c_{\omega,\tau}$ as a surface above the time-frequency plane (ω, τ) as illustrated in Fig. 2. There are inherent limitations (Heisenberg Uncertainty Principle) concerning the trade-offs between precision in frequency and precision in time. They are explained in the appendix.

Defining Time-Frequency Predicates. The STFT of a signal x thus defines a two dimensional operator taking time and frequency as arguments. By considering the frequency as a parameter, we obtain a family of signal operators $\{f_{L,\omega}\}$ such that $y = f_{L,\omega}(x)$ if $y[t] = \hat{x}_L(\omega, t)$. In other words, $f_{L,\omega}(x)$ is the projection of the L-spectrogram of x on frequency ω. It yields a *spectral signal* which tracks the evolution of the STFT coefficient at ω over time.

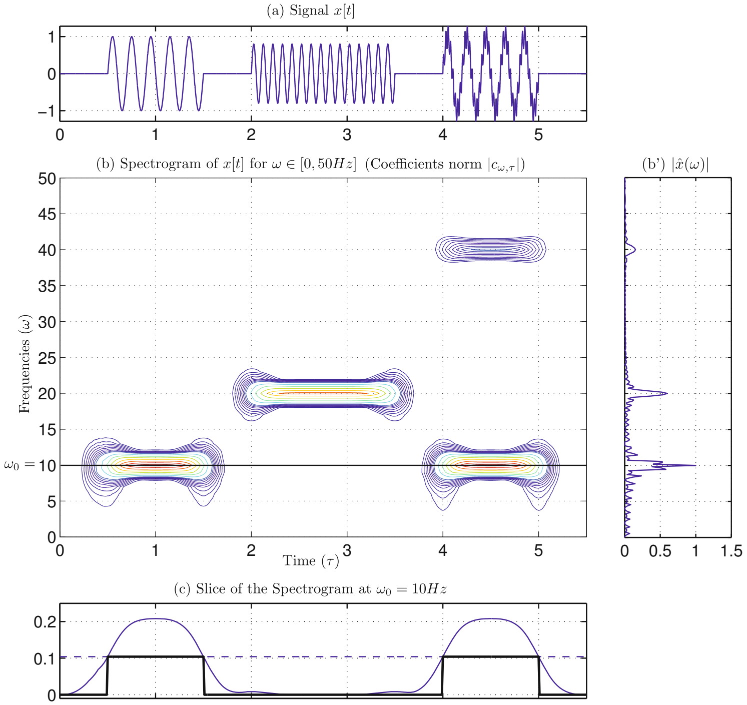


Fig. 2. Signal $x[t]$ in (a) is composed of different sinusoids at different times. A simple FT (b') exhibit peaks at frequencies 10, 20 and 40 without information about when they occur, whereas the spectrogram in (b) provide this information. We can see, e.g., that the frequency ω_0 =10 occurs at two distinct time intervals. In (c), a slice of the STFT at $\omega = \omega_0$ provides a signal that takes higher values at times when ω_0 is in the spectrum of x. Using a simple threshold, it then defines a predicate for the detection of frequency ω_0 along the time axis.

Our logic, *time-frequency logic* (TFL) is obtained by adding the operators $\{f_{L,\omega}\}$ to STL. A spectral signal $y = f_{L,\omega}(x)$, like any other signal, can participate in any TFL formula as an argument to predicates and arithmetic expressions. The monitoring machinery is similar to that of STL except for the fact that the raw signals x are pre-processed to yield the spectrogram from which spectral signals y are extracted. This can be done before the monitoring process starts or be integrated in an online procedure as in [21] where segments of y are computed incrementally upon the arrival of segments of x.

On Window Functions and Time-Frequency Resolution. To be able to detect the occurrence of a frequency ω at a given time τ, we would need a spectrogram representing a perfect matching between ω and τ for the signal x, i.e., that $|c_{\omega,\tau}|$ be non-zero if and only if x contains a component of frequency ω at time τ. Unfortunately, such ideal mapping cannot be obtained. To begin with, it is intuitively clear that low frequencies require an amount of time at least larger than the corresponding periods to be detectable. Moreover, there is an obvious technical limitation related to the sampling rate of x and the discretization of the FT by the FFT algorithm. But even if we ignore discretization errors and assume that we work in an ideal, continuous domain, there is a fundamental limitation of the time-frequency resolution than can be achieved, related to the Uncertainty Principle of Heisenberg. We sketch its development next (a thorough explanation is provided in [22]) since it provides a practical method for choosing the appropriate window function g for a desired accuracy of time-frequency detection with the STFT. The idea is as follows: assume a hypothetical signal x with an energy concentrated exactly at the time τ_0 and frequency ω_0. Then the STFT of x using a window function g will "spread" the concentrated energy of x in a box (so called an *Heisenberg box* [22]) in the time-frequency domain which has dimensions $\sigma_\tau(g) \times \sigma_\omega(g)$ given by

$$\sigma_\tau^2(g) = \int_{-\infty}^{+\infty} t^2 g(t)^2 dt \qquad \text{and} \qquad \sigma_\omega^2(g) = \int_{-\infty}^{+\infty} \nu^2 \hat{g}(\nu)^2 d\nu \tag{8}$$

The Uncertainty Principle asserts that the area of this box satisfies $\sigma_\tau \sigma_\omega \geq \frac{1}{2}$. It means that we cannot increase the accuracy in frequency detection without altering precision in time. The values of $\sigma_\tau(g)$ and $\sigma_\omega(g)$ can be easily estimated from the above formulae and can be used to optimize the trade-off between time and frequency detection. For instance, one has to make sure that the distance of two frequencies of interest is not small with respect to $\sigma_\omega(g)$.

5 Music

As observed in [10], human acoustic perception is a prime example of analyzing signals based on a combination of time and frequency features. In this section, we illustrate the applicability of TFL in formalizing and recognizing melodies starting with the basic task of note detection.

A note is characterized by a strong component at a fundamental frequency, or *pitch* ω. To obtain a note detection predicate, we thus define a spectral operator, pitch_ω, such that $\text{pitch}_\omega(x)[t]$ is the amplitude of frequency ω in signal x around time t. This operator must be able to tolerate small pitch variations while discriminating a note from its two closest neighboring notes, with pitches $\omega_1 = 2^{-\frac{1}{12}}\omega$ and $\omega_2 = 2^{\frac{1}{12}}\omega$. Thus, pitch is defined as the STFT $\text{pitch}_\omega(x)[t] = \hat{x}_L(\omega, t)$, where the size L of the window function is chosen to achieve the required time-frequency resolution (see Section 4). Using pitch_ω, a predicate detecting, e.g., the note A with pitch $\omega_A = 440\text{Hz}$ can be: $\mu_A = \text{pitch}_{\omega_A}(x) > \theta$. The only parameter which remains to be fixed is the threshold θ. It determines the robustness of the predicate to variations in volume, pitch or duration. If θ is large, it will be more sensitive to such variations, increasing the chance of false negative. Conversely, if it is small, it will tolerate more pitch variation but increase the chance of false positive, e.g., by recognising a wrong note. Fig. 3 displays the result of applying the pitch function to the detection of an F.

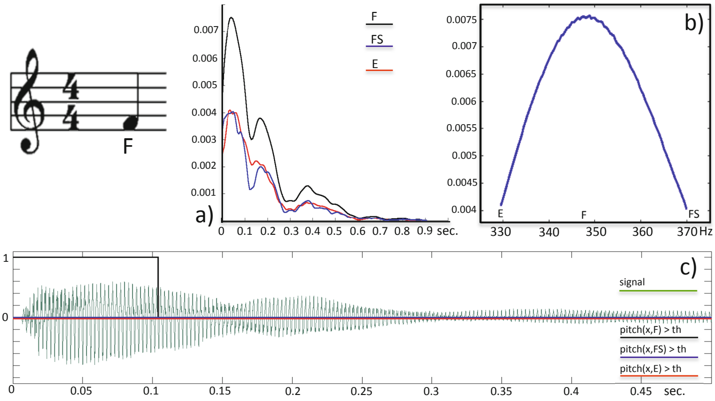


Fig. 3. Note detection: a) The STFT produced by the note F for its nominal frequency and the frequencies of its closest notes E and FS; b) Max amplitudes in time for the note F of a range of frequencies around its nominal frequency; c) The predicate μ_F is satisfied by the signal and the predicates μ_{FS} and μ_E are not.

Specifying Melodies. Music notation provides means to specify both the duration of a note and the pace of a piece, also called *tempo*. Tempo is expressed in units of *beats per minute* (bpm). The piece in Fig. 4 has a pace of 120 bpm. This means that an eighth note (F, G), a quarter note (E, A, B), a half note (D), and a whole note (C) have durations, respectively, of 0.25, 0.5, 1 and 2 seconds. The pause between G and A is one second long. In our experiments, we have generated, using a MIDI Player, two different signals that correspond to the performance of this melody by a violin and by an organ; see Fig. 4. We then checked three properties on both signals. The first two properties are: C is played for two seconds: $\square_{[0,2]}\mu_C$, and D is played for one second: $\square_{[0,1]}\mu_D$. As one can see from the corresponding satisfaction signals in Fig. 4, the first holds at the beginning of the signal and the second holds at the beginning of the second

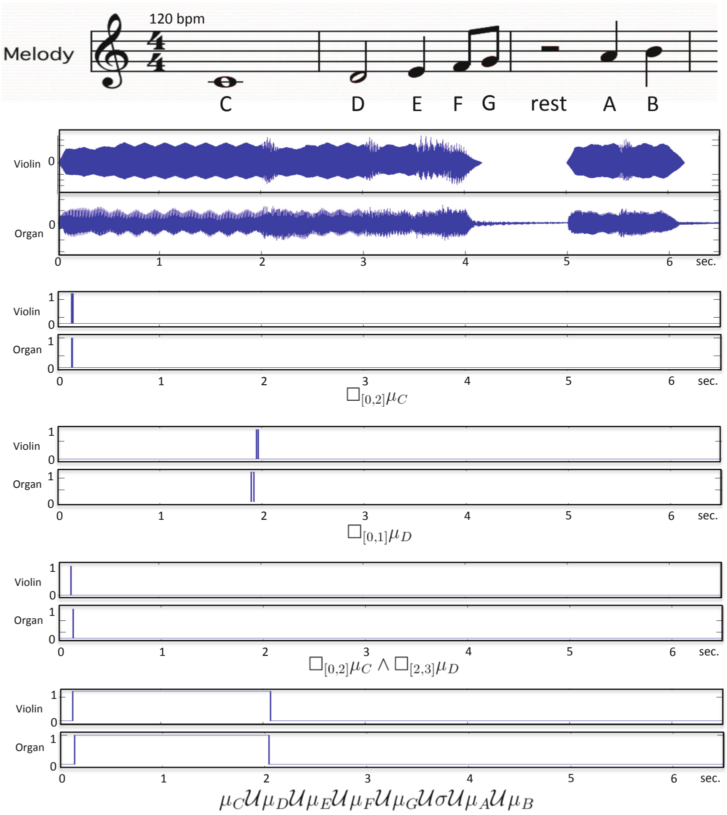


Fig. 4. Note and melody detection in violin and organ performances

note. The property $\Box_{[0,2]}\mu_C \wedge \Box_{[2,3]}\mu_D$ specifies the beginning of the melody and is found to hold at time zero. The last property ignores duration and specifies that the order of the notes corresponds to a diatonic scale of seven notes with a pause (expressed using a time domain predicate of the form $|x[t]| \leq \epsilon$ for some threshold) between G and A: $\mu_C \; \mathcal{U} \; \mu_D \; \mathcal{U} \; \mu_E \; \mathcal{U} \; \mu_F \; \mathcal{U} \; \mu_G \; \mathcal{U} \; \sigma \; \mathcal{U} \; \mu_A \; \mathcal{U} \; \mu_B$.

Recognizing a Blues Melody. For the last experiment, we tested our framework by trying to verify that a guitar melody, (imperfectly) played and recorded by one of the authors, was indeed a Blues melody. To do this, we built a formula based on the fact that standard blues is characterized by a 12-bar structure (a bar being basically four beats). In the key of E, it is as follows: `E E E E | A A E E | B A E E`. Note that a bar in E does not mean that we play four beats of E notes in a row. There can be different notes, but the overall bar should *sound like* a melody in the key of E. If we assumed that in a bar of E there should be *at least* one E note played, and similarly for A and B, it would be easy to write a formula that directly translates the above structure. However, this would be too strict in transcribing the above blues pattern for the blues line that we recorded. Indeed, our melody does not have an E in the fourth bar. Instead, we verified a simpler (but definitely bluesy) formula which looks for a starting E, an A in bars 5-6, and the so-called "turn-around" (the sequence B A E) in bars 9-11: $\varphi_{\text{blues}} = \mu_E \wedge \Diamond_{[5b,6b]}\mu_A \wedge \Diamond_{[8b,9b]}(\mu_B \wedge \Diamond_{[b,2b]}\mu_A \wedge \Diamond_{[2b,3b]}\mu_E)$. Our results are presented in Figure 5. The signal was recorded at 44 kHz for a length of 1320960 samples. The formula takes 6.9 s to be evaluated on a laptop with a Core i7 processor and 8 GB of memory.

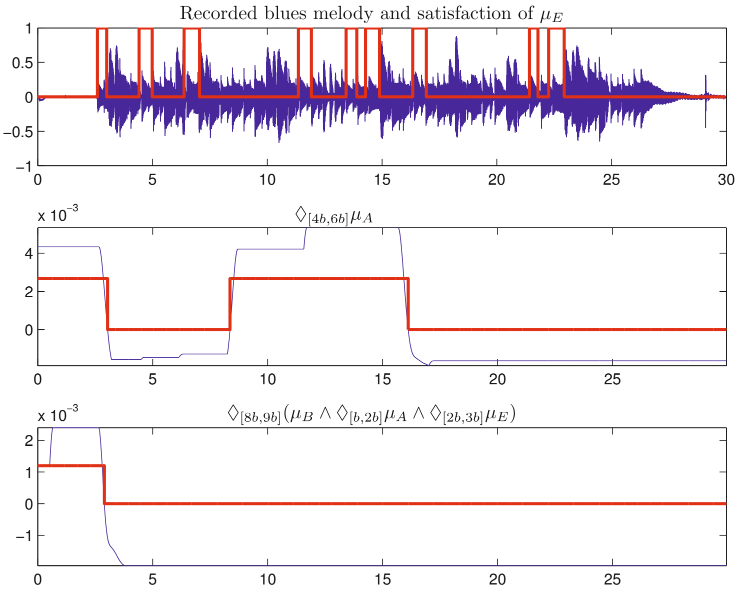


Fig. 5. Formula φ_{blues} is the conjunction of μ_E (top, with signal), $\Diamond_{[4b,6b]}\mu_A$ (middle), and $\Diamond_{[8b,9b]}(\mu_B \wedge \Diamond_{[b,2b]}\mu_A \wedge \Diamond_{[2b,3b]}\mu_E)$ (bottom). Since the three formulae are satisfied at the time of the first E, φ_{blues} is satisfied by our recording.

Implementation. We implemented the function pitch in Matlab and defined STL formulae using the Breach tool [5] (examples available at `http://www-verimag.imag.fr/~donze/breach_music_example.html`). Breach implements the full STL syntax (Boolean and temporal operators) on top of STL predicates of the form $\mu = f(x, p) > \theta$, where f is some signal operator and p is a parameter vector. The function f can be an arithmetic expression such as $2 * x[t] + p$ or a routine implemented separately and available in the Matlab environment, such as the pitch$_\omega$ routine or any other implementation of spectral operators $f_{L,\omega}$. This makes the implementation of TFL straightforward in the Breach framework.

6 Discussion

We have presented TFL, a specification formalism for time and frequency properties of signals, supported by a monitoring algorithm implemented in the Breach tool, and showed it in action on real acoustic signals. We believe that the expressivity added by the temporal operators can lead to new ways to specify music and, in particular, will allow us to define formulae that can quantify the amount of deviation of a performance from the "nominal" melody. Combining our formalism with complementary learning machinery (such as hidden Markov models, used in speech recognition), one could automatically "tune" note predicates to a given performance.

Although Music can definitely benefit from TFL, it is not necessarily the primary application domain we have in mind for this formalism. The convergence of technologies, where a typical system-on-a-chip features digital and analog components, including radio transmitters, likewise requires a convergence of modeling and analysis techniques used by the different engineering communities involved, and TFL is a step in this direction. Other extensions include the use of the more versatile Wavelet Transform for time-frequency analysis and the extension of the logic to *spatially extended* phenomena such as wave propagation in the spirit of [12,13], where dynamic cardiac conditions are specified and detected. Also note that the spectrogram is a two-dimensional entity indexed by both time and frequency, and that TFL is currently biased toward time. It would be interesting to explore a specification formalisms that can alternate more freely between temporal, frequential and spatial operators. On the engineering application side, we intend to apply the logic to specifying and monitoring the behavior of analog circuits.

References

1. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the signal language and its semantics. Sci. Comput. Program. 16(2), 103–149 (1991)
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL, pp. 178–188 (1987)
3. Chakarov, A., Sankaranarayanan, S., Fainekos, G.: Combining Time and Frequency Domain Specifications for Periodic Signals. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 294–309. Springer, Heidelberg (2012)
4. d'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: TIME, pp. 166–174. IEEE (2005)
5. Donzé, A.: Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010)
6. Donzé, A., Fanchon, E., Gattepaille, L.M., Maler, O., Tracqui, P.: Robustness analysis and behavior discrimination in enzymatic reaction networks. PLoS One 6(9) (2011)
7. Donzé, A., Maler, O.: Robust Satisfaction of Temporal Logic over Real-Valued Signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010)
8. Fourier, J.B.G.: Theórie analytique de la chaleur. Gauthier-Villars et fils (1888)
9. Frigo, M., Johnson, S.G.: The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology (September 1997)

10. Gabor, D.: Theory of communication. part 1: The analysis of information electrical engineers. Journal of the IEEE - Part III: Radio and Communication Engineering 93(26), 429–441 (1946)
11. Giorgidze, G., Nilsson, H.: Switched-On Yampa. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 282–298. Springer, Heidelberg (2008)
12. Grosu, R., Bartocci, E., Corradini, F., Entcheva, E., Smolka, S.A., Wasilewska, A.: Learning and Detecting Emergent Behavior in Networks of Cardiac Myocytes. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 229–243. Springer, Heidelberg (2008)
13. Grosu, R., Smolka, S., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. Communications of the ACM 52(3), 1–10 (2009)
14. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series, pp. 477–498. Springer (1985)
15. Hubbard, B.B.: The world according to wavelets. The story of a mathematical technique in the making, 2nd edn. CRC Press (2010)
16. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, Robots, and Functional Reactive Programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003)
17. Jones, K., Konrad, V., Nickovic, D.: Analog property checkers: a DDR2 case study. Formal Methods in System Design (2009)
18. Kesten, Y., Pnueli, A.: A compositional approach to CTL* verification. Theor. Comput. Sci. 331(2-3), 397–428 (2005)
19. Maler, O., Nickovic, D.: Monitoring Temporal Properties of Continuous Signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)
20. Maler, O., Nickovic, D., Pnueli, A.: From MITL to Timed Automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
21. Maler, O., Nickovic, D., Pnueli, A.: Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 475–505. Springer, Heidelberg (2008)
22. Mallat, S.: A Wavelet Tour of Signal Processing. Academic Press (2008)
23. Michel, M.: Composition of temporal operators. Logique et Analyse 110-111, 137–152 (1985)
24. Mukherjee, S., Dasgupta, P., Mukhopadhyay, S.: Auxiliary specifications for context-sensitive monitoring of AMS assertions. IEEE Transactions on CAD 30(10), 1446–1457 (2011)
25. Nickovic, D., Maler, O.: AMT: A Property-Based Monitoring Tool for Analog Systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007)
26. Pembeci, I., Nilsson, H., Hager, G.: Functional reactive robotics: an exercise in principled integration of domain-specific languages. In: PADP, pp. 168–179. ACM (2002)
27. Pino, J.L., Ha, S., Lee, E.A., Buck, J.T.: Software synthesis for DSP using Ptolemy. VLSI Signal Processing 9(1-2), 7–21 (1995)
28. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)

29. Pnueli, A.: The Temporal Semantics of Concurrent Programs. Theoretical Computer Science 13, 45–60 (1981)
30. Pnueli, A., Zaks, A.: On the Merits of Temporal Testers. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008)
31. Prior, A.N.: Past, Present, Future, Oxford (1969)
32. Rescher, N., Urquhart, A.: Temporal Logic. Springer (1971)
33. Vardi, M.Y.: From Philosophical to Industrial Logics. In: Ramanujam, R., Sarukkai, S. (eds.) ICLA 2009. LNCS (LNAI), vol. 5378, pp. 89–115. Springer, Heidelberg (2009)

Improved Single Pass Algorithms for Resolution Proof Reduction

Ashutosh Gupta

IST, Austria

Abstract. Unsatisfiability proofs find many applications in verification. Today, many SAT solvers are capable of producing *resolution proofs* of unsatisfiability. For efficiency smaller proofs are preferred over bigger ones. The solvers apply proof reduction methods to remove redundant parts of the proofs while and after generating the proofs. One method of reducing resolution proofs is *redundant resolution reduction*, i.e., removing repeated pivots in the paths of resolution proofs (*aka* Pivot recycle). The known single pass algorithm only tries to remove redundancies in the parts of the proof that are trees. In this paper, we present *three* modifications to improve the algorithm such that the redundancies can be found in the parts of the proofs that are DAGs. The first modified algorithm covers greater number of redundancies as compared to the known algorithm without incurring any additional cost. The second modified algorithm covers even greater number of the redundancies but it may have longer run times. Our third modified algorithm is parametrized and can trade off between run times and the coverage of the redundancies. We have implemented our algorithms in OpenSMT and applied them on unsatisfiability proofs of 198 examples from plain MUS track of SAT11 competition. The first and second algorithm additionally remove 0.89% and 10.57% of clauses respectively as compared to the original algorithm. For certain value of the parameter, the third algorithm removes almost as many clauses as the second algorithm but is significantly faster.

1 Introduction

An unsatisfiability proof is a series of applications of proof rules on an input formula to deduce *false*. Unsatisfiability proofs for a Boolean formula can find many applications in verification. For instance, one application is automatic learning of abstractions for unbounded model checking by analyzing proofs of program safety for bounded steps [14,13,10]. We can also learn unsatisfiable cores from unsatisfiability proofs, which are useful in locating errors in inconsistent specifications [22]. These proofs can be used by higher order theorem provers as sub-proofs of another proof [4].

One of the most widely used proof rules for Boolean formulas is the resolution rule, i.e., if $a \vee b$ and $\neg a \vee c$ holds then we can deduce $b \vee c$. In the application of the rule, a is known as *pivot*. A *resolution proof* is generated by applying resolution rule on the clauses of an unsatisfiable Boolean formula to deduce *false*. Modern

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 107–121, 2012.

SAT solvers (Boolean satisfiability checkers) implement some variation of DPLL that is enhanced with conflict driven clause learning [20,19]. Without incurring large additional cost on the solvers, we can generate a resolution proof from a run of the solvers on an unsatisfiable formula [23].

Due to the nature of the algorithms employed by SAT solvers, a generated resolution proof may contain redundant parts and a strictly smaller resolution proof can be obtained. Applications of the resolution proofs are sensitive to the proof size. Since minimizing resolution proofs is a hard problem [17], there has been significant interest in finding low complexity algorithms that partially minimize the resolution proofs generated by SAT solvers.

In [3], two low complexity algorithms for optimizing the proofs are presented. Our work is focused on one of the two, namely Recycle-Pivots. Lets consider a resolution step that produces a clause using some pivot p. The resolution step is called *redundant* if each deduction sequence from the clause to *false* contains a resolution step with the pivot p. A redundant resolution can easily be removed by local modifications in the proof structure. After removing the redundant resolution step, a strictly smaller proof is obtained. Detecting and removing all such redundancies is hard. Recycle-Pivots is a single pass algorithm that partially removes redundant resolutions. From each clause, the algorithm starts from the clause and follows the deduction sequences to find equal pivots. The algorithm stops looking for equal pivots if it reaches to a clause that is used to deduce more than one clause.

In this paper, we present *three algorithms* that are improved versions of Recycle-Pivots. For the first algorithm, we observe that each literal from a clause must appear as a pivot somewhere in all the deduction sequences from the clause to *false*. Therefore, we can extend search of equal pivots among the literals from the stopping clause without incurring additional cost. For the second algorithm, we observe that the condition for the redundant resolutions can be defined recursively over the resolution proof structure. This observation leads to a single pass algorithm that covers even more redundancies but it requires an expensive operation at each clause in a proof. Note that the second algorithm does not remove all such redundancies because the removal of a redundancy may lead to exposure of more. Our third algorithm is parametrized. This algorithm applies the expensive second algorithm only for the clauses that are used to derive a number of clauses smaller than the parameter. The other clauses are handled as in the first algorithm. The parametrization reduces run time for the third algorithm but also reduces the coverage of the redundancy detection.

We have implemented our algorithms in OpenSMT [5] and applied them on unsatisfiable proofs of 198 examples from plain MUS track of SAT11 competition. The original algorithm removes 11.97% of clauses in the proofs of the examples. The first and the second algorithm remove 12.86% and 22.54% of the clauses respectively. The third algorithm removes almost as many clauses as the second algorithm in lesser time for the parameter value as low as 10. We also observe similar pattern in reduction of the unsatisfiable cores of the examples.

Related Work: Two kinds of methods have been proposed in the literature for the resolution proof minimization. The first kind of methods interact with the SAT solver for proof reduction. In [23], a reduced proof is obtained by iteratively calling a SAT solver on the unsatisfiable core in the proof obtained from the last iteration. These iterations run until a fixed point is reached. Subsequently, many methods were developed to obtain minimal/minimum satisfiability core with the help of a SAT solver [12,16,11,6,15,8,9]. Their objectives were not necessarily to obtain a smaller proof but very often a consequence of a smaller unsatisfiable core is a smaller proof. The second kind of methods operate independently from the SAT solver and post-process the resolution proofs. The methods in [21,2] analyzes conflict graphs in the SAT solver (without re-running the solver) to find shared proof structures and attempts to obtain shared sub-proofs among the resolution proofs of the learned clauses. In [1], a method is presented that minimizes a resolution proof by heuristically reordering resolution steps using 'linking graph' between literals. In [18], the resolution proof rewriting rules [7] are iteratively applied to reorder resolution steps locally ([1] does it in a global context) and it is expected to expose some redundancies, which are removed by applying Recycle-Pivots after each iteration. This paper only aims to find algorithms that significantly minimize the proofs within low cost. Indeed, many of the above methods achieve more minimization as compare to our algorithms but with higher costs.

This paper is organized as follows. In section 2, we present our notation and the earlier known algorithm. In sections 3, 4, and 5, we present our three algorithms. In section 6, we discuss their complexities. We present our experimental results in section 7 and conclude in section 8. In appendix A, we present the proof of correctness of our algorithms.

2 Preliminaries

In this section, we will present our notation and one of the proof reduction algorithms presented in [3].

Conjunctive Normal Form(CNF): In the following, we will use $a, b, c, \ldots$ to denote Boolean variables. A *literal* is a Boolean variable or its negation. We will use $p, q, r, s...$ to denote literals. Let s be a literal. If $s = \neg a$ then let $\neg s = a$. Let $var(s)$ be the Boolean variable in s. A *clause* is a set of literals. A clause is interpreted as disjunction of its literals. Naturally, empty clause denotes *false*. We will use $A, B, C, ...$ to denote clauses. Let C and D be clauses. We assume for each Boolean variable b, $\{b, \neg b\} \nsubseteq C$. Let $C \vee D$ denote union of the clauses, and let $s \vee C$ denote $\{s\} \vee C$. A *CNF formula* is a set of clauses. A CNF formula is interpreted as conjunction of its clauses. We will use $P, Q, R, ...$ to denote CNF formulas. Let P be a CNF formula. Let $Atoms(P)$ be the set of Boolean variables that appear in P. Let $Lit(P) = \{a, \neg a | a \in Atoms(P)\}$. P is *satisfiable* if there exist a map $f : Atoms(P) \rightarrow \{0, 1\}$ such that for each clause $C \in P$, there is $s \in C$ for which if $s = a$ then $f(a) = 1$ and if $s = \neg a$ then $f(a) = 0$. P is *unsatisfiable* if no such map exists.

Resolution Proof: A resolution proof is obtained by applying the resolution rule to an unsatisfiable CNF formula. The resolution rule states that clauses $a \vee C$ and $\neg a \vee D$ imply clause $C \vee D$. $a \vee C$ and $\neg a \vee D$ are the *antecedent* clauses. $C \vee D$ is the *deduced* clause and a is the *pivot*. Let $C \vee D = Res(a \vee C, \neg a \vee D, a)$ if for each Boolean variable b, $\{b, \neg b\} \not\subseteq C \vee D$. We say a is the *resolving literal* between the clauses $a \vee C$ and $C \vee D$. Symmetrically, $\neg a$ is the resolving literal between $\neg a \vee D$ and $C \vee D$. Resolution is known to be sound and complete proof system for CNF formulas. In particular, a CNF formula is unsatisfiable if and only if we can deduce empty clause by applying a series of resolutions on the clauses of the the formula. The following is a definition of a labelled DAG that records the series of applications of the resolution rule.

Definition 1 (Resolution proof). *A resolution proof $\mathcal{P}$ is a labeled DAG (V, L, R, cl, piv, v_0), where V is a set of nodes, L and R are maps from nodes to their parent nodes, cl is a map from nodes to clauses, piv is a map from nodes to pivot variables, and $v_0 \in V$ is the sink node. $\mathcal{P}$ satisfies the following conditions:*

(1) V is divided into leaf and internal nodes.
(2) A leaf node *v has no parents, i.e., $L(v) = R(v) = \perp$ and $piv(v) = \perp$.*
(3) An internal node *v has exactly a pair of parents $L(v)$ and $R(v)$ such that $cl(v) = Res(cl(L(v)), cl(R(v)), piv(v))$.*
(4) v_0 is not a parent of any other node and $cl(v_0) = \emptyset$.

$\mathcal{P}$ is *derived* from unsatisfiable CNF formula P if for each leaf $v \in V$, $cl(v) \in P$. Let $Lit(\mathcal{P}) = Lit(\{cl(v) | v \in V\})$. Let $children(v) = \{v' \in V | v = L(v') \vee v = R(v')\}$. If $v' \in children(v)$ then let $rlit(v, v')$ be the resolving literal between v and v', i.e., if $v = L(v')$ then $rlit(v, v') = piv(v')$ else $rlit(v, v') = \neg piv(v')$.

Since we will be dealing with the algorithms that modify resolution proofs, we may refer to a resolution proof that satisfies all the conditions except the third. We will call such an object as *proof DAG*.

Proof Reduction: The resolution proofs obtained from SAT solvers may have redundant parts, which can be partially removed during the post-processing using low complexity algorithms. We focus on such an algorithm introduced in [3], namely RECYCLE-PIVOTS. The observation behind the algorithm is that if there is a node $v \in V$ such that each path from v to v_0 contains a node v' such that $piv(v) = piv(v')$ then the resolution at node v is redundant. v can be removed using an inexpensive transformation of the resolution proof. The transformed resolution proof is a strictly smaller than the original resolution proof. We will call this minimization *redundant pivot reduction*.

In figure 1, we present an algorithm RMREDUNDANCIES, which is a reproduction of RECYCLE-PIVOTS from [3]. RMREDUNDANCIES takes a resolution proof $\mathcal{P}$ as input and only removes the redundancies in parts of $\mathcal{P}$ that are trees. This algorithm traverses $\mathcal{P}$ twice using two algorithms, namely RMPIVOTS and RESTORERESTREE. RMPIVOTS detects and flags the redundant clauses in tree like parts of resolution. RESTORERESTREE traverses the flagged resolution proof and removes the redundant clauses using appropriate transformations.

```
global variables
$(V, L, R, cl, piv, v_0)$ : resolution proof        $visited : V \to \mathbb{B} = \lambda x.false$

fun RmRedundancies($\mathcal{P}$)
begin
   $(V, L, R, cl, piv, v_0) := \mathcal{P}$
   RmPivots($v_0, \emptyset$)
   $visited := \lambda x.false$
   $v_0 :=$ RestoreResTree($v_0$)
end

fun RmPivots($v$: node, $D$: literals)
begin
1   if $visited(v)$ then return
2   $visited(v) := true$
3   if $piv(v) = \bot$ then return
4   if $|children(v)| > 1$ then $D := \emptyset$
5   if $piv(v) \in D$ then
6      $R(v) := \bot$
7      RmPivots($L(v), D$)
8   elsif $\neg piv(v) \in D$ then
9      $L(v) := \bot$
10     RmPivots($R(v), D$)
11  else
12     RmPivots($L(v), D \cup \{piv(v)\}$)
13     RmPivots($R(v), D \cup \{\neg piv(v)\}$)
end

fun RestoreResTree($v$: node)
begin
1   if $visited(v)$ then return $v$
2   $visited(v) := true$
3   if $piv(v) = \bot$ then return $v$
4   if $L(v) = \bot$ then
5      $v' :=$ RestoreResTree($R(v)$)
6   elsif $R(v) = \bot$ then
7      $v' :=$ RestoreResTree($L(v)$)
8   else
9      $v_l :=$ RestoreResTree($L(v)$)
10     $v_r :=$ RestoreResTree($R(v)$)
11     match $(piv(v) \in cl(v_l), \neg piv(v) \in cl(v_r))$ with
12        | $(true, true)$ -> $v' := v$
13        | $(true, false)$ -> $v' := v_r$
14        | $(false, \_)$ -> $v' := v_l$
15  if $v = v'$ then
16     $cl(v) := Res(v_l, v_r, piv(v))$
17  else
18     for each $u : v = L(u)$ do $L(u) := v'$ done
19     for each $u : v = R(u)$ do $R(u) := v'$ done
20  return $v'$
end
```

Fig. 1. RmRedundancies, a dual pass resolution proof reduction algorithm from [3]

RmPivots recursively traverses the proof DAG in depth first manner. RmPivots takes a node v as the first input argument. At line 1-2 using map *visited*, it ensures that v is visited only once. At line 3, if v is a leaf then the algorithm returns. The algorithm also takes a set of literals D as the second input argument. D is a subset of the resolving literals that have appeared along the path using which DFS has reached v via the recursive calls at lines 7, 10, 12, and 13. At line 4, D is assigned empty set if v has multiple children. Consequently, D contains only the resolving literals that appeared after the last node with multiple children was visited. At line 5 and 8, if $piv(v)$ or $\neg piv(v)$ is found in D, then we have detected a redundant resolution step. The algorithm flags the detected redundant clause by removing one of the parent relations at lines 6 or 9. This modification in parent relations violates the conditions of a resolution proof.

After running RmPivots, RmRedundancies calls RestoreResTree to remove the flagged clauses. RestoreResTree traverses the proof DAG in the order of parents first. RestoreResTree takes a node v as input and returns a node v' that has a valid sub-proof (line 15–16) and replaces v in the resolution proof (line 17–20). If v is a flagged node then v' is the node that also replaces the remaining parent of v (line 4–6). If v was originally not flagged then it may

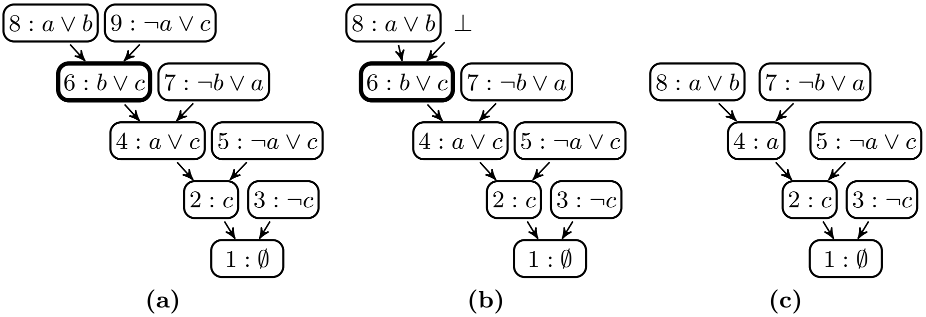


Fig. 2. Each node is labelled with a clause and assigned a number as a node id. The left parent corresponds to L parent and the right parent corresponds to R parent. The pivot used for producing a node can be inferred by looking at the clauses of the node and its parents. (a) An example resolution proof with a redundant resolution at node 6. (b) A proof DAG obtained after running RmPivots. (c) A resolution proof obtained after running RmRedundancies.

happen that one of its new parents v_l and v_r may not contain the literals corresponding to $piv(v)$ (line 9–12). In this case, v is treated as a flagged node(line 13–14). Please look in [3] for more detailed description of RestoreResTree.

Example 1. Consider the node 6 of the resolution proof presented in figure 2(a). The resolution at node 6 is redundant because the path from 6 to sink node 1 contains node 2 and both nodes 6 and 2 have pivot a. RmPivots will reach to node 6 with $D = \{a, b, c\}$. Therefore, $R(6)$ will be assigned $\perp$. In figure 2(b), we show a proof DAG obtained after running RmPivots. The subsequent run of RestoreResTree produces a resolution proof shown in figure 2(c).

For efficiency, RmPivots not so eagerly flags clauses for removal. In the following three sections, we will present three new algorithms to replace RmPivots that will detect more redundancies without adding much additional cost. RestoreResTree is general enough such that in all the three following algorithms it will be able to subsequently restore the resolution proof.

3 Using Literals of Clauses for Redundancy Detection

In this section, we will present our first modification in RmPivots that leads to detection of more redundant resolutions without additional cost.

For each node $v \in V$, we observe that each literal in $cl(v)$ has to act as a resolving literal in some resolution step along each path to the sink v_0 because $cl(v_0)$ is empty and a literal is removed in the descendants only by some resolution (Lemma 1 in appendix A). Now consider a run of RmPivots that reaches to a node v that has multiple children. At this point of execution, RmPivots resets parameter D to empty set. Due to the above observation, we are not required to fully reset D and can safely reset D to $cl(v)$.

In figure 3, we present our first algorithm RmPivots* which is a modified version of RmPivots. The only modification is at line 4 that changes the reset operation. This modification does not require any changes in RestoreResTree.

Example 2. Consider the resolution proof presented in figure 4(a). The resolution at node 7 is redundant but RmPivots will fail to remove it because node 5 has multiple descendants and the algorithm will not look further. In RmPivots*, the literals of $cl(5)$ are added in D therefore our modification enables it to detect the redundancy and remove it. The minimized proof is presented in figure 4(b).

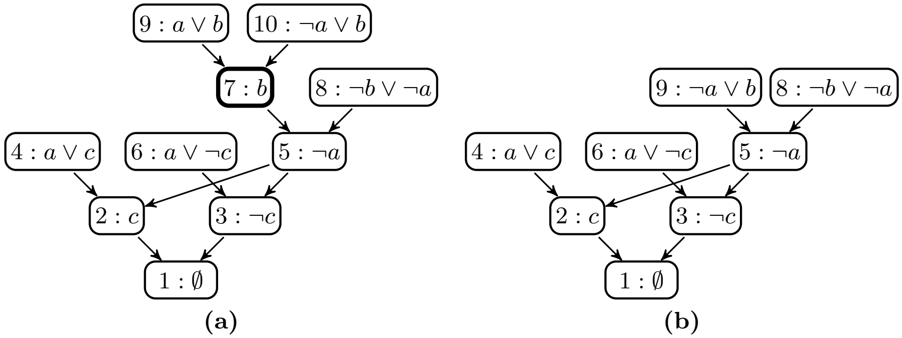
```
fun RmPivots*($v$: node, $D$: literals)
begin
1  if $visited(v)$ then return
2  $visited(v) := true$
3  if $piv(v) = \bot$ then return
4  if $|children(v)| > 1$ then $D := cl(v)$
5  if $piv(v) \in D$ then
6    $R(v) := \bot$
7    RmPivots*$(L(v), D)$
8  elsif $\neg piv(v) \in D$ then
9    $L(v) := \bot$
10   RmPivots*$(R(v), D)$
11 else
12   RmPivots*$(L(v), D \cup \{piv(v)\})$
13   RmPivots*$(R(v), D \cup \{\neg piv(v)\})$
end
```

Fig. 3. RmPivots*, our first improved version of RmPivots

4 All Path Redundancy Detection

In this section, we present our second modification in function RmPivots that considers all paths from a node to the sink to find the redundancies. This modification leads to even greater coverage in a single pass of a resolution proof but it may lead to a longer run time.

In the second modification, we additionally compute a set of literals, which we call the expansion set, for each node to guide the removal of redundant resolutions. For a node $v \in V$, the *expansion set* $\rho(v)$ is the largest set of literals such that if some proof transformation among ancestors of v leads to appearance of literals from $\rho(v)$ in $cl(v)$ then the resolution proof remains valid. Due to the



(a) **(b)**

Fig. 4. (a) An example of resolution proof on which RmPivots fails to detect the redundancy at node 7. (b) Minimized form of the example.

global variables

$\rho : V \to$ literal set $:= (\lambda x.Lit(\mathcal{P}))[v_0 \mapsto \emptyset]$ k : integer (parameter)

```
fun ALL-RMPIVOTS($v$: node)
begin
  if $visited(v) \vee piv(v) = \bot \vee$
    $\exists v' \in children(v).\neg visited(v')$
  then
    return
  $visited(v) := true$
  for each $v' \in children(v)$ do
    $\rho(v) := \rho(v) \cap (\rho(v') \cup \{rlit(v, v')\})$
  done
  $v_L := L(v)$
  $v_R := R(v)$
  if $piv(v) \in \rho(v)$ then $R(v) := \bot$
  if $\neg piv(v) \in \rho(v)$ then $L(v) := \bot$
  ALL-RMPIVOTS($v_L$)
  ALL-RMPIVOTS($v_R$)
end
```

```
fun K-RMPIVOTS($v$: node)
begin
  if $visited(v) \vee piv(v) = \bot \vee$
    $\exists v' \in children(v).\neg visited(v')$
  then
    return
  $visited(v) := true$
  if $children(v) \geq k$ then
    $\rho(v) := cl(v)$
  else
    for each $v' \in children(v)$ do
      $\rho(v) := \rho(v) \cap (\rho(v') \cup \{rlit(v, v')\})$
    done
  $v_L := L(v)$
  $v_R := R(v)$
  if $piv(v) \in \rho(v)$ then $R(v) := \bot$
  if $\neg piv(v) \in \rho(v)$ then $L(v) := \bot$
  K-RMPIVOTS($v_L$)
  K-RMPIVOTS($v_R$)
end
```

Fig. 5. ALL-RMPIVOTS and K-RMPIVOTS are our second and third modified algorithms respectively. Each can replace RMPIVOTS. To find redundant resolutions, ALL-RMPIVOTS considers all the paths from a node to the sink. Depending on the parameter k, K-RMPIVOTS only considers the paths that contain nodes with less than k children.

definition, $\rho(v)$ is a subset of the resolving literals that appear in all the paths from v to v_0. We cannot add all the resolving literals that appear in the paths in $\rho(v)$ because of the following reason. In a path from v to v_0, if literals s and $\neg s$ both appear as resolving literals then only the one that appears nearest to v can be added to $\rho(v)$. Otherwise, an expansion allowed by $\rho(v)$ may lead to the internal clauses that have both s and $\neg s$. If there are two paths in which s and $\neg s$ occur in opposite orders then none of them can be added to $\rho(v)$. The following equation recursively defines the expansion set for each node.

$$\rho(v) = \begin{cases} \emptyset & \text{if } v = v_0 \\ \bigcap\limits_{v' \in children(v)} \rho(v') \cup \{rlit(v, v')\} \setminus \{\neg rlit(v, v')\} & \text{otherwise.} \end{cases} \quad (1)$$

To understand the above equation, for each $v' \in children(v)$ lets assume we have the expansion set $\rho(v')$. All the paths from v to v_0 that goes via v' must have seen resolving literals $\rho(v') \cup \{rlit(v, v')\}$. If $\neg rlit(v, v')$ appears in $\rho(v')$ then we need to remove it as noted earlier. Therefore, $\rho(v)$ is the intersection of these resolving literal sets corresponding to the children of v. If $piv(v)$ or $\neg piv(v)$ is in $\rho(v)$ then $\rho(v)$ allows removal of the resolution step at v.

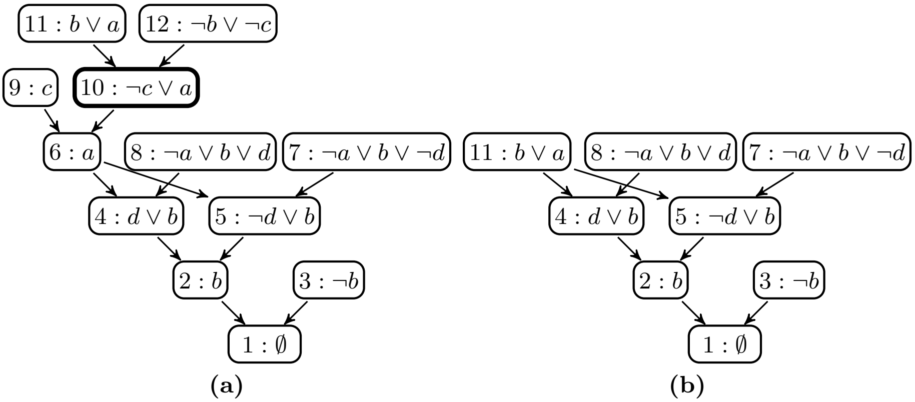


Fig. 6. (a) An example of resolution proof on which RmPivots* fails to detect the redundancy at node 10. (b) Minimized form of the example obtained by all-RMPivots.

In figure 5, we present the second modified algorithm all-RMPivots that implements the above computation of ρ and flags the resolution steps accordingly. The algorithm can replace RmPivots without any changes in RestoreResTree. In this presentation of the algorithm, we assume that initially all nodes of V are reachable from v_0. Initially, the global variable ρ maps v_0 to $\emptyset$ and rest of the nodes to all literals appear in $\mathcal{P}$. We initialize ρ in this way because if a node that eventually becomes unreachable from v_0 and has parents that are reachable from v_0 then we can consistently compute ρ for the parents. all-RMPivots takes a node v as input and decides to visit the node now or not. The condition at line 1 ensures that each node is visited only if it is an internal node, only once, and if its parents are already visited. all-RMPivots traverses $\mathcal{P}$ in the reverse topological order. During the visit of v, the loop at line 6 computes $\rho(v)$ slightly differently from the recursive equation (1). Subsequently at lines 11–12, the algorithm drops the parent relations if $\rho(v)$ contains $piv(v)$ or $\neg piv(v)$. The algorithm does not remove $\neg rlit(v, v')$ from $\rho(v')$ at line 7 as per the equation (1) because $\rho(v')$ cannot contain $\neg rlit(v, v')$. That is the case because, during the earlier visit to v', if $\rho(v')$ contained $\neg rlit(v, v')$ then the edge between v' and v must have been removed. At lines 13–14, a recursive call even for an ex-parent is made because if there were other children of the ex-parent and all of whom have been visited earlier then the ex-parent must be waiting to be visited. Due to the initialization of ρ, if $children(v)$ is empty then both the if-conditions at lines 11–12 are true and both the parent relations are removed. Therefore, if a node eventually becomes unreachable from the sink then the node also becomes isolated. Since removal of a redundancy may expose more redundancies, the algorithm removes the redundancies partially.

Example 3. Consider the resolution proof presented in figure 6(a). The resolution at node 10 is redundant because the pivot of node 10 is variable b and literal b appears as a resolving literal on both the paths from node 10 to node 1.

RmPivots* fails to detect it because node 6 has multiple descendants and $cl(6)$ does not contain b. all-RMPivots computes the following values for map ρ.

$$\rho(2) = \{b\} \quad \rho(4) = \{d, b\} \quad \rho(5) = \{\neg d, b\} \quad \rho(6) = \{a, b\} \quad \rho(10) = \{a, b, \neg c\}$$

Since $b \in \rho(10)$, all-RMPivots detects the redundancy. In figure 6(b), the minimized proof that is obtained after consecutive run of all-RMPivots and RestoreResTree is presented.

5 Redundancy Detection up to *k* Children

In this section, we present our third algorithm that only considers a fraction of paths from a node to the sink to find the redundancies. The fraction is determined by a parameter k. This algorithm only considers the paths that contain nodes with less than k children. In the following equation, we present a modified definition of the expansion set that implements the restriction.

$$\rho(v) = \begin{cases} \emptyset & \text{if } v = v_0 \\ cl(v) & \text{if } |children(v)| \geq k \\ \bigcap\limits_{v' \in children(v)} \rho(v') \cup \{rlit(v, v')\} \setminus \{\neg rlit(v, v')\} & \text{otherwise.} \end{cases}$$

If a node v has more than or equal to k children then the above equation under-approximates the expansion set by equating it to $cl(v)$. This modification may decrease the cost of computation of the expansion sets but may also lead to fewer redundancy detection.

In figure 5, we also present our third algorithm k-RMPivots using the above definition of expansion set. k-RMPivots is a modified version of all-RMPivots. At line 6, this algorithm introduces an if-condition that checks if the node v has more than or equal to k children. If the condition holds than the algorithm inexpensively computes $\rho(v)$. Otherwise, this algorithm operates as all-RMPivots.

6 Complexity

RmPivots* visits each node of the input resolution proof only once. The worst case cost of visiting each node for RmPivots* is $O(log(|Lit(\mathcal{P})|))$ because of the find and insert operations on D. Therefore, the complexity of RmPivots* is $O(|V|log(|Lit(\mathcal{P})|))$.

all-RMPivots, and k-RMPivots also visit each node of the input resolution proof only once. For each visited node, all-RMPivots iterates over children and applies the intersection operation. Since total number of edges in a resolution proof are less than twice the number of the nodes in the proof, the total number of intersection operation is linearly bounded. In worst case, each intersection may cost as much as the total number of literals in the resolution proofs.

Algorithms	avg. % reduction in proof size	avg. % reduction in unsat core size	time(s)
RmPivots	11.97	0.98	1753
RmPivots*	12.86	1.10	1772
k-RMPivots with $k = 5$	19.60	2.13	2284
k-RMPivots with $k = 10$	21.14	2.41	2599
k-RMPivots with $k = 20$	21.90	2.67	2855
all-RMPivots	22.54	2.93	5000

Fig. 7. We applied our algorithms to the resolution proofs obtained by OpenSMT for 198 examples from plain MUS track of SAT11 competition. The proofs in total contain 144,981,587 nodes.

Algorithms	avg. % reduction in proof size	avg. % reduction in unsat core size	time(s)
RmPivots	6.35	1.52	17.2
RmPivots*	6.69	1.69	18.0
k-RMPivots with $k = 5$	10.10	2.76	24.1
k-RMPivots with $k = 10$	10.43	2.88	27.8
k-RMPivots with $k = 20$	10.54	2.90	31.7
all-RMPivots	10.58	2.91	51.3

Fig. 8. We also applied our algorithms to the resolution proofs obtained by OpenSMT for 132 examples from SMTLIB. The proofs in total contain 13,629,927 nodes.

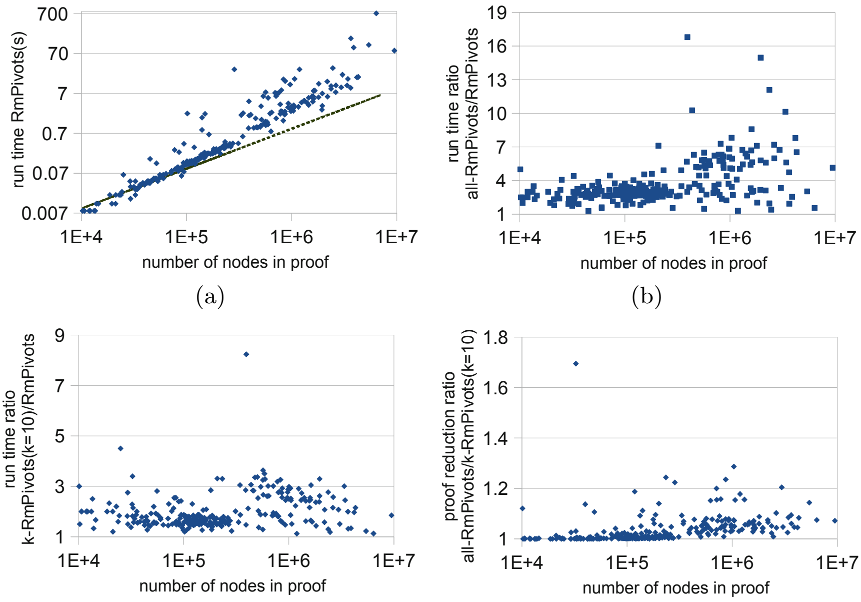
Therefore, The worst case average cost of visiting each node for all-RMPivots is $O(|Lit(\mathcal{P})|)$. k-RMPivots has worst case complexity as all-RMPivots. The complexities of all-RMPivots and k-RMPivots are $O(|V||Lit(\mathcal{P})|)$.

Since the number of literals are usually small compare to the number of nodes, we observe in experiments that although the intersections are expensive but the run times do not grow quadratically as number of nodes increases.

7 Experiments

We implemented our algorithms within an open source SMT solver OpenSMT [5]. We applied our algorithms to the resolution proofs obtained by OpenSMT for 198 unsatisfiable examples from plain MUS track of SAT11 competition. We selected an example if a proof is obtained with in 100 seconds using default options of OpenSMT and has resolution proof larger than 100 nodes.[1] These examples in total contain 144,981,587 nodes in the resolution proofs. The largest proof contains 9,489,571 nodes. In figure 7, we present the results of the experiments. We applied k-RMPivots with three values of k: 5, 10, and 20. The original algorithm RmPivots removes 11.97% of nodes in the proofs of the examples. RmPivots* and all-RMPivots additionally removes 0.89% and 10.57% of nodes

[1] Please find detailed results at http://www.ist.ac.at/~agupta/sat12/index.html



(a) (b)

(c) (d)

Fig. 9. (a) Run times of RmPivots vs. number of nodes in the proofs. The dotted line in the plot denotes a linear growth. (b) Ratio of the run times of all-RMPivots and RmPivots vs. number of nodes in the proofs. (c) Ratio of the run times of k-RMPivots with $k = 10$ and RmPivots vs. number of nodes in the proofs. (d) Ratio of reduction in the proofs by all-RMPivots and k-RMPivots with $k = 10$ vs. number of nodes in the proofs. For these plots, we only used the examples from the earlier benchmarks that have more than 10000 nodes.

respectively. Even for small values of k, k-RMPivots reduces the proofs about as much as all-RMPivots, but within significantly less run times. We observe the similar pattern in reduction of the unsat cores.

The run time of RmPivots* is almost equal to RmPivots as expected. Due to the costly computations of the intersections of sets of literals, all-RMPivots shows significantly increased run time as compared to RmPivots. k-RMPivots provides a parameter that allows one to achieve the proof reduction almost as much as all-RMPivots and within the run times almost as less as RmPivots*.

We also selected 132 unsatisfiable examples from smtlib benchmarks in the theory of QF_UF to test the performance of our algorithms in another setting.[1] These examples in total produce 13,629,927 nodes in the resolution proofs. The largest proof in the examples contains 275,786 nodes. In figure 8, we present the results of applying our three algorithms to the examples. We observe the similar pattern in the results as observed in previous example set.

We note that the % of proof reduction may vary a lot for individual examples (from 0% to 48%).[1] We also observe that the two example sets have different absolute % reduction in proof sizes.

In figure 9, we plotted the relative performances of RmPivots, all-RMPivots, and k-RMPivots with $k = 10$ for the individual examples with proof sizes greater than 10000 nodes. In figure 9(a), we plot the run times of RmPivots verses the proof sizes. The dotted line in the plot denotes a linear growth. We observe that the run times grow non-linearly but for the most examples the run times are close to the linear line. In figure 9(b), we observe that the ratios of the run times of all-RMPivots and RmPivots have increasing trend with the increasing proof sizes, which follows from the difference in their complexities. In figure 9(c), we observe that the ratios of run times of k-RMPivots with $k = 10$ and RmPivots are fairly constant across the different proof sizes, which is the result of the heuristic. In figure 9(d), we observe that the ratios of the reductions in the proofs by all-RMPivots and k-RMPivots with $k = 10$ remain below 1.1 for most of the examples.

8 Conclusion

We presented three *new* single pass algorithms that can find redundant resolutions even in the resolution proofs that have DAG form, without causing significantly large run times. Since these algorithms do not try to escape a local minimum, the improvements in reductions due to these algorithms are limited. For an analogy, we can compare these algorithms with compiler optimizations in which an efficient and less compressing algorithm is always welcomed as compare to an expensive and more compressing algorithm.

These algorithms can be further harnessed by placing them in the iterative algorithm of [18]. We leave that for future work. We specially note that the additional rules (other than A1 and A2) for restructuring a resolution proof presented in [18] become redundant if applied in combination with our algorithms.

References

1. Amjad, H.: Compressing propositional refutations. Electr. Notes Theor. Comput. Sci. 185 (2007)
2. Amjad, H.: Data compression for proof replay. J. Autom. Reasoning 41(3-4) (2008)
3. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-Time Reductions of Resolution Proofs. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 114–128. Springer, Heidelberg (2009)
4. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
6. Dershowitz, N., Hanna, Z., Nadel, A.: A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
7. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant Strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)

8. Gershman, R., Koifman, M., Strichman, O.: Deriving Small Unsatisfiable Cores with Dominators. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 109–122. Springer, Heidelberg (2006)
9. Grégoire, É., Mazure, B., Piette, C.: Local-search extraction of muses. Constraints 12(3), 325–344 (2007)
10. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
11. Huang, J.: Mup: a minimal unsatisfiability prover. In: ASP-DAC, pp. 432–437 (2005)
12. Lynce, I., Silva, J.P.M.: On computing minimum unsatisfiable cores. In: SAT (2004)
13. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
14. 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)
15. Mneimneh, M., Lynce, I., Andraus, Z.S., Marques-Silva, J., Sakallah, K.A.: A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 467–474. Springer, Heidelberg (2005)
16. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: Amuse: a minimally-unsatisfiable subformula extractor. In: DAC, pp. 518–523 (2004)
17. Papadimitriou, C.H., Wolfe, D.: The complexity of facets resolved. J. Comput. Syst. Sci. 37(1), 2–13 (1988)
18. Rollini, S., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Conference (2010)Haifa Verification
19. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of Satisfiability, pp. 131–153 (2009)
20. Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
21. Sinz, C.: Compressing Propositional Proofs by Common Subproof Extraction. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 547–555. Springer, Heidelberg (2007)
22. Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data. AI EDAM 17(1), 75–97 (2003)
23. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiabl boolean formulas. In: SAT (2003)

A Proof of Correctness and Complexity

In this section, we will present proof of correctness of our modifications. We need to define some notation first. Let (V, L, R, cl, piv, v_0) be a proof DAG. Let ρ be defined by the recursive equation 1. Let $cl^{\rho}(v) = cl(v) \cup \rho(v)$. Let $Res^{*}(C, D, a) = (C \setminus \{a\}) \vee (D \setminus \{\neg a\})$.

In the following definition, we will define an invariant for the input of RestoreResTree. We will show that RmPivots* and all-RMPivots transforms the input resolution proof into a proof DAG that satisfies the invariant.

Definition 2 (Restorable property). *The proof DAG (V, L, R, cl, piv, v_0) is restorable if each internal node $v \in V$ satisfies the following conditions.*

(1) $L(v) \neq \bot \vee R(v) \neq \bot$
(2) *if* $L(v) = \bot \vee R(v) = \bot$ *then*
(3) $$cl^{\rho}(v) \supseteq \begin{cases} cl^{\rho}(L(v)) & \textit{if } R(v) = \bot \\ cl^{\rho}(R(v)) & \textit{if } L(v) = \bot \\ Res^{*}(cl^{\rho}(L(v)), cl^{\rho}(R(v)), piv(v)) & \textit{otherwise} \end{cases}$$

Theorem 1. *If a proof DAG is restartable then* RestoreResTree *transforms the proof DAG into a resolution proof.*

We will not provide a proof for the above theorem. Please look in [3] for the proof. To prove the correctness, we will prove that during the runs of both the algorithms maintain the following invariant.

Definition 3 (Invariant). *A proof DAG* (V, L, R, cl, piv, v_0) *satisfies this invariant if each reachable from sink and internal node* $v \in V$ *satisfies the following conditions.*

(1) $L(v) \neq \bot \vee R(v) \neq \bot$
(2) *if* $R(v) \neq \bot \wedge L(v) \neq \bot$ *then* $cl(v) = Res(cl(L(v)), cl(R(v)), piv(v))$
(3) *if* $R(v) = \bot$ *then* $cl(v) \uplus \{piv(v)\} \supseteq cl(R(v))$ *and* $piv(v) \in \rho(v)$
(4) *if* $L(v) = \bot$ *then* $cl(v) \uplus \{\neg piv(v)\} \supseteq cl(L(v))$ *and* $\neg piv(v) \in \rho(v)$

where, $\uplus$ *is disjoint union.*

It can be easily checked that the invariant is a stronger property than then the restorable property. Since both RmPivots* and all-RMPivots only remove edges from a proof DAG, conditions 1 and 2 of the invariant will be true trivially. Further, first half of 3 and 4 are also true trivially. To show validity of the rest of the conditions that we need to prove the following lemma.

Lemma 1. *If the proof DAG satisfies the invariant then for each* $v \in V$, $cl(v) \subseteq \rho(v)$.

Proof. We prove it by induction over the height of the proof DAG starting from sink node. The base case at sink node is trivially true. By induction hypothesis, for a node $v \in V$, lets assume for each $v' \in children(v)$, $cl(v') \subseteq \rho(v')$. Let $v = L(v')$ and the proof for the other case is similar. Due to condition (2) and (3) of the invariant, $cl(v) \subseteq \rho(v') \cup \{piv(v')\}$. Due to definition of ρ, we can derive $cl(v) \subseteq \rho(v)$. □

Theorem 2. RmPivots* *maintains the invariant.*

Proof. The theorem is due to the previous lemma and the correctness proof of Recycle-Pivots from [3]. □

Theorem 3. all-RMPivots *maintains the invariant.*

Proof. all-RMPivots traverses the proof DAG in reverse topological order. Therefore, a node is visited only when all the ancestors of the node has been visited. The computed value of ρ for the node will not be changed due to future changes since all the future change will not happen with in the ancestors of the node. Hence by construction, the second half of the third and fourth conditions will be satisfied. □

The above theorems are sufficient to prove the correctness of our algorithms.

Model Checking Systems and Specifications with Parameterized Atomic Propositions

Orna Grumberg[1], Orna Kupferman[2], and Sarai Sheinvald[2]

[1] Department of Computer Science, The Technion, Haifa 32000, Israel
[2] School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel

Abstract. In classical LTL model checking, both the system and the specification are over a finite set of atomic propositions. We present a natural extension of this model, in which the atomic propositions are parameterized by variables ranging over some (possibly infinite) domain. For example, by parameterizing the atomic propositions $send$ and $receive$ by a variable x ranging over possible messages, the specification $\mathsf{G}(send.x \rightarrow \mathsf{F}receive.x)$ specifies that not only each send signal is followed by a receive signal, but also that the content of the received message agrees with the content of the one sent.

Our extended setting consists of *Variable LTL* (VLTL) – a specification formalism that extends LTL with atomic propositions parameterized by variables, and *abstract systems* – systems in which atomic propositions may be parameterized by variables. We study the model-checking problem in this setting. We show that while the general setting is undecidable, some useful special cases are decidable. In particular, for fragments of VLTL that restrict the quantification over the variables, the model checking is PSPACE-complete, and thus is not harder than the LTL model checking problem. The latter result conveys the strength and advantage of our setting.

1 Introduction

In model checking, we verify that a system has a desired behavior by checking that a mathematical model of the system satisfies a formal specification of the desired behavior. Traditionally, the system is modeled by a Kripke structure – a finite-state system whose states are labeled by a finite set of atomic propositions. The specification is a temporal-logic formula over the same set of atomic propositions [4].

The complexity of model checking depends on the sizes of the system and the specification [15,12]. One source for these sizes being huge is a large or even infinite data domain over which the system, and often also the specification, need to be defined. Another source for the large sizes are systems composed of many components, such as underlying processes or communication channels, whose number may not be known in advance. In both cases, desired specifications might be inexpressible and model checking intractable.

In this work we propose a novel approach for model checking systems and specifications that suffer from the size problem described above. We do so by extending both the specification formalism and the system model with atomic propositions that are parameterized by variables ranging over some (possibly infinite) domain.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 122–136, 2012.

Let us start with a short description of the specification formalism. We introduce and study the linear temporal logic *Variable LTL* (VLTL, for short). VLTL has the syntax of LTL except that atomic propositions may be parameterized with variables over some finite or infinite domain. The variables can be quantified and assignments to them can be constrained by a set of inequalities. VLTL formulas are interpreted with respect to all assignments to the variables that respect the inequalities. For example, the VLTL formula $\psi = \forall x.\mathsf{G}(\mathit{send}.x \rightarrow \mathsf{F}\mathit{receive}.x)$ states that whenever a message with content d, taken from the domain, is sent, then a message with content d is eventually received. Note that if the domain of messages is infinite or unknown in advance, then ψ does not have an equivalent LTL formula.

In order to see the usefulness of VLTL, consider the LTL specification $\mathsf{G}(\mathit{req} \rightarrow \mathsf{F}\mathit{grant})$, stating that every request is eventually granted. We may wish to parameterize the req and grant by atomic propositions with the id of the process that invokes the request. In LTL this can only be done by having a new set of atomic propositions $\mathit{req}_1, \mathit{grant}_1, \ldots, \mathit{req}_n, \mathit{grant}_n$, where n is the number of processes. This both blows-up the specification and requires the specifier to know the number of processes in advance. Instead, in VLTL we parameterize the atomic propositions by a variable x ranging over the ids of the processes. Since it is natural to require that the property holds for *every* assignment to x, we quantify it universally, thus obtaining the formula $\psi = \forall x; \mathsf{G}(\mathit{req}.x \rightarrow \mathsf{F}\mathit{grant}.x)$.[1] Note that the negation of ψ quantifies x existentially, thus $\neg\psi = \exists x; \mathsf{F}(\mathit{req}.x \wedge \mathsf{G}\neg\mathit{grant}.x)$. Beyond the use of existential quantification for identifying counterexamples as above, such quantification is useful by itself. For example, the formula $\exists x.\mathsf{GF}\neg\mathit{idle}.x$ states that in each computation, there exists at least one process that is not idle infinitely often.

Next, consider the formula $\theta = \forall x_1; \forall x_2; \mathsf{G}((\neg\mathit{send}.x_2)\,\mathsf{U}\mathit{send}.x_1) \rightarrow ((\neg\mathit{rec}.x_2)\,\mathsf{U}\mathit{rec}.x_1)$, stating that messages are received in the order in which they are sent. To convey this, the values assigned to x_1 and x_2 should be different. This is handled by the set of inequalities that VLTL formulas include. When interpreting a formula, we consider only assignments that respect the set of inequalities. In the example above, the VLTL formula $\langle\theta, x_1 \neq x_2\rangle$ specifies that θ holds for every assignment in which the variables x_1 and x_2 are assigned different values.

As another example, consider a system with transactions interleaved in a single computation [17]. Each transaction has an id, yet the range of ids is not known in advance. We may wish to refer to the sequence of events associated with one transaction. For instance, when x stands for a transaction id, then $\forall x; \mathsf{G}(\mathit{req}.x \rightarrow \mathsf{F}\mathit{grant}.x)$ states that whenever a request is raised in a transaction, it is eventually granted in the same transaction. Alternatively, we may wish to specify that a request is granted only in a different transaction. In this case we may write $\langle\varphi, x_1 \neq x_2\rangle$, where $\varphi = \forall x_1; \forall x_2; \mathsf{G}(\mathit{req}.x_1 \rightarrow ((\neg\mathit{grant}.x_1)\,\mathsf{U}\mathit{grant}.x_2))$.

Thus, as demonstrated above, VLTL formulas are able to compactly express specifications over a large, possibly infinite domain, which would otherwise be inexpressible by LTL or lead to extremely large formulas. Moreover, VLTL is able to express

[1] Note that unlike the standard way of augmenting temporal logic with quantification [13,16], here the variables do not range over the set of atomic propositions but rather over the domain of their parameters.

properties for domains whose size is unknown in advance (e.g., number of different messages, number of different channels).

We now turn to describe in detail the way we model systems. We distinguish between *concrete* models, in which atomic propositions are parameterized with values from the domain, and *abstract* models, in which the atomic propositions are parameterized with variables. For instance, in a concrete model, the proposition $send.3$ stands for the atomic proposition $send$ that carries the value 3. In an abstract model, the proposition $send.x$ indicates that $send$ carries the value assigned to x. Assignments to variables are fixed along a computation, except when the system resets their value. More precisely, in every transition of the system, a subset of the variables is reset, meaning that these variables can change their value in the next step of the run[2]. Also, as with VLTL, the model includes a set of inequalities over the variables.

The concrete computations of an abstract system are obtained from system paths by assigning values to the occurrences of the variables in a way that respects the set of inequalities, and is consistent with the resetting along the path. That is, the value assigned to a variable may change only when a transition in which the variable is reset is taken. Note that a path of an abstract system may induce infinitely many different concrete computations of the abstract system, formed by different assignments to the occurrences of the variables. Thus, abstract systems offer a compact and simple representation of infinite-state systems in which the control is finite and the source of infinity is data that may be infinite or unknown in advance.

As a simple example, consider the abstract system in Figure 1. It describes a simple stop-and-wait communication protocol. Once a message x is sent, the system waits for an ack with identical content, confirming the arrival of the message. When this happens, the content of the message is reset and a new cycle starts. If a timeout occurs, the same message is sent. Note that if the message domain is infinite, standard finite systems cannot describe this protocol, as it involves a comparison of the content of the message in different steps of the protocol. More complex communication protocols, in which several send-receive processes can run simulateously (e.g., sliding windows), can be modeled by an abstract system with several variables, one for every process.

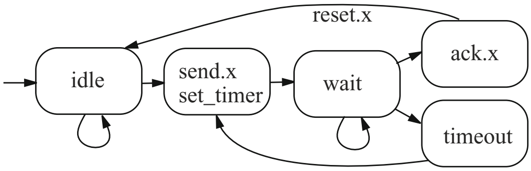


Fig. 1. A simple abstract system for the stop and wait protocol

We study the model-checking problem for VLTL with respect to concrete and abstract systems. We also consider two natural fragments of VLTL: ∀-VLTL and ∃-VLTL, containing only ∀ and ∃ quantifiers, respectively.

[2] Despite some similarity in their general description and use of resets, abstract systems have no direct relation to timed automata [22]. In abstract systems, the variables and resets are over domain values rather than clocks ranging over the reals.

We start with VLTL model checking of concrete systems. We show that given a concrete system and a VLTL formula, it is possible to reduce the infinite domain to a finite one, and reduce the problem to LTL model checking. The reduction involves an LTL formula that is exponential in the VLTL formula, and we prove that the problem is indeed EXPSPACE-complete. Hardness in EXPSPACE applies already for the fragment of ∃-VLTL. As good news, for the fragment of ∀-VLTL, a similar procedure runs in PSPACE, and the problem is PSPACE-complete in the size of the formula, as for LTL. We also consider the model-checking for a single concrete computation and show that it is PSPACE-complete.

We proceed to show that for abstract systems the model-checking problem is in general undecidable. Here too, undecidability applies already for the fragment of ∃-VLTL. Our undecidability proof is by a reduction from Post's Correspondence Problem, following the lines of the reduction in [14], showing that the universality problem for register automata is undecidable. On the other hand, for systems with no resetting, model checking is EXPSPACE-complete, and thus is not harder than model checking of concrete systems. Moreover, for the fragment of ∀-VLTL, the model-checking problem for abstract systems is PSPACE-complete, and thus is not harder than the LTL model-checking problem. This latter result conveys the strength and advantage of our model: abstract systems are able to accurately describe infinite state systems; ∀-VLTL formulas are able to express rich behaviors that are inexpressible by LTL, and may refer to infinitely many values. Yet, surprisingly, model checking in that setting is not harder than that of LTL.

Related Work. Researchers have studied several classes of infinite-state systems, generally differing in the source of infinity. For systems with a finite control and infinite (or very large) data, researchers have developed heuristics such as data abstraction [3,6]. Data abstractions are applied in order to construct a finite model and a simplified specification, and are based on mapping the large data domain into a finite and small number of classes. Our approach, on the other hand, has the abstraction built in the system, and is also part of the specification formalism. Finite control is used also in work on hybrid systems [10] and systems with an unbounded memory (e.g., pushdown systems [7]), where the underlying setting is very different from the one studied here. Closer to our abstract systems are some of the approaches to handle systems having an unbounded number of components (e.g., parameterized systems [8]). The solutions in these cases are tailored for the setting in which the parameter is the number of components, and cannot be applied, say, for parameterizing data.

Several extensions of LTL for handling infinite-state or parameterized systems have been suggested and studied. A general such extension, with a setting similar to that presented in this paper, is *first-order LTL*. First-order temporal logics are used for specifying and reasoning about temporal databases [19]. The work focuses on finding decidable fragments of the logic, its expressive power and axiomatization (c.f., [21]).

In [2,8], the authors studied an extension of temporal logic in which atomic propositions are parameterized with a variable that ranges over the set of processes ids. In [20], the authors study a restricted fragment of first-order temporal logic that is suitable for verifying parameterized systems. Again, these works are tailored for the setting of parameterized systems, and are also restricted to systems in which (almost) all components are identical. VLTL is much richer, already in the context of parameterized systems. For example, it can refer to both sender id and message content. In [5], the authors introduced Constraint LTL, in which atomic propositions may be constraints like $x < y$, and formulas are interpreted over sequences of assignments of variables to values in $\mathbb{N}$ or $\mathbb{Z}$. Thus, like our approach, Constraint LTL enables the specification to capture all assignments to the variable. Unlike our approach, the domain is restricted to numerical values. In [11], the author extends LTL with the freeze quantifier, which is used for storing values from an infinite domain in a register. As discussed in our earlier work [9], the variable-based formalism is cleaner than the register-based one, and it naturally extends traditional LTL. In addition, our extension allows variables in both the system and the specification. Finally, unlike existing work, as well as work on first order LTL, in our setting a parameterized atomic proposition may hold with several different parameter values in the same point in time.

In [9], we introduced VNFAs – nondeterministic finite automata augmented by variables. VNFAs can recognize languages over infinite alphabets. As in VLTL, the idea is simple and is based on labeling some of the transitions of the automaton by variables that range over some infinite domain. In [1], a similar use of variables was studied for finite alphabets, allowing a compact representation of regular expressions. There, the authors considered two semantics; in the "possibility" semantics, the language of an expression over variables is the union of all regular languages obtained by assignments to the variables, and in the "certainty" semantics, the language is the intersection of all such languages. Note that in VLTL, it is possible to mix universal and existential quantification of variables.

2 VLTL: LTL with Variables

Linear temporal logic (LTL) is a formalism for specifying on-going behaviors of reactive systems. We model a finite-state system by a *Kripke structure* $\mathcal{K} = \langle P, S, I, R, L \rangle$, where P is a set of atomic propositions, S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a total transitions relation, and $L : S \to 2^P$ is a labeling function. We then say that $\mathcal{K}$ *is over* P. A *path* in $\mathcal{K}$ is an infinite sequence $s_0, s_1, s_2 \ldots$ of states such that $s_0 \in I$ and $\langle s_i, s_{i+1} \rangle \in R$ for every $i \geq 0$. A *computation* of $\mathcal{K}$ is an infinite word $\pi = \pi_0 \pi_1 \ldots$ over 2^P such that there exists a path $s_0, s_1, s_2 \ldots$ in $\mathcal{K}$ with $\pi_i = L(s_i)$ for all $i \geq 0$. We denote by π^i the suffix $\pi_i \pi_{i+1}, \ldots$ of π.

We assume that the reader is familiar with the syntax and semantics of LTL. For a Kripke structure $\mathcal{K}$ and an LTL formula φ, we say that $\mathcal{K}$ satisfies φ, denoted $\mathcal{K} \models \varphi$, if for every computation π of $\mathcal{K}$, it holds that $\pi \models \varphi$. The *model-checking problem* is to decide, given $\mathcal{K}$ and φ, whether $\mathcal{K} \models \varphi$.

The logic VLTL extends LTL by allowing some of the atomic propositions to be parameterized by variables that can take values from a finite or infinite domain. Consider a finite set P of atomic propositions, a finite or infinite domain $\mathcal{D}$, a finite set T of *parameterized atomic propositions*, and a finite set X of *variables*. In VLTL, the propositions in T are parameterized by variables in X that range over $\mathcal{D}$. A VLTL formula also contains guards, in the form of inequalities over X, which restrict the set of possible assignments to the variables.

We now define VLTL formally. An *unrestricted VLTL formula* over P, T, and X is a formula ψ of the form $Q_1x_1; Q_2x_2; \cdots Q_kx_k; \theta$, where $Q_i \in \{\exists, \forall\}$, $x_i \in X$, and θ is an LTL formula over $P \cup T \cup (T \times X)$. Variables that appear in θ and are not quantified are *free*. If θ has no free variables, then ψ is *closed*. We say that a parameterized atomic proposition $a \in T$ is *primitive in* θ if there is an occurrence of a in θ that is not parameterized by a variable. An *inequality set over* X is a set $E \subseteq \{x_i \neq x_j \mid x_i, x_j \in X\}$.

A *VLTL formula* over P, T, and X is a pair $\varphi = \langle \psi, E \rangle$ where ψ is an unrestricted VLTL formula over P, T, and X, and E is an inequality set over X. The notions of closed formulas, and of a primitive parametric atomic proposition are lifted from ψ to φ. That is, φ is closed if ψ is closed, and a is primitive in φ if it is primitive in ψ.

We consider two fragments of VLTL. The logic $\exists$-VLTL is the fragment of VLTL in which $Q_i = \exists$ for all $1 \leq i \leq k$. Dually, $\forall$-VLTL is the fragment of VLTL in which only the $\forall$ quantifier is allowed.

We define the semantics for VLTL with respect to both concrete computations – infinite words over the alphabet $2^{P \cup (T \times \mathcal{D})}$, and abstract computations, defined later in this section. A position in a concrete computation is a letter $\sigma \in 2^{P \cup (T \times \mathcal{D})}$. For $a \in T$, we say that σ satisfies a, denoted $\sigma \models a$, if there exists $d \in \mathcal{D}$ such that $a.d \in \sigma$. Thus, a primitive $a \in T$ is satisfied by assignments in which a holds with at least one value.

We say that a partial function $f : X \to \mathcal{D}$ *respects* E if for every $x_i \neq x_j$ in E, it holds that $f(x_i) \neq f(x_j)$.

Consider a concrete computation π, a VLTL formula $\varphi = \langle \psi, E \rangle$, with $\psi = Q_1x_1; Q_2x_2; \ldots; Q_nx_n; \theta$, and a partial function $f : X \to \mathcal{D}$ that assigns values to all the free variables in ψ and respects E. We use $\pi \models_f \psi$ to denote that π satisfies ψ under the function f. The satisfaction relation is defined as follows.

- If $n = 0$ (that is, $\psi = \theta$ has no quantification and all its variables are free), then the formula ψ_f obtained from ψ by replacing, for every x, every occurrence of $a.x$ with $a.f(x)$, is an LTL formula over $P \cup T \cup (T \times \mathcal{D})$. Then, $\pi \models_f \varphi$ iff f respects E and $\pi \models \psi_f$.
- If $Q_1 = \exists$ (that is, $\psi = \exists x_1; Q_2x_2; \ldots; Q_nx_n; \theta$), then $\pi \models_f \varphi$ iff there exists $d \in \mathcal{D}$ such that $f[x_1 \leftarrow d]$ respects E and $\pi \models_{f[x_1 \leftarrow d]} Q_2x_2; \ldots; Q_nx_n; \theta$.
- If $Q_1 = \forall$ (that is, $\psi = \forall x_1; Q_2x_2; \ldots; Q_nx_n; \theta$), then $\pi \models_f \varphi$ iff for all $d \in \mathcal{D}$ such that $f[x_1 \leftarrow d]$ respects E, we have that $\pi \models_{f[x_1 \leftarrow d]} Q_2x_2; \ldots; Q_nx_n; \theta$.

Example 1. Consider the concrete computation

$$\pi = \{send.1\}\{send.2\}\{rec.2\}\{rec.1\}^\omega$$

and the VLTL formula $\langle \exists x; \theta, \emptyset \rangle$, where $\theta = \mathsf{G}(send.x \to \mathsf{X}rec.x)$. Then for the function $f(x) = 2$, we have that $\theta_f = \mathsf{G}(send.2 \to \mathsf{X}rec.2)$, and since $\pi \models \theta_f$, it holds that $\pi \models \langle \exists x; \theta, \emptyset \rangle$.

Next, consider the VLTL formula $\langle \forall x; \theta, \emptyset \rangle$. Then for the function $g(x) = 1$, we have that $\pi \not\models \theta_g$, and therefore $\pi \not\models \langle \forall x; \theta, \emptyset \rangle$.

Similarly to first order logic, when the formula ψ is closed, the satisfaction of E does not depend on the function f, and we use the notation $\models$.

The extension of the definition to concrete systems (rather than computations) is similar to that of LTL: For a Kripke structure $\mathcal{K}$ over $P \cup (T \times \mathcal{D})$ and a closed VLTL formula $\varphi = \langle \psi, E \rangle$, we say that $\mathcal{K}$ satisfies φ, denoted $\mathcal{K} \models \varphi$, if for every computation π of $\mathcal{K}$, it holds that $\pi \models \langle \psi, E \rangle$.

We now define abstract systems, whose computations may contain infinitely many values. An *abstract system* over P, $T \cup \{reset\}$ and X is a pair $\langle \mathcal{K}, E \rangle$, where $\mathcal{K}$ is a Kripke structure over $P \cup (T \times X)$ with a labeling $L' : R \to 2^{\{reset\} \times X}$ for the transitions, and E is an inequality set over X. Intuitively, a system variable is assigned values throughout a computation of the system. If the computation traverses a transition labeled by $reset.x$, then x can be assigned a new value, that will remain unchanged at least until the next traversal of $reset.x$. Also, if $x_i \neq x_j \in E$, then the values that are assigned to x_i and x_j in a given point in the computation must be different.

An *abstract computation* of $\langle \mathcal{K}, E \rangle$ is a pair $\langle \rho, E \rangle$, where ρ is an infinite word $\rho_0 \rho'_0 \rho_1 \rho'_1 \cdots$ over $P \cup ((T \cup \{reset\}) \times X)$ such that there exists a path $s_0, s_1, \ldots$ in $\mathcal{K}$ such that for every $i \geq 0$, it holds that $L(s_i) = \rho_i$, and $L'(\langle s_i, s_{i+1} \rangle) = \rho'_i$.

A concrete computation is obtained from an abstract computation by assigning values from $\mathcal{D}$ to the variables in X as described next.

Consider an abstract computation $\langle \rho, E \rangle$ over P, $T \cup \{reset\}$ and X, where $\rho = \rho_0 \rho'_0 \rho_1 \rho'_1 \cdots$ and a concrete computation $\pi = \pi_0, \pi_1, \ldots$ over P, T and $\mathcal{D}$. We say that π is a *concretization of* $\langle \rho, E \rangle$ if π is obtained from $\rho_0 \rho_1 \rho_2 \cdots$ by substituting every occurrence of $x \in X$ by a value in $\mathcal{D}$, according to the following rules.

- For every two consecutive occurrences of $reset.x$ in ρ'_i and ρ'_j, the values assigned to occurrences of x in $\rho_{i+1}(x), \rho_{i+2}(x) \ldots \rho_j(x)$ are identical.
- For every $x_i \neq x_j \in E$, for every position ρ_k that contains occurrences of x_i and x_j, these occurrences are assigned different values in π_i.

If π is a concretization of $\langle \rho, E \rangle$, then we say that $\langle \rho, E \rangle$ is an *abstraction* of π. Note that $\langle \rho, E \rangle$ may have infinitely many concretizations.

Notice that for a domain $\mathcal{D}$, an abstract system $\langle \mathcal{K}, E \rangle$ represents a (possibly infinite) concrete system over $P \cup (T \times \mathcal{D})$ whose computations are exactly all concretizations (w.r.t. $\mathcal{D}$) of every abstract computation of $\langle \mathcal{K}, E \rangle$.

Example 2. Consider the abstract system $\langle \mathcal{K}, \emptyset \rangle$, where $\mathcal{K}$, appearing in Figure 2, describes a protocol for two processes, a and b, that use a printer that may print a single job at a time. A token is passed around. The atomic propositions ta and tb are valid when processes a and b hold the token, respectively. The parameterized atomic propositions ra, rb, and p are parameterized by x_1 and x_2, which can get values from the range of file names. The proposition $ra.x_1$ is valid when process a requests to print x_1, and similarly for $rb.x_2$ and process b. Once a file is printed, the variable that maintains the file is reset.

Consider the path $s_1 s_2 (s_3 s_4 s_5 s_6)^\omega$ of $\mathcal{K}$. It induces the abstract computation $\langle \rho, \emptyset \rangle$, where $\rho =$

$$\{ta\}\emptyset\{ta, ra.x_1\}\emptyset(\{ta, ra.x_1, rb.x_2\}\emptyset\{p.x_1, rb.x_2\}\{reset.x_1\}$$
$$\{tb, ra.x_1, rb.x_2\}\emptyset\{p.x_2, ra.x_1\}\{reset.x_2\})^\omega$$

An example for a concretization of $\langle \rho, \emptyset \rangle$ is

$$\{ta\}\{ta, ra.doc1\}\{ta, ra.doc1, rb.data.txt\}\{p.doc1, rb.data.txt\}$$
$$\{tb, ra.doc2, rb.data.txt\}\{p.data.txt, ra.doc2\}\{ta, ra.doc2, rb.vltl.pdf\}\ldots$$

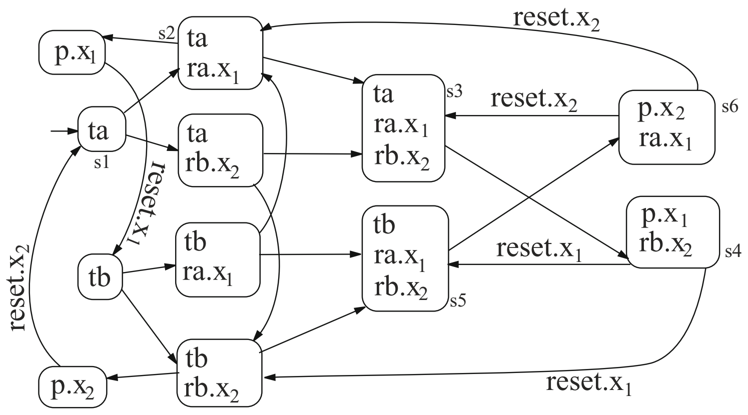


Fig. 2. The abstract system $\mathcal{K}$

We now describe the second semantics of VLTL, for abstract computations. Consider a set X' of variables, an abstract computation $\langle \rho, E' \rangle$ over $P \cup ((T \cup \{reset\}) \times X')$, and a closed VLTL formula $\varphi = \langle \psi, E \rangle$ over $P \cup T \cup (T \times X)$.[3] We say that $\langle \rho, E' \rangle$ satisfies φ, denoted $\langle \rho, E' \rangle \models \varphi$, if for every concretization π of $\langle \rho, E' \rangle$, it holds that $\pi \models \varphi$. Note that ρ and ψ are defined over different sets of variables, that are not related to each other.

Example 3. Consider the abstract system $\langle \mathcal{K}, \emptyset \rangle$ and the abstract computation $\langle \rho, \emptyset \rangle$ of Example 2.

Consider the formula $\varphi = \langle \forall z_1; \forall z_2; \mathsf{G}((ra.z_1 \wedge ta) \rightarrow ((\neg p.z_2) \mathsf{U} p.z_1)), \{z_1 \neq z_2\} \rangle$ over P, T, and $X = \{z_1, z_2\}$. In every concretization π of $\langle \rho, \emptyset \rangle$, whenever process a requests to print a document d and holds the token, then the next job that is printed is d, and no other job d' gets printed before that. This holds for all values d and d' such that $d \neq d'$, and therefore, $\langle \rho, \emptyset \rangle \models \varphi$.

For an abstract system $\langle \mathcal{K}, E' \rangle$ and a closed VLTL formula $\langle \psi, E \rangle$, we say that $\langle \mathcal{K}, E' \rangle$ satisfies $\langle \psi, E \rangle$, denoted $\langle \mathcal{K}, E' \rangle \models \langle \psi, E \rangle$, if for every abstract computation $\langle \rho, E' \rangle$ of $\mathcal{K}$, it holds that $\langle \rho, E' \rangle \models \langle \psi, E \rangle$.

3 Model Checking of Concrete Systems

In this section we present a model-checking algorithm for finite concrete systems and discuss the complexity of the model-checking problem for the fragments of VLTL. We show that the general case is EXPSPACE-complete, but is PSPACE-complete for the fragment of $\forall$-VLTL and for single computations. Thus, the problem for these latter cases is not more complex than LTL model checking.

[3] An abstract computation represents infinitely many concrete computations. For every such computation, a different function may be needed in order to satisfy the formula. Therefore, the definition does not involve a specific function from the variables to the values, and so only closed formulas are considered.

The model-checking procedure we present for concrete systems reduces the model-checking problem for VLTL to the model-checking problem for LTL. The key to this procedure is the observation that different values that do not appear in a given computation behave similarly when assigned to a formula variable. Thus, it is sufficient to consider the finite set of values that do appear in the concrete system $\mathcal{K}$, plus one additional value for every quantifier to represent the values that do not appear in $\mathcal{K}$. This means that in case of a very large, or even infinite, domain, we can check the set of assignments over a finite domain instead, resulting in a finite procedure.

We say that a concrete computation π and a VLTL formula φ *do not distinguish between two values* $d_1, d_2 \in \mathcal{D}$ *with respect to the variable* x *and a partial function* $f : X \to \mathcal{D}$ if $\pi \models_{f[x \leftarrow d_1]} \varphi$ iff $\pi \models_{f[x \leftarrow d_2]} \varphi$.

Lemma 1. *Let π be a concrete computation and let φ be a VLTL formula over P, T and X. Then, for every $x \in X$, every function $f : X \to \mathcal{D}$, and every two values d_1 and d_2 that do not appear in π, it holds that π and φ do not distinguish between d_1 and d_2 with respect to x and f.*

We now use Lemma 1 in order to reduce the VLTL model-checking problem for concrete systems to the LTL model-checking problem. We describe two algorithms. The first algorithm, ModelCheck, gets as input a single computation and a VLTL formula and decides whether the computation satisfies the formula. The second is based on a transformation of a given VLTL formula to an LTL formula such that the system satisfies the VLTL formula iff it satisfies the LTL formula. The idea behind both algorithms is the same – an inductive valuation of the formula, either (the first algorithm) by recursively trying possible assignments to the variables, or (the second algorithm) by encoding the various possible assignments using conjunctions and disjunctions in LTL. In both cases, Lemma 1 implies that the number of calls needed in the recursion or the number of conjuncts or disjuncts in the translation is finite.

Let us describe the first algorithm in more detail. Consider a computation π in the concrete system. Recall that such a system is a finite state system, and therefore π contains finitely many values from $\mathcal{D}$. Let A be the set of values that appear in π. Consider a VLTL formula $\varphi = \langle \psi, E \rangle$ and a partial function $f : X \to \mathcal{D}$ that respects E. Let $B = Image(f)$.

Intuitively, each recursive call of the algorithm evaluates φ with a different assignment to the variables. Lemma 1 enables checking assignments over a finite set of values instead of the entire domain. For every quantifier, a new value is added to this set, initially assigned $A \cup B$. According to Lemma 1, a value that is not in π and is different from every other value that has been added to the set can represent every other such value. Hence, these values are enough to cover the entire domain.[4] For the $\forall$ quantifier, we require that every respecting assignment leads to satisfaction. For the $\exists$ quantifier, we require that at least one respecting assignment leads to satisfaction.

Since different computations of a concrete system may satisfy the formula with different assignments to the same variable, ModelCheck, which checks the entire system against a single assignment, cannot be applied for checking concrete systems instead

[4] Note that we assume that $\mathcal{D}$ is sufficiently large to supply additional new values whenever the algorithm needs them. Our algorithms can be modified to account also for a small $\mathcal{D}$.

of computations. It can, however, be used to model check single paths in PSPACE. We assume, as usual for such a case, that this path is a lasso. Since we can easily reduce the problem of TQBF to model checking of a single path, a matching lower bound follows.

Theorem 1. *Let π be a lasso-shaped concrete computation, let $\varphi = \langle\psi, E\rangle$ be a VLTL formula over $P \cup T \cup (T \times X)$, and let $f : X \to \mathcal{D}$ be a partial function that respects E. Then deciding whether $\pi \models_f \langle\psi, E\rangle$ is PSPACE-complete.*

Proof: For the upper bound, using Lemma 1 we can show that for $B = Image(f)$ and for A, the finite set of values that occur in π, it holds that ModelCheck$(\pi, \langle\psi, E\rangle, f, A \cup B)$ returns *true* iff $\pi \models_f \langle\psi, E\rangle$. This procedure involves repeated runs of LTL model checking for π and a formula of the same size as ψ. Since each such run can be performed in PSPACE (in fact, in polynomial time), the entire procedure is run in PSPACE.

The lower bound is shown by a reduction from TQBF, the problem of deciding whether a closed quantified Boolean formula is valid, which is known to be PSPACE-complete. Given a quantified Boolean formula ψ, consider the single-state system $\mathcal{K}$ labeled $a.d$, where a is a parameterized atomic proposition, and d is some value, and the VLTL formula $\langle\psi', \emptyset\rangle$ obtained form ψ by replacing every variable x in ψ with $a.x$. Then, every truth assignment f to the variables of ψ is mapped to an assignment f' to the variables in ψ', where assigning *true* to x in f is equivalent to assigning d to x in f', and assigning *false* to x in f is equivalent to assigning $d' \neq d$ to x in f'. It can be shown that $f \models \psi$ iff $\mathcal{K} \models_{f'} \langle\psi', \emptyset\rangle$. Since ψ is closed, and therefore does not depend on f, showing this suffices. □

The second algorithm, VLTLtoLTL, translates the VLTL formula into an LTL formula, based on the values and the assignments of the given system $\mathcal{K}$ and function f. As in ModelCheck, a new value is added to a set C' that is maintained by the procedure (initially set to $A \cup B$, where A is the set of values in M, and $B = Image(f)$) for every quantifier in the formula. This time, every $\forall$ quantifier is translated to a conjunction of all of the recursively constructed formulas for these assignments, and every $\exists$ quantifier is translated to a disjunction.

Hence, the formula that is constructed by VLTLtoLTL contains every LTL formula that is checked by ModelCheck, and the conjunctions and disjunctions of VLTLtoLTL match the logical checks that are performed by ModelCheck.

VLTLtoLTL can then be used to model check entire concrete systems (in which case the formula is closed, and the initial function is $\emptyset$). While this leads to an exponentially large formula, this is the best that can be done, as we can show that the model checking problem for concrete systems is EXPSPACE complete, by a reduction from the acceptance problem for EXPSPACE Turing machines.

Theorem 2. *Let $\mathcal{K}$ be a concrete system over $P \cup T \times \mathcal{D}$ and let $\varphi = \langle\psi, E\rangle$ be a closed VLTL formula over $P \cup T \cup (T \times X)$. Then deciding whether $\mathcal{K} \models \varphi$ is EXPSPACE-complete.*

Proof: For the upper bound, using Lemma 1 we can show that for $B = Image(f)$ and for A, the finite set of values that occur in $\mathcal{K}$, it holds that $\mathcal{K} \models_f \langle\psi, E\rangle$ iff $\mathcal{K} \models$ VLTLtoLTL$(\langle\psi, E\rangle, f, A \cup B)$. The run VLTLtoLTL$(\langle\psi, E\rangle, f, A \cup B)$ produces an

LTL formula whose size is exponential in the size of $A \cup B$ and X. Since LTL model checking can be performed in PSPACE, checking $\mathcal{K} \models \mathsf{VLTLtoLTL}(\langle\psi, E\rangle, f, A \cup B)$ can be performed in EXPSPACE. Notice that if φ is closed, model checking is performed by using $\mathsf{VLTLtoLTL}(\langle\psi, E\rangle, \emptyset, A \cup B)$.

The lower bound is shown by a reduction from the acceptance problem for Turing machines that run in EXPSPACE. We sketch the proof. We define an encoding of runs of the Turing machine on a given input. For a Turing machine T and an input $\bar{a}$, we construct a system $\mathcal{K}$ whose computations include all encodings of potential runs of T on $\bar{a}$. We construct a VLTL formula φ that specifies computations that are not encodings of accepting runs of T on $\bar{a}$. Then, there exists an accepting run of T on $\bar{a}$ iff $\mathcal{K} \not\models \varphi$. □

The formula we construct for the lower bound in the proof of Theorem 2 is in ∃-VLTL, and so the model-checking problem is EXPSPACE-complete already for this fragment of VLTL. However, a simple variant of ModelCheck can be used for ∀-VLTL. Together with the PSPACE lower bound for LTL model checking, we have the following.

Theorem 3. *The model-checking problem for ∀-VLTL and concrete systems is PSPACE-complete.*

4 Model Checking of Abstract Systems

In this section we consider the VLTL model-checking problem for abstract systems. We begin by showing that the problem is undecidable, by proving undecidability for the fragment of ∃-VLTL. Then, we show that for certain abstract systems, as well as for ∀-VLTL, model checking is not more difficult than for concrete systems.

Theorem 4. *The model-checking problem for ∃-VLTL is undecidable.*

Proof: We sketch the proof, which is by reduction from Post's Correspondence Problem (PCP). A similar reduction is shown in [14]. An instance of PCP are two sets $\{u_1, u_2, \ldots u_n\}$ and $\{v_1, v_2, \ldots v_n\}$ of finite words over $\{a, b\}$, and the problem is to decide whether there exists a concatenation $u = u_{i_1} u_{i_2} \cdots u_{i_k}$ of words of the first set that is identical to a concatenation $v = v_{i_1} v_{i_2} \cdots v_{i_k}$ of words of the second set.

We describe an encoding of a correct solution to a PCP instance given an input. For an instance I of PCP, we construct an abstract system $\mathcal{K}$ whose computations include all possible encodings of solutions to I, and an ∃-VLTL formula φ that specifies computations that are not legal encodings to a solution to I. Then, we have that I has a solution iff $\mathcal{K} \not\models \varphi$. □

We now show that the VLTL model-checking problem for abstract systems is decidable for certain classes of systems. For the rest of the section, we consider abstract systems and abstract computations over P, $T \cup \{\mathit{reset}\}$, and X', and closed VLTL formulas over P, T, and X. We first introduce some terms.

We say that $\langle\mathcal{K}, E'\rangle$ is *bounded* if there is no occurrence of reset in $\mathcal{K}$. This means that the value of the variables does not change throughout a computation of the system. We say that $\langle\mathcal{K}, E'\rangle$ is *strict* if $E' = \{x'_i \neq x'_j | x'_i, x'_j, i \neq j \in X'\}$. Notice that a concrete computation in a bounded and strict system is obtained by an injection to the system variables.

We begin by showing that the model-checking problem for bounded systems is essentially equivalent to the model-checking problem for concrete systems.

Theorem 5. *The model-checking problem for bounded abstract systems is EXPSPACE-complete for VLTL, and PSPACE-complete for $\forall$-VLTL.*

Proof: Let $\langle \mathcal{K}, E' \rangle$ be a bounded abstract system, and let $\langle \psi, E \rangle$ be a VLTL formula.

We first prove the upper bounds for the case the system is both bounded and strict. Intuitively, assigning different values to the variables of a bounded and strict system results in a concrete system that satisfies the same set of formulas.

Formally, let $f : X' \to \mathcal{D}$ be an arbitrary injection, and let $\mathcal{K}_f$ be the concrete system that is obtained from $\langle \mathcal{K}, E' \rangle$ by substituting every occurrence of $x \in X'$ with $f(x)$. We can show that $\langle \mathcal{K}, E' \rangle \models \langle \psi, E \rangle$ iff $\mathcal{K}_f \models \langle \psi, E \rangle$.

We now turn to the general case of bounded systems, and reduce it to the case the systems are both bounded and strict. A set of bounded and strict abstract systems is obtained from $\langle \mathcal{K}, E' \rangle$ as follows. Consider the inequality set E'. Every possible setting of it induces a strict and bounded system: for every inequality $x_1 \neq x_2$ that is missing from E', both options of $x_1 \neq x_2$ and $x_1 = x_2$ are checked. For the former, a copy with $x_1 \neq x_2$ in the inequality set is constructed. For the latter, a copy with a new single variable replacing x_1 and x_2 is constructed. Then, we have that $\langle \mathcal{K}, E' \rangle \models \langle \psi, E \rangle$ iff every system in this set satisfies $\langle \psi, E \rangle$.

More specifically, consider a function $h : X' \to Z$ that respects E', where Z is a new set of variables of size $|X'|$. For every such h, an abstract system $\langle \mathcal{K}^h, E'^h \rangle$ is obtained from $\langle \mathcal{K}, E' \rangle$ by substituting every occurrence of $x \in X'$ with $h(x')$, and by setting E' to be the full inequality set. Then for $x_1, x_2 \in X'$, having $h(x_1) \neq h(x_2)$ is equivalent to setting $x_1 \neq x_2$, and $h(x_1) = h(x_2)$ is equivalent to setting $x_1 = x_2$. Every computation of $\langle \mathcal{K}, E' \rangle$ is a computation of $\langle \mathcal{K}^h, E'^h \rangle$ for some h, and vise versa.

Then, we have that $\langle \mathcal{K}, E' \rangle$does not satisfy $\langle \psi, E \rangle$ iff there exists a function h such that $\langle \mathcal{K}^h, E'^h \rangle$ does not satisfy $\langle \psi, E \rangle$. Therefore, the model-checking problem for bounded systems can be solved by guessing an appropriate function h, constructing, in linear time, a single copy of $\langle \mathcal{K}^h, E'^h \rangle$, and checking whether $\langle \mathcal{K}^h, E'^h \rangle \not\models \langle \psi, E \rangle$. This procedure is then performed in PSPACE in the size of the system and the formula.[5]

For the lower bounds, we reduce from the model-checking problem for concrete systems. Given a concrete system $\mathcal{M}$ and a VLTL formula $\langle \psi, E \rangle$, we construct in linear time a bounded (in fact, also strict) abstract system $\langle \mathcal{M}', E' \rangle$ by substituting every value d that occurs in $\mathcal{M}$ by the same unique variable x_d, and by setting E' to be the full inequality set. By a proof similar proof to the upper bound, we can show that $\langle \mathcal{M}', E' \rangle \models \langle \psi, E \rangle$ iff $\mathcal{M} \models \langle \psi, E \rangle$. □

Next, we show that the model-checking problem for abstract systems and $\forall$-VLTL formulas is, surprisingly, not more complex than the model-checking problem for LTL.

[5] For LTL, model checking is PSPACE-hard only in the size of the formula. For a fixed formula, LTL model checking can be performed in NLOGSPACE.

We do this by proving that this problem can be reduced to the model-checking problem for bounded abstract systems.

The following lemma shows that for a given assignment to the formula variables, the values in a concrete computation that are not assigned to any formula variable can be replaced with other such values without affecting the satisfiability.

Lemma 2. *Let $\langle\psi, E\rangle$ be a VLTL formula such that ψ is unquantified, and let $f : X \to \mathcal{D}$ be a function that respects E. Let π and τ be two concretizations of some abstract computation that agree on all values in $Image(f)$. Then $\pi \models_f \langle\psi, E\rangle$ iff $\tau \models_f \langle\psi, E\rangle$.*

We now show that in order to check whether an abstract computation ρ satisfies a $\forall$-VLTL formula φ, it is enough to check that every concretization of ρ that contains a bounded number of values satisfies φ.

Lemma 3. *Let $\langle\rho, E'\rangle$ be an abstract computation, and let $\langle\psi, E\rangle$ be a closed $\forall$-VLTL formula. Let C_ρ be the set of concretizations of $\langle\rho, E'\rangle$ that contains at most $|X'| + |X|$ different values. Then, $\langle\rho, E'\rangle \models \langle\psi, E\rangle$ iff for every $\pi \in C_\rho$, it holds that $\pi \models \langle\psi, E\rangle$.*

Proof: For the first direction, a computation in C_ρ is also a concretization of ρ.

For the second direction, suppose that for every $\tau \in C_\rho$, it holds that $\tau \models \langle\psi, E\rangle$. Assume by way of contradiction that there exists a concretization π of $\langle\rho, E'\rangle$ such that $\pi \not\models \langle\psi, E\rangle$. Since $\psi = \forall x_1; \ldots \forall x_k; \theta$, this means that there exists a function $f : X \to \mathcal{D}$ such that f respects E and $\pi \not\models \theta_f$.

If π contains at most $|X'| + |X|$ different values, then it is also in C_ρ. Therefore, π contains more than $|X'| + |X|$ different values. We show that there exists $\tau \in C_\rho$ such that $\tau \not\models \langle\psi, E\rangle$.

Let $a_1, a_2, \ldots a_k$ be the values that are assigned to the variables of X by f. Let $b_1, b_2, \ldots b_{k'}$, where $k' = |X'|$, be values different from the a values.

Let π' be the concrete computation obtained from ρ by assigning $a_1, a_2, \ldots a_k$ to the same occurrences of variables of X' that are assigned these values in π, and assigning every other occurrence of $x_i \in X'$ the same value b_i.

According to Lemma 2, we have that $\pi' \not\models \theta_f$. By the way we have constructed π', we have that π' is also a concretization of $\langle\rho, E'\rangle$, and that π' contains at most $|X'| + |X|$ different values. Therefore, it is also in C_ρ, a contradiction. □

Finally, we employ Lemma 3 in order to construct a model-checking procedure for $\forall$-VLTL, which runs in polynomial space.

Theorem 6. *The model-checking problem for $\forall$-VLTL and abstract systems is PSPACE-complete.*

Proof: The lower bound follows from the lower bound for LTL model checking.

Let $\langle\mathcal{K}, E'\rangle$ be an abstract system over P, $T \cup \{reset\}$, and X', where $|X'| = k'$, and let $\langle\psi, E\rangle$ be a closed $\forall$-VLTL formula over k variables. Intuitively, we construct from $\langle\mathcal{K}, E'\rangle$ a bounded system that contains exactly all computations of $\langle\mathcal{K}, E'\rangle$ that contain at most $k + k'$ different values.

Let λ be a set of new variables of size $k + k'$. Let $h : \lambda \to X'$ be an onto function. Intuitively, the function h partitions the variables of λ so that each set in the partition replaces a variable in X' in the construction.

Let $\Gamma_h = \{\{\xi_1, \xi_2 \ldots \xi_{k'}\} | \xi_i \in \lambda, h(\xi_i) = x_i, 1 \leq i \leq k'\}$.

For $\Delta \subseteq \Gamma_h$, let $\mathcal{K}_\Delta$ be the bounded system that is obtained from $\mathcal{K}$ as follows. For every set $\Gamma \in \Delta$, let $\mathcal{K}_\Gamma$ be the system obtained from $\mathcal{K}$ by replacing every occurrence of x_i with ξ_i for every $1 \leq i \leq k'$, and by removing every occurrence of $reset$. Then, in $\mathcal{K}_\Gamma$ every variable x in $\mathcal{K}$ is replaced with some variable ξ such that $h(\xi) = x$.

Let R be the set of transitions of $\mathcal{K}$. For every $\langle q, s\rangle \in R$, we add a transition from a copy of q in $\mathcal{K}_\Gamma$ to the copies of s in every $\mathcal{K}_{\Gamma'}$ such that Γ' and Γ agree on all variables in λ to which h assigns variables that are not reset in $\langle q, s\rangle$. Intuitively, a reset of a variable x is in a transition in $\mathcal{K}$ corresponds to switching from one variable in λ representing x to another.

Let $E_\Delta = \{\xi_i \neq \xi_j \,| h(\xi_i) \neq h(\xi_j) \in E'$, and $\xi_i, \xi_j \in \Gamma$ for some $\Gamma \in \Delta\}$. Then E_Δ is the inequality set induced by E' in $\mathcal{K}_\Delta$.

Note that $\mathcal{K}_\Delta$ models a copy of the system in which every occurrence of x_i between two consecutive resets is replaced by some variable in $h^{-1}(x_i)$. Also, for ξ_i and ξ_j such that $h(x_i) \neq h(x_j)$, if ξ_i, ξ_j are not in the same set in Δ, then they do not appear together in the same copy, and therefore are allowed to take the same value, even if $h(\xi_i) \neq h(\xi_j) \in E$.

Then, every abstract computation ρ of $\langle \mathcal{K}, E'\rangle$ is replaced with a set of bounded computations over $k + k'$ variables, whose set of concretizations is exactly the set of concretizations of ρ that contain at most $k + k'$ different values.

According to Lemma 3, we have that $\langle \mathcal{K}, E'\rangle \models \langle \psi, E\rangle$ iff for every h, and for every $\Delta \subseteq \Gamma_h$, it holds that $\langle \mathcal{K}_\Delta, E_\Delta\rangle \models \langle \psi, E\rangle$.

A nondeterministic procedure that runs in polynomial space guesses a function $f : X \rightarrow D$ and a function $g : \lambda \rightarrow D$, where $D \subset \mathcal{D}$ is some arbitrary set of size $2k + k'$. It follows from the proof of Theorem 5 and from Theorem 3 that a domain of the size of D is sufficient.

Next, in a procedure similar to the automata-theoretic approach to LTL model checking [18], the procedure constructs a violating path in the nondeterministic Büchi automaton $\mathcal{A}_{\neg\theta_f}$ that accepts exactly all computations that violate θ_f, constructs $\mathcal{K}_{\Delta g}$ on the fly, and guesses a violating path that is accepted by both $\mathcal{A}_{\neg\theta_f}$ and $\mathcal{K}_{\Delta g}$. While guessing a violating path in $\mathcal{K}_{\Delta g}$, the procedure must make sure that every state respects E', that is, there exist no ξ_i and ξ_j such that $h(\xi_i) \neq h(\xi_j) \in E'$, and $g(\xi_i) = g(\xi_j)$, and both ξ_i and ξ_j are in the same state along the path. Since the information needed to guess a path in both $\mathcal{A}_{\neg\theta_f}$ and $\mathcal{K}_{\Delta g}$ is polyomial, we have that the entire procedure can be performed in PSPACE in the size of the formula and of the system. □

5 Conclusions

We presented a simple, general, and natural extension to LTL and Kripke structures. Our extension allows to augment atomic propositions with variables that range over some (possibly infinite) domain. In VLTL, our extension of LTL, the extension enables the specification to refer to the domain values. In abstract systems, our extension of Kripke structures, the extension enables a compact description of infinite and complex concrete systems whose control is finite, and for which the source of infinity is the data.

We studied the model-checking problem in this setting, for both finite concrete systems and for abstract systems. We presented a model-checking procedure for VLTL and concrete systems. We showed that the general problem is EXPSPACE-complete for concrete systems and undecidable for abstract systems. As good news, we showed that even for abstract systems, the model-checking problem for the rich fragment of ∀-VLTL is not only decidable, but is of the same complexity as LTL model checking.

References

1. Barcelo, P., Libkin, L., Reutter, J.: Parameterized regular expressions and their languages. In: FSTTCS (2011)
2. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about Networks with many identical Finite-State Processes. In: PODC 1986 (1986)
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. In: TOPLAS 1994 (1994)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
5. Demri, S., D'Souza, D.: An Automata-Theoretic Approach to Constraint LTL. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 121–132. Springer, Heidelberg (2002)
6. Dingel, J., Filkorn, T.: Model Checking for Infinite State Systems Using Data Abstraction, Assumption-Commitment Style Reasoning and Theorem Proving. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 54–69. Springer, Heidelberg (1995)
7. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S., Nutzung, D.: 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. German, S., Sistla, A.P.: Reasoning about systems with many processes. JACM (1992)
9. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable Automata over Infinite Alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010)
10. Henzinger, T.A.: Hybrid Automata with Finite Bisimulations. In: Fülöp, Z. (ed.) ICALP 1995. LNCS, vol. 944, pp. 324–335. Springer, Heidelberg (1995)
11. Lazic, R.: Safely freezing LTL. In: FSTSTC (2006)
12. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. Proc. 12th POPL (1985)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1992)
14. Neven, F., Schwentick, T., Vianu, V.: Towards Regular Languages over Infinite Alphabets. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 560–572. Springer, Heidelberg (2001)
15. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. JACM (1985)
16. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theoretical Computer Science (1987)
17. Vardi, M.Y.: Personal communication (2011)
18. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and Systems Science (1986)
19. Chomicki, J., Toman, D.: Temporal Logic in Information Systems. In: Logics for Databases and Information Systems, pp. 31–70 (1998)
20. Dixon, C., Fisher, M., Konev, B., Lisitsa, A.: Efficient First-Order Temporal Logic for Infinite-State Systems. CoRR (2007)
21. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Decidable Fragments of First-Order Temporal Logics. Annals of Pure and Applied Logic (1999)
22. Alur, R.: Timed Automata. Theoretical Computer Science, pp. 183–235 (1999)

Reachability Analysis of Polynomial Systems Using Linear Programming Relaxations[⋆]

Mohamed Amin Ben Sassi[2], Romain Testylier[1],
Thao Dang[1], and Antoine Girard[2]

[1] Verimag, Université de Grenoble
[2] Laboratoire Jean Kuntzmann, Université de Grenoble
firstname.lastname@imag.fr

Abstract. In this paper we propose a new method for reachability analysis of the class of discrete-time polynomial dynamical systems. Our work is based on the approach combining the use of template polyhedra and optimization [25,7]. These problems are non-convex and are therefore generally difficult to solve exactly. Using the Bernstein form of polynomials, we define a set of equivalent problems which can be relaxed to linear programs. Unlike using affine lower-bound functions in [7], in this work we use piecewise affine lower-bound functions, which allows us to obtain more accurate approximations. In addition, we show that these bounds can be improved by increasing artificially the degree of the polynomials. This new method allows us to compute more accurately guaranteed over-approximations of the reachable sets of discrete-time polynomial dynamical systems. We also show different ways to choose suitable polyhedral templates. Finally, we show the merits of our approach on several examples.

1 Introduction

Reachability analysis has been a major research issue in the field of hybrid systems for more than a decade. Spectacular progress has been made over the past few years for a class of hybrid systems where the continuous dynamics can be described by affine differential equations [10,12,18,9]. However, dealing efficiently with systems with nonlinear dynamics remains a challenging problem that needs to be addressed. Besides, reachability analysis of non linear dynamical systems is also motivated by its numerous potential applications, in particular in systems biology [11,5,29].

In this paper, we present a new method for reachability analysis of a class of discrete-time nonlinear systems defined by polynomial maps. We follow the approach proposed in [25,7] which, by using template polyhedra, reduces the problem of reachability analysis to a set of optimization problems involving polynomials over bounded polyhedra. These are generally non-convex optimization

[⋆] This work was supported by the Agence Nationale de la Recherche (VEDECY project - ANR 2009 SEGI 015 01).

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 137–151, 2012.

problems and hence it is hard to solve them exactly. However, computing lower bounds of the solutions of these optimization problems is actually sufficient to obtain guaranteed over-approximations of the reachable sets that are usually needed for safety verification. Unlike using affine lower-bound functions in [5], in this work we use piecewise affine lower-bound functions, which allows us to obtain more accurate approximations. To this end, we essentially use the Bernstein expansions of polynomials and their properties to build linear programming relaxations of the original optimization problems. This can be roughly described as follows. First, by writing polynomials in the Bernstein basis we define a set of equivalent problems. Then, using properties of Bernstein polynomials, we show that good lower bounds of the optimal value of these problems can be computed efficiently using linear programming. This provides us with an elegant approach to reachability analysis of polynomial systems.

The rest of the paper is organized as follows. In Section 2, we show a technique for computing a lower bound of a non convex optimization problem where the cost function is a multivariate polynomial and the constraints are given by a bounded polyhedron included in the unit box. We then present a result which allows us to improve the accuracy of our lower bounds and a comparison with other relaxation methods. In Section 3, we show that reachability analysis of polynomial dynamical systems can be handled by optimizing multivariate polynomials on bounded polyhedra, and this will be used to compute guaranteed over-approximations of the reachable set. The choice of the templates, the complexity of the whole approach and a comparison with other related approach are also discussed. Finally, in Section 4, we show some experimental results including the application of our approach to reachability analysis of biological systems.

2 Optimization of Polynomials Using Linear Programming

In the following, we consider the problem of computing a guaranteed lower bound of the following optimization problem:

$$\begin{array}{ll} \text{minimize} & \ell \cdot g(y) \\ \text{over} & y \in [0,1]^n, \\ \text{subject to} & Ay \leq b. \end{array} \tag{1}$$

where $\ell \in \mathbb{R}^n$, $g : \mathbb{R}^n \to \mathbb{R}^n$ is a polynomial map, $A \in \mathbb{R}^{n\times m}$ and $b \in \mathbb{R}^m$. Let $y_1, \ldots, y_n$ denote the components of $y \in [0,1]^n$ and $\delta_1, \ldots, \delta_n$ denote the degrees of g in $y_1, \ldots, y_n$. Let $\Delta = (\delta_1, \ldots, \delta_n)$; for $I = (i_1, \ldots, i_n) \in \mathbb{N}^n$, we write $I \leq \Delta$ if $i_j \leq \delta_j$ for all $j \in \{1, \ldots, n\}$. Also, let $|I| = i_1 + \cdots + i_n$ and for $y \in \mathbb{R}^n$, $y^I = y_1^{i_1} \ldots y_n^{i_n}$. Then, the polynomial map g can be written in the form:

$$g(y) = \sum_{I \leq \Delta} g_I y^I \text{ where } g_I \in \mathbb{R}^n,\ \forall I \leq \Delta.$$

Let $\delta_{max} = \max\{|I|,\ \text{ such that } I \leq \Delta \text{ and } g_I \neq 0\}$.

2.1 Using the Bernstein Form

The main theoretical ingredient of our approach is the Bernstein expansion of polynomials [2,3]. The polynomial map g in its Bernstein form is given by:

$$g(y) = \sum_{I \leq \Delta} h_I B_{\Delta,I}(y) \text{ where } h_I \in \mathbb{R}^n,\ \forall I \leq \Delta \tag{2}$$

and the Bernstein polynomials are defined for $I \leq \Delta$ as follows:

$$B_{\Delta,I}(y) = \beta_{\delta_1,i_1}(y_1)\dots\beta_{\delta_n,i_n}(y_n)$$

with for $j = 1,\dots n$, $i_j = 0,\dots,\delta_j$: $\beta_{\delta_j,i_j}(y_j) = \begin{pmatrix}\delta_j\\ i_j\end{pmatrix} y_j^{i_j}(1-y_j)^{\delta_j - i_j}$.

The coefficients h_I of g in the Bernstein basis can be evaluated explicitly from the coefficients g_I of g in the canonical basis using the following explicit formula: for all $I \leq \Delta$,

$$h_I = \sum_{J \leq I} \frac{\begin{pmatrix}i_1\\ j_1\end{pmatrix}\dots\begin{pmatrix}i_n\\ j_n\end{pmatrix}}{\begin{pmatrix}\delta_1\\ j_1\end{pmatrix}\dots\begin{pmatrix}\delta_n\\ j_n\end{pmatrix}} g_J. \tag{3}$$

We also propose an alternative approach for computing the coefficients h_I using the interpolation at the points $\frac{J}{\Delta'} = (\frac{j_1}{\delta_1},\dots,\frac{j_n}{\delta_n})$ for $J \leq \Delta$:

$$\sum_{I \leq \Delta} h_I B_{\Delta,I}(\tfrac{J}{\Delta}) = g(\tfrac{J}{\Delta})$$

Let us denote B_Δ the matrix whose lines are indexed by $J \leq \Delta$ and columns are indexed by $I \leq \Delta$ with coefficients $B_{\Delta,I}(\frac{J}{\Delta})$. Let h be the matrix whose lines indexed by $I \leq \Delta$ are $h_I^\top$ and g the matrix whose lines indexed by $J \leq \Delta$ are $g(\frac{J}{\Delta})^\top$.Then, the previous equation under matricial form can be written as $\mathsf{B}_\Delta \mathsf{h} = \mathsf{g}$. From standard polynomial interpolation theory, the matrix B_Δ is invertible and the Bernstein coefficients are given by

$$\mathsf{h} = \mathsf{B}_\Delta^{-1}\mathsf{g}. \tag{4}$$

For determining a linear programming relaxation of (1), the most useful properties of the Bernstein polynomials are the following:

Proposition 1. *The Bernstein polynomials satisfy the following properties:*

1. *For all $y \in \mathbb{R}^n$, $\sum_{I \leq \Delta} B_{\Delta,I}(y) = 1$ and $\sum_{I \leq \Delta} \frac{I}{\Delta} B_{\Delta,I}(y) = y$.*
2. *For all $y \in [0,1]^n$, $0 \leq B_{\Delta,I}(y) \leq B_{\Delta,I}(\frac{I}{\Delta})$*
   where $B_{\Delta,I}(\frac{I}{\Delta}) = \prod_{j=1}^{j=n} \begin{pmatrix}\delta_j\\ i_j\end{pmatrix} \frac{i_j^{i_j}(\delta_j - i_j)^{\delta_j - i_j}}{\delta_j^{\delta_j}}$.

2.2 Linear Programming Relaxation

We can use the previous proposition to derive a linear programming relaxation of the problem (1):

Proposition 2. *Let $\underline{p}^*$ be the optimal value of the linear program:*

$$\begin{array}{lll} \textit{minimize} & \sum_{I\le\Delta}(\ell\cdot h_I)z_I & \\ \textit{over} & z_I\in\mathbb{R}, & I\le\Delta, \\ \textit{subject to} & 0\le z_I\le B_{\Delta,I}(\frac{I}{\Delta}), & I\le\Delta, \\ & \sum_{I\le\Delta} z_I=1, & \\ & \sum_{I\le\Delta}(A\frac{I}{\Delta})z_I\le b. & \end{array} \tag{5}$$

Then, $\underline{p}^*\le p^*$ where p^* is the optimal value of problem (1).

Proof. Using the Bernstein expansion (2) of g and the first property in Proposition 1, we can rewrite the problem (1) under the form

$$\begin{array}{ll} \text{minimize} & \sum_{I\le\Delta}(\ell\cdot h_I)B_{\Delta,I}(y) \\ \text{over} & y\in[0,1]^n, \\ \text{subject to} & \sum_{I\le\Delta}(A\frac{I}{\Delta})B_{\Delta,I}(y)\le b. \end{array}$$

Then, let y^* be the optimum of the problem (1), and let $z_I=B_{\Delta,I}(y^*)$ for all $I\le\Delta$. It is clear from Proposition 1 that these satisfy the constraints of (5), therefore the optimal value of (5) is necessary smaller than that of (1). □

Now we will show how to improve the precision at the expense of an increase in computational cost. The linear program (5) is a relaxation of the optimization problem (1). Then, the lower bound $\underline{p}^*$ on the optimal value p^* is generally not tight. We first remark that g can be seen as a higher order polynomial, possibly by adding monomials of higher degree with zero coefficients.

In the following, g is considered as a polynomial with degrees $K=(k_1,\dots,k_n)$ where $\Delta\le K$, then g can be written in the Bernstein form:

$$g(y)=\sum_{I\le K} h_I^K B_{K,I}(y),$$

We will use the following result [19]:

Proposition 3. *For $I\le K$,*

$$\|h_I^K-g(\tfrac{I}{K})\|=O(\tfrac{1}{k_1}+\cdots+\tfrac{1}{k_n}).$$

Now let $\underline{p}_K^*$ be the optimal value of the linear program (5) with degrees K instead of Δ. We want to determine the limit of $\underline{p}_K^*$ when all the k_i go to infinity.

Let $\underline{c}_K^*(y)$ be the optimal value of the linear program:

$$\begin{array}{lll} \text{minimize} & \sum_{I\le K}(\ell\cdot h_I^K)z_I & \\ \text{over} & z_I\in\mathbb{R}, & I\le K, \\ \text{subject to} & 0\le z_I\le B_{K,I}(\frac{I}{K}), & I\le K, \\ & \sum_{I\le K} z_I=1, & \\ & y=\sum_{I\le K}\frac{I}{K}z_I & \end{array} \tag{6}$$

Then,

$$\underline{p}_K^* = \min_{\substack{y\in[0,1]^n \\ Ay\leq b}} \underline{c}_K^*(y) \leq p^*.$$

Now let $C(\ell\cdot g)$ be the convex hull of the function $\ell \cdot g$ taken over the set $[0,1]^n$ (see e.g. [13]). Formally, $C(\ell\cdot g)$ is a convex function over $[0,1]^n$ such that for all $y\in[0,1]^n$, $C(\ell\cdot g)(y) \leq \ell\cdot g(y)$ and for all functions h convex over $[0,1]^n$ and such that for all $y\in[0,1]^n$, $h(y)\leq g(y)$, then for all $y\in[0,1]^n$, $h(y) \leq C(\ell\cdot g)(y)$. In other words, $C(\ell\cdot g)(y)$ is the largest function convex over $[0,1]^n$ bounding $\ell\cdot g$ from below. In particular, if $\ell\cdot g$ is convex then $C(\ell\cdot g) = \ell\cdot g$. Now, let p_C^* be the optimal value of the following optimization problem

$$\begin{array}{ll} \text{minimize} & C(\ell\cdot g)(y) \\ \text{over} & y\in[0,1]^n, \\ \text{subject to} & Ay\leq b. \end{array}$$

It is clear that $p_C^* \leq p^*$ with equality if $\ell\cdot g$ is convex. We can now state the main result of the section:

Proposition 4. *For all $K\geq\Delta$, $\underline{p}_K^* \leq p_C^*$ and*

$$|\underline{p}_K^* - p_C^*| = O(\tfrac{1}{k_1}+\cdots+\tfrac{1}{k_n}).$$

Proof. It is not hard to show that $\underline{c}_K^*$ is convex over $[0,1]^n$ and under-estimates $\ell\cdot g$. Then, by definition of the convex hull of a function we have: $\underline{c}_K{}^*(y) \leq C(\ell\cdot g)(y)$ for all $y\in[0,1]^n$. This gives directly that $\underline{p}_K^* \leq p_C^*$.

Now let $y\in[0,1]^n$, and let $z_I{}^*$, $I\leq K$ be an optimal solution of (6), then we have:

$$\underline{c}_K^*(y) = \sum_{I\leq K} (\ell\cdot h_I^K) z_I{}^* \text{ and } y = \sum_{I\leq K} z_I{}^* \frac{I}{K}.$$

Using properties of the convex hull of a function we have:

$$\begin{aligned} C(\ell\cdot g)(y) &= C(\ell\cdot g)\left(\textstyle\sum_{I\leq K} \frac{I}{K} z_I{}^*\right) \\ &\leq \sum_{I\leq K} C(\ell\cdot g)\left(\tfrac{I}{K}\right) z_I{}^* \leq \sum_{I\leq K} \ell\cdot g\left(\tfrac{I}{K}\right) z_I{}^*. \end{aligned}$$

It follows that:

$$0 \leq C(\ell\cdot g)(y) - \underline{c}_K^*(y) \leq \sum_{I\leq K} [\ell\cdot g(\tfrac{I}{K}) - \ell\cdot h_I^K] z_I{}^*,$$

Finally, using Proposition 3 we have for $y\in[0,1]^n$:

$$0 \leq C(\ell\cdot g)(y) - \underline{c}_K^*(y) \leq O(\tfrac{1}{k_1}+\cdots+\tfrac{1}{k_l})$$

which yields $|\underline{p}_K^* - p_C^*| = O(\frac{1}{k_1}+\cdots+\frac{1}{k_n})$. □

In other words, when we artificially increase the degrees of the polynomial g, we improve the computed lower bound. However, we cannot do better than p_C^* which is the optimal value of a convexified version of problem (1). We remark that if the function $\ell\cdot g$ is convex then we can approach the optimal value p^* arbitrarily close.

2.3 Related Work in Linear Relaxations

Our method of using the Bernstein form to replace expensive polynomial programming by linear programming is close to the one proposed in [7]. As mentioned earlier, the main improvement over this work is that our method uses a piecewise affine lower-bound function (the function $\underline{c}_K^*(y)$ defined in (6)) which is more accurate than a single affine lower bound function used in [7]. Moreover, our approach allows us to work directly with arbitrary polyhedral domains, while the Bernstein form in [7] works only for the unit box and this requires rectangular approximations that introduce additional error.

Let us now briefly discuss the complexity of our method and compare it with existing relaxation methods. In fact, the linear program (5) has polynomial complexity in its number of decision variables which is $N_\Delta = (\delta_1+1)\times\cdots\times(\delta_n+1)$. However, it should be noted that $N_\Delta = O({\delta_{max}}^n)$ which is exponential in the dimension n of the state space. Another known relaxation method which uses also linear programming is the reformulation-linearization-technique method (RLT) introduced by Sherali in [26,27]. The linear program given by this method is in fact a linearized version of the problem (1) by adding new variables where the constraints are given by exploiting all possible products of the original ones with respect to a fixed degree δ. In general, its number of decision variables is the same comparing to our method but the number constraints is much greater. We should mention that thanks to Bernstein properties our method can be more precise.

Also, one could use a method based on semi-definite programming and the theory of moments [17]. It should be noticed that the size of the semi-definite programs that need to be solved would be similar to the size of the linear program we solve. The quality of the lower bound obtained by semi-definite programming would certainly be better and it has been showed that an asymptotic convergence to the optimal value can be obtained; however, the approach would require more computational resources.

3 Reachability Analysis of Polynomial Dynamical Systems

Let us consider a disrete-time dynamical system of the following form:

$$x_{k+1} = f(x_k),\ k \in \mathbb{N},\ x_k \in \mathbb{R}^n,\ x_0 \in X_0 \tag{7}$$

where $f : \mathbb{R}^n \to \mathbb{R}^n$ is a polynomial map with degrees $\Delta = (\delta_1, \ldots, \delta_n)$ and X_0 is a bounded polyhedron in $\mathbb{R}^n$. In this paper, we are concerned with bounded-time reachability analysis of system (7). This consists in computing the sequence $X_k \subseteq \mathbb{R}^n$ of reachable sets at time k of the system up to some time $K \in \mathbb{N}$. It is straightforward to verify that this sequence satisfies the following induction relation $X_{k+1} = f(X_k)$. It should be noticed that even though the first element X_0 is a polyhedron, in general the other elements of the sequence are not. Actually, they are generally not even convex.

In this work, we over-approximate these sets using bounded polyhedra $\overline{X}_k$. Such a sequence can clearly be computed inductively by setting $\overline{X}_0 = X_0$ and by ensuring that for all $k = 0, \ldots, K-1$, $f(\overline{X}_k) \subseteq \overline{X}_{k+1}$. Hence, we first focus on the computation of a polyhedral over-approximation $\overline{X}_{k+1}$ of the image of a bounded polyhedron $\overline{X}_k$ by a polynomial map f.

Now, the problem we address is stated as follows: given a polyhedron $\overline{X}_k$, we want to compute $\overline{X}_{k+1}$ such that $f(\overline{X}_k) \subseteq \overline{X}_{k+1}$. In the following we propose a solution to this problem based on the use of template polyhedra and optimization.

3.1 Template Polyhedra

To represent $\overline{X}_{k+1}$, we use a template polyhedron. A template is a set of linear functions over $x \in \mathbb{R}^n$. We denote a template by a matrix $A \in \mathbb{R}^{m \times n}$, given such a template A and a coefficient vector $b \in \mathbb{R}^m$, we define the template polyhedron

$$Poly(A, b) = \{x \in \mathbb{R}^n | \; Ax \leq b\}$$

where the inequality is to be understood component-wise. By varying the value of b, we obtain a family of template polyhedra corresponding to the template A. Essentially, the template A defines the directions of the faces and the vector b define their positions. The advantage of using template polyhedra over general convex polyhedra is that the Boolean operations (union, intersection) and common geometric operations can be performed more efficiently. Indeed, such operations are crucial for most verification procedures based on reachable set computations.

In the following, we assume that $\overline{X}_{k+1}$ is a bounded polyhedron with a given template $A_{k+1} \in \mathbb{R}^{m \times n}$. We add the subscript $k+1$ to emphasize that the template may not be the same for all the polyhedra $\overline{X}_{k+1}$, $k = 0, \ldots, K-1$. Then,

$$\overline{X}_{k+1} = Poly(A_{k+1}, b_{k+1})$$

where the vector $b_{k+1} \in \mathbb{R}^m$ needs to be determined at each iteration. The choice of the templates A_{k+1} will be discussed later in Section 3.3.

From the previous discussion, it appears that the computation of the set $\overline{X}_{k+1}$ reduces to determining values for the vectors b_{k+1}. Let $b_{k+1,i}$ denote the i-th element of these vectors; and $A_{k+1,i}$ denote the i-th line of matrices A_{k+1} .

Lemma 1. *If for all $i = 1, \ldots, m$*

$$-b_{k+1,i} \leq \min_{x \in \overline{X}_k} -A_{k+1,i} f(x) \tag{8}$$

then $f(\overline{X}_k) \subseteq \overline{X}_{k+1}$ where $\overline{X}_{k+1} = Poly(A_{k+1}, b_{k+1})$

Proof. Let $y \in f(\overline{X}_k)$. For $i = 1, \ldots, m$, it is clear that we have

$$A_{k+1,i} y \leq \max_{x \in \overline{X}_k} A_{k+1,i} f(x).$$

Then, by remarking that $\max_{x \in \overline{X}_k} A_{k+1,i} f(x) = -\min_{x \in \overline{X}_k} -A_{k+1,i} f(x)$ and by (8) we obtain $A_{k+1,i} y \leq b_{k+1,i}$. □

Let us remark that the computation of the minimal values in equations (8) involves optimizing a generally non-convex multi-variable polynomial function on a bounded polyhedron. This is a difficult problem in general; however Lemma 1 shows that the computation of a lower bound for the minimal values is sufficient to obtain an over-approximation of $f(\overline{X}_k)$. Thus, we can see that the computation of $\overline{X}_{k+1}$ can be done by computing guaranteed lower bounds on the optimal values of minimization problems involving multi-variable polynomial functions on a bounded polyhedron. A solution to this problem, based on linear programming, is proposed in the previous section when $x \in [0,1]^n$. We will see how this result can be adapted to our reachability problem.

3.2 Reachability Algorithm

According to the previous discussion, in each step $k \in \mathbb{N}$ we have to compute a lower bound of the value

$$p_{k+1,i}{}^{*} = \min_{x \in \overline{X}_k} -A_{k+1,i} \cdot f(x) \text{ for all } i = 1, \ldots, m.$$

For doing so, we will propose an algorithm with essentially three steps: in the first one we compute a bounding parallelotope that will be necessary for the second step. The second one will consist in a change of variable allowing us to recast our optimization problem on the form of the one given by (1). In the last step, lower bound will be obtained using the linear program given by Proposition 2 and then an over approximation of the reachable set is computed.

Step 1: Bounding Parallelotope Computation
As $\overline{X}_k$ is a bounded polyhedron of $\mathbb{R}^n$ we can write it in the form $\overline{X}_k \cap Q_k$ where Q_k is its bounding parallelotope given by $Q_k = Poly(\tilde{C}_k, \tilde{d}_k)$ with

$$\tilde{C}_k = \begin{bmatrix} C_k \\ -C_k \end{bmatrix}, \; \tilde{d}_k = \begin{bmatrix} \overline{d}_k \\ -\underline{d}_k \end{bmatrix}.$$

$C_k \in \mathbb{R}^{n \times n}$ is an invertible matrix, $\underline{d}_k \in \mathbb{R}^n$ and $\overline{d}_k \in \mathbb{R}^n$. We assume that the matrix direction C_k is given as an input (a method describing its computation will be given later) and we compute the component $\overline{d}_{k,i}$ and $\underline{d}_{k,i}$, $i = 1, \ldots n$ of the vector position $\tilde{d}_k$ as optimal solutions of the following linear programs :

$$\overline{d}_{k,i} = \max_{x \in \overline{X}_k} C_{k,i} \cdot x \text{ and } \underline{d}_{k,i} = \max_{x \in \overline{X}_k} -C_{k,i} \cdot x \quad \forall i = 1, \ldots n$$

Step 2: Change of Variable
Now, let's proceed to the following change of variable $x = q_k(y)$ where the affine map $q_k : \mathbb{R}^n \to \mathbb{R}^n$ is given by

$$q_k(y) = C_k{}^{-1} D_k y + C_k{}^{-1} \underline{d_k}$$

where D_k is the diagonal matrix with entries $\overline{d}_{k,i} - \underline{d}_{k,i}$. The change of variable q_k essentially maps the unit cube $[0,1]^n$ to Q_k. This change of variable is the main reason for defining $\overline{X}_k$ as the intersection of a polyhedron and a parallelotope. We then define the polynomial map g_k as $g_k(y) = f(q_k(y))$. Finally, let $A'_k \in \mathbb{R}^{n\times m}$ and $b'_k \in \mathbb{R}^m$ be given by

$$A'_k = A_k {C_k}^{-1} D_k,\ b'_k = b_k - A{C_k}^{-1}\underline{d}.$$

Remark 1. It is clear that g_k is a polynomial map. As for the degrees $\Delta' = (\delta'_1, \ldots, \delta'_n)$ of g_k in the variables $y_1, \ldots, y_n$, we shall discuss two different cases depending on the nature of parallelotope Q_k. If Q_k is an axis-aligned box (i.e. if C_k is a diagonal matrix), then the degrees of g_k are the same as f: $\Delta' = \Delta$. This is not the case in general, when Q_k is not axis-aligned; in that case, the change of variable generally increases the degrees of the polynomial and g_k can be regarded as a polynomial of degrees $\Delta' = (\delta_{max}, \ldots, \delta_{max})$.

Step 3: Solving the Optimization Problem

After the change of variable we easily find the equivalent optimization problem:

$$\begin{array}{ll} \text{minimize} & -A_{k+1,i} \cdot g_k(y) \\ \text{over} & y \in [0,1]^n, \\ \text{subject to} & A'_k y \leq b'_k. \end{array}$$

Then, thanks to Proposition 2, a lower bound $-b_{k+1,i}$ can be found.
The reachable set at the step $k+1$ will be $\overline{X}_{k+1} = Poly(A_{k+1}, b_{k+1})$.

3.3 Choice of the Templates

In this section, we discuss the choice of the templates A_k for the polyhedra $\overline{X}_k$ and C_k for the parallelotope Q_k used to over-approximate the reachable sets.

Dynamical Templates for Polyhedra $\overline{X}_k$

Let us consider the template A_k for the polyhedron $\overline{X}_k = \{x \in \mathbb{R}^n |\ A_k \cdot x \leq b_k\}$. We propose a method which involves determining dynamical templates based on the dynamics of the system.

In the next iteration, we want to compute a new template A_{k+1} that reflects as much as possible the changes of the shape of $\overline{X}_k$ under the polynomial f. For that purpose, we use a local linear approximation of the dynamics of the polynomial dynamical system (7) given by the first order Taylor expansion around the centroid x_k^* of the last computed polyhedron $\overline{X}_k$:

$$f(x)\ \approx L_k(x) = f(x_k^*) + J(x_k^*)(x - x_k^*)$$

where J is the Jacobian matrix of the function f. Let us denote $F_k = J(x_k^*)$ and $h_k = f(x_k^*) - J(x_k^*)x_k^*$, then in a neighborhood of x_k^* the nonlinear dynamics can be roughly approximated by $x_{k+1} = F_k x_k + h_k$. Assuming that F_k is invertible, this gives $x_k = F_k^{-1}x_{k+1} - F_k^{-1}h_k$. Transposing the constraints on x_k given by $\overline{X}_k$ to x_{k+1}, we obtain

$$A_k F_k^{-1} x_{k+1} \leq b_k + A_k F_k^{-1} h_k$$

Then, it appears that a reasonable template for $\overline{X}_{k+1}$ should be $A_{k+1} = A_k F_k^{-1}$. This new template A_{k+1} can then be used in next iteration for the computation of the polyhedron $\overline{X}_{k+1}$ using the method described in the previous section. Let us remark that this choice implies that our reachability algorithm is exact if f is an affine map.

Dynamical Templates for Parallelotope Q_k

It can be useful in some cases to take static axis aligned boxes for Q_k (i.e. C_k is the identity matrix for all $k = 0, \dots, K$). This allows us to preserve the degrees of the polynomials when making the change of basis. However, as explained before, using static templates may produce increasingly large approximation errors. Similar to the polyhedra $\overline{X}_k$, the accuracy will be better if we use dynamical templates which take in consideration the dynamic of the system (essentially the rotation effects).

We should mention that the image of an oriented rectangular box Q_k by the linear map F_k is not necessarily an oriented rectangular box so we can not use directly the matrix F_k^{-1} computed previously. To solve this problem we will use an popular technique in interval analysis [22] based on the QR-Decomposition of matrices. Essentially, F_k will be written as the product of two matrices $F_k = \mathsf{Q_k R_k}$ where $\mathsf{Q_k}$ is an orthogonal matrix and R is an upper triangular one. Then, to choose the template C_{k+1} of the next oriented box Q_{k+1}, we apply our rotation transformation matrix $\mathsf{Q_k}$ to the given rectangular box Q_k which is equivalent to choose the template $C_{k+1} = C_k \mathsf{Q_k}^\top$. Of course, in that case, we will deal with non-aligned-axis boxes which can cause higher degrees for our polynomial but the approximation will be less conservative than using static templates for Q_k.

3.4 Computation Cost and Related Work

We have presented two approaches to compute the Bernstein form of the polynomial after a change of variable: either we compute explicitly the change of variable and then use equation (3) or we proceed by using equation (4). Both methods have polynomial time and space complexity in $N_{\Delta'} = (\delta'_1 + 1) \times \cdots \times (\delta'_n + 1)$. The size of the matrix $\mathsf{B}_{\Delta'}$ introduced in the interpolation method is $(\delta'_1 + 1) \times \cdots \times (\delta'_n + 1)$. Let us remark that in the context of reachability analysis, the inverse matrix $\mathsf{B}_{\Delta'}^{-1}$ has to be computed only once and can be used in each iteration to compute Bernstein coefficients by a simple matrix vector product operation.

In fact, we shall see from numerical experiments that both methods have their advantages depending on the form of the parallelotope Q. If Q is an axis-aligned box, we have already seen that the change of variable does not increase the degrees of the polynomial. So, the matrix $\mathsf{B}_{\Delta'}$ has a reasonnable size and the computation of the coefficients h_I using equation (4) is generally more efficient. If Q is a general parallelotope, then $\mathsf{B}_{\Delta'}$ might be much larger. In that case, since several coefficients f_I of the polynomial f might actually be zero, it will be more efficient to compute explicitly the change of variable and use equation (3) to determine the coefficients h_I.

Then, as we mentioned before, the linear program has polynomial complexity in its number of decision variables which is also $N_{\Delta'}$. Therefore, the complexity of the overall procedure is polynomial in $N_{\Delta'}$, and so is the complexity of the reachability procedure described in Lemma 1.

Finally, following the discussion in Remark 1, we would like to highlight that $N_{\Delta'}$ might be much smaller when the parallelotope Q is an axis-aligned box. Indeed, in that case $N_{\Delta'} = (\delta_1 + 1) \times \cdots \times (\delta_n + 1)$ whereas in the general case we have $N_{\Delta'} = (\delta_{max} + 1)^n$. This point might be taken into consideration in the reachability algorithm when choosing the template for parallelotopes Q_{k+1}.

One could also use interval analysis [14] for computing the reachable sets of polynomial systems. However, these methods are generally used with rectangular domains. Moreover, our approach obtains enclosures that are always finer than those obtained using Bernstein coefficients and it has been shown that these are generally more accurate than those computed using interval arithmetics [20]. Interval analysis methods can be improved using preconditionning techniques (see e.g.[22]), however these can also be used in our approach as shown in the previous section.

Also, a popular approach for nonlinear systems is to characterize reachable sets using Hamilton-Jacobi formulation [21], and then solve the resulting partial differential equations which requires expensive computations. Recent results of this approach can be seen in [15]. Another approach is based on a discretization of the state-space for abstraction (see e.g [28,16]) and approximation especially using linearization (see e.g [1,6]). For the particular class of polynomial systems,direct methods for reachability analysis [4,7] without state-space discretization has been proposed . The improvement over [7] has also been discussed in Section 2.2.

Concerning set representation, the work presented in this paper draws inspiration from the approach using template polyhedra [25]. In the hybrid systems verification, polynomial optimization can also be used to compute barrier certificates [24,25], and algebraic properties of polynomials are used to compute polynomial invariants [28] and to study computability issues of the image computation in [23].

4 Experimental Results

We implemented our reachability computation method and tested it on various examples. To solve linear programs, we use the publicly available lp_solve library.

4.1 FitzHugh-Nagumo Neuron Model

The first example we studied is a discrete time version of the FitzHugh-Nagumo model [8] which is a polynomial dynamical system modelling the electrical activity of a neuron:

$$\begin{cases} x_1(k+1) = x_1(k) + h\left((x_1(k) - \frac{x_1(k)^3}{3} - x_2(k) + I\right) \\ x_2(k+1) = x_2(k) + h\left(0.08(x_1(k) + 0.7 - 0.8x_2(k))\right) \end{cases}$$

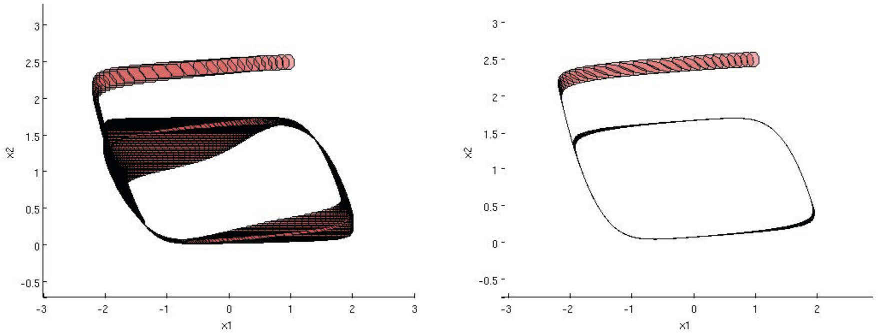


Fig. 1. Reachability computation for the FitzHugh-Nagumo neuron model using static (left) and dynamical (right) template polyhedra $\overline{X}_k$ with static bounding boxes

where the model parameter I is equal to $\frac{7}{8}$ and the time step $h = 0.05$. This system is known to have a limit cycle.

Figure 1 shows two reachable set evolutions where the initial set is a regular octagon included in the bounding box $[0.9, 1.1] \times [2.4, 2.6]$. The figure on the left was computed using static templates for polyhedra $\overline{X}_k$ where dynamical templates are used for the figure on the right. In both cases, we use axis-aligned boxes for Q_k. For a better readability, the reachable sets are plotted once every 5 steps. We observed a limit cycle after 1000 iterations. The computation time is 1.16 seconds using a static template and 1.22 seconds using the dynamical templates. We can see from the figure a significant precision improvement obtained by using dynamical templates, at little additional cost.

4.2 Prey Predator Model and Performance Evaluation

Now we consider the generalized Lotka-Volterra equations modelling the dynamics of the population of n biological species known as the prey predator model. Its equations are given by $\dot{x_i} = x_i(r_i + A_i x)$ where $i \in \{1, 2, \ldots, n\}$, r_i is the i^{th} elements of a vector $r \in \mathbb{R}^n$ and A_i is the i^{th} line of a matrix $A \in \mathbb{R}^{n \times n}$.

We performed reachable set computation for an Euler discretized Lotka-Voltera system for the case $n = 2$:

$$\begin{cases} x_1(k+1) = x_1(k) + h(0.1x_1 - 0.01x_1x_2) \\ x_2(k+1) = x_2(k) + h(-0.05x_2 + 0.001x_1x_2) \end{cases}$$

Figure 2 shows the cyclic behavior of the reachable set analysis computed using a discretization time $h = 0.3$ with an initial box included in $[49, 51] \times [14, 16]$ during 700 iterations. The figure on the left was computed in 1.87 seconds using dynamical template polyhedra and bounding boxes aligned with axis. The other one was computed in 3.46 seconds using dynamical templates and oriented boxes. A significant gain of precision using the oriented box can be observed however the computation time is almost double.

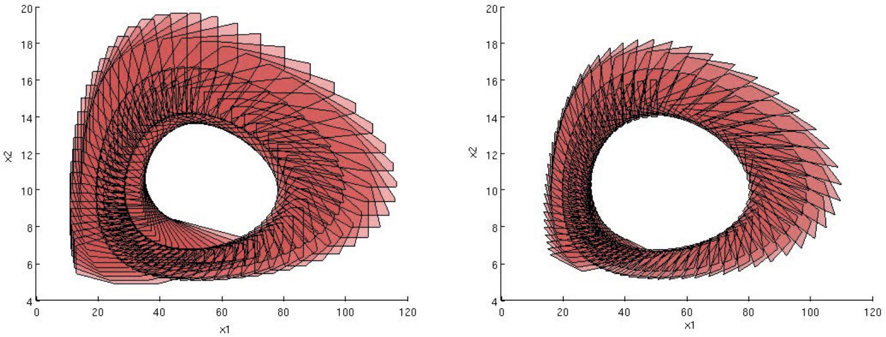


Fig. 2. Reachability computation for a 2 dimensional predator-prey model using dynamical template polyhedra $\overline{X}_k$ with axis aligned (left) and oriented (right) bounding boxes

We also evaluated the performance of our method using two ways of computing the Bernstein coefficients (explicitly and by interpolation) with recursively generated n-dimensional Lotka-Volterra equations given by:

$$\begin{cases} x_1(k+1) = x_1(k) + h\,(x_1(k)(1 - x_2(k) + x_n(k))) \\ x_i(k+1) = x_i(k) + h\,(x_i(k)(-1 - x_{i+1}(k) + x_{i-1}(k))) \\ x_n(k+1) = x_n(k) + h\,(x_n(k)(-1 - x_0(k) + x_{n-1}(k))) \end{cases}$$

where $i \in \{2, \ldots, n-1\}$. We used axis aligned bounding boxes to make the change of variable. (see tables 1).

Table 1. Computation time for one reachable set computation iteration for some generated Lotka-Voltera systems

dim	explicit	interpol	$\mathsf{B}_{\Delta'}^{-1}$
2	0.00235	0.00221	0.00001
3	0.00536	0.00484	0.00004
4	0.01112	0.01008	0.00008
5	0.02612	0.02124	0.00052
6	0.068	0.0499	0.0016
7	0.1905	0.1274	0.0099
8	0.5682	0.3674	0.0494
9	1.935	1.265	0.266
10	6.392	4.441	1.623
11	21.98	16.03	10.36

We observe that the interpolation method provides more effective results than the explicit computation of Bernstein coefficient but requires to compute the matrix $\mathsf{B}_{\Delta'}^{-1}$ before starting the analysis. A similar evaluation was done using oriented boxes but the results show that this method is not tractable over 4 dimension due to the degree elevation of polynomials by the change of variable when we don't use axis-aligned boxes.

5 Conclusion

In this paper we proposed an approach for computing reachable sets of polynomial dynamical systems. This approach combines optimization and set representation using template polyhedra. The novelty of our results lies in a efficient method for relaxing a polynomial optimization problem to a linear programming problem. On the other hand, by exploiting the evolution of the system we proposed a way to determine templates dynamically so that the reachable sets can be approximated more accurately. The approach was implemented and our experimental results are promising, compared to the existing results (see e.g. [7]). We intend to continue this work in a number of directions. One direction involves an extension of the approach to systems with parameters and uncertain inputs. Additionally, the evolution of templates can be estimated locally around each facet rather than globally at the centroid of a template polyhedron.

References

1. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Informatica 43(7), 451–476 (2007)
2. Bernstein, S.: Collected Works, vol. 1. USSR Academy of Sciences (1952)
3. Bernstein, S.: Collected Works, vol. 2. USSR Academy of Sciences (1954)
4. Dang, T.: Approximate Reachability Computation for Polynomial Systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 138–152. Springer, Heidelberg (2006)
5. Dang, T., Le Guernic, C., Maler, O.: Computing Reachable States for Nonlinear Biological Models. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 126–141. Springer, Heidelberg (2009)
6. Dang, T., Maler, O., Testylier, R.: Accurate hybridization of nonlinear systems. In: HSCC 2010, pp. 11–20 (2010)
7. Dang, T., Salinas, D.: Image Computation for Polynomial Dynamical Systems Using the Bernstein Expansion. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 219–232. Springer, Heidelberg (2009)
8. FitzHugh, R.: Impulses and physiological states in theoretical models of nerve membrane. Biophysical J. 1, 445–466 (1961)
9. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
10. 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)
11. Halasz, A., Kumar, V., Imielinski, M., Belta, C., Sokolsky, O., Pathak, S., Rubin, H.: Analysis of lactose metabolism in e.coli using reachability analysis of hybrid systems. IET Systems Biology 1(2), 130–148 (2007)
12. Han, Z., Krogh, B.H.: Reachability Analysis of Large-Scale Affine Systems Using Low-Dimensional Polytopes. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 287–301. Springer, Heidelberg (2006)

13. Horst, R., Tuy, H.: Global optimazation: Deterministic approaches, 2nd edn. Springer (1993)
14. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer (2001)
15. Kaynama, S., Oishi, M., Mitchell, I., Dumont, G.A.: The continual reachability set and its computation using maximal reachability techniques. In: CDC (2011)
16. Kloetzer, M., Belta, C.: Reachability Analysis of Multi-affine Systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 348–362. Springer, Heidelberg (2006)
17. Lasserre, J.B.: Global optimization with polynomials and the problem of moments. SIAM Journal of Optimization 11(3), 796–817 (2001)
18. 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)
19. Lin, Q., Rokne, J.G.: Interval approxiamtions of higher order to the ranges of functions. Computers Math 31, 101–109 (1996)
20. Martin, R., Shou, H., Voiculescu, I., Bowyer, A., Wang, G.: Comparison of interval methods for plotting algebraic curves. Computer Aided Geometric Design (19), 553–587 (2002)
21. Mitchell, I., Tomlin, C.J.: Level Set Methods for Computation in Hybrid Systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 310–323. Springer, Heidelberg (2000)
22. Nedialkov, N.S., Jackson, K.R., Corliss, G.F.: Validated solutions of initial value problems for ordinary differential equations. Applied Mathematics and Computation (105), 21–68 (1999)
23. Platzer, A., Clarke, E.M.: The Image Computation Problem in Hybrid Systems Model Checking. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 473–486. Springer, Heidelberg (2007)
24. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Transactions on Automatic Control 52(8), 1415–1429 (2007)
25. Sankaranarayanan, S., Dang, T., Ivančić, 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)
26. Sherali, H.D., Tuncbilek, C.H.: A global optimization algorithm for polynomial programming using a reformulation-linearization technique. Journal of Global Optimization 2, 101–112 (1991)
27. Sherali, H.D., Tuncbilek, C.H.: New reformulation-linearization/convexification relaxations for univariate and multivariate polynomial programming problems. Operation Research Letters 21, 1–9 (1997)
28. Tiwari, A., Khanna, G.: Nonlinear Systems: Approximating Reach Sets. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 600–614. Springer, Heidelberg (2004)
29. Yordanov, B., Belta, C.: Cdc. In: A Formal Verification Approach to the Design of Synthetic Gene Networks (2011)

Linear-Time Model-Checking for Multithreaded Programs under Scope-Bounding*

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], K. Narayan Kumar[3], and Prakash Saivasan[3]

[1] Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se
[2] LIAFA, Université Paris Diderot, France
abou@liafa.univ-paris-diderot.fr
[3] Chennai Mathematical Institute, India
{kumar,saivasan}@cmi.ac.in

Abstract. We address the model checking problem of omega-regular linear-time properties for shared memory concurrent programs modeled as multi-pushdown systems. We consider here boolean programs with a finite number of threads and recursive procedures. It is well-known that the model checking problem is undecidable for this class of programs. In this paper, we investigate the decidability and the complexity of this problem under the assumption of scope-boundedness defined recently by La Torre and Napoli in [24]. A computation is scope-bounded if each pair of call and return events of a procedure executed by some thread must be separated by a bounded number of context-switches of that thread. The concept of scope-bounding generalizes the one of context-bounding [31] since it allows an unbounded number of context switches. Moreover, while context-bounding is adequate for reasoning about safety properties, scope-bounding is more suitable for reasoning about liveness properties that must be checked over infinite computations. It has been shown in [24] that the reachability problem for multi-pushdown systems under scope-bounding is PSPACE-complete. We prove in this paper that model-checking linear-time properties under scope-bounding is also decidable and is EXPTIME-complete.

1 Introduction

The behaviors of concurrent programs are in general extremely intricate due to the tremendous amount of interactions between their parallel components. This complexity leads to subtle bugs, erroneous behaviors that are hard to predict and to reproduce. Therefore, it is very important to investigate the issue of providing rigorous and automated verification techniques to help programmers in this difficult task. However, the verification problem of concurrent programs is a highly challenging problem both from theoretical and practical point of view.

* Partially supported by the Swedish Research Council within UPMARC, CNRS LIA InForMel, and the project ANR-09-SEGI-016 Veridyc.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 152–166, 2012.

In this paper we consider the class of shared memory concurrent programs. More precisely, we consider programs having a finite (fixed) number of threads running in parallel and sharing a finite number of variables; each thread being a sequential process that may have (potentially recursive) procedure calls. It is well-known that, even when the manipulated data are booleans, these concurrent programs are Turing powerful [32]. In fact, it is easy to see that boolean multi-threaded programs can be translated quite naturally to multi-pushdown systems (MPDS), and vice-versa, where a MPDS is a finite control-state machine that has several stacks. (A 1-stack MPDS is a plain pushdown system, while already a 2-stack MPDS is a Turing-powerful model.) Intuitively, the control-states correspond to the configurations of the shared memory while the stacks are used to represent the local call stacks of each thread.

Since all verification problems for (boolean) multi-threaded programs are undecidable in general, many works have addressed the issue of identifying reasonable assumptions on their behaviors that lead either to (1) decidable classes of models corresponding to significant classes of programs [12,33,20,8,9], or to (2) decidable analysis allowing the efficient detection of bugs in concurrent programs encountered in practice (e.g. [31,17,13]). In the seminal paper [31], context-bounding has been proposed as a relevant concept for detecting *safety* bugs in shared memory multi-threaded programs, and it has been shown in a series of subsequent works that this concept leads indeed to efficient analysis for these programs [29,27,23,21].

However, besides safety, it is crucial to ensure for concurrent programs *termination* of certain computation phases (typically computations resulting from *retrying* mechanisms that are used for ensuring mutual exclusion and absence of interference), and *liveness* properties in general, such as *response* properties (i.e., an action by some process is eventually followed by some action by the same or some other process) or *persistence* properties (i.e., after the occurrence of some event, the program eventually reaches some state from which some property is permanently satisfied).

Then, one interesting question is what would be a suitable concept for restricting the behaviors of multi-threaded programs when reasoning about liveness properties, and more generally about any omega-regular property expressible in a linear-time temporal logic such as LTL or by a Büchi automaton?

In fact, while context-bounding is useful for detecting safety bugs for which it is sufficient to consider finite computations, this concept is not appropriate for reasoning about liveness properties for which it is necessary to consider infinite behaviors. The reason is that, roughly speaking, context-bounding does not give a chance for every thread to be executed infinitely often, since bounding the number of context-switches necessarily implies that after some point all threads except one must be ignored while the chosen one remains active forever. For instance, in the program of Figure 1, we can not detect

```
bool g;

proc Thread1 ()
   while g do
      g := false;
   return

proc Thread2 ()
   while !g do
      g := true;
   return
```

Fig. 1. A non-terminating program [6]

the presence of a non-terminating execution under the context-bounding policy. In that respect, the concept of scope-bounding introduced in [24] is more suitable for reasoning about liveness since it allows behaviors with unbounded context-switches between threads. Informally, scope-bounding means that each pair of call and return events of a procedure executed by some thread must be separated by a bounded number of context-switches of that thread. This means that each symbol that is stored in a stack can leave the stack only within a bounded number of context switches. Actually, we adopt in this paper a slightly different, but equivalent w.r.t. model checking, definition which consists in considering that useless symbols that would stay in the bottom of the stacks forever are actually never pushed in the stacks. This implies that as long as a thread remains active, it has to empty its stack repeatedly after at most each K context-switches, where K is the considered bound on the scope; e.g., the non-terminating execution of the program in Figure 1 requires a scope-bound of size at most 1. Interestingly, it has been shown that the state reachability problem under scope-bounding is decidable (and PSPACE-complete) [24]. Then, natural questions are (1) whether model checking linear-time properties under scope-bounding is decidable, and (2) what is the complexity of this problem?

The contribution of this paper is the proof that model-checking omega-regular properties for MPDS's under scope bounding is decidable and EXPTIME-complete. To establish this result, we show that it is possible to reduce the considered problem to the emptiness problem of a Büchi (single-stack) pushdown system. Let us give briefly the intuition behind our proof. First, let us assume that we are given an MPDS M and, following the standard automata-based model-checking approach, that we are also given a Büchi automaton B corresponding to the complement (negation) of the specification (a formula of LTL). Then, our problem is to check whether in the product of M with B, there is an infinite computation satisfying the scope bounding requirement and visiting infinitely often some accepting state of B.

The basic idea of our reduction is to reason in a compositional way about *thread interfaces* corresponding to the states that are visible at context-switch points. We show that, when all threads are active infinitely often, the interface of each thread can be defined by a finite-state automaton, and then our problem can be reduced to a repeated state reachability problem in the composition of these interface automata. In general, to capture also the case where after some point all thread except one may be stopped, we need to reason about interfaces as above for the finite prefix where all threads are active, and then simulate with a single-stack pushdown automaton the rest of the infinite computation corresponding to the execution of the only thread that remains active.

To build automata recognizing thread interfaces, the basic ingredient is the *summarization*, for each thread, of computation segments relating two successive configurations where the stack of that thread is empty. Let us call such a computation segment a *cluster*. Scope-boundedness imposes that the thread must have a bounded number of context-switches in each cluster. These context-switches allow jumps in the control states, i.e., the computation of the thread is a sequence of computation segments going from a control state q_1 to another state q'_1, then jumping to state q_2 (assuming that some other threads will bring the control from q'_1 to q_2) and going to another state q'_2, and so on. Therefore, given a bound K on the scope, the summary of a cluster is defined to be the set of all finite sequences of triples of the form $(q_1, f_1, q'_1)(q_2, f_2, q'_2) \cdots (q_k, f_k, q'_k)$, called *cluster interfaces*, where $k \leq K$, the q_i and q'_i are control states in the product of M and B, and the f_i's are booleans informing about whether a repeated state of B has been encountered in the computation segment from q_i to q'_i. Clearly, a cluster summary is a finite set since interfaces have a bounded size. We show that it is possible to compute this set by solving reachability queries in a pushdown system. Then, the idea is to build for each thread an interface automaton that recognizes all the possible repetitions of its cluster interfaces and to put them together so that only consistent interleavings of these stack-wise interfaces are recognized.

Related Work: As mentioned earlier, context-bounding has been introduced by Qadeer and Rehof in [31] for detecting safety bugs in shared memory concurrent programs. Several extensions of context-bounding to other classes of programs and efficient procedures for context-bounded analysis have been proposed in [14,15,27,3,21,23,22,4,5]. Other bounding concepts allowing for larger/incomparable coverage of the explored behaviors have been proposed in [20,18,17,13,24]. In particular, scope-bounding has been introduced by La Torre and Napoli in [24].

In this line of work, the focus has been on checking safety properties. In [5] the model checking problem of ordered multi-pushdown systems (OMPDS) is shown to be decidable and 2ETIME-complete. It is not clear how this decidability result is related to the one we prove in this paper. Simulating scope-bounded computations using the order restriction on stacks does not seem to be possible. Moreover, the complexity of the model checking problem for OMPDS is clearly higher than for scope-bounded MPDS which, as shown here, is the same as for (single-stack) pushdown systems.

A procedure for detecting termination bugs using *repeated* context-bounded reachability has been proposed in [6]. The idea there is to focus on checking the existence of (fair) context-bounded *ultimately periodic* non-terminating computations, i.e., infinite computations of the form uv^ω where u and v are finite computation segments with a bounded number of context-switches. The model-checking algorithms we give in this paper are more general than the procedure in [6] since context-bounded clearly scope-bounded and ω-regular behaviors are more general than ultimately periodic computations.

La Torre and Parlato have proved in [26] that the satisfiability of MSO over *finite* computations of multi-pushdown systems is decidable under scope-bounding. The proof is by showing that the graphs of each of such computations (seen as a multi-nested word) have a bounded tree-width. This result implies that model checking regular properties (over finite-computations) for these systems is decidable. However, as the satisfiability problem of MSO is nonelementary, the result in [26] does not provide a practical algorithm for the model checking problem as we do.

Recently, independently and simultaneously, La Torre and Napoli have established in [25] the EXPTIME-completeness of model checking MPDS's against MultiCaRet, an extension of the CaRet logic [1] which is itself an extension of LTL from words to nested words. The class of properties expressible in MultiCaRet is incomparable with the class of ω-regular properties. Although the approach adopted in [25] is technically different from ours, yet the essence of the proof is close to the one provided here.

2 Preliminaries

In this section, we fix some basic definitions and notations. We assume here that the reader is familiar with language and automata theory in general.

Notations: Let $\mathbb{N}$ denote the non-negative integers, and $\mathbb{N}_\omega$ denote the set $\mathbb{N} \cup \{\omega\}$ (ω represents the first limit ordinal). For every $i, j \in \mathbb{N}_\omega$ such that $i \leq j$, we use $[i..j]$ to denote the set $\{k \in \mathbb{N}_\omega \mid i \leq k \leq j\}$.

Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^ω) the set of all finite (resp. infinite) words over Σ, and by ϵ the empty word. Let u be a word over Σ. The length of u is denoted by $|u|$; we assume that $|\epsilon| = 0$ and $|u| = \omega$ if $u \in \Sigma^\omega$. For every $j \in [1..|u|]$, we use $u(j)$ to denote the j^{th} letter of u.

Let S be a set of (possibly infinite) words over the alphabet Σ and let $w \in \Sigma^*$ be a word. We define $w.S = \{w.u \mid u \in S\}$. We define the *shuffle* over two words inductively as $\sqcup\!\sqcup(\epsilon, w) = \sqcup\!\sqcup(w, \epsilon) = \{w\}$ and $\sqcup\!\sqcup(a.u', b.v') = a.(\sqcup\!\sqcup(u', b.v') \cup b.(\sqcup\!\sqcup(a.u', v')$. Given two sets (possibly infinite) of words S_1 and S_2 (over Σ), we define shuffle over these sets as $\sqcup\!\sqcup(S_1, S_2) = \bigcup_{u \in S_1, v \in S_2} \sqcup\!\sqcup(u, v)$. The *shuffle* operator for multiple sets can be extended analogously.

Finite-State Automata: A *finite-state automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ where: (1) Q is the finite non-empty set of states, (2) Σ is the input alphabet, (3) $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times Q)$ is the transition relation, (4) $I \subseteq Q$ is the set of initial states, and (5) $F \subseteq Q$ is the set of final states. The language of finite words accepted (or recognized) by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. We may also interpret the set F as a Büchi acceptance condition, and we denote by $L^\omega(\mathcal{A})$ the language of infinite words accepted by $\mathcal{A}$. The size of $\mathcal{A}$ is defined by $|\mathcal{A}| = (|Q| + |\Sigma|)$.

Context-Free Grammars: A *context-free grammar* (CFG) G is a tuple $(\mathcal{X}, \Sigma, R, S)$ where $\mathcal{X}$ is a finite non-empty set of *variables* (or *nonterminals*), Σ is an alphabet of *terminals*, $R \subseteq \big(\mathcal{X} \times (\mathcal{X}^2 \cup \Sigma)\big) \cup (S \times \{\epsilon\})$ a finite set of *productions* (the production (X, w) may also be denoted by $X \rightarrow w$), and $S \in \mathcal{X}$ is a starting variable. The size of G is defined by $|G| = (|\mathcal{X}| + |\Sigma|)$. Observe that the form of the productions is restricted, but it has been shown in [28] that every CFG can be transformed, in polynomial time, into an equivalent grammar of this form.

Given strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we say $u \Rightarrow_G v$ if there exists a production $(X, w) \in R$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$.

We use $\Rightarrow_G^*$ for the reflexive transitive closure of $\Rightarrow$. We define the context-free language generated by $L(G)$ as $\{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

Let $k \in \mathbb{N}$. A derivation α given by $\alpha \overset{\text{def}}{=} \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \cdots \Rightarrow_G \alpha_n$ is *k-bounded* if $|\alpha_i| \leq k$ for all $i \in [1..n]$. We denote by $L^{(k)}(G)$ the subset of $L(G)$ such that for every $w \in L^{(k)}(G)$ there exists a k bounded derivation $S \Rightarrow_G^* w$. We call $L^{(k)}(G)$ the *k-bounded approximation* of $L(G)$.

Lemma 1. *Given a context-free grammar G and $k \in \mathbb{N}$, then it is possible to construct a finite state automaton $\mathcal{A}$ s.t. $L(\mathcal{A}) = L^{(k)}(G)$, with $|\mathcal{A}| = O(|G|^k)$.*

Pushdown Automata: A pushdown automaton is defined by a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where: (1) Q is the finite non-empty set of states, (2) Σ is the input alphabet, (3) Γ is the stack alphabet, (4) Δ is the finite set of transition rules of the form $(q, u) \xrightarrow{a} (q', u')$ where $q, q' \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $u, u' \in \Gamma^*$ such that $|u| + |u'| \leq 1$, (5) $I \subseteq Q$ is the set of initial states, and (6) $F \subseteq Q$ is the set of final states. The size of $\mathcal{P}$ is defined by $|\mathcal{P}| = (|Q| + |\Sigma| + |\Gamma|)$

A *configuration* of $\mathcal{P}$ is a tuple (q, σ, w) where $q \in Q$ is the current state, $\sigma \in \Sigma^*$ is the input word, and $w \in \Gamma^*$ is the stack content. We define the binary relation $\Rightarrow_\mathcal{P}$ between configurations as follows: $(q, a\sigma, uw) \Rightarrow_\mathcal{P} (q', \sigma, u'w)$ iff $(q, u) \xrightarrow{a} (q', u')$. The transition relation $\Rightarrow_\mathcal{P}^*$ is the reflexive transitive closure of the binary relation $\Rightarrow_\mathcal{P}$.

The language accepted by $\mathcal{P}$ is defined by the set of finite words $L(\mathcal{P}) = \{\sigma \in \Sigma^* \mid (q_{\mathsf{init}}, \sigma, \epsilon) \Rightarrow_\mathcal{P}^* (q_{\mathsf{final}}, \epsilon, \epsilon)\}$ where $q_{\mathsf{init}} \in I$ and $q_{\mathsf{final}} \in F$. $L(\mathcal{P})$ is a context-free language, and conversely, every context-free language can be defined as the language of some pushdown automaton. In fact, it is well-known that for every pushdown automaton $\mathcal{P}$, it is possible to construct, in polynomial time in the size of $\mathcal{P}$, a context-free grammar G such that $L(G) = L(\mathcal{P})$ [19].

Similar to the case of finite state automata, even for pushdown automata we can interpret the acceptance set F as a Büchi acceptance condition and we denote by $L^\omega(\mathcal{P})$ the language of infinite words accepted by $\mathcal{P}$.

3 Multi-Pushdown Systems

Multi-pushdown systems (or *MPDS* for short) are generalizations of pushdown systems with multiple stacks. The kinds of transitions performed by an MPDS are: (i) pushing a symbol into one stack, (ii) popping a symbol from one stack, or (iii) an internal action that changes its states while leaving the stacks unchanged.

Definition 2 (Multi-PushDown Systems). *A* multi-pushdown system *(MPDS) is a tuple $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ where $n \geq 1$ is the number of stacks, Q is the finite set of* states*, Γ is the* stack alphabet*, $\Delta \subseteq \big(Q \times [1..n] \times Q\big) \cup \big(Q \times [1..n] \times Q \times \Gamma\big) \cup \big(Q \times [1..n] \times \Gamma \times Q\big)$ is the* transition relation*, and q_{init} is the initial state. The size of $\mathcal{M}$ is defined by $|\mathcal{M}| = (|n| + |Q| + |\Gamma|)$.*

Let $q, q' \in Q$ be two states, $\gamma \in \Gamma$ a stack symbol, and $i \in [1..n]$ a stack index. A transition of the form (q, i, q') is an internal operation of the i-th stack that moves the state from q to q' while keeping the stack contents unchanged. A transition of the form (q, i, q', γ) corresponds to a push operation that changes the state from q to q' and adds the symbol γ to the top of the i-th stack of $\mathcal{M}$. Finally, a transition of the form (q, i, γ, q') corresponds to a pop operation that moves the state from q to q' while removing γ from the i-th stack of $\mathcal{M}$.

A configuration of $\mathcal{M}$ is a $(n+1)$-tuple $c = (q, w_1, \ldots, w_n)$ where $q \in Q$ is a state and for every $i \in [1..n]$, $w_i \in \Gamma^*$ is the content of the i-th stack of $\mathcal{M}$. We use $\mathit{State}(c)$ and $\mathit{Stack}_i(c)$, with $1 \leq i \leq n$, to denote the state q and the stack content w_i respectively. The initial configuration c_{init} is $(q_{\mathsf{init}}, \epsilon, \cdots, \epsilon)$. The set of all configurations of $\mathcal{M}$ is denoted by $\mathit{Conf}(\mathcal{M})$.

We define the transition relation $\rightarrow_{\mathcal{M}}$ on the set of configurations as follows. For configurations $c = (q, w_1, \ldots, w_n)$ and $c' = (q', w'_1, \ldots, w'_n)$ and a transition $t \in \Delta$, we write $c \xrightarrow{t}_{\mathcal{M}} c'$ iff one of the following properties is satisfied:

- **Internal operation:** $t = (q, i, q')$ for some $i \in [1..n]$ and $w'_j = w_j$ for all $j \in [1..n]$.
- **Push operation:** $t = (q, i, q', \gamma)$ for some $\gamma \in \Gamma$ and $i \in [1..n]$, $w'_i = \gamma \cdot w_i$, and $w'_j = w_j$ for all $j \in ([1..n] \setminus \{i\})$.
- **Pop operation:** $t = (q, i, \gamma, q')$ for some $\gamma \in \Gamma$ and $i \in [1..n]$, $w_i = \gamma \cdot w'_i$, and $w'_j = w_j$ for all $j \in ([1..n] \setminus \{i\})$.

A *computation* π of $\mathcal{M}$ from a configuration c is a (possibly infinite) sequence of the form $c_0 t_1 c_1 t_2 \cdots$ such that $c_0 = c$ and $c_{i-1} \xrightarrow{t_i} c_i$ for all $1 \leq i \leq |t_1 t_2 \cdots|$. We use $\mathit{conf}(\pi)$, $\mathit{state}(\pi)$, and $\mathit{trace}(\pi)$ to denote the sequences $c_0 c_1 \cdots$, $\mathit{State}(c_0)\mathit{State}(c_1)\cdots$, and $t_1 t_2 \cdots$ respectively.

Given a finite computation $\pi_1 = c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m$ and a (possibly infinite) computation $\pi_2 = c_{m+1} t_{m+2} c_{m+2} t_{m+3} \cdots$, π_1 and π_2 are said to be *compatible* if $c_m = c_{m+1}$. Then, we write $\pi_1 \bullet \pi_2$ to denote the computation $\pi \overset{\text{def}}{=} c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m t_{m+2} c_{m+2} t_{m+3} \cdots$.

In general, multi-pushdown systems are Turing powerful resulting in the undecidability of all basic decision problems [32]. However, it is possible to obtain decidability for some problems, such a control state reachability, by restricting the allowed set of behaviours [31,20,2,16,24]. We are concerned with the *bounded-scope* restriction introduced in [24] and we describe this next.

Contexts: A *context* of a stack $i \in [1..n]$ is a computation of the form $\pi = c_0 t_1 c_1 t_2 \cdots$ in which $t_j \in \Delta_i \overset{\text{def}}{=} \big(Q \times \{i\} \times Q\big) \cup \big(Q \times \{i\} \times Q \times \Gamma\big) \cup \big(Q \times \{i\} \times \Gamma \times Q\big)$ for all $j \in [1..|\mathit{trace}(\pi)|]$. We define $\mathit{initial}(\pi)$ to be the configuration at the beginning of π (i.e., $\mathit{initial}(\pi) = c_0$). Furthermore, for any finite context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$, we use $\mathit{target}(\pi)$ to denote the configuration at the end of the context π (i.e., $\mathit{target}(\pi) = c_m$).

Context Decomposition: Every computation can be seen as the concatenation of a sequence of contexts $\pi_1 \bullet \pi_2 \bullet \ldots$. In particular, every computation π can be written uniquely as a sequence $\pi_1 \bullet \pi_2 \bullet \ldots$ such that for all i, π_i and π_{i+1} are not contexts of the same stack. We refer to this as the *context decomposition* of π.

For every $i \in [1..n]$ and two contexts π_1 and π_2, we write $\pi_1 \bullet_i \pi_2$ to denote that $\mathit{Stack}_i(\mathit{initial}(\pi_2)) = \mathit{Stack}_i(\mathit{target}(\pi_1))$. This notation is extended in the straightforward manner to sequence of contexts. Observe that if $\pi = \pi_1 \bullet \pi_2 \bullet \ldots$, and each π_i is a context and if $i_1 < i_2 < \ldots$ are all the indices j such that π_j is a context of the stack i then, $\pi_{i_1} \bullet_i \pi_{i_2} \bullet_i \ldots$.

Cluster: A *cluster* ρ of a stack $i \in [1..n]$ of size $j \in \mathbb{N}$ is a sequence of finite contexts $\pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_j$ of the stack i such that $\mathit{Stack}_i(\mathit{initial}(\pi_1)) = \mathit{Stack}_i(\mathit{target}(\pi_j)) = \epsilon$ (i.e., the stack i at the beginning of the context π_1 and at the end of the context π_j is empty).

Context-bounded Computations: Given $k \in \mathbb{N}$, a computation $\pi = c_0 t_1 c_1 t_2 \cdots$ is said to be k *context-bounded* if it has a context decomposition $\pi_1, \pi_2, \ldots, \pi_l$ consisting of at most k contexts (i.e. $l \leq k$). Thus in a context-bounded computation the number of switches between stacks is bounded by $(k-1)$.

Scope-Bounded Computations: Intuitively, in a scope bounded computation, any value that is pushed in a stack i is removed within k contexts involving this stack i. Equivalently, if we consider a point in the computation where a stack is empty (this for instance is true at the initial configuration), and if there are at least k contexts involving this stack after this point, then this stack empties within these k contexts of this stack. We require that in any infinite scope-bounded run one of two things must happen – either there are infinitely many context switches and thus the contexts belonging to each stack is just a concatenation of clusters of size $\leq k$ or it has only a finite number of context switches, with the last context being an infinite context and in this case we require that the contexts belonging to all the other stacks is just a concatenation of clusters. (Our definition is a bit stronger than required in the second case, but it makes the presentation simpler without any loss of generality, see the discussion below.) The formal definition is as follows:

Let π be an infinite computation. We say that it is *k-scope-bounded* if it can be written as a concatenation $\pi_1 \bullet \pi_2 \bullet \ldots$ of contexts (observe that for all j, π_j and π_{j+1} could be contexts of the same stack) in such a way that if $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \pi^i_{i_3} \bullet_i \cdots$ (with $i_1 < i_2 < i_3 < \cdots$) is the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$, then, one of the following cases holds:

- There is a (possibly infinite) sequence $\rho_i = \rho^i_1 \bullet_i \rho^i_2 \bullet_i \cdots$, for each $i \in [1..n]$, of clusters of size at most k such that $\sigma_i = \rho_i$ for all $i \in [1..n]$. Moreover, there is at least two distinct indices $i_1, i_2 \in [1..n]$ such that σ_{i_1} and σ_{i_2} are infinite (and all the stacks for which σ_i is finite are empty beyond a point).
- There is an index $i \in [1..n]$, a finite sequence $\rho_j = \rho^j_1 \bullet_i \rho^j_2 \bullet_i \cdots$ of clusters of size at most k for all $j \in [1..n]$, and a finite sequence $\sigma'_i = {\pi'}_1 \bullet_i {\pi'}_2 \bullet_i \cdots \bullet_i \pi'_\ell$ of contexts, with π'_ℓ is an infinite context and $\ell < k$, such that $\sigma_i = \rho_i \bullet_i \sigma'_i$ and $\sigma_r = \rho_r$ for all $r \in ([1..n] \setminus \{i\})$. This corresponds to the case where beyond some point all stacks except i are empty and there is a final infinite context involving the stack i.

Observe that we adopt in this paper a slightly different definition of scope-bounded computations than the one given in [24,26] where each symbol that is stored in a stack can be only popped iff it has been pushed within a bounded number of context switches performed by this stack. In our definition, we consider that a stack symbol that will never be popped in the context of [24,26] will not be pushed in our case. Thus, each pushed symbol in to the stack should be removed within k context-switches performed by this stack. Furthermore, we can easily show that the linear-time model checking problem for MPDS under scope-bounding w.r.t. the definition given in [24,26] can be easily reduced, in linear time, to the same problem for MPDS under scope-bounding w.r.t. the definition considered in this paper.

Moreover, in our definition, we consider that each finite "stack computation" consists of a finite sequence of clusters. This is only done for the purpose of simplifying the presentation and our results can be easily extended to the general case where each finite stack computation is a sequence of clusters followed by at most k contexts.

4 Scope-Bounded Repeated Reachability for MPDS

We present in this section a procedure for solving the repeated reachability problem for MPDS under scope-bounding. Let $k \in \mathbb{N}$ be a natural number. The *k-scope bounded repeated reachability problem* is to determine for any given MPDS $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ and a set of states $F \subseteq Q$, whether there is an infinite k-scope-bounded computation of $\mathcal{M}$ starting at the initial configuration that visits some state in F infinitely often (where a computation $\pi = c_0 t_1 c_1 t_2 \cdots$ visits a state q infinitely often if and only if for every $j \in \mathbb{N}$ there is an index $\ell > j$ such that $State(c_\ell) = q$.) In the following, we show that this problem can be reduced to the emptiness problem for Büchi pushdown automata.

Theorem 3. *Let $k \in \mathbb{N}$ be a natural number, $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ an MPDS, and $F \subseteq Q$ a set of states. Then it is possible to construct a Büchi pushdown automaton $\mathcal{P}$ such that M has a k-scope bounded computation that visits infinitely often a state in F if and only if the language $L^\omega(\mathcal{P})$ is not empty. Moreover, the size of $\mathcal{P}$ is $O(|F|(k|\mathcal{M}|)^{dkn})$ for some constant d.*

To prove Theorem 3, we assume w.l.o.g. that F contains a single state q_{final}. The key idea behind the proof is the following: Pick any scope-bounded computation and fix a stack i. This stack starts as empty and repeatedly turns empty (we deal with the case where some stack has an infinite context later). We *summarize* the execution pertaining to this stack, between two consecutive points where it is empty, as a sequence of triples of the form $(q_1, f_1, q_1')(q_2, f_2, q_2') \dots (q_l, f_l, q_l')$ for some $l < k$. This represents an execution where the MPDS executed a context of stack i starting at state q_1 to reach q_1' with a stack content, say γ_1, at which point a context-switch occurred. At the next context involving stack i, it resumed at state q_2 (and stack content γ_1) and switched context at the state q_2' and so on and eventually the l-th context involving stack i reached a state q_l' with the empty stack. The f_i component records if that context involved visits the state q_{final}. A full computation can be summarized w.r.t. to the stack i as a sequence of such summaries. We then show that using standard ideas from pushdown automata and CFGs we can compute an over-approximation of the set of summaries of each stack arising from executions of the MPDS. It is an over-approximation as it is done without verifying whether the *gaps* from q'_1 to q_2, q'_2 to q_3 and so on, can be filled out by a real execution involving the other stacks in a scope bounded manner. We then show that we may put together these over-approximations and in doing check for consistency across stacks and hence produce only combined summaries (for all stacks) that arise from real scope-bounded computations. The case where there is a final infinite context is solved similarly, except that the summarization is broken up into two parts, an initial part computed using ideas described above and the last part is the execution of a single stack pushdown system.

Before we present the details, we introduce some notations and definitions that will be useful. For any finite context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$, we can associate a tuple $\mathit{Interface}(\pi) = (q, f, q')$ of the pair of states encountered at the beginning and end of the context π (i.e., $q = \mathit{State}(c_0)$ and $q' = \mathit{State}(c_m)$), and a flag f indicating if the final state q_{final} was encountered along the context π (i.e., $f = 1$ if there is an index $r \in [1..m]$ such that $\mathit{State}(c_r) = q_{\mathsf{final}}$, otherwise $f = 0$).

For any given infinite context $\pi = c_0 t_1 c_1 t_2 \cdots$, we define $\mathit{Interface}(\pi)$ as the sequence of tuple of the form $(q_1, f_1, q'_1)(q_2, f_2, q'_2) \cdots$ such that for every $j \in \mathbb{N}$, we have $\mathit{State}(c_j) = q_{j+1}$, $\mathit{State}(c_{j+1}) = q'_{j+1}$, and if $q_j = q_{\mathsf{final}}$ then $f_{j+1} = 1$, otherwise $f_{j+1} = 0$.

Let $\rho = \pi_1 \bullet_i \pi_2 \bullet_i \cdots$ be a sequence of contexts for some $i \in [1..n]$, then we can extend the definition of context interfaces to sequence of contexts as follows: $\mathit{Interface}(\rho) = \mathit{Interface}(\pi_1)\mathit{Interface}(\pi_2) \cdots$. The function *Interface* is also extended in straightforward manner to clusters and sequences of clusters and contexts.

Let $w = (q_1, f_1, q'_1)(q_2, f_2, q'_2) \cdots$ be an infinite word over the interface alphabet $Q \times \{0, 1\} \times Q$. The word (or interface) w is said to be *well-formed* if $q_1 = q_{\mathsf{init}}$ and $q'_j = q_{j+1}$ for all $j > 1$. Moreover, the interface w visits the state q_{final} infinitely often if for every natural number $j \in \mathbb{N}$, there is a natural number $i > j$ such that $f_i = 1$.

Let ρ be a k-scope bounded computation that visits the state q_{final} infinitely often. We can assume that ρ is of the form $\pi_1 \bullet \pi_2 \bullet \pi_3 \bullet \cdots$ where each π_j, with $j \in \mathbb{N}$, is a stack context. Furthermore, we assume w.l.o.g. that, if at all, only the first stack can have an infinite context in π. Then, let $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \pi^i_{i_3} \bullet_i \cdots$ (with $i_1 < i_2 < i_3 < \cdots$) be the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$.

For any stack $i \in [2..n]$, we recall that σ_i is a sequence of clusters $\rho^i_1 \bullet_i \rho^i_2 \bullet_i \rho^i_3 \bullet_i \cdots$. Moreover, there is a sequence of clusters $\rho_1 = \rho^1_1 \bullet_1 \rho^1_2 \bullet_1 \cdots$ such that σ_1 is one of the following forms: (1) $\sigma_1 = \rho_1$ and each σ_i (with $i > 1$) can be an infinite sequence, or (2) all σ_i (with $i > 1$) are finite and there is a finite sequence $\sigma'_1 = \pi'_1 \bullet_1 \pi'_2 \bullet_1 \cdots \bullet_1 \pi'_\ell$ of contexts, with π'_ℓ is an infinite context and $\ell < k$, such that $\sigma_1 = \rho_1 \bullet_1 \sigma'_1$.

Case 1: Let us consider the case where $\sigma_1 = \rho_1$. For every stack $i \in [1..n]$, we associate the interface $\mathit{Interface}(\sigma_i)$ to the sequence σ_i. Then, it is easy to see that, in this case, there is a well-formed word in $\sqcup\!\sqcup(\{\mathit{Interface}(\sigma_1)\}, \ldots, \{\mathit{Interface}(\sigma_n)\})$ which visits q_{final} infinitely often.

Case 2: Let us consider the case where $\sigma_1 = \rho_1 \bullet_1 \sigma'_1$. Then, there is a finite word w in $\sqcup\!\sqcup(\{\mathit{Interface}(\rho_1 \bullet_1 \pi'_1 \bullet_1 \cdots \bullet_1 \pi'_{\ell-1})\}, \{\mathit{Interface}(\sigma_2)\}, \ldots, \{\mathit{Interface}(\sigma_n)\})$ such that $w \cdot \mathit{Interface}(\pi'_\ell)$ is well-formed and visits the state q_{final} infinitely often.

In general, we can show:

Lemma 4. *M has a k-scope bounded computation that visits infinitely often the state q_{final} if and only if one of the following two cases holds:*

- *For every stack $i \in [1..n]$, there is a (possibly infinite) sequence σ_i of clusters of the stack i such that there is a well-formed word in $\sqcup\!\sqcup(\{\mathit{Interface}(\sigma_1)\}, \ldots, \{\mathit{Interface}(\sigma_n)\})$ which visits q_{final} infinitely often.*
- *There is a finite sequence σ_i of clusters of the stack i for all $i \in [1..n]$, a finite sequence of clusters ρ_1 of the first stack, and a finite sequence $\sigma'_1 = \pi'_1 \bullet_1 \pi'_2 \bullet_1 \cdots \bullet_1 \pi'_\ell$ of contexts, with π'_ℓ an infinite context and $\ell < k$, s.t. there is a finite word w in $\sqcup\!\sqcup(\{\mathit{Interface}(\rho_1 \bullet_1 {\pi'}_1 \bullet_1 \cdots \bullet_1 \pi'_{\ell-1})\}, \{\mathit{Interface}(\sigma_2)\}, \ldots, \{\mathit{Interface}(\sigma_n)\})$ such that the word $\big(w \cdot \mathit{Interface}(\pi'_\ell)\big)$ is well-formed and visits q_{final} infinitely often.*

Now, we can show that checking both of these two cases can be reduced to the emptiness problem for Büchi (pushdown) automata whose size is $O((k|\mathcal{M}|)^{dkn})$ for some constant d. In fact, this is an immediate consequence of the two following lemmas. Lemma 5 refers to the first case of Lemma 4 and shows that this case can be reduced to the emptiness problem of a Büchi automata.

Lemma 5. *The problem of checking whether there is a sequence σ_i of clusters of the stack i for all $i \in [1..n]$ such that there is a well-formed word in $\sqcup\!\sqcup(\{\mathit{Interface}(\sigma_1)\}, \ldots, \{\mathit{Interface}(\sigma_n)\})$ which visits q_{final} infinitely often can be reduced to the emptiness problem for a Büchi automaton whose size is $O((k|\mathcal{M}|)^{dk})$ for some constant d.*

Proof (sketch). First of all, the set $L_i^k(\mathcal{M})$ of all the finite words of the form $\mathit{Interface}(\rho)$ where ρ is a cluster of size at most k of a stack $i \in [1..n]$ can be seen as the language of a pushdown automaton $\mathcal{P}_i$. This is a finite language of words of length $\leq k$ and hence regular, but by suitably modifying the given MPDS we can construct an explicit pushdown automaton recognizing these words. Moreover the size of $\mathcal{P}_i$ is polynomial in $k|\mathcal{M}|$. Now, we can use Lemma 1 and the equivalence between context-free grammars and pushdown automata [28,19], to show that it is possible to construct a finite state automaton A_i such that $L(A_i) = L_i^k(\mathcal{M})$, where in the worst case the size of A_i is $O((k|\mathcal{M}|)^{dk})$ for some constant d. As an immediate consequence of this result, we can show that the set of possible interfaces generated by a sequence of clusters of the stack i can be characterized by a Büchi automaton $\mathcal{B}_i$ (constructed by a special concatenation operation of the finite state automaton A_i). Observe that the size of the Büchi automaton $\mathcal{B}_i$ is $O((k|\mathcal{M}|)^{dk})$ for some constant d. Finally, we can use standard automata construction, to show that we can construct a Büchi automaton $\mathcal{B}$ that accepts all the well-formed words in $\sqcup\!\sqcup(\{Interface(\sigma_1)\}, \ldots, \{Interface(\sigma_n)\})$, by ensuring that adjacent letters in the shuffle being generated are compatible, and also verify that the resulting sequence visits q_{final} infinitely often. Moreover, the size of $\mathcal{B}$ is akin to the size of the product of $\mathcal{B}_1, \ldots, \mathcal{B}_n$ and thus is $O((k|\mathcal{M}|)^{dkn})$ for some constant d. □

The following Lemma shows that the second case of Lemma 4 can be reduced to the emptiness problem of a Büchi pushdown automata.

Lemma 6. *The problem of checking whether there are finite sequences σ_i of clusters for each stack i with $i \in [2..n]$, a finite sequence ρ_1 of clusters of the first stack, and a finite sequence $\sigma'_1 = \pi'_1 \bullet_1 \pi'_2 \bullet_1 \cdots \bullet_1 \pi'_\ell$ of contexts, with π'_ℓ an infinite context and $\ell < k$, such that there is a finite word w in $\sqcup\!\sqcup(\{Interface(\rho_1 \bullet_1 \pi'_1 \bullet_1 \cdots \bullet_1 \pi'_{\ell-1})\}, \{Interface(\sigma_2)\}, \ldots, \{Interface(\sigma_n)\})$ such that $w \cdot Interface(\pi'_\ell)$ is well-formed and visits infinitely often the state q_{final} can be reduced to the emptiness problem for a Büchi pushdown automaton whose size is $O((k|\mathcal{M}|)^{dkn})$ for some constant d.*

Proof (sketch). As in the proof of Lemma 5 , it is possible to construct a finite state automaton A_i accepting exactly all the finite words of the form $Interface(\rho)$, where ρ is a cluster of size at most k of the stack $i \in [1..n]$. Observe that the size of A_i is $O((k|\mathcal{M}|)^{dk})$ for some constant d. On the other hand, we can construct a Büchi pushdown automaton $\mathcal{P}_1$ accepting the set of infinite words of the form $Interface(\pi'_1 \bullet_1 \pi'_2 \bullet_1 \cdots \bullet_1 \pi'_\ell)$, where $\pi'_1 \bullet_1 \cdots \bullet_1 \pi'_\ell$ is a sequence of contexts of the first stack such that π'_ℓ is an infinite context and $\ell < k$. Observe that the size of such a Büchi pushdown automaton $\mathcal{P}_1$ is polynomial in $k|\mathcal{M}|$. Finally, we can use standard automaton constructions, to show that we can construct a Büchi pushdown $\mathcal{P}$ that accepts all the well-formed words of the form $w \cdot Interface(\pi'_\ell)$ visiting the state q_{final} infinitely often where $w \in \sqcup\!\sqcup(\{Interface(\rho_1 \bullet_1 \pi'_1 \bullet_1 \cdots \bullet_1 \pi'_{\ell-1})\}, \{Interface(\sigma_2)\}, \ldots, \{Interface(\sigma_n)\})$ Moreover, $\mathcal{P}$ can be constructed from $\mathcal{P}_1$ and $A_2, \ldots, A_n$ such that the size of $\mathcal{P}$ is $O((k|\mathcal{M}|)^{dkn})$ for some constant d. □

5 Scope-Bounded Model Checking for MPDS

We consider in this section the linear-time model checking problem for MPDS's under scope-bounding. We consider that we are given ω-regular properties expressed in linear-time propositional temporal logic (LTL) [30] or in the linear-time propositional μ-calculus [34]. Let us fix a set of atomic propositions *Prop*, and let $k \in \mathbb{N}$ be a natural number. The *k-scope bounded model-checking* problem is the following: Given a formula φ (in LTL or in the linear-time μ-calculus) with atomic propositions from *Prop*, and a MPDS $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ along with a labeling function $\Lambda : Q \to 2^{Prop}$ associating to each state $q \in Q$ the set of atomic propositions that are true in it, check whether all infinite k-scope-bounded computations of $\mathcal{M}$ from the initial configuration c_{init} satisfy φ.

To solve this problem, we adopt an automata-based approach similar to the one used in [11,10] to solve the analogous problem for pushdown systems. We construct a Büchi automaton $\mathcal{B}_{\neg\varphi}$ over the alphabet 2^{Prop} accepting the negation of φ [36,35]. Then, we compute the product of the MPDS $\mathcal{M}$ and the Büchi automaton $\mathcal{B}_{\neg\varphi}$ to obtain a MPDS $\mathcal{M}_{\neg\varphi}$ with a Büchi accepting set of states F and leaving us with the task if any of its its k-scope bounded runs is accepting. We can then reduce our model-checking problem to the k-scope bounded repeated reachability problem for MPDSs, which, by Theorem 3, can be solved.

Theorem 7. *The problem of scope-bounded model checking ω-regular properties of multi-pushdown systems is EXPTIME-complete.*

The lower bound of Theorem 7 follows immediately from the fact that the model-checking problem for LTL and linear time μ-calculus for pushdown systems (i.e., MPDS with only one stack) are EXPTIME-complete [10].

For the upper bound, it is well known that, given a MPDS $\mathcal{M}$ and an ω-regular formula φ, it is possible to construct a MPDS $\mathcal{M}'$ and a set of repeating states F of $\mathcal{M}'$ such that the problem of scope-bounded model checking of $\mathcal{M}$ w.r.t. the formula φ is reducible to the k-scope-bounded repeated state reachability problem of a MPDS $\mathcal{M}'$ w.r.t. F. Moreover, the size of $\mathcal{M}'$ and F is exponential in the size of φ and polynomial in the size of $\mathcal{M}$ and k. Applying Theorem 3 to the MPDS $\mathcal{M}'$ and F, we obtain our complexity result.

6 Conclusion

We have shown that model checking linear-time properties (expressed as formulas of LTL or the linear-time propositional mu-calculus) for multithreaded programs with recursive procedures is decidable under scope-bounding, and furthermore, we have established that this problem is EXPTIME-complete. Therefore, model-checking of multithreaded programs with recursive procedures under scope-bounding is as hard as the same problem for (single-stack) pushdown systems. Our algorithm is in fact based on a reduction using a compositional reasoning about interfaces of threads at context-switch points, to a repeated reachability problem in a pushdown system.

Notice that, concerning the problem of checking whether a MPDS has a computation recognized by a given Büchi automaton (i.e., the emptiness of the intersection between MPDSs and Büchi automata), it is possible to use the same techniques we develop in this paper to show that it is PSPACE-complete. So, if we are given directly the automaton of the complement of the specification, we can check if the system has no bad scope-bounded behaviors in PSPACE. (Again, this is similar to the case of single-stack pushdown systems.)

Surprisingly, these positive results for linear time model-checking don't seem to carry over to branching-time properties [7].

Future work includes implementing our construction and using it in the verification of various kinds of properties, including termination and typical liveness properties such as response and persistence properties, over nontrivial examples of concurrent programs.

References

1. Alur, R., Etessami, K., Madhusudan, P.: A Temporal Logic of Nested Calls and Returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
2. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of Multi-pushdown Automata Is 2ETIME-Complete. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 121–133. Springer, Heidelberg (2008)
3. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
4. Atig, M.F.: From Multi to Single Stack Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 117–131. Springer, Heidelberg (2010)
5. Atig, M.F.: Global model checking of ordered multi-pushdown systems. In: FSTTCS. LIPIcs, vol. 8, pp. 216–227. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
6. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting Fair Non-termination in Multithreaded Programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012)
7. Atig, M.F., Bouajjani, A., Narayan Kumar, K., Saivasan, P.: Model checking branching-time properties of multi-pushdown systems is hard. CoRR abs/ 1205.6928 (2012)
8. Atig, M.F., Bouajjani, A., Touili, T.: Analyzing asynchronous programs with preemption. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008). LIPIcs, vol. 2, pp. 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)
9. Atig, M.F., Bouajjani, A., Touili, T.: On the Reachability Analysis of Acyclic Networks of Pushdown Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 356–371. Springer, Heidelberg (2008)
10. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model-Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
11. Bouajjani, A., Maler, O.: Reachability analysis of pushdown automata. In: Proc. Intern. Workshop on Verification of Infinite-State Systems, Infinity 1996 (1996)
12. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
13. Bouajjani, A., Emmi, M., Parlato, G.: On Sequentializing Concurrent Programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
14. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability Analysis of Multithreaded Software with Asynchronous Communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
15. Bouajjani, A., Fratani, S., Qadeer, S.: Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
16. Breveglieri, L., Cherubini, A., Citrini, C., Crespi Reghizzi, S.: Multi-push-down languages and grammars. Intl. Journal of Foundations of Computer Science 7(3), 253–292 (1996)

17. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL 2011: Proc. 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 411–422. ACM (2011)
18. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)
19. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
20. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: Proceedings of LICS, pp. 161–170. IEEE (2007)
21. La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
22. La Torre, S., Madhusudan, P., Parlato, G.: Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
23. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 211–222. ACM (2009)
24. La Torre, S., Napoli, M.: Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 203–218. Springer, Heidelberg (2011)
25. La Torre, S., Napoli, M.: A temporal logic for multi-threaded programs. In: IFIP TCS. IFIP. Springer (to appear, 2012)
26. La Torre, S., Parlato, G.: Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. Technical report, University of Southampton (March 2012)
27. Lal, A., Reps, T.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
28. Lange, M., Leiß, H.: To CNF or not to CNF ? An efficient yet presentable version of the CYK algorithm. Informatica Didactica 8 (2008-2010)
29. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM (2007)
30. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
31. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
32. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
33. Sen, K., Viswanathan, M.: Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
34. Vardi, M.Y.: A temporal fixpoint calculus. In: POPL, pp. 250–259 (1988)
35. Vardi, M.Y.: Alternating Automata and Program Verification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 471–485. Springer, Heidelberg (1995)
36. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, pp. 332–344. IEEE Computer Society (1986)

Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data[⋆]

Ahmed Bouajjani[1], Cezara Drăgoi[2], Constantin Enea[1], and Mihaela Sighireanu[1]

[1] Univ Paris Diderot, Sorbonne Paris Cité, LIAFA CNRS UMR 7089, France
{abou,cenea,sighirea}@liafa.univ-paris-diderot.fr
[2] IST Austria
cezarad@ist.ac.at

Abstract. We propose a logic-based framework for automated reasoning about sequential programs manipulating singly-linked lists and arrays with unbounded data. We introduce the logic SLAD, which allows combining shape constraints, written in a fragment of Separation Logic, with data and size constraints. We address the problem of checking the entailment between SLAD formulas, which is crucial in performing pre-post condition reasoning. Although this problem is undecidable in general for SLAD, we propose a sound and powerful procedure that is able to solve this problem for a large class of formulas, beyond the capabilities of existing techniques and tools. We prove that this procedure is complete, i.e., it is actually a decision procedure for this problem, for an important fragment of SLAD including known decidable logics. We implemented this procedure and shown its preciseness and its efficiency on a significant benchmark of formulas.

1 Introduction

Programs can manipulate dynamic data structures carrying data over infinite domains. Reasoning about the behaviors of such programs is a challenging problem due to the difficulty of representing (potentially infinite) sets of configurations, and of manipulating these representations for the analysis of the execution of program statements. For instance, pre/post-condition reasoning (checking the validity of Hoare triples) requires being able, given pre- and post-conditions ϕ and ψ, and a program statement τ, (1) to compute the strongest post-condition of executing τ starting from ϕ, denoted $\mathtt{post}(\tau,\phi)$, and (2) to check that it entails ψ. Moreover, showing that τ is executable starting from ϕ amounts in checking that $\mathtt{post}(\tau,\phi)$ is satisfiable (i.e., corresponds to a nonempty set of configurations). Therefore, an important issue is to investigate logic-based formalisms where pre/post conditions are expressible for the class of programs under interest, and for which it is possible to compute effectively (strongest) post-conditions, and to check satisfiability and entailment. (Notice that both of these problems have to be considered since it is not required that the logic is closed under negation.)

In this paper, we propose such a framework for the case of programs manipulating singly-linked lists and arrays with data. Several works have addressed this issue, proposing various decidable logics for reasoning about programs with data structures, e.g. [4,8,9,13,14,15,17]. Some of these logics [1,3,2,9,17] focus mainly on shape constraints

[⋆] This work has been partially supported by the French ANR project Veridyc.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 167–182, 2012.

assuming a bounded data domain, e.g. the Separation logic fragment in [3,9] for which the entailment problem has a polynomial time complexity. The works in [4,8,13,14,15] focus on reasoning about programs manipulating data structures with unbounded data. They introduce decidability results concerning the satisfiability problem for logics that describe the shape and the data of heap configurations. The formulas in these logics have a quantifier prefix of the form $\exists^*\forall^*$. They are different w.r.t. the type of the variables which are quantified or the basic predicates which are allowed. The validity of an entailment $\phi_1 \Rightarrow \phi_2$ is reduced to the unsatisfiability of $\phi_1 \wedge \neg\phi_2$. It can be performed only between boolean combinations of formulas with quantifier prefixes $\exists^*$ or $\forall^*$.

We define a logic, called SLAD, allowing to express the specifications of all common programs manipulating lists and arrays. This logic combines shape constraints, written in Separation Logic [17], and universally-quantified first-order formulas expressing constraints on the size and the values of the data sequences associated with arrays or sharing-free list segments in the heap. The logic is parametrized by a logic on data; for simplicity, we suppose that data is of integer type and that the logic on data is Presburger arithmetics. The Separation logic formulas are in a fragment that extends the one in [3,9] with existential quantification and disjunction. Existential quantification is useful to describe for instance lasso-shaped lists, and disjunction is crucial for specifications that involve data. For example, the post-condition of a loop that searches an integer *val* in a list pointed to by *x*, where the variable *xi* is used to traverse the list, states that: either *val* is not in the list, e.g., $xi = \texttt{NULL}$, or *val* is in the list and *xi* points to the corresponding element, e.g., $xi \neq \texttt{NULL}$, *xi* is reachable from *x*, and $xi \rightarrow data = val$.

In general, SLAD formulas have quantifier prefixes of the form $\exists^*\forall^*$ (e.g., the formula in Fig.1 that describes a sorted lasso-shaped list). The validity of entailments between such formulas can not be reduced to the satisfiability of an $\exists^*\forall^*$ formula and thus, it can not be decided using approaches like in [4,8,13,14,15]. Also, in many cases, relevant program assertions are beyond the identified decidable fragments (e.g., relations between the values of the data fields and the size of the allocated list). We define a procedure for checking entailments between SLAD formulas, which exploits their syntax. The entailment between shape constraints is checked using a slightly modified version of the decision procedure for Separation Logic in [9]. If this entailment holds, the procedure reduces the entailment between the data constraints to the validity of a formula in the data logic. The novelty of this procedure is that (1) it does not reduce entailment checking in SLAD to satisfiability checking in the same logic, (2) it is applied to $\exists^*\forall^*$ formulas, and (3) it is sound when applied to *any* SLAD formulas and complete for a relevant fragment of SLAD. Note that the same procedure is also a sound decision procedure for unsatisfiability. The fragment of SLAD, called $\mathsf{SLAD}_{\leq}$, for which the unsatisfiability check is complete includes for instance, the logic APF [8] and the restriction of LISBQ [14] to singly-linked lists. The decision procedure for $\mathsf{SLAD}_{\leq}$ has the same complexity as the decision procedures in [8,14] (NP-complete, if we fix the number of universal quantifiers). The entailment problems for which our procedure is complete consider $\mathsf{SLAD}_{\leq}$ formulas (satisfying some syntactic restrictions) and are beyond the scope of all existing decision procedures that we are aware of.

Besides decidability results, we show that our approach deals efficiently with a variety of examples, including programs whose specifications are given by SLAD formulas that are not in $\mathsf{SLAD}_{\leq}$ and which can't be handled by existing tools. This is an important feature of our approach: it provides uniform and efficient techniques that are applicable to a large class of formulas, and that are complete for a significant set of formulas.

For the simplicity of the exposition, we begin by defining SLD, a logic on singly-linked lists, and then, in Section 6, we define its extension to arrays, SLAD.

2 Logic SLD, a Logic for Programs with Singly-Linked Lists

We introduce hereafter the class of programs with singly-linked lists considered in this paper and the syntax of *Singly-linked list with Data Logic* (SLD, for short), a logic to describe sets of program configurations (we relegate the definition of the formal semantics to the long version [5]). Then, we introduce fragments of SLD relevant for the results presented in the following sections. Finally, we give the properties of SLD relevant for program verification. In the following, *PVar* and *DVar* are disjoint sets of pointer resp. integer program variables. `NULL` is a distinguished pointer variable in *PVar*.

2.1 Programs with Singly-Linked Lists

We consider sequential programs manipulating singly-linked lists of the same type, i.e., pointer to a record called `list` formed of one recursive pointer field `next` and one data field `data` of integer type. However, the results presented in this paper work also for *lists with multiple data fields of different types*. As usual, the allocated memory (heap) is represented by a directed labeled graph where nodes represent list cells and edges represent values of the field `next`. The constant `NULL` is represented by a distinguished node $\sharp$ with no output edge. Nodes are labeled with values of the field `data` and pointer variables. For example, Fig. 1(a) represents a heap containing a lasso-shaped list pointed to by x. Formally, a program configuration consists of a directed graph representing the heap and a valuation of the integer program variables in *DVar*.

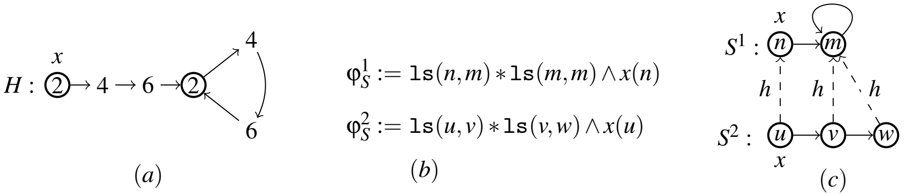
Definition 1 (Program configuration). *A* program configuration *is a tuple* $H = (V, E_n, \ell_P, \ell_D, D)$, *where (1)* V *is a finite set of nodes containing a distinguished node* $\sharp$, *(2)* $E_n : V \rightharpoonup V$ *is a partial function s.t.* $E_n(\sharp)$ *is undefined, (3)* $\ell_P : PVar \rightharpoonup V$ *is a partial function labeling nodes with pointer variables such that* $\ell_P(\mathtt{NULL}) = \sharp$, *(4)* $\ell_D : (V \setminus \{\sharp\}) \to \mathbb{Z}$ *is a function labeling nodes with integers, and (5)* $D : DVar \to \mathbb{Z}$ *is a valuation of the integer variables.* □

Definition 2 (Simple/Crucial node). *A node labeled with a pointer variable or which has at least 2 predecessors is called* crucial. *Otherwise, it's called a* simple node. □

For example, in the program configuration from Fig. 1(a) (*DVar* is empty) the circled nodes are crucial nodes and the other nodes are simple.

2.2 Syntax of SLD

The main features of SLD are introduced through an example. (A detailed presentation of SLD is given [5]). The heap in Fig. 1(a) consists of a lasso shaped list whose cyclic part is equal in size and values of the data fields to the non-cyclic part. The data in the cyclic part is strictly sorted. These properties are expressed in SLD as follows:



$$\varphi_S^1 := \mathtt{ls}(n,m) * \mathtt{ls}(m,m) \wedge x(n)$$

$$\varphi_S^2 := \mathtt{ls}(u,v) * \mathtt{ls}(v,w) \wedge x(u)$$

(b)

$$\begin{aligned}
\varphi_D^1 := {} & \mathtt{dt}(n) = \mathtt{dt}(m) \wedge \mathtt{len}(n) = \mathtt{len}(m) \\
& \wedge \forall y,y'.\ \underbrace{\left(0 < y < \mathtt{len}(n) \wedge 0 < y' < \mathtt{len}(m) \wedge y = y'\right)}_{G_{m,n}(y,y')} \Rightarrow n[y] = m[y'] \qquad \text{list equality} \\
& \wedge \forall y_1,y_2.\ \underbrace{0 < y_1 < y_2 < \mathtt{len}(m)}_{G_m(y_1,y_2)} \Rightarrow \mathtt{dt}(n) = \mathtt{dt}(m) < m[y_1] < m[y_2] \qquad \text{strict sortedness} \\
\varphi_D^2 := {} & \underbrace{\forall y_1,y_2.\ 0 < y_1 < y_2 < \mathtt{len}(v) \Rightarrow \mathtt{dt}(v) \leq v[y_1] \leq v[y_2]}_{sorted(v)} \qquad \text{sortedness}
\end{aligned}$$

(d)

Fig. 1. A program configuration (a) specified by two SLD formulas $\varphi^1 := \exists n,m.\ \left(\varphi_S^1 \wedge \varphi_D^1\right)$ and $\varphi^2 := \exists u,v,w.\ \left(\varphi_S^2 \wedge \varphi_D^2\right)$ given in (b) and (d) such that $\varphi^1 \vdash \varphi^2$. In (c), S^1 and S^2 are homomorphic SL-graphs representing the SLD graph formulas φ_S^1 resp. φ_S^2.

Shape Formulas: The shape of the heap is characterized using the formula φ_S^1 in Fig. 1(b), written in a fragment of Separation logic (SL) [17], where (1) n and m are *node variables* interpreted as nodes in the heap, (2) $\mathtt{ls}(n,m)$ denotes a *possibly-empty path* between the nodes denoted by n and m; such a path is called a *list segment*, (3) the *separating conjunction* $*$ expresses the fact that the two list segments are disjoint except for their end nodes, and (4) $x(n)$ says that the pointer variable x labels n.

Let *NVar* be a set of *node variables* interpreted as nodes in program configurations. The syntax of shape formulas is given in Fig. 2. The node $\sharp$ is represented by a constant with the same name in the syntax. For simplicity, we consider the intuitionistic model of Separation logic [17]: if a formula is true on a graph then it remains true for any extension of that graph with more nodes. Our techniques can be adapted to work also for the non-intuitionistic model. Inequalities are important to express properties like list disjointness. For example, the formula $\mathtt{ls}(n,u) * \mathtt{ls}(m,v) \wedge\ x(n) \wedge z(m)$ has as model a heap with only one list when m,u,v are interpreted in the same node, while $\mathtt{ls}(n,u) * \mathtt{ls}(m,v) \wedge\ x(n) \wedge z(m) \wedge\ n \neq m \wedge\ u \neq v$ specifies models that contain two disjoint lists.

The restriction Det from Fig. 2 and the omission of the "points to" predicate $u \mapsto v$ from Separation logic [9] (which denotes the fact that v is the successor of the node u) are adopted only for simplicity.

$x \in PVar$ program pointer variables

$n,m \in NVar$ node variables
$u,v \in NVar \cup \{\sharp\}$ node variables or $\sharp$

$\varphi_E ::= \mathtt{ls}(n,u) \mid \varphi_E * \varphi_E$
$\varphi_P ::= x(u) \mid m \neq u \mid \varphi_P \wedge \varphi_P$
$\varphi_S ::= \varphi_E \wedge \varphi_P,$ where φ_E satisfies Det

Det : "φ_E does not contain two predicates
$\mathtt{ls}(n,u)$ and $\mathtt{ls}(n,v)$
where n,u,v are pairwise distinct."

Fig. 2. Syntax of **shape formulas**

To simplify the presentation, a shape formula is represented by an SL-graph, which is a slight adaptation of the notion introduced in [9]. Each node of an SL-graph corresponds to a node variable, each edge corresponds to an `ls` predicate, and nodes are labeled by pointer variables. For example, the graphs S^1 and S^2 in Fig. 1(c) are the SL-graphs associated to the formulas φ_S^1 resp. φ_S^2 in Fig. 1(b).

Definition 3 (SL-graphs). *The* SL-graph *associated to a shape formula φ_S is either $\bot$ or a graph $S = (N \cup \{\sharp\}, E_\ell, \ell_P, E_d)$, where N is the set of node variables in φ_S, $E_\ell : N \rightharpoonup N \cup \{\sharp\}$ defines a set of directed edges by $E_\ell(n) = m$ iff $\mathtt{ls}(n,m)$ appears in φ_S, $\ell_P : PVar \rightharpoonup N \cup \{\sharp\}$ is defined by $\ell_P(x) = u$ iff $x(u)$ appears in φ_S, and $E_d \subseteq N \times N$ is a disequality relation which defines a set of undirected edges such that $(n,u) \in E_d$ iff $n \neq u$ appears in φ_S. A* path *in an SL-graph is formed only of directed edges.*

The SL-graph $\bot$ represents unsatisfiable shape formulas for which an SL-graph can not be built. For an SL-graph $S \neq \bot$, we use superscripts to denote their components, e.g., E_ℓ^S, and we note N_+^S for $N^S \cup \{\sharp\}$. Also, $Vars^S(n)$ denotes the set of pointer variables labeling n, $\{x \in PVar \mid \ell_P^S(x) = n\}$, and $Vars(S)$ denotes the set of pointer variables in S, $\mathtt{dom}(\ell_P)$. The notions of simple and crucial node are defined similarly for SL-graphs.

Sequence Formulas: Consider again the formula φ_S^1 in Fig. 1(b). The size and the data values of the list segments specified by φ_S^1 are characterized by φ_D^1 in Fig. 1(d), which is a first-order formula over integer sequences. The equality between the list segments corresponding to $\mathtt{ls}(n,m)$ and $\mathtt{ls}(m,m)$ is stated using (1) $\mathtt{len}(n) = \mathtt{len}(m)$, where $\mathtt{len}(n)$ ($\mathtt{len}(m)$) denotes the length, i.e., the number of edges, of the list segment associated to $\mathtt{ls}(n,m)$ ($\mathtt{ls}(m,m)$), (2) $\mathtt{dt}(n) = \mathtt{dt}(m)$, where $\mathtt{dt}(n)$ represents the integer labeling the node n, and (3) a universally quantified formula of the form $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$, where the variables in the set $\mathbf{y}$, called *position variables*, are interpreted as integers and $n[y]$ is interpreted as the integer labeling the node at distance y from n.

For every predicate $\mathtt{ls}(n,m)$ we call *integer sequence associated with n*, for short *sequence of n*, the element of $\mathbb{Z}^*$ obtained by concatenating the integers labeling all the nodes except the last one (i.e., the one represented by m) in the list segment corresponding to $\mathtt{ls}(n,m)$. A term $n[y]$ appears in $U(\mathbf{y})$ only if the guard $G(\mathbf{y})$ contains the constraint $0 < y < \mathtt{len}(n)$. This restriction is used to avoid undefined terms. (For

$N \cup \{n, n_y\} \subseteq NVar$ node variables $\quad d \in DVar$ integer variable

$\mathbf{y} \cup \{y\} \subseteq Pos$ position variables $\quad k \in \mathbb{Z}$ integer constant

Position terms:	E-terms:	U-terms:
$p ::= k \mid y \mid \mathtt{len}(n) \mid p+p$	$e ::= k \mid d \mid \mathtt{dt}(n) \mid \mathtt{len}(n) \mid e+e$	$t ::= e \mid y \mid n[y] \mid t+t$

Existential constraints: $E ::= e \leq e' \mid \neg E \mid E \wedge E \mid \exists d.\ E,$ where e and e' are E-terms

Constraints on positions: $C ::= p \leq p' \mid \neg C \mid C \wedge C$ where p and p' are position terms

Guards: $G(\mathbf{y}) ::= C \wedge \bigwedge_{y \in \mathbf{y}} 0 < y < \mathtt{len}(n_y),$

Data properties: $U(\mathbf{y}) ::= t \leq t' \mid \neg U \mid U \wedge U \mid \exists d.\ U,$ where t and t' are U-terms containing position variables from $\mathbf{y}$

Sequence formulas: $\varphi_D ::= E \mid \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y}) \mid \varphi_D \wedge \varphi_D$

Fig. 3. Syntax of **sequence formulas**

instance, if the length of the list segment starting in n equals 2 then the term $n[y]$ with y interpreted as 3 is undefined.) The strict sortedness is specified using a universal formula of the same form. Intuitively, $U(\mathbf{y})$ constrains the integers labeling a set of nodes determined by the guard $G(\mathbf{y})$. A formula $\rho = \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ is called a *guarded formula* and $G(\mathbf{y})$ is the *guard* of ρ. The syntax of sequence formulas is given in Fig. 3.

SLD Formulas: A formula in SLD is a disjunction of formulas of the form $\exists N_1.\ (\varphi_S^1 \wedge \varphi_D^1) * \cdots * \exists N_k.\ (\varphi_S^k \wedge \varphi_D^k)$, where each N_i is the set of all node variables in φ_S^i (which include all the node variables in φ_D^i). Note that such a formula is equivalent to $\exists N_1 \cup \cdots \cup N_k.\ (\varphi_S^1 * \cdots * \varphi_S^k \wedge \varphi_D^1 \wedge \cdots \wedge \varphi_D^k)$. W.l.o.g, in the following, we will consider only SLD formulas which are disjunctions of formulas of the form $\exists N.\ \varphi_S \wedge \varphi_D$.

The set of program configurations which are models of a formula ψ is denoted $[\psi]$.

2.3 Fragments of SLD

Succinct SLD Formulas: An SLD formula ψ is *succinct* if every SL-graph associated to a disjunct of ψ has no simple nodes. For example, $\varphi^1 := \exists n,m.\ \varphi_S^1 \wedge \varphi_D^1$ in Fig. 1 is succinct, but the formula φ^5 whose SL-graph is given in the top of Fig. 4 is not succinct.

SLD$_\leq$ Formulas: A guard $G(\mathbf{y})$ is called a $\leq$*-guard* if it has the following syntax:

$$\bigwedge_{1\leq j\leq q} p_j \leq p'_j \wedge \bigwedge_{y\in\mathbf{y}} 0 < y < \mathtt{len}(n_y), \tag{i}$$

where p_j and p'_j are either position variables or position terms that do not contain position variables. That is, a $\leq$-guard contains only inequalities of the form $y_1 \leq y_2$, $y_1 \leq p$, $p \leq y_1$, or $p \leq p'$, where $y_1, y_2 \in \mathit{Pos}$ and p, p' are position terms without position variables. Thus, a $\leq$-guard can define only ordering or equality constraints between positions variables in one or several sequences; it can not define, e.g., the successor relation between positions variables. The fragment **SLD$_\leq$** is the set of all SLD formulas ψ such that for any sub-formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ of φ, (1) $G(\mathbf{y})$ is a $\leq$-guard and (2) any occurrence of a position variable y in $U(\mathbf{y})$ belongs to a term $n[y]$ with $n \in \mathit{NVar}$.

2.4 Closure under Post Image Computation

For any program statement St and any set of program configurations $\mathbb{H}$, $\mathtt{post}(\mathit{St},\mathbb{H})$ denotes the postcondition operator. The closure of SLD under the computation of the strongest postcondition w.r.t. basic statements (which don't contain "while" loops and deallocation statements) is stated in the following theorem (the proof is given in [5]).

Theorem 1. *Let St be a basic statement and ψ an SLD formula. Then,* $\mathtt{post}(\mathit{St},[\psi])$ *is SLD-definable and it can be computed in linear time. Moreover, if ψ is an SLD$_\leq$ formula then* $\mathtt{post}(\mathit{St},[\psi])$ *is SLD$_\leq$-definable and it can be computed in linear time.*

3 Checking Entailments between SLD Formulas

For any ψ_1 and ψ_2 two SLD formulas, ψ_1 *semantically entails* ψ_2 (denoted $\psi_1 \vdash \psi_2$) iff $[\psi_1] \subseteq [\psi_2]$. The following result states that checking the semantic entailment between SLD formulas is undecidable. It is implied by the fact that even the satisfiability problem for SLD is undecidable (by a reduction to the halting problem of 2-counter machines).

Theorem 2. *The satisfiability problem for* SLD *is undecidable. The problem of checking the semantic entailment between (succinct)* SLD *formulas is also undecidable.*

We present a procedure for checking entailments between SLD formulas, called *simple syntactic entailment* and denoted $\sqsubseteq_S$, which in general is only sound.

Checking Entailments of Shape Formulas: For SLD formulas without data constraints (i.e., disjunctions of shape formulas), the simple syntactic entailment is a slight extension to disjunctions and existential quantification of the decision procedure for Separation logic introduced in [9]. Thus, given two shape formulas φ and φ', $\varphi \sqsubseteq_S \varphi'$ iff the SL-graph of φ is $\perp$ or there exists an homomorphism from the SL-graph of φ' to the SL-graph of φ. This homomorphism preserves the labeling with program variables, the edges that denote inequalities, and it maps edges of φ' to (possibly empty paths) of φ such that any two disjoint edges of φ' are mapped to two disjoint paths of φ. For example, the dotted edges Fig. 1(c) represent the homomorphism h which proves that $\exists n,m.\ \varphi_S^1 \vdash \exists u,v,w.\ \varphi_S^2$. This holds because v is not required to be different from w. We have that $\varphi \vdash \varphi'$ iff $\varphi \sqsubseteq_S \varphi'$. Formally, the homomorphism between SL-graphs is defined as follows:

Definition 4 (Homomorphic shape formulas). *Given two SL-graphs S_1 and S_2, S_1 is* homomorphic *to S_2, denoted by $S_1 \mapsto_h S_2$, if $S_1 = S_2 = \perp$ or there exists a function $h : N_+^{S_1} \to N_+^{S_2}$, called homomorphism, such that: (1) $h(\sharp) = \sharp$; (2) for any $n \in N_+^{S_1}$, $Vars^{S_1}(n) \subseteq Vars^{S_2}(h(n))$; (3) for any $e = (n,u) \in E_\ell^{S_1}$, there is a (possibly empty) path π_e in S_2 starting in $h(n)$ and ending in $h(u)$; (4) for any two distinct edges $e_1 = (n,u) \in E_\ell^{S_1}$ and $e_2 = (m,v) \in E_\ell^{S_1}(m,v)$, the corresponding paths π_{e_1} and π_{e_2} associated by h in S_2 don't share any edge; (5) for any $e = (n,u) \in E_d^{S_1}$, $(h(n),h(u)) \in E_d^{S_2}$.* □

For any two SLD formulas ψ and ψ', which are disjunctions of shape formulas, $\psi \sqsubseteq_S \psi'$ iff for any disjunct φ of ψ there exists a disjunct φ' of ψ' such that $\varphi \sqsubseteq_S \varphi'$. One can prove that $\sqsubseteq_S$ is sound, i.e., for any ψ and ψ', if $\psi \sqsubseteq_S \psi'$ then $\psi \vdash \psi'$.

Adding Data Constraints: For SLD formulas with data constraints, the definition of the simple syntactic entailment is guided by the syntax of SLD. We illustrate it on the formulas from Fig. 1$(b),(d)$. First, the procedure checks if the simple syntactic entailment holds between the SL-formulas φ_S^1 and φ_S^2. Then, because the homomorphism in Fig. 1(c) maps every edge in S^2 to an edge in S^1, it checks that φ_D^1 entails $\varphi_D^2[h]$, where $\varphi_D^2[h]$ is obtained from φ_D^2 by applying the substitution $[u \mapsto n, v \mapsto m, w \mapsto m]$ defined by the homomorphism h (if the homomorphism h does not satisfy this condition then the simple syntactic entailment does not hold). The entailment between two sequence formulas φ_D and φ'_D is reduced to the entailment in the logic on data by checking that for any guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U'(\mathbf{y})$ in φ'_D there exists a guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ in φ_D such that $U(\mathbf{y}) \Rightarrow U'(\mathbf{y})$. *This entailment check between sequence formulas is also denoted by* $\sqsubseteq_S$. In Fig. 1(d), this test is satisfied for the guarded formula in $\varphi_D^2[h]$ by the last guarded formula in φ_D^1 (i.e., strict sortedness implies sortedness). This procedure is efficient because it requires no transformation on the input formulas and the decision procedure on data is applied on small instances and a number of times which is linear in the size of the input formulas.

The simple syntactic entailment for disjunctions of formulas of the form $\exists N.\ \varphi_S \wedge \varphi_D$ is defined as in the case of disjunctions of shape formulas. Clearly, $\sqsubseteq_S$ is only sound.

In the following, we will introduce a more precise procedure for checking entailments between SLD formulas, denoted $\sqsubseteq$ and called *syntactic entailment*. The presentation is done in two steps depending on the class of homomorphisms discovered while proving the entailment between shape formulas. First, we will consider *edge homomorphisms*, that map edges of an SL-graph to edges of another SL-graph (i.e., for any edge (u,v), $(h(u),h(v))$ is an edge) and then, we will consider the case of arbitrary homomorphisms.

4 Syntactic Entailment w.r.t. Edge Homomorphisms

The simple syntactic entailment fails to prove some relevant entailments encountered in practice, e.g, the equality of two lists pointed to by x and z, resp., and the fact that the list pointed to by z is sorted implies that the list pointed to by x is also sorted:

$$\exists n,m.\, (\mathtt{ls}(n,\sharp) * \mathtt{ls}(m,\sharp) \land x(n) \land z(m) \land \varphi_D^1) \vdash \exists u.\, \big(\mathtt{ls}(u,\sharp) \land x(u) \land sorted(u)\big). \qquad \text{(ii)}$$

Above, φ_D^1 is the formula in Fig. 1(c) and the entailment between the shape formulas is proven by the homomorphism h defined by $h(u) = n$. Checking the entailment between φ_D^1 and $sorted(u)[h]$ fails because the sets of guards in the two formulas are different. More precisely, φ_D^1 does not contain a guarded formula of the form $\forall y_1,y_2.\, G_n(y_1,y_2) \Rightarrow U$, where $G_n(y_1,y_2) := 0 < y_1 < y_2 < \mathtt{len}(n)$.

Saturation Procedure: The problem is that SLD does not have a normal form in the sense that the same property can be expressed using SLD formulas over different sets of guards. In our example, one can add to φ_D^1 the guarded formula $\rho := \forall y_1,y_2.\, 0 < y_1 < y_2 < \mathtt{len}(n) \Rightarrow \mathtt{dt}(n) < n[y_1] < n[y_2]$ while preserving the same set of models. Adding this guarded formula makes explicit the constraints on the integer values in the list segment starting in n, which are otherwise implicit in φ_D^1. If all constraints were explicit then, the simple syntactic entailment would succeed in proving the entailment.

Based on these remarks, we extend the simple syntactic entailment such that before applying the syntactic check between two sequence formulas φ_D and φ_D' presented above, we apply a saturation procedure to the SLD formula in the left hand side of the entailment, called `saturate`. This procedure makes explicit in φ_D all the properties expressed with guards that appear in φ_D'. For example, by applying this procedure the formula ρ is added to φ_1. More precisely, we add to φ_D a trivial formula $\forall \mathbf{y}.\, G(\mathbf{y}) \Rightarrow true$, for every guard in φ_D', and then, we call `saturate` which strengthens every guarded formula in φ_D. Roughly, the strengthening of $\forall \mathbf{y}.\, G(\mathbf{y}) \Rightarrow U_G(\mathbf{y})$ relies on the following principle: to find a formula U such that $G \Rightarrow U$ is implied by $E \land (G_1 \Rightarrow U_1) \land \ldots \land (G_k \Rightarrow U_k)$, one has to find a (negation-free) boolean combination $\mathbb{C}[G_1,\ldots,G_k]$ of $G_1,\ldots,G_k$ such that $(E \land G) \Rightarrow \mathbb{C}[G_1,\ldots,G_k]$, and then set U to $\mathbb{C}[U_1,\ldots,U_k]$. This principle is extended to boolean combinations of guards where some position variables are existentially-quantified (see [5] for more details). Going back to the example in (ii), we add to φ_D^1 the formula $\rho_0 := \forall y_1,y_2.\, G_n(y_1,y_2) \Rightarrow true$. Then, following the principle described above, we have that

$$\big(\mathtt{len}(n) = \mathtt{len}(m) \land G_n(y_1,y_2)\big) \Rightarrow \exists y_1',y_2'.\, \big(G_m(y_1',y_2') \land G_{m,n}(y_1,y_1') \land G_{m,n}(y_2,y_2')\big),$$

where G_m and $G_{m,n}$ are the guards from φ_D^1 given in Fig. 1(d). Therefore, the right part of the implication in ρ_0 can be replaced by

$$\exists y_1', y_2'. \left(\mathtt{dt}(n) = \mathtt{dt}(m) < m[y_1'] < m[y_2'] \wedge m[y_1'] = n[y_1] \wedge m[y_2'] = n[y_2]\right),$$

which is equivalent to the right hand side of ρ, $\mathtt{dt}(n) < n[y_1] < n[y_2]$.

Correctness and Precision Results: The next result shows that the saturation procedure returns a formula equivalent to the input one and that, for the fragment $\mathsf{SLD}_{\leq}$ of SLD, `saturate` computes the strongest guarded formulas which are implied by the input formula. The precision result holds because for the class of $\leq$-guards, it suffices to reason only with representatives for the set of tuples of positions satisfying some guard.

Theorem 3. *Let* $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ *be a disjunction-free* SLD *formula. Then,* $\mathtt{saturate}(\varphi)$ *is equivalent to* φ *and* $\mathtt{saturate}(\varphi) \sqsubseteq_S \varphi$*. Moreover, for any* $\mathsf{SLD}_{\leq}$ *formula* φ, $\mathtt{saturate}(\varphi) = \exists N.\ \varphi_S \wedge \varphi_D'$ *such that the following hold:*

- *the existential constraint of* φ_D', E', *is the strongest existential constraint such that* $\varphi \vdash (\exists N.\ \varphi_S \wedge E')$, *and*
- *for any guard* $G(\mathbf{y})$ *in* φ_D, φ_D' *contains the strongest universal formula of the form* $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ *such that* $\varphi \vdash (\exists N.\ \varphi_S \wedge \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y}))$. □

This procedure will be used to define a sound and complete decision procedure for the satisfiability of $\mathsf{SLD}_{\leq}$ and a sound and complete decision procedure for checking entailments between formulas in a fragment of $\mathsf{SLD}_{\leq}$ (see Th. 6 for more details).

5 Syntactic Entailment w.r.t Arbitrary Homomorphisms

Suppose that we want to check the entailment between two SLD formulas $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ and $\varphi' = \exists N'.\ \varphi_S' \wedge \varphi_D'$ and that h is an homomorphism that is a witness for the fact that $\varphi_S \vdash \varphi_S'$. If h is not an edge homomorphism then, when proving the entailment between φ_D and $\varphi_D'[h]$, one encounters two difficulties: (1) edges of φ_S' mapped to nodes of φ_S (i.e., edges (u,v) such that $h(u) = h(v)$) and (2) edges of φ_S' mapped to paths in φ_S containing at least two edges (i.e., edges (u,v) such that the nodes $h(u)$ and $h(v)$ are connected by a path of length at least 2).

Procedure `split`**:** In the first case, the edges of φ_S' mapped to nodes of φ_S pose the following problem: the sequence formula $\varphi_D'[h]$ may contain guarded formulas that describe list segments that don't have a correspondent in φ_D. For example, let

$$\begin{aligned}\varphi^3 &:= \exists n.\ x(n) \quad \wedge \qquad\qquad\ \mathtt{dt}(n) = 3 \text{ and}\\ \varphi^4 &:= \exists u,v.\ \mathtt{ls}(u,v) \wedge x(u) \wedge\ \mathtt{dt}(u) \geq 2 \wedge \forall y.\ 0 < y < \mathtt{len}(u) \Rightarrow u[y] \geq 1\end{aligned}$$

Note that $\varphi^3 \vdash \varphi^4$ and that there exists an homomorphism h between the shape formula of φ^4 and the shape formula of φ^3 given by $h(u) = h(v) = n$. In order to be able to use the same approach as before (i.e., applying `saturate` on φ^3 and then checking the entailment between guarded formulas with similar guards), we define a procedure `split` that transforms the formula φ^3 such that the homomorphism h becomes injective. That is, `split` transforms φ^3 into:

$$\overline{\varphi^3} := \exists n, nn.\ x(n) \wedge \mathtt{ls}(n,nn) \wedge \mathtt{dt}(n) = 3 \wedge \mathtt{len}(n) = 0,$$

where the new node variable nn is added such that $h'(u) = n$ and $h'(v) = nn$ is an injective homomorphism. Note that the two formulas φ^3 and $\overline{\varphi^3}$ are equivalent. Now, as described in the previous section, we can add the trivial formula $\forall y.\ 0 < y < \texttt{len}(n) \Rightarrow$ *true* to $\overline{\varphi^3}$ and then apply `saturate`, which strengthens it into $\forall y.\ 0 < y < \texttt{len}(n) \Rightarrow$ *false* because the list segment starting in n is empty. Now, $\texttt{dt}(n) = 3 \Rightarrow dt(n) \geq 2$ and $\mathit{false} \Rightarrow n[y] \geq 1$ which is enough to prove that $\varphi^3 \vdash \varphi^4$.

If the SL-graph of φ_S and φ'_S do not contain cycles then `split` returns a triple $(\overline{\varphi}, h', \varphi')$. The formula $\overline{\varphi}$ is obtained by (1) adding to φ_S a new node variable denoted nn for every two node variables n and m in φ'_S such that $h(n) = h(m)$, (2) placing nn between $h(n)$ and its successor in φ_S (i.e., $\texttt{ls}(h(n), x)$ is replaced by $\texttt{ls}(h(n), nn) * \texttt{ls}(nn, x)$) (3) substituting $h(n)$ with nn in the sequence formula φ_D (in this way, all constraints on $h(n)$ are transferred to the node nn) (4) adding the constraint that the length of the list segment starting in $h(n)$ is 0. The homomorphism h' is injective and it is defined like h except for n and m where $h'(n) = h(n)$ and $h'(m) = nn$. In the general case, $\texttt{split}(\varphi, h, \varphi')$ returns a set of triples $(\overline{\varphi}, h', \overline{\varphi'})$ with h' an injective homomorphism between the shape formula of $\overline{\varphi'}$ and the shape formula of $\overline{\varphi}$ ($\overline{\varphi}$ is an over-approximation of φ and $\overline{\varphi'}$ is an under-approximation of φ'). We have that $\varphi \vdash \varphi'$ iff $\overline{\varphi} \vdash \overline{\varphi'}$, for some triple $(\overline{\varphi}, h, \overline{\varphi'}) \in \texttt{split}(\varphi, h, \varphi')$ (see [5] for a complete definition of `split`).

Procedure `fold`: The second case is illustrated on the entailment $\varphi^5 \vdash \varphi^6$, where

$$\varphi^5 := \exists n, m, p.\ \big(\varphi^5_S \wedge \mathit{sorted}(n) \wedge \mathit{sorted}(m) \wedge \forall y.\ 0 < y < \texttt{len}(n) \Rightarrow n[y] \leq \texttt{dt}(m)\big)$$
$$\varphi^6 := \exists u, v.\ \big(\varphi^6_S \wedge \mathit{sorted}(u)\big)$$

and the graph formulas φ^5_S and φ^6_S are given by the SL-graphs S^5 and S^6 in Fig. 4.

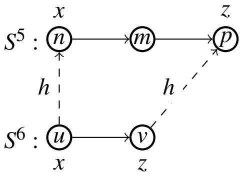


Fig. 4.

The homomorphism h from S^6 to S^5 defined by $h(u) = n$ and $h(v) = p$ maps the edge (u, v) to the path $(n, m), (m, p)$. Intuitively, the entailment $\varphi^5 \vdash \varphi^6$ holds because, in any model of φ^5, the concatenation between the sequence of integers in the list segment from n to m and the sequence of integers in the list segment from m to p is sorted (i.e., it satisfies the property of the list segment from u to v in φ^6.)

Let $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ and $\varphi' = \exists N'.\ \varphi'_S \wedge \varphi'_D$ be two SLD formulas and h an homomorphism from φ'_S to φ_S that maps edges of φ'_S to non-empty paths of φ_S. We denote by $\varphi'_D[\![h]\!]$ the sequence formula obtained from φ'_D by (1) substituting u by $h(u)$ in all terms except for $\texttt{len}(u)$ and (2) substituting $\texttt{len}(u)$ by $\sum_{(n,m) \in \overrightarrow{h(u), h(v)}} \texttt{len}(n)$ with v being the successor of u in φ'_S and $\overrightarrow{(h(u), h(v))}$ the path between $h(u)$ and $h(v)$ in the SL-graph of φ_S. For example, $\mathit{sorted}(u)[\![h]\!]$ is the formula:

$$\mathit{sorted}(n+m) := \forall y_1, y_2.\ \underbrace{0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m)}_{G_{n+m}(y_1, y_2)} \Rightarrow \texttt{dt}(n) \leq n[y_1] \leq n[y_2].$$

Remark that the substitution of $\texttt{len}(u)$ by $\texttt{len}(n) + \texttt{len}(m)$ makes the formula $\mathit{sorted}(u)[\![h]\!]$ contain properties of concatenations of list segments.

Note that the entailment $\varphi \vdash \varphi'$ holds if φ_D entails $\varphi'_D[\![h]\!]$. For example, $\varphi^5 \vdash \varphi^6$ holds because the sequence formula of φ^5, denoted φ^5_D, entails $\mathit{sorted}(u)[\![h]\!]$.

The difficulty in proving the entailment between φ^5_D and $\mathit{sorted}(n+m)$ is that φ^5_D does not contain a guarded formula having as guard $0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m)$. In the following, we describe a procedure called `fold` which computes properties of such concatenations of list segments. In this particular case, we add to φ^5_D the trivial formula $\forall y_1, y_2.\ 0 < y_1 < y_2 < \mathtt{len}(n) + \mathtt{len}(m) \Rightarrow \mathit{true}$ which is strengthened by `fold` into a formula equivalent to $\mathit{sorted}(n+m)$.

For every $G(\mathbf{y})$ as above, `fold` begins by computing a set of auxiliary guards, one for every way of placing positions that satisfy $G(\mathbf{y})$ on the list segments that are concatenated. Then, for every such satisfiable guard $G'(\mathbf{y}')$, it calls the saturation procedure `saturate` to compute a guarded formula of the form $\forall \mathbf{y}'.\ G'(\mathbf{y}') \Rightarrow U'(\mathbf{y}')$ implied by φ. Finally, it defines $U(\mathbf{y})$ as the disjunction of all formulas which are in the right hand side of a guarded formula computed in the previous step (see [5] for more details).

We exemplify the procedure `fold` on the formula $\varphi^5_D \wedge \forall y_1, y_2.\ 0 < y_1 < y_2 < \mathtt{len}(n) + \mathtt{len}(m) \Rightarrow \mathit{true}$ involved in proving the entailment $\varphi^5 \vdash \varphi^6$ above. A value for y_1 or y_2 that satisfies the constraints in $G_{n+m}(y_1, y_2)$, i.e. it is a position between 1 and $\mathtt{len}(n) + \mathtt{len}(m) - 1$ on the list starting in n, can either correspond to (1) a position on the sequence associated with the list segment $\mathtt{ls}(n,m)$ (when it is less than $\mathtt{len}(n)$) or to (2) the first element of the sequence associated with the list segment $\mathtt{ls}(m,p)$ (when it equals $\mathtt{len}(n)$) or to (3) a position on the tail of the sequence associated with the list segment $\mathtt{ls}(m,p)$ (when it is greater than $\mathtt{len}(n)$). In each case, an auxiliary guard is computed by adding some constraints to $G_{n+m}(y_1, y_2)$ and by substituting the variables y_1 and y_2 as follows. If y_i is considered to be a position on the tail of some list segment α then the constraint $0 < y_i < \mathtt{len}(\alpha)$ is added to $G_{n+m}(y_1, y_2)$ and y_i is substituted by $y_i + \sum_j \mathtt{len}(n_j)$, where n_j are all the list segments from n to the predecessor of α. Concretely, if y_i corresponds to a position on the tail of the list segment $\mathtt{ls}(n,m)$ then $0 < y_i < \mathtt{len}(n)$ is added to $G_{n+m}(y_1, y_2)$ and y_i remains unchanged. If y_i corresponds to a position on the tail of the list segment $\mathtt{ls}(m,p)$ then $0 < y_i < \mathtt{len}(m)$ is added to $G_{n+m}(y_1, y_2)$ and y_i is substituted by $y_i + \mathtt{len}(n)$. If y_i is considered to be the first element of the list segment $\mathtt{ls}(m,p)$ then it is substituted by the exact value of this position, i.e. $\mathtt{len}(n)$. Below, we consider three cases and give the auxiliary guard computed in each case:

"y_1 is the first element of $\pi_{m,p}$", "y_2 is a position on the tail of $\pi_{m,p}$"

$$0 < \mathtt{len}(n) < y_2 + \mathtt{len}(n) < \mathtt{len}(n) + \mathtt{len}(m) \wedge 0 < y_2 < \mathtt{len}(m)$$
$$\equiv 0 < y_2 < \mathtt{len}(m)$$

"y_1 is a position on the tail of $\pi_{n,m}$", "y_2 is a position on the tail of $\pi_{m,p}$"

$$0 < y_1 < y_2 + \mathtt{len}(n) < \mathtt{len}(n) + \mathtt{len}(m) \wedge 0 < y_1 < \mathtt{len}(n) \wedge 0 < y_2 < \mathtt{len}(m)$$
$$\equiv 0 < y_1 < \mathtt{len}(n) \wedge 0 < y_2 < \mathtt{len}(m)$$

"y_1 is a position on the tail of $\pi_{m,p}$", "y_2 is the first element of $\pi_{m,p}$"

$$0 < y_1 + \mathtt{len}(n) < \mathtt{len}(n) < \mathtt{len}(n) + \mathtt{len}(m) \wedge 0 < y_1 < \mathtt{len}(m)$$
$$\equiv \mathit{false}$$

Notice that the third situation is not possible and it corresponds to an unsatisfiable guard which will be ignored in the following. The procedure `saturate` infers from φ^5_D the following properties: $\gamma_1 := \forall y_2.\ 0 < y_2 < \mathtt{len}(m) \Rightarrow \mathtt{dt}(n) \leq \mathtt{dt}(m) \leq m[y_2]$ and $\gamma_2 := \forall y_1, y_2.\ (0 < y_1 < \mathtt{len}(n) \wedge 0 < y_2 < \mathtt{len}(m)) \Rightarrow \mathtt{dt}(n) \leq \mathtt{dt}(n) \leq n[y_1] \leq m[y_2]$. The other possible cases for the placement of the positions denoted by y_1 and y_2 are handled in a similar manner.

The right parts of all the generated guarded formulas are "normalized" such that they characterize the terms $n[y_1]$ and $n[y_2]$, where y_1 and y_2 satisfy the constraints in $G_{n+m}(y_1,y_2)$. For example, in the right part of γ_1, $\mathtt{dt}(m)$ is substituted by $n[y_1]$ (because y_1 was considered to be the first element of the list segment starting in m) and $m[y_2]$ is substituted by $n[y_2]$ (because y_2 was considered to be a position on the tail of the list segment starting in m) and in the right part of γ_2, $n[y_1]$ remains unchanged and $m[y_2]$ is substituted by $n[y_2]$. The procedure `fold` returns $\forall y_1,y_2.\ G_{n+m}(y_1,y_2) \Rightarrow U(y_1,y_2)$, where $U(y_1,y_2)$ is the disjunction of all the obtained formulas. In this case, $U(y_1,y_2)$ is equivalent to $\mathtt{dt}(n) \leq n[y_1] \leq n[y_2]$.

Syntactic Entailment: Given two SLD formulas ψ and ψ', the syntactic entailment $\psi \sqsubseteq \psi'$ is defined as follows: for any disjunct φ of ψ there exists a disjunct φ' of ψ' such that $\varphi \sqsubseteq \varphi'$ holds, where the relation $\sqsubseteq$ is defined in Fig. 5.

Correctness and Precision Results: The following theorem gives precision and correctness results for `fold`. The precision result is implied by the precision of `saturate` for $\mathsf{SLD}_{\leq}$ formulas.

Theorem 4. *Let $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ be a disjunction-free SLD formula. Then,* $\mathtt{fold}(\varphi)$ *is equivalent to φ and* $\mathtt{fold}(\varphi) \sqsubseteq_S \varphi$. *Moreover, for any $\mathsf{SLD}_{\leq}$ formula φ,* $\mathtt{fold}(\varphi) =$ *$\exists N.\ \varphi_S \wedge \varphi'_D$ such that for any guard $G(y)$ in φ_D, which describes concatenations of list segments, φ'_D contains the strongest universal formula of the form $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ such that $\varphi \vdash (\exists N.\ \varphi_S \wedge \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y}))$.*

The correctness result for `fold` and `saturate` implies that $\sqsubseteq$ is sound. Next, we identify entailment problems $\psi_1 \vdash \psi_2$, where ψ_1 and ψ_2 belong to $\mathsf{SLD}_{\leq}$, for which the procedure $\sqsubseteq$ is complete. Roughly, we impose restrictions on ψ_1 and ψ_2 such that a disjunction-free SLD formula in ψ_1 may entail at most one disjunct in ψ_2. For example, we require that ψ_2 is unambiguous. An SLD formula ψ is called *unambiguous* if for any disjunct φ of ψ, the SL-graph of φ contains an undirected (inequality) edge between every two nodes.

Theorem 5 (Soundness). *Let ψ_1 and ψ_2 be SLD formulas. If $\psi_1 \sqsubseteq \psi_2$ then $\psi_1 \vdash \psi_2$.*

Theorem 6 (Completeness). *Let ψ_1 and ψ_2 be two $\mathsf{SLD}_{\leq}$ formulas. If ψ_1 is unambiguous, ψ_2 is succinct, and for every disjunct φ_1 of ψ_1 there exists at most one disjunct φ_2 of ψ_2 homomorphic to φ_1 then $\psi_1 \vdash \psi_2$ implies* $\mathtt{saturate}(\psi_1) \sqsubseteq \psi_2$.

ALGORITHM Syntactic entailment $\varphi \sqsubseteq \varphi'$

```
Require: $\varphi := \exists N.\ \varphi_S \wedge \varphi_D,\ \varphi' := \exists N'.\ \varphi'_S \wedge \varphi'_D$
choose $h$ an homomorphism from $\varphi'_S$ to $\varphi_S$
choose $(\overline{\varphi}, h, \overline{\varphi'})$ in split$(\varphi, h, \varphi')$
add to $\overline{\varphi}$ missing guards from $\overline{\varphi'}[\![h]\!]$
$\varphi^1 :=$ fold$(\overline{\varphi})$
$\varphi^2 :=$ saturate$(\varphi_1)$
check $\varphi^2_D \sqsubseteq_S \overline{\varphi'_D}$, where $\varphi^2_D$ and $\overline{\varphi'_D}$ is the
    sequence formula of $\varphi^2$ and $\overline{\varphi'}$, respectively.
```

Fig. 5

The procedure `saturate` can also be used to check satisfiability of SLD formulas. Notice that an SLD formula $\varphi := \exists N.\ \varphi_S \wedge \varphi_D$ is unsatisfiable iff either the SL-graph of φ_S is $\perp$ or the sequence formula φ_D is unsatisfiable. The latter condition means that the strongest existential constraint E s.t. $\varphi \vdash \exists N.\ (\varphi_S \wedge E)$ is equivalent to *false*.

Theorem 7. *An SLD formula ψ is unsatisfiable iff for any disjunct φ of ψ either the SL-graph of φ is $\perp$ or the existential constraint E of* $\mathtt{saturate}(\varphi)$ *is unsatisfiable.*

We give in [5] an extension of these results to more general SLD formulas that contain guarded formulas describing concatenations of list segments.

6 Logic SLAD, Extension of SLD with Arrays

The class of programs considered in Section 2 can be extended to manipulate, besides lists, a fixed set of arrays. We consider that the arrays are not overlapping (e.g., like in Java), and that they are manipulated by operations like allocation, read/write an element, and read the length. A configuration of such programs is also represented by a directed graph and a valuation for the integer program variables. For lack of space, we present here the main ideas of this extension, a detailed presentation of is provided in [5].

Let *AVar* be a set of array variables, disjoint from the sets *PVar* and *DVar*. Also, let *IVar* be the set of integer program variables in *DVar* used in order to access array elements. A variable in *IVar* is called an *index variable*. The syntax of *shape formulas* in SLAD is the one given in Fig. 2 for SLD; only sequence formulas are specifying array properties. The syntax of *sequence formulas* in SLAD extends the one given in Fig. 3 by allowing the following new terms and guards:

Position terms:	E-terms:	U-terms:	Guards:
$p \ ::= \ \dots \mid i$	$e \ ::= \ \dots \mid a[p]$	$t \ ::= \ \dots \mid a[y]$	$G(\mathbf{y}) \ ::= C \wedge \bigwedge_{y \in \mathbf{y}} 0 < y < \ell(y)$
$i \in \mathit{IVar}$	$a, a_y \in \mathit{AVar}$		$\ell(y) \ ::= \mathtt{len}(n_y) \mid \mathtt{len}(a_y)$

The definition of the SLAD guards includes constraints on position variables $y \in \mathbf{y}$ used with arrays. The same condition for U-terms $n[y]$ is applied to terms $a[y]$: they appear in $U(\mathbf{y})$ only if the guard includes the constraint $0 < y < \mathtt{len}(a_y)$.

The procedure for checking the syntactic entailment $\varphi_1 \sqsubseteq \varphi_2$ between two disjunction-free formulas $\varphi_1, \varphi_2 \in$ SLAD translates φ_1 and φ_2 into equivalent SLD formulas and then, it applies the syntactic entailment for SLD defined in Fig. 5. Roughly, the translation procedure applied on an SLAD formula φ adds to the shape formula the list segments corresponding to array variables used in the sequence formula, and it soundly translates the terms and guards over arrays into terms and guards over lists. The resulting SLD formula $\overline{\varphi}$ is equivalent to φ and of size polynomial in the size of φ.

The fragment SLAD$_{\leq}$ of SLAD is defined similarly to the fragment SLD$_{\leq}$ of SLD (the only difference is that, for any guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$, any occurrence of a position variable y in $U(\mathbf{y})$ belongs to a term of the form $n[y]$ or $a[y]$). The following results are straightforward consequences of Th. 6.

Corollary 1. *Let ψ_1, ψ_2 be two formulas in SLAD. If $\psi_1 \sqsubseteq \psi_2$ then $\psi_1 \vdash \psi_2$. Moreover, if ψ_1, ψ_2 are SLAD$_{\leq}$ formulas satisfying the restrictions in Th. 6, then $\psi_1 \vdash \psi_2$ implies* $\mathtt{saturate}(\psi_1) \sqsubseteq \psi_2$.

Corollary 2. *Checking the semantic entailment $\psi_1 \vdash \psi_2$, where ψ_1 and ψ_2 are two SLAD$_{\leq}$ formulas satisfying the restrictions in Th. 6, is decidable. Also, checking the satisfiability of an SLAD$_{\leq}$ formula is decidable. If we consider SLAD$_{\leq}$ formulas with a fixed number of universal quantifiers s.t. the logic on data is quantifier-free Presburger arithmetics then the two problems are NP-complete.*

7 Experimental Results

We have implemented in CELIA [6] the algorithm $\sqsubseteq$ and the postcondition operator for SLAD, and we have applied the tool to the verification of a significant set of programs manipulating lists and arrays. These programs need invariants and pre/post conditions beyond the decidable fragment SLAD$_{\leq}$. For example, we have verified a C library implementing sets of integers using strictly sorted lists (procedures/clients of the library are named *setlist-** in Tab. 1). The guarded formulas used by the specifications of this library need guards of the form $y_2 = y_1 + 1$ or $y_1 < y_2$ which are not $\leq$-guards. The verification of the clients is done in a modular way by applying an extension for SLAD of the frame rule from Separation logic.

The verification tool extends the implementation of the abstract domain of universal formulas defined in [6] to which we have added the procedures `saturate`, `split`, `fold`, and the computation of SL-graph homomorphism. The decision procedures $\sqsubseteq$ and `saturate` are also available in an independent tool SLAD whose input are formulas in the SMTLIB2 format. Table 1 provides the characteristics and the experimental results obtained (on a Pentium 4 Xeon at 4 GHz) for an illustrative sample of the verified programs. The full list of verified programs is available at `www.liafa.jussieu.fr/celia/verif/`.

Table 1. Experimental results. Size of formulas $n\vee \times m\forall$ means n disjuncts (i.e., shape formulas) with at most m guarded formulas per disjunct. `copyAddV` creates a list from an input list with all data increased by some data parameter. `initSeq` initializes data in a list with consecutive values starting from some data parameter. `initFibo` initializes data in a list with the Fibonacci sequence. `setlist-*` are procedures/clients of a library implementing sets as strictly sorted lists. `svcomp-list-prop` is an example from the SV-COMP competition. `array2list` creates a list from an array.

Program	pre-cond		inv		`post`(inv)		post-cond		Verif.
	size	logic	size	logic	size	logic	size	logic	time
copyaddV	$1\vee \times 0\forall$	SLD$_{\leq}$	$3\vee \times 1\forall$	SLD$_{\leq}$	$2\vee \times 1\forall$	SLD$_{\leq}$	$1\vee \times 1\forall$	SLD$_{\leq}$	< 1s
initSeq	$1\vee \times 0\forall$	SLD$_{\leq}$	$3\vee \times 1\forall$	SLD	$2\vee \times 1\forall$	SLD	$1\vee \times 1\forall$	SLD	< 1s
initFibo	$1\vee \times 0\forall$	SLD$_{\leq}$	$3\vee \times 2\forall$	SLD	$4\vee \times 2\forall$	SLD	$1\vee \times 1\forall$	SLD$_{\leq}$	< 1s
setlist-contains	$1\vee \times 2\forall$	SLD	$3\vee \times 4\forall$	SLD	$4\vee \times 4\forall$	SLD	$2\vee \times 3\forall$	SLD	< 1s
setlist-add	$1\vee \times 2\forall$	SLD	$3\vee \times 7\forall$	SLD	$4\vee \times 7\forall$	SLD	$1\vee \times 1\forall$	SLD	< 1s
setlist-union	$1\vee \times 4\forall$	SLD	$4\vee \times 13\forall$	SLD	$5\vee \times 13\forall$	SLD	$1\vee \times 6\forall$	SLD	< 2s
setlist-intersect	$1\vee \times 4\forall$	SLD	$3\vee \times 13\forall$	SLD	$4\vee \times 13\forall$	SLD	$1\vee \times 6\forall$	SLD	< 2s
setlist-client	$0\vee \times 0\forall$	SLD	$2\vee \times 2\forall$	SLD	$3\vee \times 2\forall$	SLD	$1\vee \times 2\forall$	SLD	< 1s
svcomp-list-prop	$1\vee \times 0\forall$	SLD$_{\leq}$	$3\vee \times 2\forall$	SLD	$4\vee \times 2\forall$	SLD	$1\vee \times 1\forall$	SLD	< 1s
array2list	$1\vee \times 0\forall$	SLAD$_{\leq}$	$2\vee \times 1\forall$	SLAD$_{\leq}$	$2\vee \times 1\forall$	SLAD$_{\leq}$	$1\vee \times 1\forall$	SLAD$_{\leq}$	< 1s
array-insertsort	$1\vee \times 4\forall$	SLAD$_{\leq}$	$3\vee \times 5\forall$	SLAD$_{\leq}$	$2\vee \times 5\forall$	SLAD$_{\leq}$	$1\vee \times 4\forall$	SLAD$_{\leq}$	< 3s

8 Related Work and Conclusions

Various frameworks have been developed for the verification of programs based on logics for reasoning about data structures, e.g., [1,4,6,7,8,9,12,13,14,15,16,17,18].

Decidable Logics for Unbounded Data Domains: Several works have addressed the issue of reasoning about programs manipulating data structures with unbounded data, e.g. [4,8,13,14,15,19]. The logics in [8,13] allow to reason about arrays and they are fragments of SLAD (see [5]). The fragment $\mathsf{SLAD}_{\leq}$, for which the satisfiability problem is decidable, includes the Array Property Fragment [8] when defined over finite arrays but, it is incomparable to the logic LIA [13].

The logics in [4,14] to reason about composite data structures are more expressive concerning the shape constraints but they are less expressive than SLAD when restricted to heaps formed of singly-linked lists and arrays. The restriction of LISBQ [14] to this class of heaps is included in $\mathsf{SLAD}_{\leq}$ but, the similar restriction of CSL [4] is not included in $\mathsf{SLAD}_{\leq}$; the latter does not permit guards that contain strict inequalities. The decidable fragment of STRAND [15] can describe other recursive data structures than lists but, its restriction to lists is incomparable to SLD: it does not include properties on data expressed using the immediate successor relation between nodes in the lists.

Sound Decision Procedures: Decision procedures which are sound but, in general, not complete, have been proposed in [11,16,10,18]. The work in [18] targets functional programs and it is not appropriate for imperative programs that mutate the heap.

The framework in [16] considers recursive programs on trees and it defines a sound decision procedure for proving Hoare triples. The validity of the Hoare triple is reduced through some abstraction mechanism to the validity of a formula in a decidable logic. In this paper, we describe a sound procedure for checking entailments between formulas in SLAD, which is independent of the fact that these entailments are obtained from some Hoare triples. Moreover, SLAD is incomparable to the logic used in [16].

Current state of the art SMT solvers do not include a theory for lists having the same expressiveness as SLD. For arrays, most of the SMT solvers deal with formulas in the Array Property Fragment [8]. However, they may prove entailments between array properties in SLAD but not in $\mathsf{SLAD}_{\leq}$ by using heuristics for quantifier instantiation, see e.g. Z3 [11,10]. Our entailment procedure, which is based on the saturation procedure `saturate`, is more powerful because it is independent of the type of constraints that appear in the right hand side of the guarded formulas. The heuristics used in Z3 work well when the entailment can be proved using some boolean abstraction of the formulas or when the right hand side of the guarded formulas contains only equalities.

In our previous work [6,7], we introduced a logic on lists called SL3, which is included in SLAD. In SL3, data properties are also described by universal implications $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ but the guard $G(\mathbf{y})$ is not as expressive as in SLAD. Any two node variables in an SL3 formula denote distinct vertices in the heap. This can lead to an exponential blow-up for the size of the formulas which implies a blow-up in the complexity of the decision procedure. Checking an entailment between SL3 formulas is reduced to the abstract analysis of a program that traverses the allocated lists and thus, it is impossible to characterize its preciseness using completeness results.

Conclusions: We have defined an approach for checking entailment and satisfiability of formulas on lists and arrays with data. Our approach deals with complex assertions that are beyond the reach of existing techniques and decision procedures. Although we have considered only programs with singly-linked lists and arrays, our techniques can be extended to other classes of data structures (doubly-linked lists, trees) using appropriate embeddings of heap graphs into finite abstract graphs.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape Analysis of Single-Parent Heaps. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 91–105. Springer, Heidelberg (2007)
2. Benedikt, M., Reps, T., Sagiv, M.: A Decidable Logic for Describing Linked Data Structures. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 2–19. Springer, Heidelberg (1999)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
4. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A Logic-Based Framework for Reasoning about Composite Data Structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
5. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. Technical report, LIAFA (2011), http://www.liafa.univ-paris-diderot.fr/~cenea/SLAD.pdf
6. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI, pp. 578–589. ACM (2011)
7. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract Domains for Automated Reasoning about List-Manipulating Programs with Infinite Data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
8. 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 (2005)
9. Cook, B., Haase, C., Ouaknine, J., Parkinson, M.J., Worrell, J.: Tractable Reasoning in a Fragment of Separation Logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
10. de Moura, L., Bjørner, N.: Efficient E-Matching for SMT Solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
11. Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
12. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246. ACM (2008)
13. 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)
14. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL, pp. 171–182. ACM (2008)
15. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data d. In: POPL, pp. 283–294. ACM (2011)
16. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: POPL, pp. 123–136. ACM (2012)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
18. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL, pp. 199–210. ACM (2010)
19. Wies, T., Muñiz, M., Kuncak, V.: An Efficient Decision Procedure for Imperative Tree Data Structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011)

A Verifier for Functional Properties of Sequence-Manipulating Programs

Carlo A. Furia

ETH Zurich, Switzerland
caf@inf.ethz.ch

Abstract. Many programs operate on data structures whose models are sequences, such as arrays, lists, and queues. When specifying and verifying functional properties of such programs, it is convenient to use an assertion language and a reasoning engine that incorporate sequences natively. This paper presents qfis, a program verifier geared to sequence-manipulating programs. qfis is a command-line tool that inputs annotated programs, generates the verification conditions that establish their correctness, tries to discharge them by calls to the SMT-solver CVC3, and reports the outcome back to the user. qfis can be used directly or as a back-end of more complex programming languages.

1 Overview

Many programs use data structures whose functional properties are expressible in terms of *sequences* of values from a certain domain. For example, lists, queues, and stacks are all modeled by sequences accessed according to specific patterns. To specify and reason about such programs, it is convenient to use first-order languages that support sequences natively, and that are amenable to automated reasoning.

In previous work [2], we introduced a first-order theory of integer sequences $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ whose quantifier-free fragment is decidable. $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ also includes Presburger arithmetic on sequence elements, and it is sufficiently expressive to specify several functional properties of sequence-manipulating programs. The present paper describes qfis, an automated verifier for programs annotated with $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ formulas. qfis inputs programs written in a simple imperative Algol-like procedural language, supporting integers and sequences as primitive types. Each routine may include a functional specification in the form of pre- and postcondition, written in a logic language including native functions and predicates on sequences—such as the concatenation and length functions.

The overall usage of qfis is similar to that of general-purpose program verifiers such as Dafny [3] or Why [4]. First, the user writes the program as a collection of routines with pre- and postconditions. Whenever useful, she also provides a collection of logic axioms (also expressed in the theory of integer sequences $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$) that define the semantics of predicates mentioned in the specification. For example, when proving the correctness of a sorting algorithm, it is customary to introduce a predicate *sorted*?(X) with the expected meaning. When called on the input file, qfis generates the *verification conditions* (VC): a set of first-order formulas whose validity entails the correctness of the program with respect to its specification. qfis encodes the VC in the input language

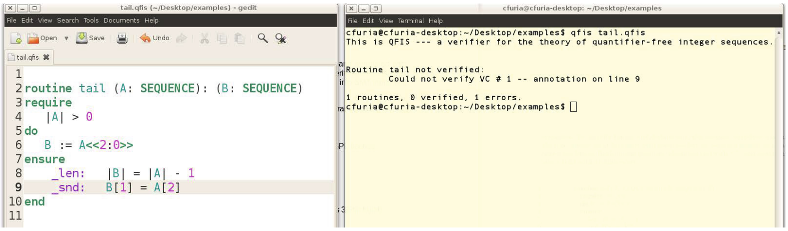
S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 183–186, 2012.

of the SMT-solver CVC3 [1], and calls the solver to discharge the VC. Then, qfis filters back CVC3's output and reports the outcome to the user. In case of unsuccessful verification, qfis points to specific annotations in the input program that could not be verified. Figure 1 shows a screenshot of qfis, with the input program displayed in the editor on the left, and the verifier's output in the shell on the right.

qfis is written in Eiffel, distributed under GPL, and available for download at:

http://se.inf.ethz.ch/people/furia/software/qfis.html

The download page includes pre-compiled binaries for Linux; the source code for compilation; a user manual with installation instructions and a tutorial examples; a demo video; a collection of annotated example programs; and a syntax-highlighting GTK source-view specification of its input language (used in Figure 1).


tail.qfis (~/Desktop/examples) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
tail.qfis
cfuria@cfuria-desktop: ~/Desktop/examples
File Edit View Terminal Help

```
1
2 routine tail (A: SEQUENCE): (B: SEQUENCE)
3 require
4     |A| > 0
5 do
6     B := A<<2:0>>
7 ensure
8     _len:    |B| = |A| - 1
9     _snd:    B[1] = A[2]
10 end
11
```

```
cfuria@cfuria-desktop:~/Desktop/examples$ qfis tail.qfis
This is QFIS --- a verifier for the theory of quantifier-free integer sequences.


Routine tail not verified:
        Could not verify VC # 1 -- annotation on line 9

1 routines, 0 verified, 1 errors.
cfuria@cfuria-desktop:~/Desktop/examples$ 
```

Fig. 1. qfis verifying routine *tail*

2 Using qfis

qfis's input language features sequences as a primitive type; unlike other verifiers that also support sequence types (e.g. Dafny [3]), qfis's sequences are usable in both specification and imperative constructs.

2.1 Input Language

qfis inputs text files containing a collection of routines (functions and procedures) and declarations of global variables (accessible from any routine), predicates (usable in annotations), and axioms (defining the semantics of user-defined predicates). For example:

```
1   routine tail (A: SEQUENCE): (B: SEQUENCE)
2     require |A| > 0
3     do B := A≪2:0≫
4     ensure
5       _len: |B| = |A| − 1
6       _snd: B[1] = A[2]
7   end
```

is a partially-specified function *tail* that returns the input sequence without the first element (line 3, i.e., from the second element to the last one). *tail*'s precondition (line 2) requires that the input sequence *A* has positive length; the postcondition has two clauses (lines 5 and 6) that assert that the returned sequence B has one less element than A, and that B's first element equals A's second element.

qfis annotations include axioms, pre- and postconditions (**require**, **ensure**), intermediate assertions (**assert**, **assume**), loop invariants (**invariant**), and frame clauses (**modify**) to specify the effect of routines on global variables. The assertions themselves share the same language of Boolean expressions also usable in imperative statements. It is very similar to the quantifier-free fragment of the theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ with some restrictions (for performance in the current implementation) but also some additions. In particular, it includes element ($A[3]$) and range ($A\ll 2{:}5\gg$) selection, sequence length ($|A|$), concatenation ($A \circ B$) and sequence equality ($A \doteq B$), as well as full integer arithmetic among sequence elements and lengths. Postconditions may also include the **old** keyword to refer to the values of global variables before invocation of the current routine. Axioms may also include arbitrary quantifiers (**forall**, **exist**) to define predicate semantics. To simplify typing and declarations, qfis assumes different identifier styles according to the type: integer identifiers start with lowercase letters, sequence identifiers with uppercase letters, Boolean and predicate identifiers end with "?", and assertion labels start with an underscore.

2.2 Verification Condition Generation

Given a collection of annotated input routines, qfis generates *verification conditions* by weakest precondition calculation (performed by visiting the AST of the input). The backward propagated assertions hold references to their source line numbers in the input program; this information is used when some VC cannot be discharged, to trace back the error to the location in the source. For example, the backward substitution of *tail*'s postcondition clause *_len* determines the VC $|A| > 0 \Longrightarrow |A\ll 2{:}0\gg| = |A| - 1$, which can be proven valid.

Similarly to other program provers, qfis performs *modular reasoning*: the semantics of a call to some routine *foo* within another routine is entirely determined by *foo*'s precondition P, postcondition Q, and frame F: check that P holds before the call (**assert** P); nondeterministically assign values to variables in *foo*'s frame (and arguments) **havoc** F; constrain the nondeterministic assignment to satisfy Q (**assume** Q).

2.3 SMT Encoding

qfis does not implement the decision procedure for the quantifier-free fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ presented in [2], but it directly encodes the VC in the input language of the SMT-solver CVC3. This design choice provides overall more flexibility and a more robust implementation, relying on a carefully engineered tool such as CVC3. The input language is deliberately relaxed to include undecidable components (full-fledged integer arithmetic and unrestricted quantifiers in axioms), but this is not much of a problem in practice thanks to the powerful instantiation heuristics provided by SMT-solvers—as long as the departure from the basic decidable kernel is reasonably restricted.

The translation of VC to CVC3 uses a list **DATATYPE** definition to encode sequences. A set of standard axioms provides an axiomatization of the concatenation function *cat* applied on lists; for example, an axiom asserts that **nil** is the neutral element of the concatenation function *cat* with formulas such as $cat(x, \mathbf{nil}) = x$.

The CVC3 encoding of expressions involving element selection and subranges uses *unrolled* definitions that are handled efficiently by the reasoning engine.

3 Examples

Table 1 lists 11 programs verified using qfis with CVC3 2.2 as back-end, running on a GNU/Linux Ubuntu box (kernel 2.6.32) with an Intel Quad-Core2 CPU at 2.40 GhZ with 4 GB of RAM. For each program, the table shows the number of routines in the input file (# R), the number of user-defined predicates and specification functions (# P), of user-written axioms (# A), the total lines of the input (# L), and the real time (in seconds) taken by verification, including both the VC generation and the call to CVC3. All the programs are included in the qfis distribution.

The most complex program is the second version of *merge sort*, which required an increase of the standard timeout of 10 seconds (per VC, before the SMT solver gives up proving validity). *tail* is the very simple program of Section 2.1, and it is also the only program where verification fails (as shown in Figure 1); qfis reports that it cannot verify the postcondition clause *_snd*: $B[1] = A[2]$: in fact, if A has only one element (which is possible, because the precondition only requires that it is not empty), $A[2]$ is undefined.

Table 1. Programs verified with qfis

PROGRAM	# R	# P	# A	# L	TIME [S]
binary search	2	3	11	87	3.1
linear search	1	3	5	41	2.3
linked list	15	4	6	208	15.6
merge sort (v. 1)	1	1	4	49	12.3
merge sort (v. 2)	2	1	4	67	31.0
quick sort	2	3	11	87	6.8
reversal	1	1	2	24	4.0
stack reversal	1	2	4	37	5.5
sum & max (v. 1)	3	2	8	83	6.8
sum & max (v. 2)	1	2	9	52	3.8
tail	1	0	0	11	2.2

Acknowledgements. Work partially supported by SNF project 200021-134976 (Automated Support for Invariant Inference) and by the Hasler foundation on related projects.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Furia, C.A.: What"s Decidable about Sequences? In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 128–142. Springer, Heidelberg (2010)
3. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
4. Why3: Where programs meet provers, http://why3.lri.fr

Accelerating Interpolants*

Hossein Hojjat[1], Radu Iosif[2],
Filip Konečný[2,4], Viktor Kuncak[1], and Philipp Rümmer[3]

[1] Swiss Federal Institute of Technology Lausanne (EPFL)
[2] Verimag, Grenoble, France
[3] Uppsala University, Sweden
[4] Brno University of Technology, Czech Republic

Abstract. We present Counterexample-Guided *Accelerated* Abstraction Refinement (CEGAAR), a new algorithm for verifying infinite-state transition systems. CEGAAR combines *interpolation-based predicate discovery* in counterexample-guided predicate abstraction with *acceleration* technique for computing the transitive closure of loops. CEGAAR applies acceleration to dynamically discovered looping patterns in the unfolding of the transition system, and combines overapproximation with underapproximation. It constructs inductive invariants that rule out an infinite family of spurious counterexamples, alleviating the problem of divergence in predicate abstraction without losing its adaptive nature. We present theoretical and experimental justification for the effectiveness of CEGAAR, showing that inductive interpolants can be computed from classical Craig interpolants and transitive closures of loops. We present an implementation of CEGAAR that verifies integer transition systems. We show that the resulting implementation robustly handles a number of difficult transition systems that cannot be handled using interpolation-based predicate abstraction or acceleration alone.

1 Introduction

This paper contributes to the fundamental problem of precise reachability analysis for infinite-state systems. Predicate abstraction using interpolation has emerged as an effective technique in this domain. The underlying idea is to verify a program by reasoning about its *abstraction* that is easier to analyse, and is defined with respect to a set of predicates [17]. The set of predicates is refined to achieve the precision needed to prove the absence or the presence of errors. A key difficulty in this approach is to automatically find predicates to make the abstraction sufficiently precise [2]. A breakthrough technique is to generate predicates based on *Craig interpolants* [13] derived from the proof of unfeasibility of a spurious trace [19].

While empirically successful on a variety of domains, abstraction refinement using interpolants suffers from the unpredictability of interpolants computed by provers,

* Supported by the Rich Model Toolkit initiative, `http://richmodels.org`, the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (COST OC10009 and MSM 0021630528), the BUT project FIT-S-12-1 and the Microsoft Innovation Cluster for Embedded Software.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 187–202, 2012.

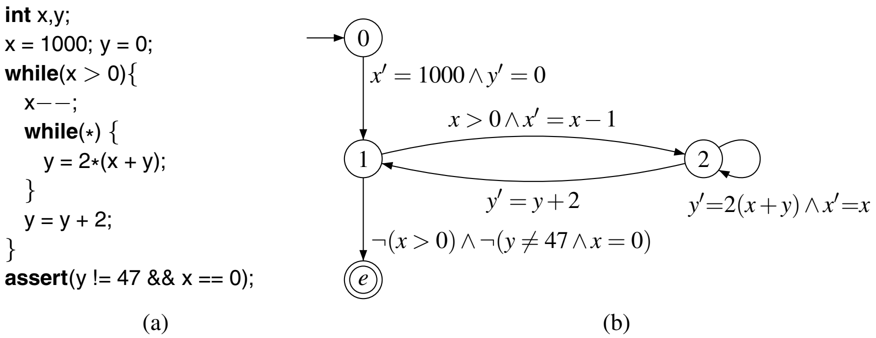
which can cause the verification process to diverge and never discover a sufficient set of predicates (even in case such predicates exist). The failure of such a refinement approach manifests in a sequence of predicates that rule out longer and longer counterexamples, but still fail to discover inductive invariants.

Following another direction, researchers have been making continuous progress on techniques for computing the transitive closure of useful classes of relations on integers [7,10,14]. These *acceleration* techniques can compute closed form representation of certain classes of loops using Presburger arithmetic.

A key contribution of this paper is an algorithmic solution to apply these specialized analyses for particular classes of loops to rule out an infinite family of counterexamples during predicate abstraction refinement. An essential ingredient of this approach are interpolants that not only rule out one path, but are also *inductive* with respect to loops along this path. We observe that we can start from any interpolant for a path that goes through a loop in the control-flow graph, and apply a postcondition (or, equivalently a weakest precondition) with respect to the transitive closure of the loop (computed using acceleration) to generalize the interpolant and make it inductive. Unlike previous theoretical proposals [12], our method treats interpolant generation and transitive closure computation as black boxes: we can start from any interpolants and strengthen it using any loop acceleration. We call the resulting technique Counterexample-Guided *Accelerated* Abstraction Refinement, or CEGAAR for short. Our experience indicates that CEGAAR works well in practice.

Motivating Example. To illustrate the power of the technique that we propose, consider the example in Figure 1. The example is smaller than the examples we consider in our evaluation (Section 6), but already illustrates the difficulty of applying existing methods.

Note that the innermost loop requires a very expressive logic to describe its closed form, so that standard techniques for computing exact transitive closure of loops do not apply. In particular, the acceleration technique does not apply to the innermost loop, and the presence of the innermost loop prevents the application of acceleration to the outer loop. On the other hand, predicate abstraction with interpolation refinement also fails to solve this example. Namely, it enters a very long refinement loop, considering



```
int x,y;
x = 1000; y = 0;
while(x > 0){
  x−−;
  while(∗) {
    y = 2∗(x + y);
  }
  y = y + 2;
}
assert(y != 47 && x == 0);
```

0
$x' = 1000 \wedge y' = 0$
$x > 0 \wedge x' = x - 1$
1
2
$y' = y + 2$
$y' = 2(x+y) \wedge x' = x$
$\neg(x > 0) \wedge \neg(y \neq 47 \wedge x = 0)$
e

(a) (b)

Fig. 1. Example Program and its Control Flow Graph with Large Block Encoding

increasingly longer spurious paths with CFG node sequences of the form $\mathbf{0}(\mathbf{12})^i\mathbf{1e}$, for $0 \leq i < 1000$. The crux of the problem is that the refinement eliminates each of these paths one by one, constructing too specific interpolants.

Our combined CEGAAR approach succeeds in proving the assertion of this program by deriving the loop invariant $y\%2 == 0 \wedge x \geq 0$. Namely, once predicate abstraction considers a path where the CFG node **1** repeats (such as **0121e**), it applies acceleration to this path. CEGAAR then uses the accelerated path to construct an inductive interpolant, which eliminates an infinite family of spurious paths. This provides predicate abstraction with a crucial predicate $y\%2 = 0$, which enables further progress, leading to the discovery of the predicate $x \geq 0$. Together, these predicates allow predicate abstraction to construct the invariant that proves program safety. Note that this particular example focuses on proving the absence of errors, but our experience suggests that CEGAAR can, in many cases, find long counterexamples faster than standard predicate abstraction.

Related Work. Predicate abstraction has proved is a rich and fruitful direction in automated verification of detailed properties of infinite-state systems [17, 19]. The pioneering work in [3] is, to the best of our knowledge, the first to propose a solution to the divergence problem in predicate abstraction. More recently, sufficient conditions to enforce convergence of refinement in predicate abstraction are given in [2], but it remains difficult to enforce them in practice. A promising direction for ensuring completeness with respect to a language of invariants is parameterizing the syntactic complexity of predicates discovered by an interpolating *split prover* [21]. Because it has the flavor of invariant enumeration, the feasibility of this approach in practice remains to be further understood.

To alleviate relatively weak guarantees of refinement in predicate abstraction in practice, researchers introduced *path invariants* [5] that rule out a family of counterexamples at once using constraint-based analysis. Our CEGAAR approach is similar in the spirit, but uses acceleration [7, 10, 14] instead of constraint-based analysis, and therefore has complementary strengths. Acceleration naturally generates precise *disjunctive invariants*, needed in many practical examples, while constraint-based invariant generation [5] resorts to an ad-hoc unfolding of the path program to generate disjunctive invariants. Acceleration can also infer expressive predicates, in particular modulo constraints, which are relevant for purposes such as proving memory address alignment.

The idea of generalizing spurious error traces was introduced also in [18], by extending an infeasible trace, labeled with interpolants, into a finite interpolant automaton. The method of [18] exploits the fact that some interpolants obtained from the infeasibility proof happen to be inductive w.r.t. loops in the program. In our case, given a spurious trace that iterates through a program loop, we *compute* the needed inductive interpolants, combining interpolation with acceleration. The method that is probably closest to CEGAAR is proposed in [12]. In this work the authors define *inductive interpolants* and prove the existence of effectivelly computable inductive interpolants for a class of affine loops, called *poly-bounded*. The approach is, however, limited to programs with one poly-bounded affine loop, for which initial and error states are specified. We only consider loops that are more restricted than the poly-bounded ones, namely loops for which transitive closures are Presburger definable. On the other hand, our method is more general in that it does not restrict the number of loops occurring in the path program, and benefits from regarding both interpolation and transitive closure computation as black boxes. The ability to compute closed forms of certain loops is also exploited in algebraic approaches [6]. These approaches can also naturally be generalized to perform useful over-approximation [1] and under-approximation.

2 Preliminaries

Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a set of variables ranging over integer numbers, and $\mathbf{x}'$ be the set $\{x'_1, \ldots, x'_n\}$. A *predicate* is a first-order arithmetic formula P. By $FV(P)$ we denote the set of free variables in P, i.e. variables not bound by a quantifier. By writing $P(\mathbf{x})$ we intend that $FV(P) \subseteq \mathbf{x}$. We write $\bot$ and $\top$ for the boolean constants false and true. A *linear term* t over a set of variables in $\mathbf{x}$ is a linear combination of the form $a_0 + \sum_{i=1}^{n} a_i x_i$, where $a_0, a_1, \ldots, a_n \in \mathbb{Z}$. An *atomic proposition* is a predicate of the form $t \leq 0$, where t is a linear term. *Presburger arithmetic* is the first-order logic over propositions $t \leq 0$; Presburger arithmetic has quantifier elimination and is decidable.For simplicity we consider only formulas in Presburger arithmetic in this paper. A valuation of $\mathbf{x}$ is a function $\nu : \mathbf{x} \rightarrow \mathbb{Z}$. If ν is a valuation of $\mathbf{x}$, we denote by $\nu \models P$ the fact that the formula obtained by replacing each occurrence of x_i with $\nu(x_i)$ is valid. Similarly, an arithmetic formula $R(\mathbf{x}, \mathbf{x}')$ defining a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$ is evaluated referring to two valuations ν_1, ν_2; the satisfaction relation is denoted $\nu_1, \nu_2 \models R$. The composition of two relations $R_1, R_2 \in \mathbb{Z}^n \times \mathbb{Z}^n$ is denoted by $R_1 \circ R_2 = \{(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \exists \mathbf{t} \in \mathbb{Z}^n \ . \ (\mathbf{u}, \mathbf{t}) \in R_1 \text{ and } (\mathbf{t}, \mathbf{v}) \in R_2\}$. Let ε be the identity relation $\{(\mathbf{u}, \mathbf{u}) \mid \mathbf{u} \in \mathbb{Z}^n \times \mathbb{Z}^n\}$. We define $R^0 = \varepsilon$ and $R^i = R^{i-1} \circ R$, for any $i > 0$. With these notations, $R^+ = \bigcup_{i=1}^{\infty} R^i$ denotes the *transitive closure* of R, and $R^* = R^+ \cup \varepsilon$ denotes the *reflexive and transitive closure* of R. We sometimes use the same symbols to denote a relation and its defining formula. For a set of n-tuples $S \subseteq \mathbb{Z}^n$ and a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, let $post(S, R) = \{\mathbf{v} \in \mathbb{Z}^n \mid \exists \mathbf{u} \in S \ . \ (\mathbf{u}, \mathbf{v}) \in R\}$ denote the *strongest postcondition* of S via R, and $wpre(S, R) = \{\mathbf{u} \in \mathbb{Z}^n \mid \forall \mathbf{v} \ . \ (\mathbf{u}, \mathbf{v}) \in R \rightarrow \mathbf{v} \in S\}$ denote the *weakest precondition* of S with respect to R. We use *post* and *wpre* for sets and relations, as well as for logical formulae defining them.

We represent programs as control flow graphs. A *control flow graph* (CFG) is a tuple $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ where $\mathbf{x} = \{x_1, \ldots, x_n\}$ is a set of variables, Q is a set of *control states*, $\rightarrow$ is a set of edges of the form $q \xrightarrow{R} q'$, labeled with arithmetic formulae defining relations $R(\mathbf{x}, \mathbf{x}')$, and $I, E \subseteq Q$ are sets of *initial* and *error* states, respectively. A *path* in G is a sequence $\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$, where $q_1, q_2, \ldots, q_n \in Q$ and $q_i \xrightarrow{R_i} q_{i+1}$ is an edge in G, for each $i = 1, \ldots, n-1$. We assume without loss of generality that all variables in $\mathbf{x} \cup \mathbf{x}'$ appear free in each relation labeling an edge of G[1]. We denote the relation $R_1 \circ R_2 \circ \ldots \circ R_{n-1}$ by $\rho(\theta)$ and assume that the set of free variables of $\rho(\theta)$ is $\mathbf{x} \cup \mathbf{x}'$. The path θ is said to be a *cycle* if $q_1 = q_n$, and a *trace* if $q_1 \in I$. The path θ is said to be *feasible* if and only if there exist valuations $\nu_1, \ldots, \nu_n : \mathbf{x} \rightarrow \mathbb{Z}$ such that $\nu_i, \nu_{i+1} \models R_i$, for all $i = 1, \ldots, n-1$. A control state is said to be *reachable* in G if it occurs on a feasible trace.

[1] For variables that are not modified by a transition, this can be achieved by introducing an explicit update $x' = x$.

Acceleration. The goal of acceleration is, given a relation R in a fragment of integer arithmetic, to compute its reflexive and transitive closure, R^*. In general, defining R^* in a decidable fragment of integer arithmetic is not possible, even when R is definable in a decidable fragment such as, e.g. Presburger arithmetic. In this work we consider two fragments of arithmetic in which transitive closures of relations are Presburger definable.

An *octagonal relation* is a relation defined by a constraint of the form $\pm x \pm y \leq c$, where x and y range over the set $\mathbf{x} \cup \mathbf{x}'$, and c is an integer constant. The transitive closure of an octagonal relation has been shown to be Presburger definable and effectively computable [10]. A *linear affine relation* is a relation of the form $\mathcal{R}(\mathbf{x}, \mathbf{x}') \equiv C\mathbf{x} \geq \mathbf{d} \wedge \mathbf{x}' = A\mathbf{x} + \mathbf{b}$, where $A \in \mathbb{Z}^{n \times n}$, $C \in \mathbb{Z}^{p \times n}$ are matrices and $\mathbf{b} \in \mathbb{Z}^n$, $\mathbf{d} \in \mathbb{Z}^p$. $\mathcal{R}$ is said to have the *finite monoid property* if and only if the set $\{A^i \mid i \geq 0\}$ is finite. It is known that the finite monoid condition is decidable [7], and moreover that the transitive closure of a finite monoid affine relation is Presburger definable and effectively computable [7,14].

Predicate Abstraction. Informally, predicate abstraction computes an overapproximation of the transition system generated by a program and verifies whether an error state is reachable in the abstract system. If no error occurs in the abstract system, the algorithm reports that the original system is safe. Otherwise, if a path to an error state (counterexample) has been found in the abstract system, the corresponding concrete path is checked. If this latter path corresponds to a real execution of the system, then a real error has been found. Otherwise, the abstraction is refined in order to exclude the counterexample, and the procedure continues.

Given a CFG $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$, and a (possibly infinite) set of predicates $\mathcal{P}$, an *abstract reachability tree* (ART) for G is a tuple $T = \langle S, \pi, r, e \rangle$ where $S \subseteq Q \times 2^{\mathcal{P} \setminus \{\bot\}}$ is a set of nodes (notice that for no node $\langle q, \Phi \rangle$ in T we may have $\bot \in \Phi$), $\pi : Q \rightarrow 2^{\mathcal{P}}$ is a mapping associating control states with sets of predicates, $i \in I \times \{\top\}$ is the root node, $e \subseteq S \times S$ is a tree-structured edge relation:

- all nodes in S are reachable from the root r
- for all $n, m, p \in S$, $e(n, p) \wedge e(m, p) \Rightarrow n = m$
- $e(\langle q_1, \Phi_1 \rangle, \langle q_2, \Phi_2 \rangle) \Rightarrow q_1 \xrightarrow{R} q_2$ and $\Phi_2 = \{P \in \pi(q_2) \mid post(\bigwedge \Phi_1, R) \rightarrow P\}$

We say that an ART node $\langle q_1, \Phi_1 \rangle$ is *subsumed* by another node $\langle q_2, \Phi_2 \rangle$ if and only if $q_1 = q_2$ and $\bigwedge \Phi_1 \rightarrow \bigwedge \Phi_2$. It is usually considered that no node in an ART is subsumed by another node, from the same ART.

It can be easily checked that each path $\sigma : r = \langle q_1, \Phi_1 \rangle, \langle q_2, \Phi_2 \rangle, \ldots, \langle q_k, \Phi_k \rangle$, starting from the root in T, can be mapped into a trace $\theta : q_1 \xrightarrow{R_1} q_2 \ldots q_{k-1} \xrightarrow{R_{k-1}} q_k$ of G, such that $post(\top, \rho(\theta)) \rightarrow \bigwedge \Phi_k$. We say that θ is a *concretization* of σ, or that σ concretizes to θ. A path in an ART is said to be *spurious* if none of its concretizations is feasible.

3 Interpolation-Based Abstraction Refinement

By *refinement* we understand the process of enriching the predicate mapping π of an ART $T = \langle S, \pi, r, e \rangle$ with new predicates. The goal of refinement is to prevent spurious counterexamples (paths to an error state) from appearing in the ART. To this end, an effective technique used in many predicate abstraction tools is that of *interpolation*.

Given an unsatisfiable conjunction $A \wedge B$, an interpolant I is a formula using the common variables of A and B, such that $A \rightarrow I$ is valid and $I \wedge B$ is unsatisfiable. Intuitively, I is the explanation behind the unsatisfiability of $A \wedge B$. Below we introduce a slightly more general definition of a *trace interpolant*.

Definition 1 ([21]). *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and*

$$\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$$

be an infeasible trace of G. An interpolant *for θ is a sequence of predicates $\langle I_1, I_2, \ldots, I_n \rangle$ with free variables in $\mathbf{x}$, such that: $I_1 = \top$, $I_n = \bot$, and for all $i = 1, \ldots, n-1$, $post(I_i, R_i) \rightarrow I_{i+1}$.*

Interpolants exist for many theories, including all theories with quantifier elimination, and thus for Presburger arithmetic. Moreover, a trace is infeasible if and only if it has an interpolant. Including any interpolant of an infeasible trace into the predicate mapping of an ART suffices to eliminate any abstraction of the trace from the ART. We can thus refine the ART and exclude an infeasible trace by including the interpolant that proves the infeasibility of the trace.

Note that the refinement technique using Definition 1 only guarantees that *one* spurious counterexample is eliminated from the ART with each refinement step. This fact hinders the efficiency of predicate abstraction tools, which must rely on the ability of theorem provers to produce interpolants that are general enough to eliminate more than one spurious counterexample at the time. The following is a stronger notion of an interpolant, which ensures generality with respect to an infinite family of counterexamples.

Definition 2 ([12], Def. 2.4). *Given a CFG G, a* trace scheme *in G is a sequence:*

$$\xi : q_0 \xrightarrow{Q_1} \overset{\overset{L_1}{\curvearrowright}}{q_1} \xrightarrow{Q_2} \ldots \xrightarrow{Q_{n-1}} \overset{\overset{L_{n-1}}{\curvearrowright}}{q_{n-1}} \xrightarrow{Q_n} \overset{\overset{L_n}{\curvearrowright}}{q_n} \xrightarrow{Q_{n+1}} q_{n+1} \qquad (1)$$

where $q_0 \in I$ and:

- *$Q_i = \rho(\theta_i)$, for some non-cyclic paths θ_i of G, from q_{i-1} to q_i*
- *$L_i = \bigvee_{j=1}^{k_i} \rho(\lambda_{ij})$, for some cycles λ_{ij} of G, from q_i to q_i*

Intuitivelly, a trace scheme represents an infinite regular set of traces in G. The trace scheme is said to be *feasible* if and only if at least one trace of G of the form $\theta_1;\lambda_{1i_1}\ldots\lambda_{1i_{j_1}};\theta_2;\ldots;\theta_n;\lambda_{ni_n}\ldots\lambda_{ni_{j_n}};\theta_{n+1}$ is feasible.

The trace scheme is said to be *bounded* if $k_i = 1$, for all $i = 1, 2, \ldots, n$. A bounded[2] trace scheme is a regular language of traces, of the form $\sigma_1 \cdot \lambda_1^* \cdot \ldots \cdot \sigma_n \cdot \lambda_n^* \cdot \sigma_{n+1}$, where σ_i are acyclic paths, and λ_i are cycles of G.

Definition 3 ([12], Def. 2.5). *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and ξ be an infeasible trace scheme of the form (1). An* interpolant *for ξ is a sequence of predicates $\langle I_0, I_1, I_2, \ldots, I_n, I_{n+1} \rangle$, with free variables in $\mathbf{x}$, such that:*

[2] This term is used in analogy with the notion of bounded languages [16].

1. $I_0 = \top$ *and* $I_{n+1} = \bot$
2. $post(I_i, Q_{i+1}) \rightarrow I_{i+1}$*, for all* $i = 0, 1, \ldots, n$
3. $post(I_i, L_i) \rightarrow I_i$*, for all* $i = 1, 2, \ldots, n$

The main difference with Definition 1 is the third requirement, namely that each interpolant predicate (except for the first and the last one) must be *inductive* with respect to the corresponding loop relation. It is easy to see that each of the two sequences:

$$\langle \top,\ post(\top, Q_1 \circ L_1^*),\ \ldots,\ post(\top, Q_1 \circ L_1^* \circ Q_2 \circ \ldots Q_n \circ L_n^*) \rangle \qquad (2)$$

$$\langle wpre(\bot, Q_1 \circ L_1^* \circ Q_2 \circ \ldots Q_n \circ L_n^*),\ \ldots,\ wpre(\bot, Q_n \circ L_n^*),\ \bot \rangle \qquad (3)$$

are interpolants for ξ, provided that ξ is infeasible (Lemma 2.6 in [12]). Just as for finite trace interpolants, the existence of an inductive interpolant suffices to prove the infeasibility of the entire trace scheme.

Lemma 4. *Let* $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ *be a CFG and* ξ *be an infeasible trace scheme of* G *of the form (1). If* $T = \langle S, \pi, r, e \rangle$ *is an ART for* G*, such that there exists an interpolant* $\langle I_i \in \pi(q_i) \rangle_{i=0}^{n+1}$ *for* ξ*, then no path in* T *concretizes to a trace in* ξ*.*

4 Counterexample-Guided Accelerated Abstraction Refinement

This section presents the CEGAAR algorithm for predicate abstraction with interpolant-based accelerated abstraction refinement. Since computing the interpolant of a trace scheme is typically more expensive than computing the interpolant of a finite counterexample, we apply acceleration in a demand-driven fashion. The main idea of the algorithm is to accelerate only those counterexamples in which some cycle repeats a certain number of times. For example, if the abstract state exploration has already ruled out the spurious counterexamples $\sigma \cdot \tau$, $\sigma \cdot \lambda \cdot \tau$ and $\sigma \cdot \lambda \cdot \lambda \cdot \tau$, when it sees next the spurious counterexample $\sigma \cdot \lambda \cdot \lambda \cdot \lambda \cdot \tau$, it will accelerate it into $\sigma \cdot \lambda^* \cdot \tau$, and rule out all traces which comply to this scheme. The maximum number of cycles that are allowed to occur in the acyclic part of an error trace, before computing the transitive closure, is called the *delay*, and is a parameter of the algorithm (here the delay was 2). A smaller delay results in a more aggressive acceleration strategy, whereas setting the delay to infinity is equivalent to performing predicate abstraction without acceleration.

The main procedure is CONSTRUCTART which builds an ART for a given CFG, and an abstraction of the set of initial values (Fig. 2). CONSTRUCTART is a worklist algorithm that expands the ART according to a certain exploration strategy (depth-first, breadth-first, etc.) determined by the type of the structure used as a worklist. We assume without loss of generality that the CFG has exactly one initial vertex *Init*. The CONSTRUCTART procedure starts with *Init* and expands the tree according to the definition of the ART (lines 11 and 12). New ART nodes are constructed using NEWARTNODE, which receives a CFG state and a set of predicates as arguments. The algorithm backtracks from expanding the ART when either the current node contains $\bot$ in its set of predicates, or it is subsumed by another node in the ART (line 13). In the algorithm (Fig. 2), we denote logical entailment by $\phi \vdash \psi$ in order to avoid confusion.

```
input CFG $G = \langle \mathbf{x}, Q, \rightarrow, \{Init\}, E \rangle$
output ART $T = \langle S, \pi, Root, e \rangle$
$WorkList = []$, $S, \pi, e = \emptyset$, $Root = nil$
def ConstructART(Init, initialAbstraction) {
  node = newARTnode(Init, initialAbstraction)
  if (Root = nil) Root = node
  $WorkList.add(\langle Init, node \rangle)$
  while (!($WorkList.empty$)) {
    $\langle nextCFGvertex, nextARTnode \rangle = WorkList.remove()$
    for ($child = children(nextCFGVertex)$) {
      Let $R$ be such that $nextCFGvertex \xrightarrow{R} child$ in $G$
      $\Phi = \{p \in \pi(child) \mid \textsc{post}(\bigwedge nextARTnode.abstraction, R) \vdash p\}$
      if ($\bot \in \Phi$ or $\big(\exists$ an ART node$\langle child, \Psi \rangle$ . $\wedge \Phi \vdash \Psi\big)$)
        continue
      $node = newARTnode(child, \Phi)$
      $S = S \cup \{node\}$
      $e = e \cup \{(nextARTnode, node)\}$
      if ($child \in E$and $checkRefineError(node)$)
        report "ERROR"
      $WorkList.add(\langle child, node \rangle)$
      $WorkList.removeAll($nodes from $WorkList$ subsumed by $node)$ }}}
```

Fig. 2. The CEGAAR algorithm (a) - High-Level Structure

The refinement step is performed by the CHECKREFINEERROR function (Fig. 3). This function returns true if and only if a feasible error trace has been detected; otherwise, further predicates are generated to refine the abstraction. First, a minimal infeasible ART path to *node* is determined (line 4). This path is generalized into a trace scheme (line 6). The generalization function FOLD takes *Path* and the delay parameter δ as input and produces a trace scheme which contains *Path*. The FOLD function creates a trace scheme of the form (1) out of the spurious path given as argument. The spurious path is traversed and control states are recorded in a list. When we encounter a control state which is already in the list, we identified an elementary cycle λ. If the current trace scheme ends with at least δ occurrences of λ, where $\delta \in \mathbb{N}_\infty$ is the delay parameter, then λ is added as a loop to the trace scheme, provided that its transitive closure can be effectivelly computed. For efficiency reasons, we syntactically check the relation on the loop, namely we check whether the relation is syntactically compliant to the definition of octagonal relations. Notice that a relation can be definable by an octagonal constraint even if it is not a conjunction of octagonal constraints, i.e. it may contain redundant atomic propositions which are not of this form. Once the folded trace scheme is obtained, there are three possibilities:

1. If the trace scheme is not bounded (the test on line 7 passes), we compute a bounded overapproximation of it, in an attempt to prove its infeasibility (line 8). If the test on line 9 succeeds, the original trace scheme is proved to be infeasible and the ART is refined using the interpolants for the overapproximated trace scheme.

languagelanguage

```
1  def checkRefineError($node$): Boolean {
2    $traceScheme = []$
3    while (the ART path $Root \to \cdots \to node$ is spurious) {
4      Let $Path = \langle q_1, \Phi_1 \rangle \to \ldots \to \langle q_n, \Phi_n \rangle$ be the (unique) minimal ART path with
5      $pivot = \langle q_1, \Phi_1 \rangle$ and $\langle q_n, \Phi_n \rangle = node$ such that the CFG path $q_1 \to \cdots \to q_n$ is infeasible
6      $newScheme = Fold(Path, delay)$
7      if (!$isBounded(newScheme)$) {
8        $absScheme = Concat(Overapprox(newScheme), traceScheme)$
9        if (interpolateRefine(absScheme, pivot) return false
10         else $newScheme = Underapprox(newScheme, Path)$ }
11     $traceScheme = Concat(newScheme, traceScheme)$
12     if (interpolateRefine(traceScheme, pivot) return false
13     $node = Path.head$}
14   return true }
```

Fig. 3. The CEGAAR algorithm (b) - Accelerated Refinement

2. Else, if the overapproximation was found to be feasible, it could be the case that the abstraction of the scheme introduced a spurious error trace. In this case, we compute a bounded underapproximation of the trace scheme, which contains the initial infeasible path, and replace the current trace scheme with it (line 10). The only requirement we impose on the UNDERAPPROX function is that the returned bounded trace scheme contains *Path*, and is a subset of *newScheme*.
3. Finally, if the trace scheme is bounded (either because the test on line 7 failed, or because the folded path was replaced by a bounded underapproximation on line 10) and also infeasible (the test on line 12 passes) then the ART is refined with the interpolants computed for the scheme. If, on the other hand, the scheme is feasible, we continue searching for an infeasible trace scheme starting from the head of *Path* upwards (line 13).

Example Let $\theta : q_1 \xrightarrow{P} q_2 \xrightarrow{Q} q_2 \xrightarrow{R} q_1 \xrightarrow{P} q_2 \xrightarrow{R} q_1$ be a path. The result of applying FOLD to this path is the trace scheme ξ shown in the left half of Fig. 4. Notice that this path scheme is not bounded, due to the presence of two loops starting and ending with q_2. A possible bounded underapproximation of ξ, containing the original path θ, is shown in the right half of Fig. 4. □

The iteration stops either when a refinement is possible (lines 9, 12), in which case CHECKREFINEERROR returns false, or when the search reaches the root of the ART and the trace scheme is feasible, in which case CHECKREFINEERROR returns true (line

$$\begin{tikzcd}
q_1 \arrow[r, "P"] & q_2 \arrow[r, "R"] \arrow[loop above, "Q"] \arrow[d, shift left, "R"] & q_1 \\
 & q_1 \arrow[u, shift left, "P"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
q_1 \arrow[r, "P"] & q_2 \arrow[r, "\varepsilon"] \arrow[loop above, "Q"] \arrow[d, shift left, "R"] & q_2 \arrow[r, "R"] & q_1 \\
 & q_1 \arrow[u, shift left, "P"] & &
\end{tikzcd}$$

Fig. 4. Underapproximation of unbounded trace schemes. ε stands for the identity relation.

languagelanguage

```
1  def InterpolateRefine(traceScheme, Pivot) : Boolean {
2    metaTrace = TransitiveClosure(traceScheme)
3    interpolant = InterpolatingProverCall(metaTrace)
4    if (interpolant = $\emptyset$) return false
5    I = AccelerateInterpolant(interpolant)
6    for ($\psi \in$ I) {
7      let $v$ be the CFG vertex corresponding to $\psi$
8      $\pi = \pi[v \leftarrow (\pi(v) \cup \psi)]$
9    }
10   ConstructART(Pivot,Pivot.abstraction)
11   return true }
```

Fig. 5. The Interpolation Function

14) and the main algorithm in Figure 2 reports a true counterexample. Notice that, since we update *node* to the head of *Path* (line 13), the position of *node* is moved upwards in the ART. Since this cannot happen indefinitely, the main loop (lines 3-13) of the checkRefineError is bound to terminate.

The interpolateRefine function is used to compute the interpolant of the trace scheme, update the predicate mapping π of the ART, and reconstruct the subtree of the ART whose root is the first node on *Path* (this is usually called the *pivot* node). The InterpolateRefine (Fig. 5) function returns true if and only if its argument represents an infeasible trace scheme. In this case, new predicates, obtained from the interpolant of the trace scheme, are added to the nodes of the ART. This function uses internally the TransitiveClosure procedure (line 2) in order to generate the meta-trace scheme (5). The AccelerateInterpolant function (line 5) computes the interpolant for the trace scheme, from the resulting meta-trace scheme. Notice that the refinement algorithm is recursive, as ConstructART calls checkRefineError (line 18), which in turn calls InterpolateRefine (lines 9,12), which calls back ConstructART (line 10). Our procedure is *sound,* in the sense that whenever function ConstructART terminates with a non-error result, the input program does not contain any reachable error states. Vice versa, if a program contains a reachable error state, ConstructART is guaranteed to eventually discover a feasible path to this state, since the use of a work list ensures fairness when exploring ARTs.

5 Computing Accelerated Interpolants

This section describes a method of refining an ART by excluding an infinite family of infeasible traces at once. Our method combines interpolation with acceleration in a way which is oblivious of the particular method used to compute interpolants. For instance, it is possible to combine proof-based [23] or constraint-based [26] interpolation with acceleration, whenever computing the precise transitive closure of a loop is possible. In cases when the precise computation fails, we may resort to both over- and under-approximation of the transitive closure. In both cases, the accelerated interpolants are at least as general (and many times more general) than the classical interpolants extracted from a finite counterexample trace.

5.1 Precise Acceleration of Bounded Trace Schemes

We consider first the case of bounded trace schemes of the form (1), where the control states $q_1,\ldots,q_n$ belong to some cycles labeled with relations $L_1,\ldots,L_n$. Under some restrictions on the syntax of the relations labeling the cycles L_i, the reflexive transitive closures L_i^* are effectively computable using acceleration algorithms [7,9,14]. Among the known classes of relations for which acceleration is possible we consider: *octagonal relations* and *finite monoid affine transformations*. These are all conjunctive linear relations. We consider in the following that all cycle relations L_i belong to one of these classes. Under this restriction, any infeasible bounded trace scheme has an effectivelly computable interpolant of one of the forms (2),(3).

However, there are two problems with applying definitions (2),(3) in order to obtain interpolants of trace schemes. On one hand, relational composition typically requires expensive quantifier eliminations. The standard proof-based interpolation techniques (e.g. [23]) overcome this problem by extracting the interpolants directly from the proof of infeasibility of the trace. Alternatively, constraint-based interpolation [26] reduce the interpolant computation to a Linear Programming problem, which can be solved by efficient algorithms. Both methods apply, however, only to finite traces, and not to infinite sets of traces defined as trace schemes. Another, more important, problem is related to the sizes of the interpolant predicates from (2), (3) compared to the sizes of interpolant predicates obtained by proof-theoretic methods (e.g. [22]), as the following example shows.

Example Let $R(x,y,x',y') : x' = x+1 \wedge y' = y+1$ and $\phi(x,y,\ldots)$, $\psi(x,y,\ldots)$ be some complex Presburger arithmetic formulae. The trace scheme:

$$q_0 \xrightarrow{z=0\wedge z'=z\wedge\phi} \overset{z'=z+2\wedge R}{\overset{\curvearrowright}{q_1}} \xrightarrow{z=5\wedge\psi} q_2 \qquad (4)$$

is infeasible, because z remains even, so it cannot become equal 5. One simple interpolant for this trace scheme has at program point q_1 the formula $z\%2 = 0$. On the other hand, the strongest interpolant has $(z = 0 \wedge z' = x \wedge \phi) \circ (z' = z+2 \wedge R)^*$ at q_1, which is typically a much larger formula, because of the complex formula ϕ. Note however that ϕ and R do not mention z, so they are irrelevant. □

To construct useful interpolants instead of the strongest or the weakest ones, we therefore proceed as follows. Let ξ be a bounded trace scheme of the form (1). For each control loop $q_i \xrightarrow{R_i} q_i$ of ξ, we define the corresponding *meta-transition* $q_i' \xrightarrow{R_i^*} q_i''$ labeled with the reflexive and transitive closure of R_i. Intuitively, firing the meta-transition has the same effect as iterating the loop an arbitrary number of times. We first replace each loop of ξ by the corresponding meta-transition. The result is the *meta-trace*:

$$\overline{\xi} : q_0 \xrightarrow{Q_1} q_1' \xrightarrow{L_1^*} q_1'' \xrightarrow{Q_2} q_2' \ \ldots\ q_{n-1}'' \xrightarrow{Q_n} q_n' \xrightarrow{L_n^*} q_n'' \xrightarrow{Q_{n+1}} q_{n+1} \qquad (5)$$

Since we supposed that ξ is an infeasible trace scheme, the (equivalent) finite meta-trace $\overline{\xi}$ is infeasible as well, and it has an interpolant $I_{\overline{\xi}} = \langle \top, I_1', I_1'', I_2', I_2'', \ldots, I_n', I_n'', \bot \rangle$

in the sense of Definition 1. This interpolant is not an interpolant of the trace scheme ξ, in the sense of Definition 3. In particular, none of I'_i, I''_i is guaranteed to be inductive with respect to the loop relations L_i. To define compact inductive interpolants based on $I_{\overline{\xi}}$ and the transitive closures L_i^*, we consider the following sequences:

$$\begin{aligned} I_\xi^{post} &= \langle \top, post(I'_1, L_1^*), post(I'_2, L_2^*), \ldots, post(I'_n, L_n^*), \bot \rangle \\ I_\xi^{wpre} &= \langle \top, wpre(I''_1, L_1^*), wpre(I''_2, L_2^*), \ldots, wpre(I''_n, L_n^*), \bot \rangle \end{aligned}$$

The following lemma proves the correctness of this approach.

Lemma 5. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and ξ be an infeasible trace scheme of the form (1). Then I_ξ^{post} and I_ξ^{wpre} are interpolants for ξ, and moreover $I_{\xi_i}^{wpre} \rightarrow I_{\xi_i}^{post}$, for all $i = 1, 2, \ldots, n$.*

Notice that computing I_ξ^{post} and I_ξ^{wpre} requires n relational compositions, which is, in principle, just as expensive as computing directly one of the extremal interpolants (2),(3). However, by re-using the meta-trace interpolants, one potentially avoids the worst-case combinatorial explosion in the size of the formulae, which occurs when using (2), (3) directly.

Example. Let us consider again the trace scheme (4). The corresponding infeasible finite trace $\overline{\xi}$ is:

$$q_0 \xrightarrow{z=0 \wedge z'=z \wedge \phi} q'_1 \xrightarrow{\exists k \geq 0 \ . \ z'=z+2k \ \wedge \ x'=x+k \ \wedge \ y'=y+k} q''_1 \xrightarrow{z=5 \wedge \psi} q_2$$

A possible interpolant for this trace is $\langle \top, z = 0, \exists k \geq 0 \ . \ z = 2k, \bot \rangle$. An inductive interpolant for the trace scheme, derived from it, is $I_\xi^{post} = \langle \top, post(z = 0, \exists k \geq 0 . z' = z + 2k \wedge x' = x + k \wedge y' = y + k), \bot \rangle = \langle \top, \ z\%2 = 0, \ \bot \rangle$. □

5.2 Bounded Overapproximations of Trace Schemes

Consider a trace scheme (1), not necessarily bounded, where the transitive closures of the relations L_i labeling the loops are not computable by any available acceleration method [7, 9, 14]. One alternative is to find abstractions $L_i^\sharp$ of the loop relations, i.e. relations $L_i^\sharp \leftarrow L_i$, for which transitive closures are computable. If the new abstract trace remains infeasible, it is possible to compute an interpolant for it, which is an interpolant for the original trace scheme. However, replacing the relations L_i with their abstractions $L_i^\sharp$ may turn an infeasible trace scheme into a feasible one, where the traces introduced by abstraction are spurious. In this case, we give up the overapproximation, and turn to the underapproximation technique described in the next section.

The overapproximation method computes an interpolant for a trace scheme ξ of the form (1) under the assumption that the abstract trace scheme:

$$\xi^\sharp : q_0 \xrightarrow{Q_1} \overset{L_1^\sharp}{\overset{\curvearrowright}{q_1}} \xrightarrow{Q_2} \ldots \xrightarrow{Q_{n-1}} \overset{L_{n-1}^\sharp}{\overset{\curvearrowright}{q_{n-1}}} \xrightarrow{Q_n} \overset{L_n^\sharp}{\overset{\curvearrowright}{q_n}} \xrightarrow{Q_{n+1}} q_{n+1} \qquad (6)$$

is infeasible. In this case one can effectivelly compute the interpolants $I^{post}_{\xi^\sharp}$ and $I^{wpre}_{\xi^\sharp}$, since the transitive closures of the abstract relations labeling the loops are computable by acceleration. The following lemma proves that, under certain conditions, computing an interpolant for the abstraction of a trace scheme is sound.

Lemma 6. *Let G be a CFG and ξ be a trace scheme (1) such that the abstract trace scheme $\xi^\sharp$ (6) is infeasible. Then the interpolants $I^{post}_{\xi^\sharp}$ and $I^{wpre}_{\xi^\sharp}$ for $\xi^\sharp$ are also interpolants for ξ.*

5.3 Bounded Underapproximations of Trace Schemes

Let ξ be a trace scheme of the form (1), where each relation L_i labeling a loop is a disjunction $L_{i1} \vee \ldots \vee L_{ik_i}$ of relations for which the transitive closures are effectively computable and Presburger definable. A *bounded underapproximation scheme* of a trace scheme ξ is obtained by replacing each loop $q_i \xrightarrow{L_i} q_i$ in ξ by a bounded trace scheme of the form:

$$\begin{tikzcd} q_i^1 \arrow[loop above, "L_{i1}"] \arrow[r, "\varepsilon"] & q_i^2 \arrow[loop above, "L_{i2}"] \arrow[r, "\varepsilon"] & \ldots\ q_i^{k_i} \arrow[loop above, "L_{ik_i}"] \end{tikzcd}$$

where ε denotes the identity relation. Let us denote[3] the result of this replacement by $\xi^\flat$. It is manifest that the set of traces $\xi^\flat$ is included in ξ.

Since we assumed that the reflexive and transitive closures L^*_{ij} are effectivelly computable and Presburger definable, the feasibility of $\xi^\flat$ is a decidable problem. If $\xi^\flat$ is found to be feasible, this points to a real error trace in the system. On the other hand, if $\xi^\flat$ is found to be infeasible, let $I_{\xi^\flat} = \langle \top, I_1^1, \ldots, I_1^{k_1}, \ldots, I_n^1, \ldots, I_n^{k_n}, \bot \rangle$ be an interpolant for $\xi^\flat$. A refinement scheme using this interpolant associates the predicates $\{I_i^1, \ldots, I_i^{k_i}\}$ with the control state q_i from the original CFG. As the following lemma shows, this guarantees that any trace that follows the pattern of $\xi^\flat$ is excluded from the ART, ensuring that a refinement of the ART using a suitable underapproximation (that includes a spurious counterexample) is guaranteed to make progress.

Lemma 7. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG, ξ be an infeasible trace scheme of G (1) and $\xi^\flat$ a bounded underapproximation of ξ. If $T = \langle S, \pi, r, e \rangle$ is an ART for G, such that $\{I_i^1, \ldots, I_i^{k_i}\} \subseteq \pi(q_i)$, then no path in T concretizes to a trace in $\xi^\flat$.*

Notice that a refinement scheme based on underapproximation guarantees the exclusion of those traces from the chosen underapproximation trace scheme, and not of all traces from the original trace scheme. Since a trace scheme is typically obtained from a finite counterexample, an underapproximation-based refinement still guarantees that the particular counterexample is excluded from further searches. In other words, using underapproximation is still better than the classical refinement method, since it can potentially exclude an entire family of counterexamples (including the one generating the underapproximation) at once.

[3] The choice of the name depends on the ordering of particular paths $L_{i1}, L_{i2}, \ldots, L_{ik_i}$, however we shall denote any such choice in the same way, in order to keep the notation simple.

6 Experimental Results

We have implemented CEGAAR by building on the predicate abstraction engine Eldarica[4] [20], the FLATA verifier[5] [20] based on acceleration, and the Princess interpolating theorem prover [11,25]. Tables in Figure 6 compares the performance of the Flata, Eldarica, *static acceleration* and CEGAAR on a number of benchmarks (the platorm used for experiments is Intel® Core™2 Duo CPU P8700, 2.53GHz with 4GB of RAM).

Model	Time [s]			
	F.	E.	S.	D.
(a) Examples from [21]				
anubhav (C)	0.8	3.0	4.0	3.1
copy1 (E)	2.0	7.2	5.8	5.9
cousot (C)	0.6	-	6.2	5.9
loop1 (E)	1.7	7.1	5.2	5.4
loop (E)	1.8	5.9	4.8	5.4
scan (E)	3.3	-	5.1	5.0
string_concat1 (E)	5.3	-	10.1	7.3
string_concat (E)	4.9	-	7.0	7.5
string_copy (E)	4.6	-	6.3	5.7
substring1 (E)	0.6	9.4	18.2	8.3
substring (E)	2.1	3.3	6.3	3.5
(b) Verification conditions for array programs [9]				
rotation_vc.1 (C)	0.6	2.0	9.5	2.0
rotation_vc.2 (C)	1.6	2.2	18.5	2.2
rotation_vc.3 (C)	1.2	0.3	18.3	0.3
rotation_vc.1 (E)	1.1	1.3	10.2	1.3
split_vc.1 (C)	3.9	3.7	91.1	3.6
split_vc.2 (C)	3.0	2.3	74.1	2.2
split_vc.3 (C)	3.3	0.6	75.0	0.6
split_vc.1 (E)	28.5	2.3	185.6	2.4

Model	Time [s]			
	F.	E.	S.	D.
(c) Examples from [24]				
boustrophedon (C)	-	-	-	14.4
gopan (C)	0.4	-	-	6.4
halbwachs (C)	-	-	7.3	7.0
rate_limiter (C)	31.7	6.1	8.1	5.5
(d) Examples from L2CA [8]				
bubblesort (E)	14.9	9.9	9.5	9.3
insdel (E)	0.1	1.3	2.5	1.4
insertsort (E)	2.0	4.2	5.0	4.0
listcounter (C)	0.3	-	1.9	3.7
listcounter (E)	0.3	1.4	1.6	1.4
listreversal (C)	4.5	3.0	6.0	3.3
listreversal (E)	0.8	2.7	8.1	2.8
mergesort (E)	1.2	7.7	21.3	7.4
selectionsort (E)	1.5	8.1	13.7	7.7
(e) NECLA benchmarks				
inf1 (E)	0.2	2.0	2.0	2.0
inf4 (E)	0.9	3.7	3.7	3.7
inf6 (C)	0.1	2.0	2.0	2.0
inf8 (C)	0.3	3.6	3.4	3.9

Model	Time [s]			
	F.	E.	S.	D.
(f) Examples from [15]				
h1 (E)	-	5.1	5.6	5.1
h1.optim (E)	0.8	2.9	5.5	2.9
h1h2 (E)	-	9.4	10.1	12.2
h1h2.optim (E)	1.1	3.3	4.4	3.4
simple (E)	-	6.4	7.0	8.4
simple.optim (E)	0.8	3.0	5.1	2.9
test0 (C)	-	23.0	23.4	29.2
test0.optim (C)	0.3	3.2	5.4	3.2
test0 (E)	-	5.4	5.9	5.7
test0.optim (E)	0.6	3.0	5.8	2.9
test1.optim (C)	0.9	4.7	5.9	7.8
test1.optim (E)	1.5	4.4	5.9	4.7
test2_1.optim (E)	1.6	5.2	5.5	5.6
test2_2.optim (E)	2.9	4.6	5.9	4.6
test2.optim (C)	6.4	27.2	30.1	30.0
wrpc.manual (C)	0.6	1.2	1.4	1.2
wrpc (E)	-	7.9	8.4	8.2
wrpc.optim (E)	-	5.1	8.5	5.2
(g) VHDL models from [27]				
counter (C)	0.1	1.6	1.6	1.6
register (C)	0.2	1.1	1.1	1.1
synlifo (C)	16.6	22.1	21.4	22.0

Fig. 6. Benchmarks for **F**lata, **E**ldarica without acceleration, Eldarica with acceleration of loops at the CFG level (**S**tatic) and CEGAAR (**D**ynamic acceleration). The letter after the model name distinguishes **C**orrect from models with a reachable **E**rror state. Items with "-" led to a timeout for the respective approach.

The benchmarks are all in the Numerical Transition Systems format[6] (NTS). We have considered seven sets of examples, extracted automatically from different sources: (a) C programs with arrays provided as examples of divergence in predicate abstraction [21], (b) verification conditions for programs with arrays, expressed in the SIL logic of [9] and translated to NTS, (c) small C programs with challenging loops, (d) NTS extracted from programs with singly-linked lists by the L2CA tool [8], (e) C programs provided as benchmarks in the NECLA static analysis suite, (f) C programs with asynchronous procedure calls translated into NTS using the approach of [15] (the examples with extension .optim are obtained via an optimized translation method [Pierre Ganty, personal communication], and (g) models extracted from VHDL models of circuits following the method of [27]. The benchmarks are available from the home page of our tool. The results on this benchmark set suggest that we have arrived at a fully automated verifier that is robust in verifying automatically generated integer programs

[4] http://lara.epfl.ch/w/eldarica

[5] http://www-verimag.imag.fr/FLATA.html

[6] http://richmodels.epfl.ch/ntscomp_ntslib

with a variety of looping control structure patterns. An important question we explored is the importance of dynamic application of acceleration, as well as of overapproximation and underapproximation. We therefore also implemented static acceleration [12], a lightweight acceleration technique generalizing large block encoding (LBE) [4] with transitive closures. It simplifies the control flow graph prior to predicate abstraction. In some cases, such as mergesort from the (d) benchmarks and split_ vc.1 from (b) benchmarks, the acceleration overhead does not pay off. The problem is that static acceleration tries to accelerate every loop in the CFG rather than accelerating the loops occurring on spurious paths leading to error. Acceleration of inessential loops generates large formulas as the result of combining loops and composition of paths during large block encoding. The CEGAAR algorithm is the only approach that could handle all of our benchmarks. There are cases in which the Flata tool outperforms CEGAAR such as test2.optim from (f) benchmarks. We attribute this deficiency to the nature of predicate abstraction, which tries to discover the required predicates by several steps of refinement. In the verification of benchmarks, acceleration was exact 11 times in total. In 30 case the over-approximation of the loops was successful, and in 15 cases over-approximation failed, so the tool resorted to under-approximation. This suggests that all techniques that we presented are essential to obtain an effective verifier.

7 Conclusions

We have presented CEGAAR, a new automated verification algorithm for integer programs. The algorithm combines two cutting-edge analysis techniques: interpolation-based abstraction refinement and acceleration of loops. We have implemented CEGAAR and presented experimental results, showing that CEGAAR handles robustly a number of examples that cannot be handled by predicate abstraction or acceleration alone. Because many classes of systems translate into integer programs, our advance contributes to automated verification of infinite-state systems in general.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2) (February 2011)
2. 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, pp. 158–172. Springer, Heidelberg (2002)
3. Bensalem, S., Lakhnech, Y.: Automatic generation of invariants. Form. Methods Syst. Des. 15(1), 75–92 (1999)
4. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32 (2009)
5. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309 (2007)
6. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: Algebraic Bound Computation for Loops. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 103–118. Springer, Heidelberg (2010)

7. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces, PhD Thesis, vol. 189. Collection des Publications de l'Université de Liège (1999)
8. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
9. Bozga, M., Habermehl, P., Iosif, R., Konečný, 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)
10. Bozga, M., Iosif, R., Konečný, F.: Fast Acceleration of Ultimately Periodic Relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
11. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
12. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating Interpolation-Based Model-Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 428–442. Springer, Heidelberg (2008)
13. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic 22(3), 250–268 (1957)
14. Finkel, A., Leroux, J.: How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
15. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. CoRR, abs/1011.0551 (2010)
16. Ginsburg, S., Spanier, E.: Bounded algol-like languages. Trans. of the AMS 113(2), 333–368 (1964)
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. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of Trace Abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
19. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: 31st POPL (2004)
20. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems (tool paper). In: FM (2012)
21. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
22. Kroening, D., Leroux, J., Rümmer, P.: Interpolating Quantifier-Free Presburger Arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 489–503. Springer, Heidelberg (2010)
23. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1) (2005)
24. Monniaux, D.: Personal Communication
25. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
26. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
27. Smrčka, A., Vojnar, T.: Verifying Parametrised Hardware Designs Via Counter Automata. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 51–68. Springer, Heidelberg (2008)

FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization*

Ondrej Sery[1,2], Grigory Fedyukovich[1], and Natasha Sharygina[1]

[1] University of Lugano, Switzerland
name.surname@usi.ch
[2] D3S, Faculty of Mathematics and Physics, Charles University, Czech Rep.

Abstract. This paper presents FunFrog, a tool that implements a function summarization approach for software bounded model checking. It uses interpolation-based function summaries as over-approximation of function calls. In every successful verification run, FunFrog generates function summaries of the analyzed program functions and reuses them to reduce the complexity of the successive verification. To prevent reporting spurious errors, the tool incorporates a counterexample-guided refinement loop. Experimental evaluation demonstrates competitiveness of FunFrog with respect to state-of-the-art software model checkers.

1 Introduction

Bounded model checkers (BMC) [1] search for errors in a program within the given bound on the maximal number of loop iterations and recursion depth. Typically, the check is repeated for different properties to be verified and thus large amount of the work is repeated. This raises a problem of constructing an incremental model checker. In this paper, we present a tool, FunFrog, that serves this goal. From a successful verification run, FunFrog extracts function summaries using Craig interpolation [3]. The summaries are then used to represent the functions in subsequent verification runs, when the same code is analyzed again (e.g., with respect to different properties). Significant time savings can be achieved by reusing summaries between the verification runs.

To be able to use interpolation for function summarization, FunFrog converts the unwound program into a partitioned bounded model checking (PBMC) formula. For each function to be summarized, this formula is partitioned into two parts. The first part symbolically encodes the function itself and all its callee functions. The second part encodes the remaining functions, i.e., the calling context of the function. Given the two parts, a Craig interpolant that constitutes the function summary is then computed. Our function summaries are over-approximations of the actual behavior of the functions. As a result, spurious errors may occur due to a too coarse over-approximation. To discard spurious errors, FunFrog implements a counterexample-guided refinement loop.

The paper provides an architectural description of the tool implementing the function summarization approach to bounded model checking and discusses the tool usage and experimentation on various benchmarks[1].

* This work is partially supported by the European Community under the call FP7-ICT-2009-5 — project PINCETTE 257647.

[1] Further details on interpolation-based function summarization can be found in [4].

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 203–207, 2012.

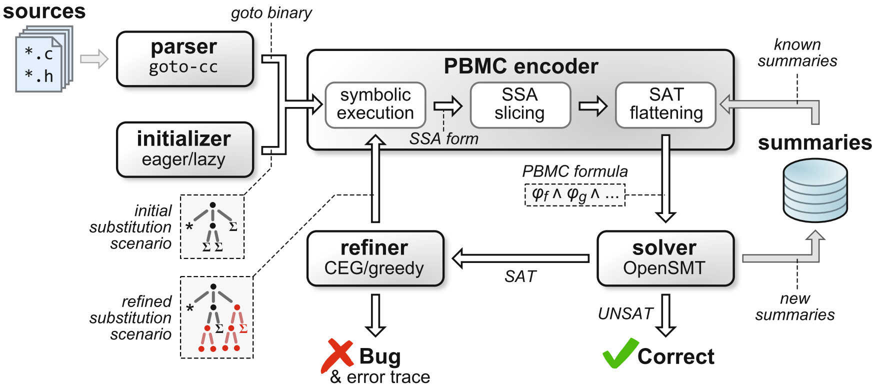


Fig. 1. FunFrog architecture overview

2 Tool Architecture

The architecture of FunFrog is depicted in Fig. 1. The tool takes a C program and uses the *parser* for pre-processing. The parser produces an intermediate code representation, which is then used for encoding into a PBMC formula by *PBMC encoder*. Encoding is achieved using *symbolic execution*, which unwinds the program and prepares its static single assignment (SSA) form, *SSA slicing* that removes the SSA steps irrelevant to the property, and *SAT flattening* that produces the final formula by encoding it into propositional logic. FunFrog loads function summaries from a persistent storage and attempts to use them during encoding as over-approximations of the corresponding program functions. The tool passes the resulting formula to a *solver*. If the formula is unsatisfiable, the program is safe and FunFrog uses interpolation to generate new function summaries and stores them for use in later runs. In case of a satisfiable formula, FunFrog asks *refiner* whether a refinement is necessary. If so, FunFrog continues by precisely encoding the functions identified by the refiner. If a refinement is not necessary (i.e., no summarized function call influences the property along the counterexample), the counterexample is real, and the program is proven unsafe. In the following, we describe each step of FunFrog in more detail.

Parsing. As the first step, the source codes are parsed and transformed into a *goto-program*, where the complicated conditional statements and loops are simplified using only guards and goto statements. For this purpose, FunFrog uses `goto-cc`[2] , i.e., a parser specifically designed to produce intermediate representation suitable for formal verification. Other tools from the CProver[2] framework can be used to alter this representation. For example, `goto-instrument` injects additional assertions (e.g., array bound tests) to be checked during analysis.

Symbolic execution. In order to unwind the program, the intermediate representation is symbolically executed tracking the number of iterations of loops. The result of this step is the SSA form of the unwound program, i.e., a form where every variable is assigned at most once. This is achieved by adding version numbers to the variables. In FunFrog,

[2] `http://www.cprover.org/`

this step is also influenced by the choice of an *initial substitution scenario*. Intuitively, it defines how different functions should be encoded (e.g., using precise encoding or using a summary).

Slicing. After the symbolic execution step, slicing is performed on the resulting SSA form. It uses dependency analysis in order to figure out which variables and instructions are relevant for the property being analyzed. The dependency analysis also takes summaries into account. Whenever an output variable of a function is not constrained by a function summary, its dependencies need not be propagated and a more aggressive slicing is achieved.

SAT flattening. When the SSA form is pruned, the PBMC formula is created by flattening into propositional logic. The choice of using SAT allows for bit-precise reasoning. However, in principle, the SAT flattening step could be substituted by encoding into a suitable SMT theory that supports interpolation.

Solving. The PBMC formula is passed to a SAT solver to decide its satisfiability. FunFrog uses OpenSMT [2] in the SAT solver mode for both satisfiability checks and as an interpolating engine. Certain performance penalties follow from the additional bookkeeping in order to produce a proof of unsatisfiability used for interpolation.

Summaries extraction. When the PBMC formula is unsatisfiable, FunFrog extracts function summaries using interpolation using the proof of unsatisfiability. The extracted summaries are serialized in a persistent storage so that they are available for other FunFrog runs. In this step, FunFrog also compares the new summaries with any existing summaries for the same function and the same bound, and keeps the more precise (tighter over-approximation) one.

Refiner. The refiner is used to identify and to mark summaries directly involved in the error trace. We call this strategy CEG (counterexample-guided). Alternatively, the refiner can avoid identification of summaries in the error trace and can mark all summaries for refinement (greedy strategy). In other words, greedy strategy falls back to the standard BMC, when the summaries are not strong enough to prove the property.

3 Tool Usage

When running FunFrog, the user can choose the preferred initial substitution scenario, a refinement strategy and whether summaries optimization and slicing should be performed. The user can also specify the unwinding bound; the overall bound as well as bounds for particular loops. The input code is expected to contain user provided assertions to be checked for violations. The user can choose which assertion(s) should be checked by FunFrog. Linux binaries of FunFrog as well as the benchmarks used for evaluation are available online for other researchers[3]. The webpage also contains a tutorial explaining how to use FunFrog and explanation of the most important parameters.

Experiments. In order to evaluate FunFrog, we compared it with other state-of-the-art C-model checkers CBMC (v4.0), SATABS (v3.0 with Cadence SMV v10-11-02p46),

[3] www.verify.inf.usi.ch/funfrog

Table 1. Verification times [s] of FunFrog, CBMC, SATABS, and CPAchecker, where '∞' is a timeout (1h), '×' - bug in safe code, '†' - other failure (We notified the tool authors about the issues), number of lines of code, preprocessed code instructions in `goto-cc`, function calls, and assertions.

benchmark					FunFrog details						total			
	#LoC	#Instructions	#func. calls	#assertions	#ref. iter.	symb. ex.	slicing	flattening	solving	interpol.	FunFrog	CBMC	SATABS	CPAchecker
`floppy`	10288	2164	227	8	0	4.80	0.07	6.05	2.94	0.57	14.54	19.59	918.25	383.97
`kbfiltr`	12247	1052	64	8	0	1.72	0.01	2.38	0.23	0.15	4.60	5.33	91.37	†
`diskperf`	6324	2037	182	5	0	2.41	0.01	2.31	0.42	0.36	5.60	21.42	146.82	259.26
`no_sprintf`	178	68	6	2	0	0.01	0.00	0.03	0.03	0.01	0.08	0.01	125.69	2.96
`gd_simp`	207	82	4	5	0	0.03	0.00	0.07	0.05	0.01	0.17	0.03	∞	×
`do_loop`	126	176	12	7	3	7.74	2.66	2.58	2.29	0.11	15.78	19.52	∞	×
`goldbach`	268	344	22	6	0	0.41	0.00	1.53	2.03	0.78	5.78	15.44	∞	†

and CPAchecker (v1.1). CBMC and FunFrog are BMC tools, provided with the same bound. We evaluated all tools (with default options) on both real-life industrial benchmarks (including Windows device drivers) and on smaller crafted examples designed to stress-test the implementation of our tool and verified them for user defined assertions (separate run for each assertion). The assertions held, so FunFrog had the opportunity to extract and reuse function summaries.

Table 1 reports the running times of all the tools[4]. In case of FunFrog, the summaries were generated after the first run (for the first assertion in each group) and reused in the consecutive runs (for the rest of (#asserts - 1) assertions). To demonstrate the performance of FunFrog, the running times of different phases of its algorithm were summed across all runs for the same benchmark. Note that the time spent in counterexample analysis (i.e., the only computation, needed for refinement) is negligible, and thus not reported in a separate column, but still included to the total.

As expected, FunFrog was outperformed by CBMC on the smaller examples without many function calls, but FunFrog's running times were still very competitive. On majority of the larger benchmarks, FunFrog outperformed all the other tools. These benchmarks feature large number of function calls so FunFrog benefited from function summarization.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[4] The complete set of experiments can be found at `www.verify.inf.usi.ch/funfrog`.

2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
3. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic, 269–285 (1957)
4. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC 2011. LNCS (2011) (to appear)

Synthesis of Succinct Systems*

John Fearnley[1], Doron Peled[2], and Sven Schewe[1]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Synthesis of correct by design systems from specification has recently attracted much attention. The theoretical results imply that this problem is highly intractable, e.g., synthesizing a system is 2EXPTIME-complete for an LTL specification and EXPTIME-complete for CTL. An argument in favor of synthesis is that the temporal specification is highly compact, and the complexity reflects the large size of the system constructed. A careful observation reveals that the size of the system is presented in such arguments as the size of its state space. This view is a bit biased, in the sense that the state space can be exponentially larger than the size of a reasonable implementation such as a circuit or a program. Although this alternative measure of the size of the synthesized system is more intuitive (e.g., this is the standard way model checking problems are measured), research on synthesis has so far stayed with measuring the system in terms of the explicit state space. This raises the question of whether or not there exists a small bound on the circuits or programs. In this paper, we show that this is the case if, and only if, PSPACE = EXPTIME.

1 Introduction

Synthesis of reactive systems is a research direction inspired by Church's problem [1]. It focuses on systems that receive a constant stream of inputs from an environment, and must, for each input, produce some output. Specifically, we are given a logical specification that dictates how the system must react to the inputs, and we must construct a system that satisfies the specification for all possible inputs that the environment could provide.

While the verification [2,6] (the validation or refutation of the correctness of such a system) has gained many algorithmic solutions and various successful tools, the synthesis problem [3,4,5,13,17] has had fewer results. One of the main problems is the complexity of the synthesis problem. A classical result by Pnueli and Rosner [14] shows that synthesis of a system from an LTL specification is 2EXPTIME-complete. It was later shown by Kupferman and Vardi that synthesis for CTL specifications is EXPTIME-complete [9]. A counter argument

* This work was supported by the Engineering and Physical Science Research Council grant EP/H046623/1 'Synthesis and Verification in Markov Game Structures' and a short visit grant 'Circuit Complexity for Synthesis' within the framework of the ESF activity 'Games for Design and Verification'. A full version of this paper is available at `http://arxiv.org/abs/1202.5449`

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 208–222, 2012.

against the claim that synthesis has prohibitive high complexity is that the size of the system produced by the synthesis procedure is typically large. Some concrete examples [8] show that the size of the system synthesized may need to be doubly exponentially larger than the LTL specification. This, in fact, shows that LTL specifications are a very compact representation of a system, rather than simply a formalism that is intrinsically hard for synthesis.

As we are interested in the relationship between the specification and the synthesized system, a question arises with respect to the nature of the system representation. The classical synthesis problem regards the system as a *transition system* with an *explicit* state space, and the size of this system is the number of states and transitions. This is, to some extent, a biased measure, as other system representations, such as programs or circuits with memory, often have a much more concise representation: it is often possible to produce a circuit or program that is exponentially smaller than the corresponding transition system. For example, it is easy to produce a small program that implements an n bit binary counter, but a corresponding transition system requires 2^n distinct states to implement the same counter. We ask the question of *what is the size of the minimal system representation in terms of the specification?*

We look at specifications given in CTL, LTL, or as an automaton, and study the relative synthesized system complexity. We choose to represent our systems as online Turing machines with a bounded storage tape. This is because there exist straightforward translations between online Turing machines and the natural representations of a system, such as programs and circuits, with comparable representation size. The online Turing machine uses a read-only input tape to read the next input, a write-only output tape to write the corresponding output, and its storage tape to serve as the memory required to compute the corresponding output for the current input.

The binary-counter example mentioned above showed that there are instances in which an online Turing machine model of the specification is exponentially smaller than a transition system model of that formula. In this paper we ask: is this always the case? More precisely, for every CTL formula ϕ, does there always exist an online Turing machine $\mathcal{M}$ that models ϕ, where the amount of space required to describe $\mathcal{M}$ is polynomial in ϕ? We call machines with this property *small*. Our answer to this problem is the following:

> Every CTL formula has a small online Turing machine model (or no model at all) if, and only if, PSPACE = EXPTIME.

This result can be read in two ways. One point of view is that, since PSPACE is widely believed to be a proper subset of EXPTIME, the "if" direction of our result implies that it is unlikely that every CTL formula has a small online Turing machine model. However, there is an opposing point of view. It is widely believed that finding a proof that PSPACE $\neq$ EXPTIME is an extremely difficult problem. The "only if" direction of our result implies that, if we can find a family of CTL formulas that provably requires super-polynomial sized online Turing machine models, then we have provided a proof that PSPACE $\neq$ EXPTIME. If it is difficult to find a proof that PSPACE $\neq$ EXPTIME, then it must also be difficult to find such CTL formulas. This indicates that most CTL formulas, particularly those that are likely to arise in practice, might have small online Turing machine models.

Using an online Turing machine raises the issue of the time needed to respond to an input. In principle, a polynomially-sized online Turing machine can take exponential time to respond to each input. A small model may, therefore, take exponential time in the size of the CTL formula to produce each output. This leads to the second question that we address in this paper: for CTL, does there always exist a small online Turing machine model that is *fast*? The model is fast if it always responds to each input in polynomial time. Again, our result is to link this question to an open problem in complexity theory:

> Every CTL formula has a small and fast (or no) online Turing machine model if, and only if, EXPTIME $\subseteq$ P/poly.

P/poly is the class of problems solvable by a polynomial-time Turing machine with an advice function that provides advice strings of polynomial size. It has been shown that if EXPTIME $\subseteq$ P/poly, then the polynomial time hierarchy collapses to the second level, and that EXPTIME itself would be contained in the polynomial time hierarchy [7].

Once again, this result can be read in two ways. Since many people believe that the polynomial hierarchy is strict, the “if” direction of our result implies that it is unlikely that all CTL formulas have small and fast models. On the other hand, the “only if” direction of the proof implies that finding a family of CTL formulas that do not have small and fast online Turing machine models is as hard as proving that EXPTIME is not contained in P/poly. As before, if finding a proof that EXPTIME $\nsubseteq$ P/poly is a difficult problem, then finding CTL formulas that do not have small and fast models must also be a difficult problem. This indicates that the CTL formulas that arise in practice might have small and fast models.

We also replicate these results for specifications given by Co-Büchi automata, which then allows us to give results for LTL specifications.

2 Preliminaries

2.1 CTL Formulas

Given a finite set Π of atomic propositions, the syntax of a CTL formula is defined as follows:

$$\begin{aligned}\phi &::= p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi,\\ \psi &::= \mathcal{X}\phi \mid \phi\, \mathcal{U}\, \phi,\end{aligned}$$

where $p \in \Pi$. For each CTL formula ϕ we define $|\phi|$ to give the length of ϕ.

Let $T = (V, E)$ be an infinite directed tree, with all edges pointing away from the root. Let $l : V \to 2^{\Pi}$ be a labelling function. The semantics of CTL are defined as follows. For each $v \in V$ we have:

- $v \models p$ if and only if $p \in l(v)$.
- $v \models \neg\phi$ if and only if $v \not\models \phi$.
- $v \models \phi \vee \psi$ if and only if either $v \models \phi$ or $v \models \psi$.
- $v \models A\psi$ if and only if for all paths π starting at v we have $\pi \models \psi$.
- $v \models E\psi$ if and only if there exists a path π starting at v with $\pi \models \psi$.

Let $\pi = v_1, v_2, \ldots$ be an infinite path in T. We have:

- $\pi \models \mathcal{X}\psi$ if and only if $v_2 \models \psi$.
- $\pi \models \phi \, \mathcal{U} \, \psi$ if and only if there exists $i \in \mathbb{N}$ such that $v_i \models \psi$ and for all j in the range $1 \leq j < i$ we have $v_j \models \phi$.

The pair (T, l), where T is a tree and l is a labelling function, is a model of ϕ if and only if $r \models \phi$, where $r \in V$ is the root of the tree. If (T, l) is a model of ϕ, then we write $T, l \models \phi$.

2.2 Mealy Machines

The synthesis problem is to construct a model that satisfies the given specification. We will give two possible formulations for a model. Traditionally, the synthesis problem asks us to construct a model in the form of a transition system. We will represent these transition systems as *Mealy machines*, which we will define in this section. In a later section we will give online Turing machines as an alternative, and potentially more succinct, model of a specification.

A Mealy machine is a tuple $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l, \mathsf{start}, \mathsf{input})$. The set S is a finite set of states, and the state $\mathsf{start} \in S$ is the starting state. The set Σ_I gives an input alphabet, and the set Σ_O gives an output alphabet. The transition function $\tau : S \times \Sigma_I \to S$ gives, for each state and input letter, an outgoing transition. The function $l : S \times \Sigma_I \to \Sigma_O$ is a labelling function, which assigns an output letter for each transition. The letter $\mathsf{input} \in \Sigma_I$ gives an initial input letter for the machine.

Suppose that ϕ is a CTL formula that uses Π_I as a set *input* propositions, and Π_O as a set of *output* propositions. Let $\mathcal{T} = (S, 2^{\Pi_I}, 2^{\Pi_O}, \tau, l, \mathsf{start}, \mathsf{input})$ be a Mealy machine that uses sets of these propositions as input and output alphabets. A sequence of states $\pi = s_0, s_1, s_2, \ldots$ is an infinite path in $\mathcal{T}$ if $s_0 = \mathsf{start}$, and if, for each i, there is a letter $\sigma_i \in \Sigma_I$ such that $\tau(s_i, \sigma_i) = s_{i+1}$. We define ω_i to be the set of input and output propositions at position i in the path, that is we define $\omega_i = \sigma_i \cup l(s_i, \sigma_i)$, where we take $\sigma_0 = \mathsf{input}$. Then, for each infinite path π, we define the word $\sigma(\pi) = \omega_0, \omega_1, \omega_2 \ldots$ to give the sequence of joint inputs and outputs along the path π. Furthermore, let (T, l) be the infinite tree corresponding to the set of words $\sigma(\pi)$, over all possible infinite paths π. We say that $\mathcal{T}$ is a model of ϕ if $T, l \models \phi$. Given a CTL formula ϕ and a Mealy machine $\mathcal{T}$, the CTL model checking problem is to decide whether $\mathcal{T}$ is a model of ϕ.

Theorem 1 ([11]). *Given a Mealy machine $\mathcal{T}$ and a CTL-formula ϕ, the CTL model checking problem can be solved in space polynomial in $|\phi| \cdot \log |\mathcal{T}|$.*

Given a CTL formula ϕ, the *CTL synthesis problem* is to decide whether there exists a Mealy machine that is a model of ϕ. This problem is known to be EXPTIME-complete.

Theorem 2 ([9]). *The CTL synthesis problem is EXPTIME-complete.*

2.3 Tree Automata

Universal Co-Büchi tree automata will play a fundamental role in the proofs given in subsequent sections, because we will translate each CTL formula ϕ into a universal Co-Büchi tree automaton $\mathcal{U}(\phi)$. The automaton will accept Mealy machines, and the language of the tree automaton will be exactly the set of models accepted by ϕ. We will then use these automata to obtain our main results.

A *universal Co-Büchi tree automaton* is $\mathcal{A} = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F)$, where S denotes a finite set of states, Σ_I is a finite input alphabet, Σ_O is a finite output alphabet, $\mathsf{start} \in S$ is an initial state, δ is a transition function, $F \subseteq S$ is a set of final states. The transition function $\delta : S \times \Sigma_O \to 2^{S \times \Sigma_I}$ maps a state and an output letter to a set of pairs, where each pair consists of a successor state and an input letter.

The automaton accepts Mealy machines that use Σ_I and Σ_O as their input and output alphabets, and the acceptance mechanism is defined in terms of run graphs. We define a *run graph* of a universal Co-Büchi tree automaton $\mathcal{A} = (S_\mathcal{A}, \Sigma_I, \Sigma_O, \mathsf{start}_\mathcal{A}, \delta, F_\mathcal{A})$ on a Mealy machine $\mathcal{T} = (S_\mathcal{T}, \Sigma_I, \Sigma_O, \tau, l_\mathcal{T}, \mathsf{start}_\mathcal{T})$ to be a minimal directed graph $G = (V, E)$ that satisfies the following constraints:

- The vertices of G satisfy $V \subseteq S_\mathcal{A} \times S_\mathcal{T}$.
- The pair of initial states $(\mathsf{start}_\mathcal{A}, \mathsf{start}_\mathcal{T})$ is contained in V.
- Suppose that for a vertex $(q, t) \in V$, we have that $(q', \sigma_I) \in \delta(q, l_\mathcal{T}(\sigma_I, t))$ for some input letter σ_I. An edge from (q, t) to $(q', \tau(t, \sigma_I))$ must be contained in E.

A run graph is *accepting* if every infinite path $v_1, v_2, v_3, \dots \in V^\omega$ contains only finitely many states in $F_\mathcal{A}$. A Mealy machine $\mathcal{T}$ is accepted by $\mathcal{A}$ if it has an accepting run graph. The set of Mealy machines accepted by $\mathcal{A}$ is called its *language*, and is denoted by $\mathcal{L}(\mathcal{A})$. The automaton is empty if, and only if, its language is empty.

A universal Co-Büchi tree automaton is called a *safety* tree automaton if $F = \emptyset$. Therefore, for safety automata, we have that every run graph is accepting, and we drop the $F = \emptyset$ from the tuple defining the automaton. A universal Co-Büchi tree automaton is *deterministic* if $|\delta(s, \sigma_O))| = 1$, for all states s, and output letters σ_O.

2.4 Online Turing Machines

We use *online* Turing machines as a formalisation of a concise model. An online Turing machine has three tapes: an infinite input tape, an infinite output tape, and a storage tape of bounded size. The input tape is read only, and the output tape is write only. Each time that a symbol is read from the input tape, the machine may spend time performing computation on the storage tape, before eventually writing a symbol to the output tape.

We can now define the synthesis problem for online Turing machines. Let ϕ be a CTL formula defined using Π_I and Π_O, as the sets of input, and output, propositions, respectively. We consider online Turing machines that use 2^{Π_I} as the input tape alphabet, and 2^{Π_O} as the output alphabet. Online Turing machines are required, after receiving an input symbol, to produce an output before the next input symbol can be read. Therefore, if we consider the set of all possible input words that could be placed on the input tape, then the set of possible outputs made by the online Turing machine forms a tree. If this tree is a model of ϕ, then we say that the online Turing machine is a model of ϕ.

Given a CTL formula ϕ, we say that an online Turing machine $\mathcal{M}$ is a *small* model of ϕ if:

- $\mathcal{M}$ is a model of ϕ,
- the storage tape of $\mathcal{M}$ has length polynomial in $|\phi|$, and
- the discrete control (i.e. the action table) of $\mathcal{M}$ has size polynomial in $|\phi|$.

For this reason, we define $|\mathcal{M}|$ to be size of the discrete control of $\mathcal{M}$ plus the length of the storage tape of $\mathcal{M}$. Note that a small online Turing machine may take an exponential number of steps to produce an output for a given input. We say that an online Turing machine is a *fast* model of ϕ if, for all inputs, it always responds to each input in time polynomial in $|\phi|$.

3 Small Models Imply PSPACE = EXPTIME

In this section we show that, if every satisfiable CTL formula ϕ has a small online Turing machine model, then PSPACE = EXPTIME. Our approach is to guess a polynomially sized online Turing machine $\mathcal{M}$, and then to use model checking to verify whether $\mathcal{M}$ is a model of ϕ. Since our assumption guarantees that we only need to guess polynomially sized online Turing machines, this gives a NPSPACE = PSPACE algorithm for solving the CTL synthesis problem. Our proof then follows from the fact that CTL synthesis is EXPTIME-complete.

To begin, we show how model checking can be applied to an online Turing machine. To do this, we first unravel the online Turing machine to a Mealy machine.

Lemma 3. *For each CTL formula ϕ, and each online Turing machine $\mathcal{M}$ that is a model of ϕ, there exists a Mealy machine $\mathcal{T}(\mathcal{M})$, which uses the same input and output alphabet, such that $\mathcal{T}(\mathcal{M})$ is a model of ϕ.*

The size of $\mathcal{T}(\mathcal{M})$ is exponential in the size of $\mathcal{M}$, because the number of storage tape configurations of $\mathcal{M}$ grows exponentially with the length of the tape. However, this is not a problem because there exists a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using only $O(|\mathcal{M}|)$ space.

Lemma 4. *There is a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using $O(|\mathcal{M}|)$ space.*

Since the model checking procedure given in Theorem 1 uses polylogarithmic space, when it is applied to $\mathcal{T}(\mathcal{M})$, it will use space polynomial in $|\mathcal{M}|$. Now, using standard techniques to compose space bounded Turing machines (see [12, Proposition 8.2], for example), we can compose the deterministic Turing machine given by Lemma 4 with the model checking procedure given in Theorem 1 to produce a deterministic Turing machine that uses polynomial space in $|\mathcal{M}|$. Hence, we have shown that each online Turing machine $\mathcal{M}$ can be model checked against ϕ in space polynomial in $|\mathcal{M}|$. Since Theorem 2 implies that CTL synthesis is EXPTIME-complete, we have the following theorem.

Theorem 5. *If every satisfiable CTL formula ϕ has an online Turing machine $\mathcal{M}$ model, where $|\mathcal{M}|$ is polynomial in ϕ, then PSPACE = EXPTIME.*

4 PSPACE = EXPTIME Implies Small Models

In this section we show the opposite direction of the result given in Section 3. We show that if PSPACE = EXPTIME, then, for every CTL formula ϕ that has a model, there exists a polynomially sized online Turing machine that is a model of ϕ. We start our proof of this result with a translation from CTL to universal Co-Büchi tree automata. In [9] it was shown that every CTL formula ϕ can be translated to an *alternating* Co-Büchi tree automaton $\mathcal{A}(\phi)$, whose language is the models of ϕ. Using standard techniques [10,15], we can translate this alternating tree automaton $\mathcal{A}(\phi)$ through a universal tree automaton $\mathcal{U}(\phi)$ with the same states to a safety automaton $\mathcal{F}(\phi)$. Both transformations preserve emptiness, and it is simple to obtain a model for $\mathcal{A}(\phi)$ from a model of $\mathcal{F}(\phi)$.

One complication of the first transformation is that the output alphabet of $\mathcal{U}(\phi)$ is not 2^{Π_O}. This is because the reduction to universal tree automata augments each output letter with additional information, which is used by $\mathcal{U}(\phi)$ to resolve the nondeterministic choices made by $\mathcal{A}(\phi)$. Hence, each output letter of $\mathcal{U}(\phi)$ contains an actual output $\sigma_O \in 2^{\Pi_O}$, along with some extra information, which can be encoded in polynomially many bits in the length of ϕ.

Let $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l_\mathcal{T}, \mathsf{start}, \mathsf{input})$ be a Mealy machine, where each output $\sigma_O \in \Sigma_O$ contains some element $a \in 2^{\Pi_O}$ with $a \subseteq \sigma_O$. We define $\mathcal{T} \restriction \Pi_O$ to be a modified version of $\mathcal{T}$ that only produces outputs from the set 2^{Π_O}. Formally, we define $\mathcal{T} \restriction \Pi_O$ to be the Mealy machine $\mathcal{T}' = (S, \Sigma_I, 2^{\Pi_O}, \tau, l_{\mathcal{T}'}, \mathsf{start}, \mathsf{input})$ where, if $l_\mathcal{T}(s, \sigma_I) = \sigma_O$, then we define $l_{\mathcal{T}'}(s, \sigma_I) = \sigma_O \cap 2^{\Pi_O}$ for all $s \in S$.

Lemma 6. *Let ϕ be a CTL formula, which is defined over the set Π_I of input propositions, and the set Π_O of output propositions. We can construct a universal Co-Büchi tree automaton $\mathcal{U}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$ such that:*

- *For every model $\mathcal{T} \in \mathcal{L}(\mathcal{U}(\phi))$, we have that $\mathcal{T} \restriction 2^{\Pi_O}$ is a model of ϕ.*
- *For every model $\mathcal{T}'$ of ϕ there is a model $\mathcal{T} \in \mathcal{L}(\mathcal{U}(\phi))$ with $\mathcal{T} \restriction 2^{\Pi_O} = \mathcal{T}'$.*

- *The size of the set S is polynomial in $|\phi|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|\phi|$.*
- *The transition function δ can be computed in time polynomial in $|\phi|$.*
- *The state* start *can be computed in polynomial time.*

The techniques used in [16] show how the automaton given by Lemma 6 can be translated into a safety tree automaton $\mathcal{F}(\phi)$ such that the two automata are emptiness equivalent.

Lemma 7 ([16]). *Given the universal Co-Büchi tree automaton $\mathcal{U}(\phi)$, whose state space is $S_{\mathcal{U}}$, we can construct a deterministic safety tree automaton $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, \mathsf{input})$ such that:*

- *If $\mathcal{L}(\mathcal{U}(\phi))$ is not empty, then $\mathcal{L}(\mathcal{F}(\phi))$ is not empty. Moreover, if $\mathcal{T}$ is in $\mathcal{L}(\mathcal{F}(\phi))$, then $\mathcal{T} \restriction 2^{\Pi_O}$ is a model of ϕ.*
- *Each state in S can be stored in space polynomial in $|S_{\mathcal{U}}|$.*
- *The transition function δ can be computed in time polynomial in $|S_{\mathcal{U}}|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|S_{\mathcal{U}}|$.*
- *The state* start *can be computed in time polynomial in $|S_{\mathcal{U}}|$.*

We will use the safety automaton $\mathcal{F}(\phi)$ given by Lemma 7 to construct a polynomially sized model of ϕ. This may seem counter intuitive, because the number of states in $\mathcal{F}(\phi)$ may be exponential in ϕ. However, we do not need to build $\mathcal{F}(\phi)$. Instead our model will solve language emptiness queries for $\mathcal{F}(\phi)$.

For each state $s \in S$ in $\mathcal{F}(\phi)$, we define $\mathcal{F}^s(\phi)$ to be the automaton $\mathcal{F}(\phi)$ with starting state s. The *emptiness problem* takes a CTL formula ϕ and a state of $\mathcal{F}(\phi)$, and requires us to decide whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$. Note that the input has polynomial size in $|\phi|$. We first argue that this problem can be solved in exponential time. To do this, we just construct $\mathcal{F}^s(\phi)$. Since $\mathcal{F}^s(\phi)$ can have at most exponentially many states in $|\phi|$, and the language emptiness problem for safety automata can be solved in polynomial time, we have that our emptiness problem lies in EXPTIME.

Lemma 8. *For every CTL formula ϕ, and every state s in $\mathcal{F}(\phi)$ we can decide whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$ in exponential time.*

The algorithm that we construct for Lemma 8 uses exponential time and exponential space. However, our key observation is that, under the assumption that PSPACE = EXPTIME, Lemma 8 implies that there must exist an algorithm for the emptiness problem that uses polynomial space. We will use this fact to construct $\mathcal{M}(\phi)$, which is a polynomially sized online Turing machine that models ϕ.

Let ϕ be a CTL formula that over the set $\Pi_I \cup \Pi_O$ of propositions. Suppose that ϕ has a model, and that $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$. The machine $\mathcal{M}(\phi)$ always maintains a current state $s \in S$, which is initially set so that $s = \mathsf{start}$. Lemma 7 implies that s can be stored in polynomial space, and that setting $s = \mathsf{start}$ can be done in polynomial time, and hence polynomial space.

Every time that $\mathcal{M}(\phi)$ reads a new input letter $\sigma_I \in 2^{\Pi_I}$ from the input tape, the following procedure is executed. The machine loops through each possible output letter $\sigma_O \in \Sigma_O$ and checks whether there is a pair $(s', \sigma'_I) \in \delta(s, \sigma_O)$ such that $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$. When an output symbol σ_O and state s' with this property are found, then the machine outputs $\sigma_O \cap 2^{\Pi_O}$, moves to the state s', and reads the next input letter.

The fact that a suitable pair σ_O and s' always exists can be proved by a simple inductive argument, which starts with the fact that $\mathcal{L}(\mathcal{F}^{\mathsf{start}}(\phi)) \neq \emptyset$, and uses the fact that we always pick a successor that satisfies the non-emptiness check. Moreover, it can be seen that $\mathcal{M}(\phi)$ is in fact simulating some Mealy machine $\mathcal{T}$ that is contained in $\mathcal{L}(\mathcal{F}(\phi))$. Therefore, by Lemma 6, we have that $\mathcal{M}(\phi)$ is a model of ϕ.

The important part of our proof is that, if PSPACE = EXPTIME, then this procedure can be performed in polynomial space. Since each letter in Σ_O can be stored in polynomial space, we can iterate through all letters in Σ_O while using only polynomial space. By Lemma 8, the check $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$ can be performed in exponential time, and hence, using our assumption that PSPACE = EXPTIME, there must exist a polynomial space algorithm that performs this check. Therefore, we have constructed an online Turing machine that uses polynomial space and models ϕ. Thus, we have shown the following theorem.

Theorem 9. *Let ϕ be a CTL formula that has a model. If PSPACE = EXPTIME then there is an online Turing machine $\mathcal{M}$ that models ϕ, where $|\mathcal{M}|$ is polynomial in ϕ.*

Theorem 9 is not constructive. However, if a polynomially sized online Turing machine that models ϕ exists, then we can always find it in PSPACE by guessing the machine, and then model checking it.

5 Small and Fast Models Imply EXPTIME $\subseteq$ P/poly

In this section we show that, if all satisfiable CTL formulas have a polynomially sized model that responds to all inputs within polynomial time, then EXPTIME $\subseteq$ P/poly, where P/poly is the is the class of problems solvable by a polynomial-time Turing machine with an advice function that, for each input length, provides a polynomially sized advice string.

Let $\mathcal{A}_b$ be an alternating Turing machine with a tape of length b that is written in binary. Since the machine is alternating, its state space Q must be split into $Q_\forall$, which is the set of universal states, and $Q_\exists$, which is the set of existential states. The first step of our proof is to construct a CTL formula ϕ_b, such that all models of ϕ_b are forced to simulate $\mathcal{A}_b$. The formula will use a set of input propositions Π_I such that $|\Pi_I| = b$. This therefore gives us enough input propositions to encode a tape configuration of $\mathcal{A}_b$. The set of output propositions will allow us to encode a configuration of $\mathcal{A}_b$. More precisely, the output propositions use:

- b propositions to encode the current contents of the tape,
- $\log_2(b)$ propositions to encode the current position of the tape head, and
- $\log_2(Q)$ propositions to encode q, which is the current state of the machine.

Our goal is to simulate $\mathcal{A}_b$. The environment, which is responsible for providing the input in each step, will perform the following tasks in our simulation. In the first step, the environment will provide an initial tape configuration for $\mathcal{A}_b$. In each subsequent step, the environment will resolve the nondeterminism, which means that it will choose the specific existential or universal successor for the current configuration. In response to these inputs, our CTL formula will require that the model should faithfully simulate $\mathcal{A}_b$. That is, it should start from the specified initial tape, and then always follow the existential and universal choices made by the environment. It is not difficult to write down a CTL formula that specifies these requirements.

In addition to the above requirements, we also want our model to predict whether $\mathcal{A}_b$ halts. To achieve this we add the following output propositions:

- Let $C = |Q| \cdot 2^b \cdot b$ be the total number of configurations that $\mathcal{A}_b$ can be in. If the machine halts, it halts within C steps (avoiding cycles). We add $\log(C)$ propositions to encode a counter c for the number of simulated steps.
- We add a proposition h, and we will require that h correctly predicts whether $\mathcal{A}_b$ halts from the current configuration.

The counter c can easily be enforced by a CTL formula. To implement h, we will add the following constraints to our CTL formula:

- If q is an accepting state, then h must be true.
- If c has reached its maximum value, then h must be false.
- If q is non-accepting and c has not reached its maximum value then:
  - If q is an existential state, then $h \leftrightarrow E\mathcal{X}h$.
  - If q is a universal state, then $h \leftrightarrow A\mathcal{X}h$.

These conditions ensure that, whenever the machine is in an existential state, there must be at least one successor state from which $\mathcal{A}_b$ halts, and whenever the machine is in a universal state, $\mathcal{A}_b$ must halt from all successors. We will use ϕ_b to denote the CTL formula that we have outlined.

Suppose that there is an online Turing machine $\mathcal{M}$ model of ϕ_b that is both small and fast. We argue that, if this assumption holds, then we can construct a polynomial time Turing machine $\mathcal{T}$ that solves the halting problem for $\mathcal{A}_b$. Suppose that we want to decide whether $\mathcal{A}_b$ halts on the input word I. The machine $\mathcal{T}$ does the following:

- It begins by giving I to $\mathcal{M}$ as the first input letter.
- It then proceeds by simulating $\mathcal{M}$ until the first output letter is produced.
- Finally, it reads the value of h from the output letter, and then outputs it as the answer to the halting problem for $\mathcal{A}_b$.

By construction, we know that h correctly predicts whether $\mathcal{A}_b$ halts on I. Therefore, this algorithm is correct. Since $\mathcal{M}$ is both small and fast, we have that $\mathcal{T}$ is a polynomial time Turing machine. Thus, we have the following lemma.

Lemma 10. *If ϕ_b has an online Turing machine model that is small and fast, then there is a polynomial-size polynomial-time Turing machine that decides the halting problem for $\mathcal{A}_b$.*

We now use Lemma 10 to prove the main result of this section: if every CTL formula has a small and fast model, then EXPTIME $\subseteq$ P/poly. We will do this by showing that there is a P/poly algorithm for solving an EXPTIME-hard problem.

We begin by defining our EXPTIME-hard problem. We define the problem HALT-IN-SPACE as follows. Let $\mathcal{U}$ be a universal[1] alternating Turing machine. We assume that $\mathcal{U}$ uses space polynomial in the amount of space used by the machine that is simulated. The inputs to our problem are:

- An input word I for $\mathcal{U}$.
- A sequence of blank symbols B, where $|B| = \mathit{poly}(|I|)$.

Given these inputs, HALT-IN-SPACE requires us to decide whether $\mathcal{U}$ halts when it is restricted to use a tape of size $|I|+|B|$. Since B can only ever add a polynomial amount of extra space, it is apparent that this problem is APSPACE-hard, and therefore EXPTIME-hard.

Lemma 11. *HALT-IN-SPACE is EXPTIME-hard.*

A P/poly algorithm consists of two parts: a polynomial-time Turing machine $\mathcal{T}$, and a polynomially-sized *advice function* f. The advice function maps the length of the input of $\mathcal{T}$ to a polynomially-sized advice string. At the start of its computation, the polynomial-time Turing machine $\mathcal{T}$ is permitted to read $f(i)$, where i is the length of input, and use the resulting advice string to aid in its computation. A problem lies in P/poly if there exists a machine $\mathcal{T}$ and advice function f that decide that problem.

We now provide a P/poly algorithm for HALT-IN-SPACE. We begin by defining the advice function f. Let $\mathcal{U}_b$ be the machine $\mathcal{U}$, when it is restricted to only use the first b symbols on its tape. By Lemma 10, there exists a polynomial-size polynomial-time deterministic Turing machine $\mathcal{T}_b$ that solves the halting problem for $\mathcal{U}_b$. We define $f(b)$ to give $\mathcal{T}_b$. Since $\mathcal{T}_b$ can be described in polynomial space, the advice function f gives polynomial size advice strings.

The second step is to give a polynomial-time algorithm that uses f to solve HALT-IN-SPACE. The algorithm begins by obtaining $\mathcal{T}_{i+b} = f(|I| + |B|)$ from the advice function. It then simulates $\mathcal{T}_{i+b}$ on the input word I, and outputs the answer computed by $\mathcal{T}_{i+b}$. By construction, this algorithm takes polynomial time, and correctly solves HALT-IN-SPACE. Therefore, we have shown that an EXPTIME-hard problem lies in P/poly, which gives the main theorem of this section.

Theorem 12. *If every satisfiable CTL formula has a polynomially sized online Turing machine model that responds to all inputs in polynomial time, then EXPTIME $\subseteq$ P/poly.*

[1] Here, the word "universal" means an alternating Turing machine that is capable of simulating all alternating Turing machines.

6 EXPTIME $\subseteq$ P/poly Implies Small and Fast Models

Let ϕ be a CTL formula that has a model. In this section we show that if EXPTIME $\subseteq$ P/poly, then there always exists an polynomially sized online Turing machine that is a model of ϕ, that also responds to every input within a polynomial number of steps.

The proof of this result closely follows the proof given in Section 4. In that proof, we looped through every possible output letter and solved an instance of the emptiness problem. This was sufficient, because we can loop through every output letter in polynomial space. However, when we wish to construct a fast model, this approach does not work, because looping through all possible output letters can take exponential time.

For this reason, we introduce a slightly modified version of the emptiness problem, which we call the *partial letter emptiness problem.* The inputs to our problem will be a CTL formula ϕ, a state s, an input letter $\sigma_I \in 2^{\Pi_I}$ of $\mathcal{F}(\phi)$, an integer l, and a bit string w of length l. Given these inputs, the problem is to determine whether there is a letter $\sigma_O \in \Sigma_O$ such that:

- the first l bits of σ_O are w, and
- there exists $(s', \sigma_I) \in \delta(s, \sigma_O)$ such that $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$.

Lemma 7 implies that the input size of this problem is polynomial in $|\phi|$. In fact, if the CTL formula ϕ is fixed, then Lemma 7 implies that all other input parameters have bounded size. For a fixed formula ϕ, let $(\phi, s, \sigma_I, l, w)$ be the input of the partial letter emptiness problem that requires the longest representation. We pad the representation of all other inputs so that they have the same length as $(\phi, s, \sigma_I, l, w)$.

Next, we show how our assumption that EXPTIME $\subseteq$ P/poly allows us to argue that partial letter emptiness problem lies in P/poly. Note that the partial letter emptiness problem can be solved in exponential time, simply by looping through all possible letters in $\sigma_O \in \Sigma_O$, checking whether the first n bits of σ_O are w, and then applying the algorithm of Lemma 8. Also note that our padding of inputs does not affect this complexity. Therefore, the partial letter emptiness problem lies in EXPTIME, and our assumption then implies that it also lies in P/poly.

Let $\mathcal{T}$ and f be the polynomial time Turing machine and advice function that witness the inclusion in P/poly. Our padding ensures that we have, for each CTL formula ϕ, a unique advice string in f that is used by $\mathcal{T}$ to solve all partial letter emptiness problems for ϕ. Hence, if we append this advice string to the storage tape of $\mathcal{T}$, then we can construct a polynomial time Turing machine (with no advice function) that solves all instances of the partial letter emptiness problem that depend on ϕ. Therefore, we have shown the following lemma.

Lemma 13. *If EXPTIME $\subseteq$ P/poly then, for each CTL formula ϕ, there is a polynomial time Turing machine that solves all instances of the partial letter emptiness problem that involve ϕ.*

The construction of an online Turing machine that models ϕ is then the same as the one that was provided in Section 4, except that we use the polynomial time Turing machine from Lemma 13 to solve the partial letter emptiness problem in each step. More precisely, we iteratively solve $|\sigma_O|$ instances of the partial letter emptiness problem to find the appropriate output letter σ_O in each step. Since the size of σ_O is polynomial in $|\phi|$, this can obviously be achieved in polynomial time. Moreover, our online Turing machine still obviously uses only polynomial space. Thus, we have established the main result of this section.

Theorem 14. *Let ϕ be a CTL formula that has a model. If EXPTIME $\subseteq$ P/poly then there is a polynomially sized online Turing machine $\mathcal{M}$ that models ϕ that responds to every input after a polynomial number of steps.*

7 LTL Specifications

In this section we extend our results to LTL. Since LTL specifications can be translated into universal Co-Büchi tree automata [10], our approach is to first extend our results so that they apply directly to universal Co-Büchi tree automata, and then to use this intermediate step to obtain our final results for LTL.

We start with universal Co-Büchi tree automata. Recall that the arguments in Sections 4 and 6 start with a CTL formula, translate the formula into a universal Co-Büchi tree automaton, and then provide proofs that deal only with the resulting automaton. Reusing the proofs from these sections immediately gives the following two properties.

Theorem 15. *Let $\mathcal{U}$ be a universal Co-Büchi tree (or safety word) automaton that accepts Mealy machines.*

1. *If PSPACE = EXPTIME then there is an online Turing machine $\mathcal{M}$ with $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{U})$, where $|\mathcal{M}|$ is polynomial in the states and a representation of the transition function of $\mathcal{U}$.*
2. *If EXPTIME $\subseteq$ P/poly then there is a polynomially-sized online Turing machine $\mathcal{M}$ with $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{U})$, which responds to every input after a polynomial number of steps.*

The other two results can be generalized using slight alterations of our existing techniques. An analogue of the result in Section 3 can be obtained by using the fact that checking whether online Turing machine $\mathcal{M}$ is accepted by a universal Co-Büchi tree automaton $\mathcal{U}$ can be done in in $O\big((\log|\mathcal{U}|+\log|\mathcal{T}(\mathcal{M})|)^2\big)$ time [18, Theorem 3.2]. For the result in Section 5, we can obtain an analogue by using a very similar proof. The key difference is in Lemma 10, where we must show that there is universal Co-Büchi tree automaton with the same properties as ϕ_b. The construction of a suitable universal Co-Büchi tree automaton appears in the full version of the paper. Thus, we obtain the other two directions.

Theorem 16. *Let $\mathcal{U}$ be a universal safety word (or universal Co-Büchi tree) automaton that accepts Mealy machines.*

1. *If there is always an online Turing machine $\mathcal{M}$ in the language of $\mathcal{U}$, where $|\mathcal{M}|$ has polynomial size in the states of $\mathcal{U}$, then PSPACE=EXPTIME.*
2. *If there is always an online Turing machine $\mathcal{M}$ in the language of $\mathcal{U}$, where $|\mathcal{M}|$ has polynomial size in the states of $\mathcal{U}$, which also responds to every input after polynomially many steps, then EXPTIME $\subseteq$ P/poly.*

Now we move on to consider LTL specifications. For LTL, the situation is more complicated, because the translation from LTL formulas to universal Co-Büchi tree automata does not give the properties used in Lemma 6.

Theorem 17. *[10] Given an LTL formula ϕ, we can construct a universal Co-Büchi tree automaton $\mathcal{U}_\phi$ with $2^{O(|\phi|)}$ states that accepts a Mealy machine $\mathcal{T}$ if, and only if, $\mathcal{T}$ is a model of ϕ.*

Note that this translation gives a universal Co-Büchi tree automaton that has exponentially many states in $|\phi|$. Unfortunately, this leads to less clean results for LTL. For the results in Sections 3 and 4, we have the following analogues.

Theorem 18. *We have both of the following:*

1. *If every LTL formula ϕ has an online Turing machine model $\mathcal{M}$, where $|\mathcal{M}|$ is exponential in $|\phi|$, then EXPSPACE = 2EXPTIME.*
2. *If PSPACE = EXPTIME, then every LTL formula has an exponentially sized online Turing machine model.*

The first of these two claims is easy to prove: we can guess an exponentially sized online Turing machine, and then model check it. For the second claim, we simply apply Theorem 15.

In fact, we can prove a stronger statement for the second part of Theorem 18. QPSPACE is the set of problems that can be solved in $O(2^{(\log n_d)^c})$ space for some constant c. We claim that if EXPTIME $\subseteq$ QPSPACE, then, for every LTL formula ϕ, there is an exponentially sized online Turing machine $\mathcal{M}$ that models ϕ. This is because, in the proofs given in Section 4, if we have an algorithm that solves the emptiness problem in QPSPACE, then the online Turing machine that we construct will still run in exponential time in the formula.

We can also prove one of the two results regarding small and fast online Turing machines. The following result is implied by Theorem 15.

Theorem 19. *If EXPTIME $\subseteq$ P/poly then every LTL formula has an exponentially sized online Turing machine model, which responds to every input after an exponential number of steps.*

We can strengthen this result to "If all EXPTIME problems are polylogspace reducible to P/poly then every LTL formula has an exponentially sized online Turing machine model, which responds to every input after an exponential number of steps" with the same reason we used for strengthening the previous theorem: in the proofs given in Section 4, if we have an algorithm that solves the emptiness problem in QPTIME using an advice tape of quasi-polynomial size, then the online Turing machine that we construct will still run in exponential time in the formula.

However, we cannot prove the opposite direction. This is because the proof used in Theorem 16 would now produce an exponential time Turing machine with an advice function that gives exponentially sized advice strings. Therefore, we cannot draw any conclusions about the class P/poly.

References

1. Church, A.: Logic, arithmetic and automata. In: Proceedings of the International Congress of Mathematicians, Institut Mittag-Leffler, Djursholm, Sweden, Stockholm, August 15–22, pp. 23–35 (1962,1963)
2. Cimatti, A., Clarke, E., 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)
3. Ehlers, R.: Unbeast: Symbolic Bounded Synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
4. 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)
5. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proc. of PLDI, pp. 62–73 (2011)
6. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
7. Karp, R., Lipton, R.: Turing machines that take advice. Enseign. Math 28(2), 191–209 (1982)
8. Kupferman, O., Vardi, M.Y.: Freedom, weakness, and determinism: From linear-time to branching-time. In: Proc. of LICS (June 1995)
9. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete informatio. In: Proc. of ICTL, pp. 91–106 (July 1997)
10. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. of FOCS, pp. 531–540 (2005)
11. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)
12. Papadimitriou, C.: Computational Complexity. Addison Wesley Pub. Co. (1994)
13. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
14. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. of POPL, pp. 179–190 (1989)
15. Safra, S.: On the complexity of omega-automata. In: Proc. of FOCS, pp. 319–327 (1988)
16. 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)
17. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proc. of ASPLOS, pp. 404–415 (2006)
18. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. of LICS, Cambridge, UK, pp. 322–331 (1986)

Controllers with Minimal Observation Power (Application to Timed Systems)*

Peter Bulychev[1], Franck Cassez[2], Alexandre David[1], Kim Guldstrand Larsen[1],
Jean-François Raskin[3], and Pierre-Alain Reynier[4]

[1] CISS, CS, Aalborg University, Denmark
{pbulychev,adavid,kgl}@cs.aau.dk
[2] National ICT Australia, Sydney, Australia
Franck.Cassez@nicta.com.au
[3] Computer Science Department, Université Libre de Bruxelles (U.L.B.), Belgium
jraskin@ulb.ac.be
[4] LIF, Aix-Marseille University & CNRS, France
pierre-alain.reynier@lif.univ-mrs.fr

Abstract. We consider the problem of controller synthesis under imperfect information in a setting where there is a set of available observable predicates equipped with a cost function. The problem that we address is the computation of a subset of predicates sufficient for control and whose cost is minimal. Our solution avoids a full exploration of all possible subsets of predicates and reuses some information between different iterations. We apply our approach to timed systems. We have developed a tool prototype and analyze the performance of our optimization algorithm on two case studies.

1 Introduction

Timed automata by Alur and Dill [2] is one of the most popular formalism for the modeling of real-time systems. One of the applications of Timed Automata is *controller synthesis*, i.e. the automatic synthesis of a controller strategy that forces a system to satisfy a given specification. For timed systems, the controller synthesis problem has been first solved in [18] and progress on the algorithm obtained in [9] has made possible the application on examples of a practical interest. This algorithm has been implemented in the Uppaal-Tiga tool [3], and applied to several case studies [1,10,11,20].

The algorithm of [9] assumes that the controller has *perfect information* about the evolution of the system during its execution. However, in practice, it is common that the controller acquires information about the state of the system via a finite set of sensors each of them having only a finite precision. This motivates to study *imperfect information* games.

The first theoretical results on *imperfect information* games have been obtained in [22], followed by algorithmic progresses and additional theoretical

* Partly supported by ANR project ECSPER (JC09- 472677), ERC Starting Grant inVEST-279499, Danish-Chinese Center for Cyber Physical Systems (IDEA4CPS) and VKR Center of Excellence MT-LAB.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 223–237, 2012.

results in [21], as well as application to timed games in [6,8]. This paper extends the framework of [8] and so we consider the notion of *stuttering-invariant observation-based strategies* where the controller makes choice of actions only when changes in its observation occur. The observations are defined by the values of a finite set of *observable state predicates.* Observable predicates correspond, for example, to information that can be obtained through sensors by the controller. In [8], a symbolic algorithm for computing observation-based strategies for a *fixed* set of observable predicates is proposed, and this algorithm has been implemented in Uppaal-Tiga.

In the current paper, we further develop the approach of [8] and consider a set of *available* observation predicates equipped with a cost function. Our objective is to synthesize a winning strategy that uses a subset of the available observable predicates with a *minimal cost.* Clearly, this can be useful in the design process when we need to select sensors to build a controller.

Our algorithm works by iteratively picking different subsets of the set of the available observable predicates, solving the game for these sets of predicates and finally finding the controllable combination with the minimal cost. Our algorithm avoids the exploration of all possible combinations by taking into account the inclusion-set relations between different sets of observable predicates and monotonic properties of the underlying games. Additionally, for efficiency reasons, our algorithm reuses, when solving the game for a new set of observation predicates, information computed on previous sets whenever possible.

Related Works. Several works in the literature consider the synthesis of controllers along with some notion of optimality [5,7,4,12,16,23,13,19] but they consider the minimization of a cost along the execution of the system while our aim is to minimize a static property of the controller: the cost of observable predicates on which its winning strategy is built. The closest to our work is [13] where the authors consider the related but different problem of turning on and off sensors during the execution in order to minimize energy consumption. In [15], the authors consider games with perfect information but the discovery of interesting predicates to establish controllability. In [14] this idea is extended to games with imperfect information. In those two works the set of predicates is not fixed a priori, there is no cost involved and the problems that they consider are undecidable. In [19], a related technique is used: a hierarchy on different levels of abstraction is considered, which allows to use analysis done on coarser abstractions to reduce the state space to be explored for more precise abstractions.

Structure of the Paper. In section 2, we define a notion of *labeled transition systems* that serves as the underlying formalism for defining the semantics of the two-player safety games. In the same section we define imperfect information games and show the reduction of [22] of these games to the games with complete information. Then in section 3 we define *timed game automata*, that we use as a modeling formalism. In section 4, we state the cost-optimal controller synthesis problem and show that a natural extension of this problem (that considers a simple infinite set of observation predicates) is undecidable. In section 5, we propose an algorithm and in section 6, we present two case studies.

2 Games with Incomplete Information

2.1 Labeled Transition Systems

Definition 1 (Labeled Transition System). *A* Labeled Transition System *(LTS) A is a tuple $(S, s_{init}, \Sigma, \rightarrow)$ where:*

- *S is a (possibly infinite) set of states,*
- *$s_{init} \in S$ is the initial state,*
- *Σ is the set of actions,*
- *$\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation, we write $s_1 \xrightarrow{a} s_2$ if $(s_1, a, s_2) \in \rightarrow$.*

W.l.o.g. we assume that a transition relation is total, i.e. for all states $s \in S$ and actions $a \in \Sigma$, there exists $s' \in S$ such that $s \xrightarrow{a} s'$.

A *run* of a LTS is a finite or infinite sequence of states $r = (s_0, s_1, \ldots, s_n, \ldots)$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for some action $a_i \in \Sigma$. r^i denotes the prefix run of r ending at s_i. We denote by $Runs(A)$ the set of all finite runs of the LTS A and by $Runs^{\omega}(A)$ the set of all infinite runs of the LTS A.

A *state predicate* is a characteristic function $\varphi : S \rightarrow \{0, 1\}$. We write $s \models \varphi$ iff $\varphi(s) = 1$.

We use LTS as arenas for games: at each round of the game Player I (Controller) chooses an action $a \in \Sigma$, and Player II (Environment) resolves the nondeterminism by choosing a transition labeled with a. Starting from the state s_{init}, the two players play for an infinite number of rounds, and this interaction produces an infinite run that we call the *outcome* of the game. The objective of Player I is to keep the game in states that satisfy a state predicate φ, this predicate typically models the safe states of the system.

More formally, Player I plays according to a strategy λ (of Player I) which is a mapping from the set of finite runs to the set of actions, i.e. $\lambda : Runs(A) \rightarrow \Sigma$. We say that an infinite run $r = (s_0, s_1, s_2, \ldots, s_n, \ldots) \in Runs^{\omega}(A)$ is *consistent* with the strategy λ, if for all $0 \le i$, there exists a transition $s_i \xrightarrow{\lambda(r^i)} s_{i+1}$. We denote by $\mathsf{Outcome}(A, \lambda)$ all the infinite runs in A that are consistent with λ and start in s_{init}. An infinite run $(s_0, s_1, \ldots, s_n, \ldots)$ *satisfies* a state predicate φ if for all $i \ge 0$, $s_i \models \varphi$. A (perfect information) *safety game* between Player I and Player II is defined by a pair (A, φ), where A is an LTS and φ is a state predicate that we call a *safety state predicate*. The *safety game problem* asks to determine, given a game (A, φ), if there exists a strategy λ for Player I such that all the infinite runs in $\mathsf{Outcome}(A, \lambda)$ satisfy φ.

2.2 Observation-Based Stuttering-Invariant Strategies

In the imperfect information setting, Player I observes the state of the game using a set of *observable predicates* $obs = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$. An *observation* is a valuation for the predicates in *obs*, i.e. in a state s, Player I is given the subset of observable predicates that are satisfied in that state. This is defined by the function γ_{obs}:

$$\gamma_{obs}(s) \equiv \{\varphi \in obs \mid s \models \varphi\}$$

We extend the function γ_{obs} to sets of states that satisfy the same set of observation predicates. So, if all the elements of some set of states $v \subseteq S$ satisfy the same set of observable predicates o (i.e. $\forall s \in v \cdot \gamma_{obs}(s) = o$), then we let $\gamma_{obs}(v) = o$.

In a game with imperfect information, Player I has to play according to *observation based stuttering invariant strategies* (OBSI strategies for short). Initially, and whenever the current observation of the system state changes, Player I proposes some action $a \in \Sigma$ and this intuitively means that he wants to play the action a whenever this action is enabled in the system. Player I is not allowed to change his choice as long as the current observation remains the same.

An *Imperfect Information Safety Game* (IISG) is defined by a triple (A, φ, obs).

Consider a run $r = (s_0, s_1, \ldots, s_n)$, and its prefix r' that contains all the elements but the last one (i.e. $r = r' \cdot s_n$). A *stuttering-free* projection $r \downarrow obs$ of a run r over a set of predicates *obs* is a sequence, defined by the following inductive rules:

- if r is a singleton (i.e. $n = 0$), then $r \downarrow obs = \gamma_{obs}(s_0)$
- else if $n > 0$ and $\gamma_{obs}(s_{n-1}) = \gamma_{obs}(s_n)$, then $r \downarrow obs = r' \downarrow obs$
- else if $n > 0$ and $\gamma_{obs}(s_{n-1}) \neq \gamma_{obs}(s_n)$, then $r \downarrow obs = r' \downarrow obs \cdot \gamma_{obs}(s_n)$

Definition 2. *[8] A strategy λ is called obs-Observation Based Stuttering Invariant (obs-OBSI) if for any two runs r' and r'' such that $r' \downarrow obs = r'' \downarrow obs$, the values of λ on r' and r'' coincide, i.e. $\lambda(r') = \lambda(r'')$.*

We say that Player I wins in IISG (A, φ, obs), if there exists a *obs*-OBSI strategy λ for Player I such that all the infinite runs in $\mathsf{Outcome}(A, \lambda)$ satisfy φ.

2.3 Knowledge Games

The solution of a IISG (A, φ, obs) can be reduced to the solution of a *perfect information* safety game (G, ψ), whose states are sets of states in A and represent the *knowledge* (beliefs) of Player I about the current possible states of A.

We assume that $\varphi \in obs$, i.e. the safety state predicate is observable for Player I. This is a reasonable assumption since Player I should be able to know whether he loses the game or not.

Consider an LTS $A = (S, s_{init}, \Sigma, \rightarrow)$. We say that a transition $s_1 \xrightarrow{a} s_2$ in A is *obs*-visible, if the states s_1 and s_2 have different observations (i.e. $\gamma_{obs}(s_1) \neq \gamma_{obs}(s_2)$), otherwise we call this transition to be *obs*-invisible. Let $v \subseteq S$ be *a knowledge* (belief) of Player I in A, i.e. it is some set of states that satisfy the same observation. The set $Post_{obs}(v, a)$ contains all the states that are accessible from the states of v by a finite sequence of a-labeled *obs*-invisible transitions followed by an a-labeled *obs*-visible transition. More formally, $Post_{obs}(v, a)$ contains all the states s', such that there exists a run $s_1 \xrightarrow{a} s_2 \xrightarrow{a} \ldots \xrightarrow{a} s_n$ and $s_1 \in v$, $s_n = s'$, $\gamma_{obs}(s_i) = \gamma_{obs}(s)$ for all $1 \leq i < n$, and $\gamma_{obs}(s_n) \neq \gamma_{obs}(s)$.

The set $Post_{obs}(v, a)$ contains all the states that are visible for Player I after he continuously offers to play action a from some state in v. Player I can distinguish the states s_1 and s_2 from $Post_{obs}(v, a)$ iff they have different observations, i.e.

$\gamma_{obs}(s_1) \neq \gamma_{obs}(s_2)$. In other words, the set $\{Post_{obs}(v,a) \cap \gamma_{obs}^{-1}(o) \mid o \in \mathcal{P}(obs)\} \setminus \{\varnothing\}$ consists of all the beliefs that Player I might have after he plays the a action from the knowledge set v[1].

A game can *diverge* in the current observation after playing some action. To capture this we define the boolean function $Sink_{obs}(v,a)$ whose value is true iff there exists an infinite run $(s_0, s_1, \dots, s_n, \dots) \in Runs(A)$ such that $s_0 \in v$ and for each $i \geq 0$ we have $s_i \xrightarrow{a} s_{i+1}$ and $\gamma_{obs}(s_i) = \gamma_{obs}(s_0)$.

Definition 3. *We say, that a game (G, ψ) is the knowledge game for (A, φ, obs), if $G = (V, v_{init}, \Sigma, \rightarrow_g)$ is an LTS and*

– $V = \{v \in \mathcal{P}(S) \mid \forall s_1, s_2 \in v \cdot \gamma_{obs}(s_1) = \gamma_{obs}(s_2)\} \setminus \{\varnothing\}$ *is the set of all the beliefs of Player I in A,*
– $v_{init} = \{s_{init}\}$ *is the initial game state,*
– $\rightarrow_g$ *represents the game transition relation; a transition* $v_1 \xrightarrow{a}_g v_2$ *exists iff:*
  - $v_2 = Post_{obs}(v_1, a) \cap \gamma_{obs}^{-1}(o)$ *and* $v_2 \neq \varnothing$ *for some* $o \subseteq obs$*, or*
  - $Sink_{obs}(v_1, a)$ *is true and* $v_2 = v_1$*.*
– $v \models \psi$ *iff* $\varphi \in \gamma_{obs}(v)$*.*

Theorem 1 ([8]). *Player I wins in a IISG (A, φ, obs) iff he has a winning strategy in the safety game (G, ψ) which is the knowledge game for (A, φ, obs).*

This theorem gives us the algorithm of solution of a IISG for the case when the knowledge games for it is finite and can be automatically constructed.

3 Timed Game Automata

The knowledge game (G, ψ) for (A, φ, obs) is finite when the source game A is finite [22]. The converse is not true and there are higher level formalisms that can induce *infinite* games for which knowledge games are still *finite* and can be automatically constructed. One of such formalisms is Timed Game Automata [17], that we use as a modeling formalism and that has been proved in [8] to have finite state knowledge games.

Let X be a finite set of real-valued variables called clocks. We denote by $\mathcal{C}(X)$ the set of constraints ψ generated by the grammar: $\psi ::= x \sim k \mid x - y \sim k \mid \psi \wedge \psi$ where $k \in \mathbb{N}$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X)$ is the set of constraints generated by the following grammar: $\psi ::= \top \mid k_1 \leq x < k_2 \mid \psi \wedge \psi$ where $k, k_1, k_2 \in \mathbb{N}$, $k_1 < k_2$, $x \in X$, and $\top$ is the boolean constant *true*.

A *valuation* of the clocks in X is a mapping $X \mapsto \mathbb{R}_{\geq 0}$. For $Y \subseteq X$, we denote by $v[Y]$ the valuation assigning 0 (respectively, $v(x)$) for any $x \in Y$ (respectively, $x \in X \setminus Y$). We also use the notation $\mathbf{0}$ for the valuation that assigns 0 to each clock from X.

[1] The powerset $\mathcal{P}(S)$ is equal to the set of all subsets of S

Definition 4 (Timed Game Automata). *A* Timed Game Automaton *(TGA) is a tuple* $(L, l_{init}, X, E, \Sigma_c, \Sigma_u, I)$ *where:*

- L *is a finite set of locations,*
- $l_{init} \in L$ *is the initial location,*
- X *is a finite set of real-valued clocks,*
- Σ_c *and* Σ_u *are finite the sets of controllable and uncontrollable actions (of Player I and Player II, correspondingly),*
- $E \subseteq (L \times \mathcal{B}(X) \times \Sigma_c \times 2^X \times L) \cup (L \times \mathcal{C}(X) \times \Sigma_u \times 2^X \times L)$ *is partitioned into controllable and uncontrollable transitions*[2]*,*
- $I : L \to \mathcal{B}(X)$ *associates to each location its* invariant*.*

We first briefly recall *the non-game* semantics of TGA, that is the semantics of Timed Automata (TA) [2]. A state of TA (and TGA) is a pair (l, v) of a location $l \in L$ and a valuation v over the clocks in X. An automaton can do two types of transitions, that are defined by the relation $\hookrightarrow$:

- a **delay** $(l, v) \overset{t}{\hookrightarrow} (l, v')$ for some $t \in \mathbb{R}_{>0}$, $v' = v + t$ and $v' \models I(l)$, i.e. to stay in the same location while the invariant of this location is satisfied, and during this delay all the clocks grow with the same rate, and
- a **discrete transition** $(l, v) \overset{a}{\hookrightarrow} (l', v')$ if there is an element $(l, g, a, Y, l') \in E$, $v \models g$ and $v' = v[Y]$, i.e. to go to another location l' with resetting the clocks from Y, if the guard g and the invariant of the target location l' are satisfied.

In the remainder of this section, we define the *game semantics* of TGA. As in [8], for TGA, we let observable predicates be of the form (K, ψ), where $K \subseteq L$ and $\psi \in \mathcal{B}(X)$. We say that a state (l, v) satisfies (K, ψ) iff $l \in K$ and $v \models \psi$.

Intuitively, whenever the current observation of the system state changes, Player I proposes a controllable action $a \in \Sigma_c$ and as long as the observation does not change Player II has to play this action when it is enabled, and otherwise he can play any uncontrollable actions or do time delay. Player I can also propose a special action **skip**, that means that he lets Player II play any uncontrollable actions and do time delay. Any time delay should be stopped as soon as the current observation is changed, thus giving a possibility for Player I to choose another action to play.

Formally, the semantics of TGA is defined by the following definition:

Definition 5. *The semantics of TGA* $(L, l_{init}, X, E, \Sigma_c, \Sigma_u, I)$ *with the set of observable predicates obs is defined as the LTS* $(S, s_{init}, \Sigma_c \cup \{\mathbf{skip}\}, \to)$*, where* $S = L \times \mathbb{R}^X_{\geq 0}$*,* $s_{init} = (l_{init}, \mathbf{0})$ *and the transition relation is: (*$\hookrightarrow$ *denotes the non-game semantics of* M*)*

- $s \xrightarrow{\mathbf{skip}} s'$ *exists, iff* $s \overset{a_u}{\hookrightarrow} s'$ *for some* $a_u \in \Sigma_u$*, or there exists a delay* $s \overset{t}{\hookrightarrow} s'$ *for some* $t \in \mathbb{R}_{>0}$ *and any smaller delay doesn't change the current observation (i.e. if* $s \overset{t'}{\hookrightarrow} s''$ *and* $0 \leq t' < t$ *then* $\gamma_{obs}(s) = \gamma_{obs}(s'')$*).*

[2] We follow the definition of [8] that also assumes that the guards of the controllable transitions should be of the form $k_1 \leq x < k_2$. This allows us to use the results from that paper. In particular, we use *urgent* semantics for the controllable transitions, i.e. for any controllable transition there is an exact moment in time when it becomes enabled.

- *for* $a \in \Sigma_c$, $s \xrightarrow{a} s'$ *exists, iff:*
  - a *is enabled in* s *and there exists a discrete transition* $s \overset{a}{\hookrightarrow} s'$, *or*
  - a *is not enabled in* s, *but there exists a discrete transition* $s \overset{a_u}{\hookrightarrow} s'$ *for some* $a_u \in \Sigma_u$, *or*
  - *there exists a delay* $s \overset{t}{\hookrightarrow} s'$ *for some* $t \in \mathbb{R}_{>0}$, *and for any smaller delay* $s \overset{t'}{\hookrightarrow} s''$ *(where* $0 \leq t' < t$*) the observation is not changed, i.e.* $\gamma_{obs}(s) = \gamma_{obs}(s'')$, *and action* a *is not enabled in* s''.

For a given TGA M, set of observable predicates *obs* and a safety state-predicate φ (that can be again of the form (K, ψ)), we say that Player I wins in the Imperfect Information Safety Timed Game (IISTG) (M, φ, obs) iff he wins in the IISG (A, φ, obs), where A defines the semantics for M and *obs*.

The problem of solution of IISTG is decidable since the knowledge games are finite for TGA [8]. The paper [8] proposes a symbolic Difference Bounded Matrices (DBM)-based procedure to construct them.

4 Problem Statement

Consider that several observable predicates are available, with assigned costs, and we look for a set of observable predicates allowing controllability and whose cost is minimal. This is formalized in the next definition:

Definition 6. *Consider a TGA* M*, a finite set of* available *observable predicates* Obs *over* M*, a safety observable predicate* $\varphi \in Obs$ *and a monotonic with respect to set inclusion function* $\omega : \mathcal{P}(Obs) \to \mathbb{R}_{\geq 0}$. *The optimization problem for* $(M, \varphi, Obs, \omega)$ *consists in computing a set of observable predicates* $obs \subseteq Obs$ *such that Player I wins in the Imperfect Information Safety Timed Game* (M, φ, obs) *and* $\omega(obs)$ *is minimal.*

We present in the next section our algorithm to compute a solution to the optimization problem. In this paper, we restrict our attention to *finite* sets of available predicates. We justify this restriction by the following undecidability result: considering a reasonable infinite set of observation predicates, the easier problem of the existence of a set of predicates allowing controllability is undecidable:

Theorem 2. *Consider a TGA* M *with clocks* X*, and an (infinite) set of available predicates* $Obs = \{x < \frac{1}{q} \mid x \in X, q \in \mathbb{N}, q \geq 1\}$ *and the safety objective* φ. *Determining whether there exists a finite set of predicates* $obs \subset Obs$ *such that Player I wins in IISTG* (M, φ, obs) *is undecidable.*

5 The Algorithm

The naive algorithm is to iterate through all the possible solutions $\mathcal{P}(Obs)$, for each $obs \in \mathcal{P}(Obs)$ solve IISTG (M, φ, obs) via the reduction to the finite-state knowledge games, and finally pick a solution with the minimal cost.

In section 5.1 we propose the more efficient algorithm that avoids exploring all the possible solutions from $\mathcal{P}(Obs)$. Additionally, in sections 5.2 we describe the optimization that reuses the information between different iterations.

Algorithm 1. Lattice-based algorithm

```
//input: TGA $M$, a set of observable predicates $Obs$, a safety predicate $\varphi$
//output: a solution with a minimal cost
function $Optimize(M, \varphi, Obs, \omega)$:
    $candidates := \mathcal{P}(Obs)$ // initially, $candidates$ contains all subsets of $Obs$
    $best\_candidate := None$
    while $candidates \neq \varnothing$:
        pick $obs \in candidates$
        if $Solve(M, \varphi, obs)$:
            $best\_candidate := obs$
            $candidates = candidates \setminus \{c : c \in \mathcal{P}(Obs) \wedge \omega(c) \geq \omega(obs)\}$
        else:
            $candidates = candidates \setminus \{c : c \in \mathcal{P}(Obs) \wedge c \subseteq obs\}$
     return $best\_candidate$
```

5.1 Basic Exploration Algorithm

Consider, that we already solved the game for the observable predicates sets $obs_1, obs_2, \ldots, obs_n$ and obtained the results $r_1, r_2, \ldots, r_n$, where r_i is either $true$ or $false$, depending on whether Player I wins in IISTG (M, φ, obs_i) or not.

From now on we don't have to consider any set of observable predicates with a cost larger or equal to the cost of the optimal solution found so far. Additionally, if we know, that Player I loses for the set of observable predicates obs_i (i.e. $r_i = false$), then we can conclude that he also loses for any coarser set of observable predicates $obs \subset obs_i$ (since in this case Player I has less observation power). Therefore we don't have to consider such obs as a solution to our optimization problem. This can be formalized by the following definition:

Definition 7. *A sequence $(obs_1, r_1), (obs_2, r_2) \ldots (obs_n, r_n)$ is called a non-redundant sequence of solutions for a set of available observable predicates Obs and cost function ω, if for any $1 \leq i \leq n$ we have $obs_i \subseteq Obs$, $r_i \in \{true, false\}$, and for any $j < i$ we have:*

- *$\omega(obs_j) > \omega(obs_i)$ if $r_j = true$,*
- *$obs_i \nsubseteq obs_j$, otherwise.*

Algorithm 1 solves the optimization problem by iteratively solving the game for different sets of observable predicates so that the resulting sequence of solutions is non-redundant. The procedure $Solve(M, \varphi, obs)$ uses the knowledge game-reduction technique described in section 2. The algorithm updates the set *candidates* after each iteration and when the algorithm finishes, the *best_candidate* variable contains a reference to the solution with the minimal cost.

Algorithm 1 doesn't state, in which order we should navigate through the set of candidates. We propose the following heuristics:

- *cheap first* (and *expensive first*) — pick any element from the *candidates* with the maximal (or minimal) cost,

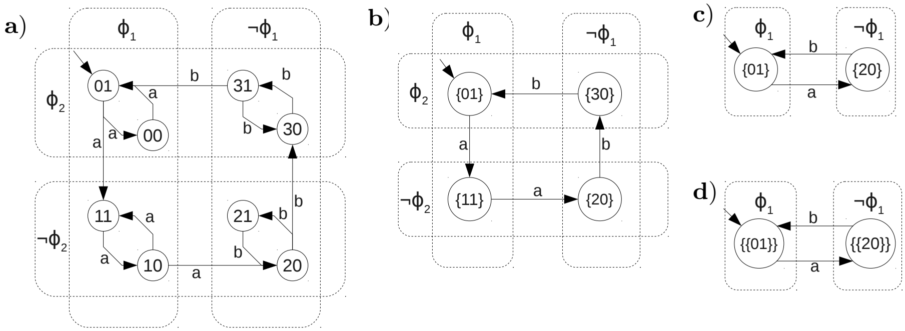


Fig. 1. a) The original LTS A and two observable predicates φ_1 and φ_2, b) the knowledge game G_f for A with observable predicates $\{\varphi_1, \varphi_2\}$, c) the knowledge game G_c^1 for A with observable predicates $\{\varphi_1\}$, d) the knowledge game G_c^2 for G_f with observable predicates $\{\varphi_1\}$

– *random* — pick a random element from the *candidates*,
– *midpoint* — pick any element, that will allow us to eliminate as many elements from the *candidates* set as it is possible. In other words, we pick an element that maximizes the value of
$\min(|\{c : c \in \mathit{candidates} \wedge w(c) \geq w(\mathit{obs})\}|, |\{c : c \in \mathit{candidates} \wedge c \subseteq \mathit{obs}\}|)$.

Algorithm 1 doesn't specify how we store the set of possible solutions *candidates*. An explicit way (i.e. store all elements) is expensive, because the *candidates* set initially contains $2^{|Obs|}$ elements. However, an efficient procedure for obtaining a next candidate may not exist as a consequence of the following theorem:

Theorem 3. *Let* $seq_n = (obs_1, r_1), (obs_2, r_2), \ldots, (obs_n, r_n)$ *be a non-redundant sequence of solutions for some set* Obs *and cost function* $\omega : \mathcal{P}(Obs) \to \mathbb{R}_{\geq 0}$*. Consider that the value of* ω *can be computed in polynomial time. Then the problem of determining whether there exists a one-element extension* $seq_{n+1} = (obs_1, r_1), (obs_2, r_2), \ldots, (obs_n, r_n), (obs_{n+1}, r_{n+1})$ *of* seq *that is still non-redundant for* Obs *and* ω *is NP-complete.*

5.2 State Space Reusage from Finer Observations

Intuitively, if we have already solved a knowledge game (G_f, ψ_f) for a set obs_f of observable predicates, then we can view a knowledge game (G_c, ψ_c) associated with a *coarser* set of observable predicates $obs_c \subset obs_f$ as an imperfect information game with respect to (G_f, ψ_f). Thus we can solve the knowledge game for *obs* without exploring the state space of the TGA M and therefore without using the expensive DBM operations. Moreover, we can build another game on top of G_c (for an observable predicates set that is coarser than *obs*) and thus construct a "Russian nesting doll" of games. This is an important contribution of our paper, since this construction can be applied not only to Timed Games, but also to any modeling formalism that have finite knowledge games.

The state space reusage method is demonstrated on a simple LTS A at Fig. 1. Suppose, that we already built the knowledge game G_f for the observable predicates $\{\varphi_1, \varphi_2\}$. Now, if we want to build a knowledge game for $\{\varphi_1\}$, we can do that in two ways. First, we can build it from scratch based on the state space of A, and the resulting knowledge game G_c^1 is given at subfigure c. Alternatively, we can build the knowledge game G_c^2 on the top of G_f (see subfigure d). The states of G_c^1 are sets of states of A and the states of G_c^2 are sets of sets of states of A. The games G_c^2 and G_c^1 are bisimilar, thus Player I wins in G_c^1 iff he wins in G_c^2 (for any safety predicate). The latter is true for any LTS A, that is stated by the following theorem and corollary:

Theorem 4. *Suppose that* $obs_c \subset obs_f$, (G_f, ψ_f) *is the knowledge game for* (A, φ, obs_f), (G_c^1, ψ_c^1) *is the knowledge game for* (A, φ, obs_c) *and* (G_c^2, ψ_c^2) *is the knowledge game for* (G_f, ψ_f, obs_c). *Then the relation* $R = \{(v, v') | v = \bigcup_{s' \in v'} s'\}$ *between the states of* G_c^1 *and* G_c^2 *is a bisimulation.*

Corollary 1. *Player I wins in* (G_c^1, ψ_c^1) *iff Player I wins in* (G_c^2, ψ_c^2).

This reusage method is also correct for the case when an input model is defined as a TGA (since we can apply the theorem to the underlying LTS).

Implementation. Our Python prototype implementation of this algorithm (see `https://launchpad.net/pytigaminobs`) explicitly stores the set of candidates and uses the on-the-fly DBM-based algorithm of [8] for the construction and solution of knowledge games for IISTG (the algorithm stops early when it detects that the initial state is losing).

6 Case Studies

We applied our implementation to two case studies.

The first is a “Train-Gate Control”, where two trains tracks merge together on a bridge and the goal of the controller is to prevent their collision. The trains can arrive in any order (or don’t arrive at all), thus the challenge for the controller is to handle all possible cases.

The second is “Light and Heavy boxes”, where a box is being processed on the conveyor in several steps, and the goal of the controller is to move the box to the next step within some time bound after it has been processed at the current step.

6.1 Train-Gate Control

The model of a single (first) train is depicted at Fig. 2. There are two semaphore lights before the bridge on each track. A train passes the distance between semaphores within 1 to 2 time units. A controller can switch the semaphores to red (actions `stop1` and `stop2` depending on the track number), and to green (actions `go1` and `go2`). These semaphores are intended to prevent the trains from

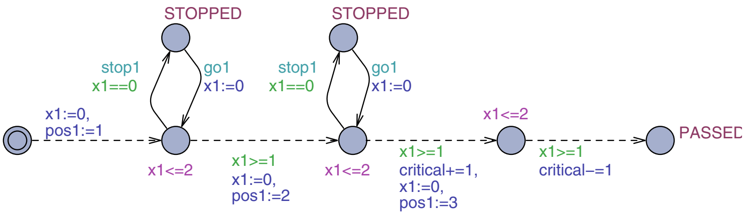


Fig. 2. A model of a single train

colliding on the bridge. When the red signal is illuminated, a train will stop at the next semaphore and wait for the green signal.

It is possible to mount sensors on the semaphores, and these sensors will detect if a train approaches the semaphore. This is modeled with observable predicates $(pos1 \geq 1)$, $(pos2 \geq 1)$, $(pos1 \geq 2)$ and $(pos2 \geq 2)$.

The controller has a discrete timer that is modeled using the clock y. At any time this clock can be reset by the controller (action `reset`). There is an available observable predicate $(y < 2)$ that becomes false when the value of y reaches 2. This allows the controller to measure time with a precision 2 by resetting y each time this predicate becomes false and counting the number of such resets.

The integer variable *critical* contains the number of trains that are currently on the bridge. The safety property is that no more than one train can be at the critical section (bridge) at the same time and the trains should not be stopped for more than 2 time units:

$$(critical < 2) \wedge ((Train1.STOPPED) \rightarrow (x1 \leq 2)) \wedge ((Train2.STOPPED) \rightarrow (x2 \leq 2))$$

The optimal controller uses the following set of observable predicates: $(pos1 \geq 2)$, $(pos2 \geq 2)$ and $(y < 2)$. Such a controller waits until the second (in time) train comes to the second semaphore, then pauses this train and lets it go after 2 time units.

Figure 3a reports the time needed to find this solution for different parameters of the algorithm. Figure 3b contains the average number of iterations of

exploration order	expensive first		cheap first		midpoint		random	
state space reusage	with	without	with	without	with	without	with	without
minimum	10m	1h03m	50m	49m	24m	41m	10m	48m
maximum	11m	1h36m	1h30m	1h34m	55m	1h36m	1h26m	1h44m
average	10m	1h18m	1h0m	1h12m	33m	1h03m	37m	1h05m

(a) Running time (the average is computer on 10 runs)

exploration order	expensive first	cheap first	midpoint	random
without state space reusage	1	21.69	5.27	6.17
with state space reusage	7.1	0	2.7	3.46

(b) The average number of iterations

Fig. 3. Results for the Train-Gate model

Algorithm 1 (i.e. game checks for different sets of observable predicates). You can see that it requires only a fraction of the total number of all possible solutions $2^5 = 32$. Additionally, the state space reusage heuristic allows to improve the performance, especially for the "expensive first" exploration order. For this model the most efficient way to solve the optimization problem is to first solve the game with all the available predicates being observed, and then always reuse the state space of this knowledge game. The numbers of 0 and 1 at Figure 3b reflect that we *don't* reuse the state space exactly once for the "expensive first" order, and we never reuse the state space for the "cheap first" exploration order.

The game size ranges from 5 states for the game when only the safety state predicate is observable to 9202 for the case when all the available predicates are observable. The number of the symbolic states of TGA (i.e. different pairs of reachable locations and DBMs that form the states of a knowledge game) ranges from 1297 to 31171, correspondingly.

6.2 Light and Heavy Boxes

Consider a conveyor belt on which Light and Heavy boxes can be put. A box is processed in n steps (n is a parameter of the model), and the processing at each step takes from 1 to 2 time units for the Light boxes, and from 4 to 5 time units for the Heavy boxes. The goal of the controller is to move a box to the next step (by rotating the conveyor, with an action `move`) within 3 time units after the box has been processed at the current step. At the last step the controller should remove (action `remove`) the box from the conveyor within 3 time units. If the controller rotates the conveyor too early (before the box has been processed), too late (after more than 3 time units), or does not move it at all, then the Controller loses (similar is true for the removing of the box at the last step). Additionally, the controller should not rotate the conveyor when there is no box on it, and should not try to remove the box when the box is not at the last step. Our model is depicted at Fig. 4, and the goal of the controller is to avoid the `BAD` location.

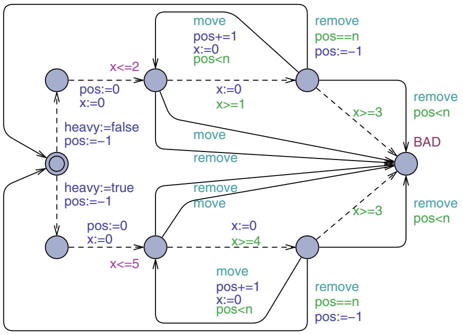


Fig. 4. Light and heavy boxes model

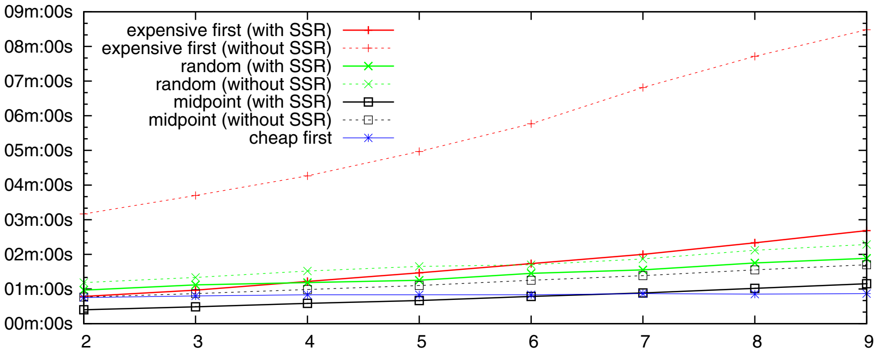


Fig. 5. Average running time (SSR states for State Space Reusage)

A box can arrive on the conveyor at any time, and there is an observable predicate ($pos = 0$) with cost 1 which becomes true when the box is put on the conveyor. Additionally, there is predicate ($heavy = true$) with cost 1 that becomes true if a heavy box arrives. The model is cyclic, i.e. another box can be put on the conveyor after the previous box has been removed from it.

As in the Traingate model, the controller can measure time using a special clock y. We assume that a controller can measure time with different granularity, and more precise clocks cost more. We model this by having three available observable predicates: ($y < 1$) with cost 3, ($y < 2$) with cost 2, and ($y < 3$) with cost 1.

A naive controller works with the observable predicates $\{(heavy = true), (pos = 0), (y < 1)\}$, resets the clock y each time a new box is arrived, and then move it to the next step (remove after the last iteration) each 2 time units if the box is *light* and 5 time units if the box is *heavy*. However, it is not necessary to use the expensive ($y < 1$) observable predicate, since a controller can move a box after each 3 (6 for heavy box) time units, thus the time granularity of 3 is enough and there is a controller that uses the observable predicates $\{(heavy = true), (pos = 0), (y < 3)\}$. Our implementation detects such an optimal solution, and Fig. 5 demonstrates an average time needed to compute this solution for different numbers of box processing steps n. You can see that the state space reusage heuristics improves the performance of the algorithm.

The game size for this model ranges from 4 knowledge game states and 51 symbolic NTA states when there are 2 processing steps and only safety predicate is observable to 6417 knowledge game states and 15554 symbolic NTA states for 9 processing steps and when all the available predicate are observable.

7 Conclusions

In this paper we have developed, implemented and evaluated an algorithm for the cost-optimal controller synthesis for timed systems, where the cost of a controller is defined by its observation power.

Our important contributions are two optimizations: the one that helps to avoid exploration of all possible solutions and the one that allows to reuse the state space and solve the imperfect information games on top of each other. Our experiments showed that these optimizations allow to improve the performance of the algorithm.

In the future, we plan to apply our method to other modeling formalisms that have finite state knowledge games.

References

1. AlAttili, I., Houben, F., Igna, G., Michels, S., Zhu, F., Vaandrager, F.W.: Adaptive scheduling of data paths using uppaal tiga. In: QFM, pp. 1–11 (2009)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
4. Bouyer, P., Brihaye, T., Bruyre, V., Raskin, J.-F.: On the optimal reachability problem of weighted timed automata. Formal Methods in System Design 31(2), 135–175 (2007)
5. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal Strategies in Priced Timed Game Automata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
6. 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)
7. Brihaye, T., Bruyère, V., Raskin, J.-F.: On Optimal Timed Strategies. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 49–64. Springer, Heidelberg (2005)
8. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed Control with Observation Based and Stuttering Invariant Strategies. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 192–206. Springer, Heidelberg (2007)
9. 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)
10. 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)
11. Cesta, A., Finzi, A., Fratini, S., Orlandini, A., Tronci, E.: Flexible Timeline-Based Plan Verification. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 49–56. Springer, Heidelberg (2009)
12. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: QUASY: Quantitative Synthesis Tool. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 267–271. Springer, Heidelberg (2011)
13. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Controller Synthesis with Budget Constraints. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 72–86. Springer, Heidelberg (2008)

14. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In: FSTTCS. LIPIcs, vol. 2, pp. 175–186. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)
15. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided Control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003)
16. Majumdar, R., Tabuada, P. (eds.): HSCC 2009. LNCS, vol. 5469. Springer, Heidelberg (2009)
17. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
18. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
19. Malinowski, J., Niebert, P., Reynier, P.-A.: A Hierarchical Approach for the Synthesis of Stabilizing Controllers for Hybrid Systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 198–212. Springer, Heidelberg (2011)
20. Orlandini, A., Finzi, A., Cesta, A., Fratini, S.: TGA-Based Controllers for Flexible Plan Execution. In: Bach, J., Edelkamp, S. (eds.) KI 2011. LNCS, vol. 7006, pp. 233–245. Springer, Heidelberg (2011)
21. Raskin, J.-F., Chatterjee, K., Doyen, L., Henzinger, T.A.: Algorithms for omega-regular games with imperfect information. Logical Methods in Computer Science 3(3) (2007)
22. Reif, J.H.: The complexity of two-player games of incomplete information. Journal of Computer and System Sciences 29(2), 274–301 (1984)
23. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158, 343–359 (1996)

Counterexample Guided Synthesis of Monitors for Realizability Enforcement

Matthias Güdemann[1], Gwen Salaün[2,1], and Meriem Ouederni[3]

[1] INRIA Rhône-Alpes, Grenoble, France
[2] Grenoble INP, France
[3] LINA, University of Nantes, France

Abstract. Many of today's software systems are built using distributed services, which evolve in different organizations. In order to facilitate their integration, it is necessary to provide a *contract* that the services participating in a composition should adhere to. A contract specifies interactions among a set of services from a global point of view. One important problem in a top-down development process is figuring out whether such a contract can be implemented by a set of services, obtained by projection and communicating via message passing. It was only recently shown, that this problem, known as *realizability*, is decidable if asynchronous communication (communication via FIFO buffers) is considered. It can be verified using the *synchronizability* property. If the system is not synchronizable, the system is not realizable either. In this paper, we propose a new, automatic approach, which enforces both synchronizability and realizability by generating *local* monitors through successive equivalence checks and refinement.

1 Introduction

Many software systems are now built using independently developed services, which are mostly geographically and organizationally distributed. The specification and analysis of interactions among such distributed systems is a major concern for ensuring their correctness and reliability. In order to simplify the construction of these systems, their design often relies on a *contract*, which describes from a global point of view the admissible interaction sequences exchanged between the participants. In the area of Service Oriented Computing (SOC), this contract is called *choreography* and the participants are called *peers*. The peers correspond to a distributed implementation of this choreography, and can be derived by *projection*, *i.e.*, by projecting the choreography specification to each peer by ignoring the messages that are not sent or received by that peer. A crucial question in this context is to check whether the peers behave exactly as required in the choreography. This property is called *realizability* [10,1] and particularly matters when the system is developed following a top-down development process.

Figure 1 presents a simple example of choreography involving three peers (identified using 1, 2, and 3), which exchange three messages in sequence (a between 1 and 2, b between 2 and 3, and c between 1 and 2) and loops. On the right

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 238–253, 2012.

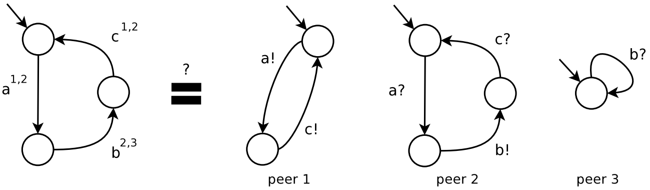


Fig. 1. Choreography, Peers, Realizability

hand side of Figure 1, we give the projection obtained from this choreography, where question marks correspond to receptions and exclamation marks to emissions. Realizability aims at checking whether the distributed implementation respects the ordering constraints specified in the global choreography.

Most distributed systems interact asynchronously where messages are sent and received through unbounded FIFO buffers. In this context, checking the realizability is a very difficult issue, because the distributed version of the system can generate infinite state spaces. This is the case of the distributed system given in Figure 1 for instance where peer 1 can infinitely send messages. Whether realizability is decidable was an open problem for several years. However, it was recently shown that it is decidable, verifying the *synchronizability* property [3]. A set of peers is synchronizable if and only if the system behavior, considering the send messages, preserves the same message sequences under synchronous and 1-bounded asynchronous communication. If a set of peers is synchronizable, one can check if it conforms to a choreography specification. If the system is not synchronizable then it is also not realizable. Both synchronizability and realizability checking involves finite state spaces and can be verified using equivalence checking techniques. The system described in Figure 1 is not synchronizable for example, because peer 1 can send a and c in sequence in the asynchronous system, whereas b occurs before c in the synchronous system as specified in the choreography.

Although this result is a significant step forward for formally analysing choreographies, there are still open issues that deserve to be studied. One of them arises when the realizability check returns *false*, due to one (or several) message exchange(s) violating the choreography ordering constraints. In this situation, there is no established solution for enforcing realizability and the designer is supposed to *patch* the choreography manually. However, correcting ordering issues may be a real burden for a designer, who just wants that the distributed implementation of his/her system behaves as specified in the choreography. This means that we need a way to control the distributed system to make it respect the global requirements. It is worth observing that, in this paper, when we refer to a *problem* in the choreography, this will always be an issue in the order of messages. Finding bugs (other than ordering issues) in choreographies can be achieved using existing verification tools.

In this paper, we propose a new approach, which identifies all problems which prevent synchronizability and realizability of a choreography, and provides a possible solution to enforce them. To do so, we generate *monitors*, which act as local controllers interacting with their peer and the rest of the system in order to make the peers respect the choreography requirements. These monitors are obtained by first generating the set of distributed peers by projection from the choreography specification. Then, we check in sequence the system synchronizability and realizability using equivalence checking. If one of these properties is violated, we exploit the generated counterexample to augment the monitors with a new synchronization message. Monitors are obtained through an iterative process, automatically refining their behaviors. The successive addition of these messages will finally enforce both synchronizability and realizability.

Our approach can be automated using any existing verification toolbox handling Labeled Transition Systems and providing an equivalence checker. We chose to encode choreographies into the value-passing process algebra LNT [6], one of the input languages of the CADP verification toolbox [12]. By doing so, we reuse existing state space exploration tools for generating peers and distributed systems, and equivalence checking techniques for verifying synchronizability and realizability. The process is fully supported (no human intervention) by calling various tools, some we reused from CADP, others we implemented ourselves, *e.g.*, for automating the iterative part of the process. We have validated our approach on hundreds of examples, some of them borrowed from real-world scenarios found in the literature.

Our monitor synthesis solution presents several advantages compared to existing results. Our approach goes beyond realizability checking by enforcing the system to respect the choreography. It is non-intrusive (peers are not modified or extended) and preserves the system parallelism by generating distributed monitors. It finds *all* problems in the choreography which prevent its realizability and suggests a distributed, implementable way to fix it. This is helpful in Service Oriented Computing or Component Based Software Engineering where black-box components are assumed. In the Web service domain, BPEL wrappers [2] can be automatically generated from our monitor models for controlling the distributed peers. In domains where the direct usage of the monitors is not an acceptable solution, the generated synchronization messages can serve to augment the choreography and provide a suggestion of how to fix it manually.

2 Background

We use conversation protocols [10] as choreography specification language in this paper. A conversation protocol is a low-level formal model, which can be computed from other existing specification formalisms such as collaboration diagrams [4], BPMN 2.0 choreographies [19], Singularity channels [22], or Message Sequence Charts (MSC) [1].

A conversation protocol is a Labeled Transition System (LTS) specifying the desired set of interactions from a global point of view. Each transition specifies an interaction between two peers $\mathcal{P}_{sender}$, $\mathcal{P}_{receiver}$ on a specific message m. A conversation protocol makes explicit the execution order of interactions. Sequence, choice, and loops are modeled using a sequence of transitions, several transitions going out from the same state and a cycle in the LTS, respectively.

Definition 1 (Conversation protocol). *A conversation protocol CP for a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is an LTS $CP = (S_{CP}, s^0_{CP}, L_{CP}, T_{CP})$ where S_{CP} is a finite set of states and $s^0_{CP} \in S_{CP}$ is the initial state; L_{CP} is a set of labels where a label $l \in L_{CP}$ is a tuple $m^{\mathcal{P}_i,\mathcal{P}_j}$ such that $\mathcal{P}_i$ and $\mathcal{P}_j$ are the sending and receiving peers, respectively, $\mathcal{P}_i \neq \mathcal{P}_j$, and m is a message on which those peers interact; finally, $T_{CP} \subseteq S_{CP} \times L_{CP} \times S_{CP}$ is the transition relation. We require that each message has a unique sender and receiver: $\forall m^{\mathcal{P}_i,\mathcal{P}_j}, m'^{\mathcal{P}'_i,\mathcal{P}'_j} \in L_{CP}$: $m = m' \implies \mathcal{P}_i = \mathcal{P}'_i \wedge \mathcal{P}_j = \mathcal{P}'_j$.*

In the remainder of this paper, we denote a transition $t \in T_{CP}$ as $s \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'$ where s and s' are source and target states and $m^{\mathcal{P}_i,\mathcal{P}_j}$ is the transition label.

We use LTSs for specifying the peer interaction model. This behavioral model defines the order in which the peer messages are executed. A label is a tuple (m, d) where m is the message name and d stands for the communication direction (either an emission ! or a reception ?). The set of messages in one peer LTS constitutes the peer *alphabet*.

Definition 2 (Peer). *A peer is an LTS $\mathcal{P} = (S, s^0, \Sigma, T)$ where S is a finite set of states, $s^0 \in S$ is the initial state, $\Sigma = \Sigma^! \cup \Sigma^?$ is a finite alphabet partitioned into a set of send and receive messages, and $T \subseteq S \times \Sigma \times S$ is a transition relation. We write $m!$ for a message $m \in \Sigma^!$ and $m?$ for $m \in \Sigma^?$.*

Peers are obtained by projection from a conversation protocol. After the projection they are determinized and minimized using standard algorithms [14], which is possible as the number of states and messages is finite.

Definition 3 (Projection). *Peer LTSs $\mathcal{P}_i = (S_i, s^0_i, \Sigma_i, T_i)$ are obtained by replacing in $CP = (S_{CP}, s^0_{CP}, L_{CP}, T_{CP})$ each label $m^{\mathcal{P}_j,\mathcal{P}_k} \in L_{CP}$ with $m!$ if $j = i$, with $m?$ if $k = i$, and with τ (internal action) otherwise; and finally removing the τ-transitions by applying standard minimization algorithms [14].*

The synchronous composite system corresponds to the distributed system computed over a set of peers communicating synchronously. In this context, a communication between two peers holds if and only if both agree on a synchronization label, *i.e.*, if one peer is in a state in which a message can be sent, then the other peer must be in a state in which that message can be received.

Definition 4 (Synchronous System). *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ with $\mathcal{P}_i = (S_i, s^0_i, \Sigma_i, T_i)$, the synchronous system $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n)$ is the LTS (S, s^0, Σ, T) where:*

- *$S = S_1 \times \ldots \times S_n$*
- *$s^0 \in S$ such that $s^0 = (s^0_1, \ldots, s^0_n)$*
- *$\Sigma = \cup_i \Sigma_i$*
- *$T \subseteq S \times \Sigma \times S$, and for $s = (s_1, \ldots, s_n) \in S$ and $s' = (s'_1, \ldots, s'_n) \in S$*

(interact) *$s \xrightarrow{m} s' \in T$ if $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma^!_i \cap \Sigma^?_j$ where $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$, and $s_j \xrightarrow{m?} s'_j \in T_j$ such that $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$*

In the asynchronous composite system, the peers communicate with each other asynchronously through FIFO buffers, *i.e.*, each peer $\mathcal{P}_i$ is equipped with a k-bounded message buffer Q_i^k. If k is not made explicit, noted Q_i, it means that $k = \infty$ and stands for unbounded buffers. A peer can either send a message $m \in \Sigma^!$ to the tail of the receiver buffer Q_j at any state where this send message is available, or read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head.

Definition 5 (Asynchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$, *and* Q_i *being its associated buffer, the asynchronous system* $((\mathcal{P}_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q_n))$ *is the LTS* (S, s^0, Σ, T) *defined as follows:*

- $S \subseteq S_1 \times Q_1 \times \ldots \times S_n \times Q_n$ *where* $\forall i \in \{1, \ldots, n\}$, $Q_i \subseteq (\Sigma_i^?)*$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \emptyset, \ldots, s_n^0, \emptyset)$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$,
  and for $s = (s_1, Q_1, \ldots, s_n, Q_n) \in S$ *and* $s' = (s_1', Q_1', \ldots s_n', Q_n') \in S$

(send) $s \xrightarrow{m!} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \Sigma_j^?$, *(i)* $s_i \xrightarrow{m!} s_i' \in T_i$, *(ii)* $Q_j' = Q_j m$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q_k' = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s_k' = s_k$

(read) $s \xrightarrow{m?} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$, *(i)* $s_i \xrightarrow{m?} s_i' \in T_i$, *(ii)* $mQ_i' = Q_i$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q_k' = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s_k' = s_k$

A system is synchronizable [11,3] when its behavior remains the same under both synchronous and asynchronous communication semantics. This is checked by bounding buffers to $k = 1$ and comparing interactions in the synchronous system with send messages in the asynchronous system.

Definition 6 (Synchronizability). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, *the synchronous system* $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n) = (S_s, s_s^0, L_s, T_s)$, *and the 1-bounded asynchronous system* $((\mathcal{P}_1, Q_1^1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q_n^1)) = (S_a, s_a^0, L_a, T_a)$, *two states* $r \in S_s$ *and* $s \in S_a$ *are synchronizable if there exists a relation* R *such that* $R(r, s)$ *and:*

- *for each* $r \xrightarrow{m} r' \in T_s$, *there exists* $s \xrightarrow{m!} s' \in T_a$, *such that* $R(r', s')$;
- *for each* $s \xrightarrow{m!} s' \in T_a$, *there exists* $r \xrightarrow{m} r' \in T_s$, *such that* $R(r', s')$;
- *for each* $s \xrightarrow{m?} s' \in T_a$, $R(r, s')$.

The set of peers is synchronizable if $R(s_s^0, s_a^0)$.

The approach presented in [3] proposes a sufficient and necessary condition showing that the realizability of conversation protocols is decidable.

Definition 7 (Realizability). *A conversation protocol CP is realizable if and only if (i) the peers computed by projection from this protocol are synchronizable, (ii) the 1-bounded system resulting from the peer composition is well-formed, and (iii) the synchronous version of the distributed system* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *is equivalent to CP.*

Well-formedness states that whenever the i-th peer buffer Q_i is non-empty, the system can eventually move to a state where Q_i is empty. For every synchronizable set of peers, if the peers are deterministic, *i.e.*, for every state, the possible send messages are unique, well-formedness is implied.

Both synchronizability and realizability properties are checked automatically using equivalence checking (weak trace equivalence in [3,17]). This check requires the modification of the asynchronous system for hiding receptions ($m? \rightsquigarrow \tau$), renaming emissions into interactions ($m! \rightsquigarrow m$), and removing τ-transitions using standard minimization techniques.

Running Example. For illustration purposes we specify the use of an application in the cloud. This system involves four peers: a client (cl), a Web interface (int), a software application (appli), and a database (db). We show first a conversation protocol (Figure 2) describing the requirements that the designer expects from the composition-to-be. The conversation protocol starts with a login interaction (connect) between the client and the interface, followed by the setup of the application triggered by the interface (setup). Then, the client can access and use the application as far as necessary (access). Finally, the client decides to logout from the interface (logout) and the application stores some information (start/end time, used resources, etc.) into a database (log).

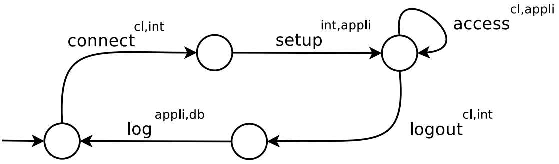


Fig. 2. Running Example: Choreography Specification

Figure 3 shows the four peers obtained by projection. This set of peers seems to respect the behavior specified in the conversation protocol, yet this is difficult to be sure using only visual analysis, even for such a simple example. In addition, as the choreography involves looping behavior, it is hard to know whether the resulting distributed system is bounded and finite, which would allow its formal analysis using existing verification techniques. Actually, this set of peers is not synchronizable (and therefore not realizable), because the trace of send messages "connect, access" is present in the 1-bounded asynchronous system, but is not present in the synchronous system. Synchronous communication enforces the sequence "connect, setup, access" as specified in the choreography, whereas in the asynchronous system peer cl can send connect! and access! in sequence.

In the rest of this paper, we propose an automated technique to identify all problematic messages in a choreography. Our approach augments the system with new participants and interactions in order to restore the correct message sequences as specified in the global contract.

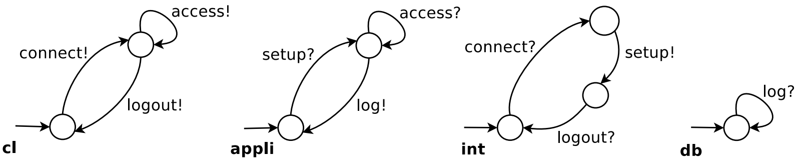


Fig. 3. Peer Projection

3 Counterexample Guided Realizability Enforcement

In our approach, we augment each peer by an accompanying monitor, which observes the behavior of the peer, and if necessary, controls the send messages according to the temporal ordering of the global specification. Adding monitors guarantees that the local behavior of the peers is not changed at all. The monitors locally receive the messages sent by their peer and relay them later after synchronization with the other monitors. They are refined by an iterative process, when it terminates the choreography is realized by the set of peers and monitors.

3.1 Monitors

A monitor interacts with its corresponding peer, with the other monitors and with receiving peers (via their buffers in the asynchronous system). The interaction with other monitors is done via synchronization messages, either incoming synchronizations of the form $m^{\leftarrow}$ for the synchronized monitor or outgoing synchronizations of the form $m^{\rightarrow}$, initiated by the synchronizing monitor. We call a message synchronized if there exists a synchronization message which delays it.

The monitor interacts with its corresponding peer over the send messages. The monitor locally receives the message from the peer. If the message needs to be synchronized, it first waits for the incoming synchronization message and then relays the message to the receiver, otherwise it relays the message directly to its receiver. If required, it will emit an outgoing synchronization message afterwards.

Definition 8 (Monitor). *A monitor is an LTS $M = (\overline{S}, \overline{s^0}, \overline{\Sigma}, \overline{T})$ where $\overline{S}$ is a finite set of states, $\overline{s^0}$ is the initial state, $\overline{\Sigma} = \overline{\Sigma^!} \cup \overline{\Sigma^?} \cup \overline{\Sigma^{\leftarrow}} \cup \overline{\Sigma^{\rightarrow}}$is a finite alphabet partitioned into sets of sending, locally receiving, incoming and outgoing synchronization messages and $\overline{T} \subseteq \overline{S} \times \overline{\Sigma} \times \overline{S}$ is a transition relation.*

The synchronous parallel composition of the peers and their monitors describes the system where all participants interact using synchronous communication.

Definition 9 (Monitored Synchronous System). *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ with $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ and a set of monitors $\{M_1, \ldots, M_n\}$ with $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma_i}, \overline{T_i})$, the monitored synchronous system $((\mathcal{P}_1, M_1) \mid \ldots \mid (\mathcal{P}_n, M_n))$ is the LTS $\mathcal{SS}' = (S, s^0, \Sigma, T)$ where:*

- $S = S_1 \times \overline{S_1} \times \ldots \times S_n \times \overline{S_n}$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \overline{s_1^0} \ldots, s_n^0, \overline{s_n^0})$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$, *for* $s = (s_1, \overline{s_1} \ldots, s_n, \overline{s_n}) \in S$ *and* $s' = (s'_1, \overline{s'_n} \ldots, s'_n, \overline{s'_n}) \in S$

(send) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \overline{\Sigma_i^?}$ *where* $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$, *and* $\overline{s_i} \xrightarrow{m?} \overline{s'_i} \in \overline{T_i}$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k \wedge \overline{s'_k} = \overline{s_k}$

(interact) $s \xrightarrow{m} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^!} \cap \Sigma_j^?$ *where* $\exists\ \overline{s_i} \xrightarrow{m!} \overline{s'_i} \in \overline{T_i}$, *and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\} : (k \neq j \Rightarrow s'_k = s_k) \wedge (k \neq i \Rightarrow \overline{s'_k} = \overline{s_k})$

(sync) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^{\rightarrow}} \cap \overline{\Sigma_j^{\leftarrow}}$ *where* $\overline{s_i} \xrightarrow{m^{\rightarrow}} \overline{s'_i} \in \overline{T_i}$ *and* $\overline{s_j} \xrightarrow{m^{\leftarrow}} \overline{s'_j} \in \overline{T_j}$ *and* $\forall k \in \{1, \ldots, n\} : s'_k = s_k \wedge (k \neq i \wedge k \neq j \Rightarrow \overline{s'_k} = \overline{s_k})$

and finally removing the τ*-transitions.*

In the monitored asynchronous system, each pair $(\mathcal{P}_i, Q_i)$ is composed with the LTS of its monitor M_i. The asynchronous behavior of the peers and monitors corresponds to the distributed system where the sending peers communicate with their monitors, which relay the messages to the buffers of the receiving peers. This is shown in Figure 4 for two peers. The remote interactions between the monitors, local interactions between peers and their buffers or between peers and their monitors are marked with dashed lines. They are not observable from an external point of view. The visible interactions are the messages sent from one peer to the other. These are relayed by the monitor of the sending peer and are stored in the buffer of the receiving peer.

Definition 10 (Monitored Asynchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$, Q_i *its associated buffer and a set of corresponding monitors* $\{M_1, \ldots, M_n\}$ *with* $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma}_i, \overline{T_i})$, *the asynchronous system* $((\mathcal{P}_1, M_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, M_n, Q_n))$ *is the LTS* $\mathcal{AS}' = (S, s^0, \Sigma, T)$ *where:*

- $S \subseteq S_1 \times \overline{S_1} \times Q_1 \times \ldots \times S_n \times \overline{S_n} \times Q_n$ *where* $\forall i \in \{1, \ldots, n\}$, $Q_i \subseteq (\Sigma_i^?)*$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \overline{s_1^0}, \emptyset, \ldots, s_n^0, \overline{s_n^0}, \emptyset)$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$, *and for*
  $s = (s_1, \overline{s_1}, Q_1, \ldots, s_n, \overline{s_n}, Q_n) \in S$ *and* $s' = (s'_1, \overline{s'_1}, Q'_1, \ldots s'_n, \overline{s'_n}, Q'_n) \in S$

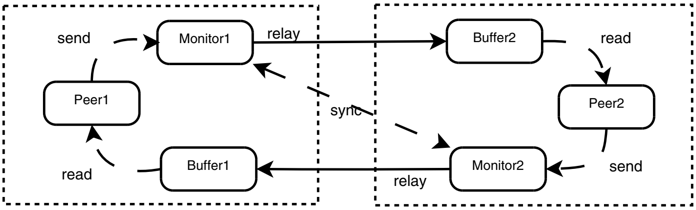


Fig. 4. Interactions in the Monitored Asynchronous System

(send) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \overline{\Sigma_i^?}$, *(i)* $s_i \xrightarrow{m!} s'_i \in T_i$ *and* $\overline{s_i} \xrightarrow{m?} \overline{s'_i} \in \overline{T_i}$, *(ii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *and (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$ *and* $\overline{s'_k} = \overline{s_k}$

(relay) $s \xrightarrow{m!} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma_j^? \cap \overline{\Sigma_i^!}$, *(i)* $\overline{s_i} \xrightarrow{m!} \overline{s'_i} \in \overline{T_i}$, *(ii)* $Q'_j = Q_j m$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q'_k = Q_k$, *(iv)* $\forall k \in \{1, \ldots, n\} : s'_k = s_k$, *and (v)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow \overline{s'_k} = \overline{s_k}$

(read) $s \xrightarrow{m?} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$, *(i)* $s_i \xrightarrow{m?} s'_i \in T_i$, *(ii)* $mQ'_i = Q_i$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$, *(iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$, *and (v)* $\forall k \in \{1, \ldots, n\} : \overline{s'_k} = \overline{s_k}$

(sync) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^{\rightarrow}} \cap \overline{\Sigma_j^{\leftarrow}}$, *(i)* $\overline{s_i} \xrightarrow{m^{\rightarrow}} \overline{s'_i} \in \overline{T_i}$ *and* $\overline{s_j} \xrightarrow{m^{\leftarrow}} \overline{s'_j} \in \overline{T_j}$, *(ii)* $\forall k \in \{1, \ldots, n\} : s'_k = s_k$, *(iii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i, j \Rightarrow \overline{s'_k} = \overline{s_k}$

and finally removing the τ*-transitions.*

Using Def. 9 and 10, synchronizability and realizability are checked as follows: For synchronizability, we check the equivalence between the monitored synchronous and monitored asynchronous system with 1-bounded buffers. For realizability we check the equivalence between the monitored synchronous system and the choreography.

3.2 Iterative Construction of the Monitors

We use an iterative approach to identify all the problematic messages in a choreography. At each iteration an equivalence check is conducted. If the check fails, its result is analyzed to decide which synchronization message must be added to the choreography. This results in the extended conversation protocol (ECP).

Definition 11 (Extended Conversation Protocol). *An extended conversation protocol ECP for a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and corresponding set of monitors* $\{M_1, \ldots, M_n\}$ *is an LTS* $(S_{ECP}, s^0_{ECP}, L_{ECP} \cup L^+_{ECP}, T_{ECP})$ *where* S_{ECP}, s^0_{ECP}, L_{ECP} *are defined analogous to Def. 1; a synchronization label* $l \in L^+_{ECP}$ *is a tuple* sync^{M_j, M_k} *where* M_j *and* M_k *are the synchronizing and synchronized monitor* $(j \neq k)$*; finally,* $T_{ECP} \subseteq S_{ECP} \times (L_{ECP} \cup L^+_{ECP}) \times S_{ECP}$ *is the transition relation.*

The extended conversation protocol is augmented iteratively with synchronization messages until the choreography becomes realizable. This works for all non-faulty choreographies. Those which involve divergent choices are considered as faulty [22]. Realizability cannot be enforced in that case, as it is impossible to control divergent choices in a distributed system without changing the local behavior of the peers. Faulty choreographies are identified beforehand by detecting non-confluent diamonds of interactions in the conversation protocol using the executable temporal logic (XTL) [13].

The complete approach to enforce realizability of a choreography is shown as activity diagram in Figure 5. In a first step, we discard faulty choreographies.

Then, we project the peers and start with the synchronizability check. At each iteration, the equivalence between the monitored synchronous and the 1-bounded monitored asynchronous system is checked. If this check fails, we analyze the counterexample, identify the problematic message, and augment the ECP with the necessary synchronization message. The synchronizability loop of the activity is executed as long as the system is not synchronizable. When the choreography is finally synchronizable, we proceed with the realizability check of the activity. Here, we check the equivalence between the monitored synchronous system and the original CP. The analysis of the counterexamples and the introduction of the synchronization messages is done as before, and we continue the activity until the realizability check succeeds.

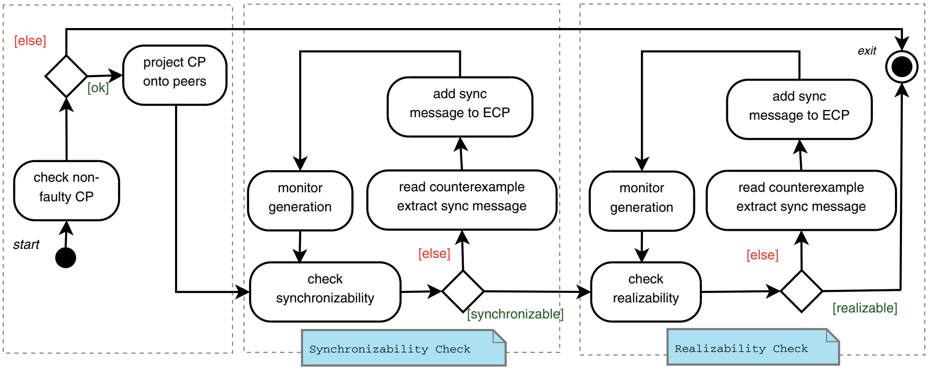


Fig. 5. Approach Overview

Now, we explain how we augment the ECP with synchronization messages for the monitors. If the equivalence check does not succeed in iteration k, a counterexample is returned. This is a finite trace, whose prefix is contained in both systems, but the sending of the last message m' is only possible in one of them. Therefore the sending of this message m' must be controlled by a monitor, in order to adhere to the specification. To do so, we introduce synchronization messages into ECP_k as follows:

1. Locate in ECP_k the message m', its sending peer $\mathcal{P}_i$ and the states $s^*, s^{*'}$ for which there exists $s^* \xrightarrow{m'^{\mathcal{P}_i,\mathcal{P}}} s^{*'} \in T_{\mathrm{ECP}_k}$
2. Add a new state s_{new} to the set of states $S_{\mathrm{ECP}_{k+1}}$
3. Replace each $s^* \xrightarrow{m^{P_i,P_l}} s^{*'} \in T_{\mathrm{ECP}_k}$ with $s_{new} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_l}} s^{*'}$ (with $s_{new} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_l}} s_{new}$ if $s^* = s^{*'}$) in $T_{\mathrm{ECP}_{k+1}}$
4. For every incoming transition to s^*, $s \xrightarrow{m^{\mathcal{P}_j,\mathcal{P}_{j'}}} s^* \in T_{\mathrm{ECP}_k}$, add a new transition $s^* \xrightarrow{\mathrm{sync}_{m'}^{\mathcal{P}_j,\mathcal{P}_i}} s_{new}$ to $T_{\mathrm{ECP}_{k+1}}$ (for M_j to M_i, where $\mathrm{sync}_{m'}$ is a new name) and add the synchronization message $\mathrm{sync}_{m'}^{\mathcal{P}_j,\mathcal{P}_i}$ to $L^+_{\mathrm{ECP}_{k+1}}$

After each iteration, we derive the monitors from the extended conversation protocol. This can be achieved by using a process similar to the peer projection.

Definition 12 (Monitor Projection). *Monitor LTSs* $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma}_i, \overline{T}_i)$ *are obtained by replacing in* $ECP = (S_{ECP}, s^0_{ECP}, L_{ECP} \cup L^+_{ECP}, T_{ECP})$ *each transition* $s \xrightarrow{m^{P_j,P_k}} s'$ *(i) with a sequence of transitions* $s \xrightarrow{m?} s^*$, $s^* \xrightarrow{m!} s'$ *if* $m \notin L^+_{ECP}$ *and* $P_j = P_i$, *(ii) with a sequence of transitions* $s \xrightarrow{m'?} s^*$, $s^* \xrightarrow{sync^{\leftarrow}_{m'}} s^{*'}$, $s^{*'} \xrightarrow{m'!} s'$ *if* $m = sync_{m'} \in L^+_{ECP}$ *and* $P_k = P_i$, *(iii) with* $s \xrightarrow{sync^{\rightarrow}_{m'}} s'$ *if* $m = sync_{m'} \in L^+_{ECP}$ *and* $P_j = P_i$, *and (iv) with* τ *otherwise; adding the new states* $s^*, s^{*'}$ *to* $\overline{S_i}$, *and finally removing the* τ*-transitions.*

Note that this projection does result in a correct monitor, but not necessarily in the most permissive one. Due to the lack of space, we do not give its formal definition here. Intuitively, we use an additional state machine composed with the monitor. This creates all possible interleavings of the monitor behavior and of the outgoing synchronization messages.

The iterative extension of the conversation protocol is guaranteed to terminate after a finite number of steps and to result in a realizable choreography. We must omit the proofs here, but the basic argument is as follows: the number of messages that may be synchronized is bounded and no message can be synchronized more than once; the equivalence checks assure that we find the right message to synchronize.

Complexity. In theory it can be necessary to synchronize every message $m \in L_{CP}$ of the conversation protocol. As the parallel composition and equivalence checks have a worst case complexity exponential in the number of peers $\#\mathcal{P}$, the worst case complexity of our approach is $O(|L_{CP}| \cdot |S_{CP}|^{\#\mathcal{P}})$. Nevertheless, our experience showed that this is unlikely in practical cases. Most often the number of additional synchronization messages is rather small and compositional verification techniques help to reduce the complexity of the parallel composition (for experimental details see Section 4).

Running Example. We illustrate the construction of the most permissive monitors for the example choreography shown in section 2, which is is not synchronizable. The message sequence "connect, access" is possible in the asynchronous system, but not in the synchronous one. The message access can only be sent from cl to appli after setup, therefore it must be deferred to be sent after that. To do so, we add a synchronization message for access to the choreography. This synchronization message is emitted by the monitor for int, who is the sender of the message setup. The left hand side of Figure 6 shows the extended conversation protocol with the first synchronization message.

The right hand side of Figure 6 shows the monitor for the peer int. In the initial state it accepts the message sent by its peer (setup?). It relays this message to its receiver (setup!) and sends an outgoing synchronization message ($\text{sync}^{\rightarrow}_{\text{access}}$) afterwards. What may seem counter-intuitive is the possibility of the message sequence "setup?, setup!, setup?, setup!" followed by two synchronization messages.

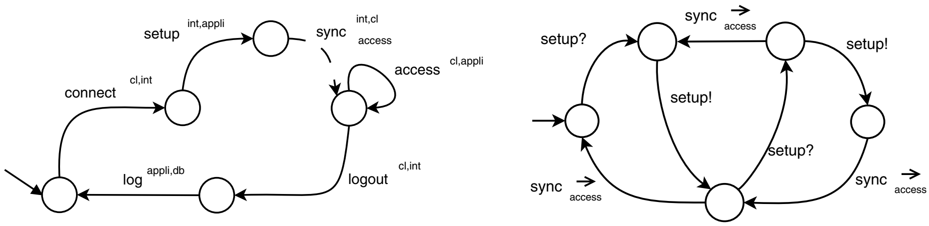


Fig. 6. After First Iteration

This is the result of constructing the most permissive monitor. As the peer is not blocked after it sends the first message, it may proceed to send it again. The monitor can relay both these messages. Nevertheless, after it relays the second one without an outgoing synchronization message, both must be synchronized, as synchronization messages are not buffered.

After the introduction of the first synchronization message, the choreography is synchronizable but not realizable. The equivalence check returns the counterexample "connect, setup, log", but logout must always precede log. A second synchronization message is therefore introduced right after the logout message. It is exchanged between the monitors for the peer cl (who sends logout) and for the peer appli. The left hand side of Figure 7 shows the monitor for appli after the second iteration. It accepts the local emission of the log message from its peer, waits for the incoming synchronization message, and then relays log.

Still, the choreography is not realizable in this form. The next counterexample is "connect, setup, logout, connect", *i.e.*, the peer cl starts a new connection attempt, before the log message is sent to db. A third synchronization message is introduced directly after log, between the monitor for appli and the monitor for cl. The right hand side of Figure 7 shows the monitor for appli after the third iteration. After the integration of the three synchronization messages, the choreography is finally both synchronizable and realizable.

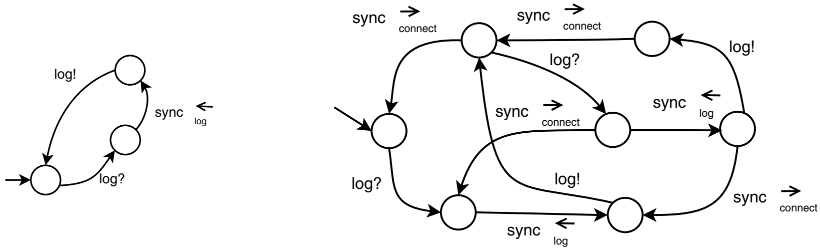


Fig. 7. Iterative Monitor Construction for Peer appli

4 Tool Support

Implementation. Our approach is tool-independent; every formal verification tool for equivalence checking of LTSs is usable. To automate the process, we chose the formal language LOTOS NT (LNT) [6]. It enables the description of concurrent processes, communicating via messages. It is fully integrated into CADP [12], which includes efficient methods for minimization under different equivalence relations, equivalence and model checking. The encoding into LNT also permits to analyze choreographies for bugs (other than message ordering issues) using CADP tools, *e.g.*, temporal properties expressed in MCL [16].

The conversation protocol, peers, monitors are encoded via the state machine pattern as LNT processes. We exploit the parallel composition operator of LNT to construct the most permissive monitors with all possible interleavings of the synchronization messages. The buffer behaviors are also encoded using LNT processes; the buffer operations are specified as LNT data types. The projection from the ECP onto the distributed peers is realized using label hiding and LTS reduction. The parallel composition of the FIFO buffers, peers and monitors, as well as of the monitored peers is done using the parallel composition and rendez-vous synchronization of LNT.

Experiments. We developed a test case generator, which we used to generate hundreds of conversation protocols with varying parameters, *e.g.*, number of peers, states and transitions. Our database of examples also includes 65 choreographies taken from the literature, as well as variants of them.

Table 1 shows the results for some of the experiments we conducted. For each example it shows the number of peers involved, the number of transitions and states in the choreography, and the number of additional synchronization messages. The fifth column shows the number of states and transitions of the largest intermediate LTS while creating the monitored asynchronous system. We use compositional verification, in particular smart parallel composition [8], where reductions are applied during the parallel composition and a composition sequence is decided heuristically. The final column shows the time for the longest iteration as well as the overall time for all computations and checks on a 3 Ghz Xeon CPU with 12 Gbyte RAM.

The number of peers has a significant influence on the state space of the intermediate LTSs, more so than the number of transitions, *e.g.*, see examples cp0031 and cp0032. The asynchronous behavior of many peers with only few messages generates many possible interleavings, while the behavior of few peers but more messages generally creates much less. This is the case, *e.g.*, in cp0153, which has a small number of peers, but a higher number of transitions, yet the intermediate state space of the LTS is rather small.

Table 1. Experimental Results

example	|peers|	$\|T\|/\|S\|$	|sync|	parallel	time
				composition	max / total
cp0121	3	12 / 8	0	355 / 931	- / 54s
cp0016	3	4 / 3	1	121 / 337	46s / 1m 31s
cp0063	4	5 / 4	3	337 / 988	58s / 3m 54s
cp0153	3	29 / 16	5	15,182 / 59,033	53s / 7m 03s
cp0031	7	11 / 11	6	158,741 / 853,559	5m 47s / 19m 31s
cp0032	9	11 / 12	5	105,598 / 856,617	25m 53s / 1h 25m 10s

5 Related Work

There exists much work on the verification of realizability, *e.g.*, [10,1,4,21,15,3], but none provides a solution if the choreography is not realizable. Let us focus on related approaches, which propose solutions for ensuring realizability of a choreography. In [5], the authors identify three principles for global descriptions under which they define a sound and complete end-point projection, *i.e.*, the generation of distributed processes from the choreography description. If these rules are respected, the distributed system obtained by projection will behave exactly as specified in the choreography. The same approach is chosen for BPMN 2.0 choreographies [18]. In [20], the authors propose to modify their choreography language to include new constructs (dominated choice and loop). During projection of these new operators, some communication is added to make the peers respect the choreography specification. However, these solutions prevent the designer from specifying what (s)he wants to, and complicates the design by obliging the designer to make explicit extra-constraints in the specification, *e.g.*, by associating *dominant roles* to certain peers. In [9], the authors propose a Petri Net-based formalism for choreographies and algorithms to check realizability and local enforceability. A choreography is locally enforceable if interacting peers are able to satisfy a subset of the requirements of the choreography. To ensure this, some message exchanges in the distributed system are disabled. In [21], the authors propose automated techniques to check the realizability of collaboration diagrams for different communication models. In case of non-realizability messages are added directly to the peers to enforce realizability. Collaboration diagrams are much less expressive than conversation protocols, as choices and loops cannot be specified, except for repetition of the same interaction.

Beyond advocating a solution for enforcing realizability, our contribution differs from these related works as follows. We focus on asynchronous communication and choreographies involving loops that may result in infinite state spaces. Our approach is non-intrusive; we do not add any constraints on the choreography language or specification, and the designer neither has to modify the original choreography specification, nor the peer models. Instead, we generate local monitors that preserve the system parallelism and control the peer behaviors to make them respect the choreography requirements.

The technique we rely on here shares some similarities with counterexample-guided abstraction refinement (CEGAR) [7]. In CEGAR, an abstract system is analyzed for temporal logic properties. If a property holds, the abstraction mechanism guarantees that the property also holds in the concrete design. If the property does not hold, the reason may be a too coarse approximation by the abstraction. In this case, the counterexample generated by the model checker, is used to refine the system to a finer abstraction and the process is iterated.

To the best of our knowledge, our approach is the first application of equivalence checking for a technique inspired from CEGAR. Moreover, our contribution goes beyond CEGAR related approaches, because we do not only automatically find problems in the model, but also offer a fix for (all of) them. Our approach allows to solve a problem, namely automatically fixing message ordering issues in a distributed system modeled using global contracts, for which no solution has been yet suggested.

6 Conclusion

In this paper, we have presented a new solution to identify all necessary changes to choreographies and synthesize distributed, local monitors which enforce realizability. Our approach is directly applicable to all notations which are transformable into conversation protocols. This is the case for most existing languages such as BPMN 2.0, collaboration diagrams, WS-CDL, Singularity channels, and MSC. We generate the monitors in successive iterations by checking both the synchronizability and realizability properties on the distributed system obtained by projection from the choreography specification. If one of these two properties is not satisfied, we use the counterexample resulting from this check to extend the monitors with additional synchronization messages. When both properties are finally ensured, we know that the system is bounded, synchronizable, and realizable. This assures the correct behavior of the distributed system according to the choreography, without making any change in the services themselves. Our main perspective aims at working with models closer to implementations that consider not only message passing communications, but also data exchanged between peers. This impacts choreography semantics and raises new issues such as dead code detection.

Acknowledgements. The authors would like to thank Samik Basu, Tevfik Bultan, Frédéric Lang, and Radu Mateescu for interesting discussions on the topics of this paper.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Realizability and Verification of MSC Graphs. Theoretical Computer Science 331(1), 97–114 (2005)
2. Andrews, T.: et al. Business Process Execution Language for Web Services (WS-BPEL). BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems (2005)
3. Basu, S., Bultan, T., Ouederni, M.: Deciding Choreography Realizability. In: Proc. of POPL 2012. ACM Press (2012)
4. Bultan, T., Fu, X.: Specification of Realizable Service Conversations using Collaboration Diagrams. Service Oriented Computing and Applications 2(1), 27–39 (2008)
5. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
6. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 149 pages (2011)
7. Clarke, E., 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, pp. 154–169. Springer, Heidelberg (2000)

8. Crouzen, P., Lang, F.: Smart Reduction. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 111–126. Springer, Heidelberg (2011)
9. Decker, G., Weske, M.: Local Enforceability in Interaction Petri Nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
10. Fu, X., Bultan, T., Su, J.: Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. Theoretical Computer Science 328(1-2), 19–37 (2004)
11. Fu, X., Bultan, T., Su, J.: Synchronizability of Conversations among Web Services. IEEE Transactions on Software Engineering 31(12), 1042–1055 (2005)
12. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
13. Garavel, H., Mateescu, R.: XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In: Proc. STTT 1998 (1998)
14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (1979)
15. Lohmann, N., Wolf, K.: Realizability Is Controllability. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 110–127. Springer, Heidelberg (2010)
16. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
17. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice-Hall (1989)
18. OMG. Business Process Model and Notation (BPMN) – Version 2.0 (2011)
19. Poizat, P., Salaün, G.: Checking the Realizability of BPMN 2.0 Choreographies. In: Proc. of SAC 2012. ACM Press (2012)
20. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the Theoretical Foundation of Choreography. In: Proc. of WWW 2007. ACM Press (2007)
21. Salaün, G., Bultan, T.: Realizability of Choreographies Using Process Algebra Encodings. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 167–182. Springer, Heidelberg (2009)
22. Stengel, Z., Bultan, T.: Analyzing Singularity Channel Contracts. In: Proc. of ISSTA 2009. ACM (2009)

Parallel Assertions for Architectures with Weak Memory Models

Daniel Schwartz-Narbonne[1], Georg Weissenbacher[1,2], and Sharad Malik[1]

[1] Princeton University
[2] Vienna University of Technology, Austria

Abstract. Assertions are a powerful and widely used debugging tool in sequential programs, but are ineffective at detecting concurrency bugs. We recently introduced parallel assertions which solve this problem by providing programmers with a simple and powerful tool to find bugs in parallel programs. However, while modern computer hardware implements weak memory models, the sequentially consistent semantics of parallel assertions prevents these assertions from detecting some feasible bugs. We present a formal semantics for parallel assertions that accounts for the effects of weak memory models. This new formal semantics allows us to prove the correctness of two key optimizations which significantly increase the speed of a runtime assertion checker on a set of PARSEC benchmarks. We discuss the probe effect caused by checking these assertions at runtime, and show how our new semantics allows the detection of bugs that were undetectable in the previous semantics.

1 Introduction

Assertions are a powerful and widely used debugging tool. They allow programmers to state their expectations about program executions, and provide a mechanism to indicate when these expectations are violated. However, while assertions have proven to be a valuable tool for debugging sequential programs, they have fundamental limitations that restrict their utility in parallel programs.

If a programmer asserts a property ϕ in a sequential program, he can safely assume that the code *after* the assertion executes in an environment in which ϕ is true. In a parallel program, however, other threads can interfere and invalidate the property ϕ. The programmer wants to state "while this code is executing, ϕ must hold", but only has a way to say "before this code executes, ϕ holds".

Parallel Assertions [13] provide an elegant solution to this problem. Instead of evaluating an assertion at a single point in time, a parallel assertion is associated with a syntactic scope, delineated with the keywords `thru` and `passert(`ϕ`)`. The assertion ϕ must hold at all times between the begin and end of the scope.

Atomicity violations and order violations comprise the majority of concurrency bugs [11]. These bugs have the same ultimate cause: a read or a write was issued by a thread at a time when it should not have been able to interfere. In order to allow the specification of such non-interference properties, we allow parallel assertions to refer to memory accesses by means of the operators LocalWrite

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 254–268, 2012.

LW(x), RemoteWrite RW(x), LocalRead LR(x), RemoteRead RR(x), which indicate whether a given variable is read or written by a local or remote thread. The history operator HasOccurred (HO) allows the programmer to make assertions about a (limited) history of the execution. The assertion if Fig. 1, for example, checks whether the variable x is initialised by the local thread before it is read by any remote thread. These features make parallel assertions highly expressive, allowing them to capture 14 out of 17 real world bugs from the University of Michigan bug bench [17].

```
thru{
     ... ; x = 1; ...
} passert(!RR(x)||HO(LW(x))
```

Fig. 1. A simple assertion

We recently presented a runtime checker for parallel assertions [12]. This work, however, is limited by the fact that the semantics of parallel assertions in [13] is based on a sequentially consistent (SC) memory model. Modern processor architectures do not satisfy this requirement, and enforcing SC on a weaker memory model using fences slows down the execution and effectively *masks bugs*.

Contributions: We present a formal operational semantics for parallel assertions that accounts for the effects of weak memory models (§2, §3). This model enables two key optimisations (event filtering and relaxed timing) which we present and prove correct in (§4). We discuss the impact of fences in our model in (§5). We implemented a run-time checker for weak memory systems, and show the significant (order 2×) speedups enabled by our optimisations (§6).

2 Observing Program Executions

The evaluation of an assertion is based on the observation of an execution of the program under test. An execution is characterised by a series of events (discussed in §2.1) and the order in which they are observed (§2.2).

2.1 Program Events

A parallel program comprises a set of threads, each of which generates a series of *observable* events. Events are generated by instructions executed by the processor, and each instruction may result in a number of events. The execution of the assignment x:=y+z, for instance, may give rise to two read and one write events. We intentionally base the specification of our assertion language on program events rather than on the instructions of the underlying programming language. The rationale for this decision is that the C++ standard [7] does not provide a semantics for programs with race conditions. In practice, though, a compiler would still generate assembly code (albeit with a compiler-specific behaviour) for such a program. It is exactly in such corner cases that parallel assertions enabling the programmer to debug the flawed program are particularly valuable.

We use $\mathbb{E}$ to denote the set of all observable program events. Formally, an event is a tuple comprising a unique identifier *uid*, the thread identifier *tid* of the thread that generated the event, the *type* of the event, and any data associated with that particular type of event. We distinguish the following types of events:

Memory events $\mathbb{M}$ are physical memory accesses, such as reads and writes. A memory event is a tuple $\langle uid, tid, type, \ell, v\rangle$, where $uid, tid \in \mathbb{N}$ are unique identifiers, $type \in \{\text{READ}, \text{WRITE}\}$ represents the direction of the access, ℓ is the memory location accessed, and $v \in \mathbb{V}$ (where $\mathbb{V}$ is a set of values) is the value read or written. We use $W_{tid}\,\ell\, v$ to denote the write access $\langle uid, tid, \text{WRITE}, \ell, v\rangle$ and $R_{tid}\,\ell$ to denote a read access $\langle uid, tid, \text{READ}, \ell, v\rangle$ when uid (and v, respectively) is not relevant in the given context.

Fence (barrier) events $\mathbb{F}$ affect the legal orderings in an execution by enforcing ordering constraints on memory operations issued before and after the fence instruction (c.f. §2.2). A fence event is a tuple comprising a unique identifier *uid*, a thread identifier *tid*, an architecture dependent fence-type *type*, and an optional (architecture-dependent) set of ordering constraints (c.f. §5).

Scope events $\mathbb{S}$ are generated upon entry or exit of a syntactic assertion scope. A scope entry event is a tuple $\langle uid, tid, \text{ENTRY}, \phi_{uid}\rangle$, where ϕ_{uid} is an assertion (defined in §3.1). A scope exit event is a tuple $\langle uid, tid, \text{EXIT}, uid_{\text{ENTRY}}\rangle$, where uid_{ENTRY} represents the unique entry event corresponding to the scope exit.

We use `skip` to denote events that are irrelevant for assertion evaluation.

2.2 Observation Order for Threads

An observation of a program execution is a sequence of program events. From the viewpoint of a thread P_n, each event occurs at a particular point in time (which determines its location in the sequence). In general, there is no global notion of time, and therefore two threads may disagree on the order in which they observe events. Note that we do not distinguish between "thread-local" and "global" events — conceptually, a thread observes *all* events in some order, though in practice it is typically infeasible (and unnecessary) to record all observations.

The following definition (borrowed from [6] and consistent with [1]) determines what constitutes an observation of a memory event — from the point of view of a given thread — in terms of the *local* time of that thread.

Definition 1 (Observability). *A read or write event is observed when the respective memory access takes effect from the point of view of the observer:*

- *A write to a location in memory is said to be observed by an observer P_n when a subsequent read of the location by P_n would return the value written by the write.*
- *A read of a location in memory is said to be observed by an observer P_n when a subsequent write to the location by P_n would have no effect on the value returned by the read.*[1]

Fence events have no side effect on the execution other than the constraints they impose on the ordering of events (c.f. §3 and §5). Accordingly, *when* a fence event

[1] This definition is *not* cyclic, since a read observation is defined in terms of the potential *effect* of a write rather than in terms of the observation of a write.

is observed is architecture-dependent and determined by the respective ordering constraints. Therefore, we do not provide a general definition.

Scope events are only visible to the thread which generates them (see §3). Since they are side-effect free, we assume they are observed by P_n at the point in time when they are generated.

Observations induce a per-thread total order of events, reflecting the order in which they became visible to a particular thread P_n. We represent this order using an irreflexive transitive relation $\xrightarrow{\mathrm{obs}_n} : \mathbb{E} \times \mathbb{E}$. For every pair of events e_1, e_2 and thread P_n, there is a thread-local observation edge $e_1 \xrightarrow{\mathrm{obs}_n} e_2$ iff e_1 is before e_2 in this total order, i.e., the thread observes e_1 before e_2. We use $e_1 \overset{\mathrm{obs}_n}{\not\longrightarrow} e_2$ to abbreviate $\neg(e_1 \xrightarrow{\mathrm{obs}_n} e_2)$ (which implies $e_2 \xrightarrow{\mathrm{obs}_n} e_1$ for $e_1 \neq e_2$).

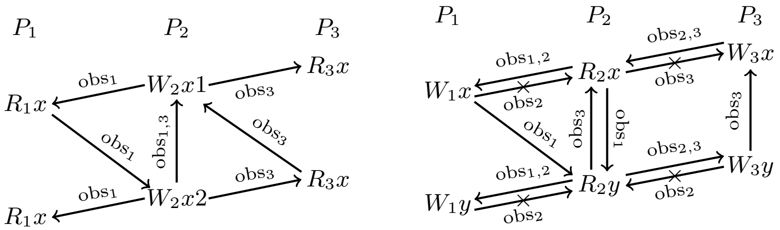
Note that since $\xrightarrow{\mathrm{obs}_n}$ is irreflexive (i.e., $e_i \overset{\mathrm{obs}_n}{\not\longrightarrow} e_i$) and transitive, cycles are not allowed. Accordingly, executions characterised by a cyclic ordering relation are *infeasible*. While the definition of $\xrightarrow{\mathrm{obs}_n}$ does not impose any further restriction on executions, the program structure imposes certain restrictions as to the order in which events are generated; the thread executing `x:=y+z`, for example, has to perform reads from `y` and `z` before it can issue a write to `x`. We refer to these (thread-local) constraints as the *program order* $\xrightarrow{\mathrm{po}_n} : \mathbb{E} \times \mathbb{E}$. The program order of a thread P_n enforces that certain events are observed in a specific order, i.e., $e_1 \xrightarrow{\mathrm{po}_n} e_2 \Rightarrow e_1 \xrightarrow{\mathrm{obs}_n} e_2$. The relation $\xrightarrow{\mathrm{po}_n}$ corresponds to $\xrightarrow{\mathrm{po}}$ in [2] and to the *sequenced-before* relation of [4] and is determined by the semantics of the language. The diagram to the right, for instance, shows the program order derived from the code fragment `x=1; y=x+x` in the C++ language [7].

$$\begin{tikzcd}
& W_1\,x\,1 \arrow[dl, "\mathrm{po}_1"'] \arrow[dr, "\mathrm{po}_1"] & \\
R_1\,x \arrow[dr, "\mathrm{po}_1"'] & & R_1\,x \arrow[dl, "\mathrm{po}_1"] \\
& W_1\,y\,2 &
\end{tikzcd}$$

In addition to the restrictions imposed by the semantics of the programming language, we require that events are not reordered across assertion scope boundaries. For events e_{bef} and e_{aft} generated by instructions before and after the beginning of a scope, respectively, and the corresponding scope entry event e_{entry}, we impose $e_{\mathrm{bef}} \xrightarrow{\mathrm{po}_n} e_{\mathrm{entry}} \xrightarrow{\mathrm{po}_n} e_{\mathrm{aft}}$ (and similarly for the end of the scope). We emphasise that the relation $\xrightarrow{\mathrm{obs}_n}$ does not impose ordering constraints on other threads, i.e., $\xrightarrow{\mathrm{obs}_n}$ is *not* global in the sense that $\left(\exists n \,.\, e_i \xrightarrow{\mathrm{obs}_n} e_j\right)$ does not imply $\left(\forall n \,.\, e_i \xrightarrow{\mathrm{obs}_n} e_j\right)$.

The underlying memory model, however, may impose ordering constraints across threads. A common assumption is that the memory model guarantees memory coherence in that for each location, there is a global total order over the writes to that location (c.f. [2]):

$$\begin{aligned}
&\left(\exists n \,.\, \langle \mathit{uid}_1, \mathit{tid}_1, \mathrm{WRITE}, \ell, v_1\rangle \xrightarrow{\mathrm{obs}_n} \langle \mathit{uid}_2, \mathit{tid}_2, \mathrm{WRITE}, \ell, v_2\rangle\right) \\
&\qquad \Rightarrow \left(\forall n \,.\, \langle \mathit{uid}_1, \mathit{tid}_1, \mathrm{WRITE}, \ell, v_1\rangle \xrightarrow{\mathrm{obs}_n} \langle \mathit{uid}_2, \mathit{tid}_2, \mathrm{WRITE}, \ell, v_2\rangle\right) \qquad (1)
\end{aligned}$$



(a) Write serialisation

(b) Read order

Fig. 2. P_1 and P_3 observing (a) two instances of W_2x or (b) R_2x and R_2y

Example 1 (Memory Coherence). Fig. 2(a) depicts an execution invalidated by the coherence constraint (1). Threads P_1 and P_3 observe the write events $W_2\,x\,1$ and $W_2\,x\,2$ in opposite order. By transitivity, $W_2\,x\,2 \xrightarrow{\text{obs}_3} W_2\,x\,1$ follows from $W_2\,x\,2 \xrightarrow{\text{obs}_3} R_3\,x \xrightarrow{\text{obs}_3} W_2\,x\,1$. Similarly, $W_2\,x\,1 \xrightarrow{\text{obs}_1} R_1\,x \xrightarrow{\text{obs}_1} W_2\,x\,2$ implies that $W_2\,x\,1 \xrightarrow{\text{obs}_1} W_2\,x\,2$, and $W_2\,x\,2 \xrightarrow{\text{obs}_1} W_2\,x\,1$ follows from (1) and $W_2\,x\,2 \xrightarrow{\text{obs}_3} W_2\,x\,1$. This contradicts $W_2\,x\,1 \xrightarrow{\text{obs}_1} W_2\,x\,2$. ◁

The coherence constraint (1) also implies that the observation orders for reads of the same location are consistent for all threads. There is, however, no such constraint for write (or read) accesses to *different* locations. Fig. 2(b), for instance, depicts a valid execution in which the threads P_1 and P_3 observe two subsequent reads from x and y in opposite order. We point out that this does *not* contradict the definition in [2, §2.4] that "a read is globally performed as soon as it is performed." This apparent discrepancy stems from the fact that [2] defines when read events are *performed* ("the point when the value of the read is determined" [2, §2.3]) whereas we define when they are *observed*: it is possible to conceive a cache hierarchy in which thread P_n has already *observed* a read while thread P_m may still influence its outcome, i.e., according to Definition 1, a read can be observed before it is actually performed.

3 Operational Semantics of Parallel Assertions

Parallel assertions are evaluated over a given thread's observation of the program state and *execution history*. We provide the syntax and semantics of assertions in §3.1, and subsequently cover executions in §3.2.

3.1 Structural Operational Semantics for Assertions

Each scope event $e \in \mathbb{S}$ has a unique identifier *uid* and an assertion expression $\phi_{uid} \in AExpr$, where *AExpr* is the set of all side-effect-free C++ expressions (defined in [7, §A.4]) augmented with a number of operators (described below). Table 1 shows the (simplified) syntax of assertions in *AExpr*. We hide

Table 1. Simplified syntax of assertion expressions

$$\begin{array}{rl} \mathit{assertion} ::= & \mathrm{HO}\ (\mathit{assertion})\ | \\ & \mathrm{LR}\ (\mathit{lvalue})\ |\ \mathrm{LW}\ (\mathit{lvalue})\ |\ \mathrm{RR}\ (\mathit{lvalue})\ |\ \mathrm{RW}\ (\mathit{lvalue})\ | \\ & \mathit{assertion\ infix\text{-}op\ asssertion}\ |\ \mathit{unary\text{-}op\ assertion}\ |\ \mathit{rvalue}\ |\ \mathit{lvalue} \end{array}$$

the complexity of C++ expressions by omitting details about unary and binary operations (*unary-op* and *infix-op*), operator precedence, and type correctness.

In accordance with the C++ standard [7, §3.10], the non-terminal *lvalue* represents an expression that "designates [...] an object," and an *rvalue* is "a value that is not associated with an object." As in §2.1, we use $v \in \mathbb{V}$ to refer to *rvalue*s and $\ell \in \mathbb{L}$ to refer to *lvalue*s. The access operators LR, LW, RR, and RW in Table 1 take a single *lvalue* as a parameter and check for the occurrence of a memory access to the respective object (c.f. §1). The operator HO takes an assertion as a parameter and returns a Boolean indicating whether this assertion evaluated to true at some point in the respective scope. The use of these operators is demonstrated in Fig. 1 and [13].

We use $\mathit{lvalues}(\mathit{expr}) \subseteq \mathbb{L}$ to denote the set of memory locations (*lvalue*s, respectively) referenced by $\mathit{expr} \in \mathit{AExpr}$ outside an access operator. The operator *lvalues* is defined inductively as follows:

- If $\mathit{lvalues}(\mathit{expr}) = \mathcal{L}$, then $\mathit{lvalues}(\mathrm{HO}(\mathit{expr})) = \mathit{lvalues}(\mathit{unary\text{-}op\ expr}) = \mathcal{L}$.
- Similarly, $\mathit{lvalues}(\mathit{expr}_1\ \mathit{infix\text{-}op}\ \mathit{expr}_2) = \mathit{lvalues}(\mathit{expr}_1) \cup \mathit{lvalues}(\mathit{expr}_2)$.
- If $\mathit{expr} \in \{\mathrm{LR}(\ell), \mathrm{LW}(\ell), \mathrm{RR}(\ell), \mathrm{RW}(\ell)\}, \ell \in \mathbb{L}$, then $\mathit{lvalues}(\mathit{expr}) = \emptyset$.
- Finally, $\mathit{lvalues}(\ell) = \{\ell\}$ for $\ell \in \mathbb{L}$ and $\mathit{lvalues}(v) = \emptyset$ for $v \in \mathbb{V}$.

Similarly, $\mathit{accessops}(\mathit{expr}) \subseteq \mathit{AExpr}$ is the set of all sub-expressions of $\mathit{expr} \in \mathit{AExpr}$ of the form $\mathrm{LR}(\ell)$, $\mathrm{LW}(\ell)$, $\mathrm{RR}(\ell)$, or $\mathrm{RW}(\ell)$ (where $\ell \in \mathbb{L}$). Intuitively, $\mathit{lvalues}(\mathit{expr})$ represents all memory locations whose value is relevant to the evaluation of *expr*, and $\mathit{accessops}(\mathit{expr})$ represents all access events in an expression.

An *assertion set* α is a set of *tagged* assertion expressions $\mathit{uid} : \mathit{expr}$ (where $\mathit{expr} \in \mathit{AExpr}$). Each set α can be partitioned into sets $\alpha|_{\mathit{uid}}$, which denotes the restriction of α to elements tagged with *uid*. The set $\alpha|_{\mathit{uid}}$ itself is inductively defined for each *uid* as the smallest set satisfying the following rules:

- The assertion $\mathit{uid} : \phi_{\mathit{uid}}$ as well as its negation $\mathit{uid} : (\neg\phi_{\mathit{uid}})$ are in $\alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\neg \mathit{expr}) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr} \in \alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\mathit{expr}_1\ \mathit{bop}\ \mathit{expr}_2) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr}_1$ and $\mathit{uid} : \mathit{expr}_2$ are in $\alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\mathrm{HO}(\mathit{expr})) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr} \in \alpha|_{\mathit{uid}}$

Here, $\mathit{expr}, \mathit{expr}_1, \mathit{expr}_2 \in \mathit{AExpr}$ and *bop* represents the Boolean connectives supported by the programming language (e.g., $\wedge$ or $\vee$). Intuitively, $\alpha|_{\mathit{uid}}$ contains the assertion ϕ_{uid} and its negation $\neg\phi_{\mathit{uid}}$, as well as all sub-expressions of ϕ_{uid}. $\mathcal{A}$ denotes the set of all conceivable assertion sets.

Example 2. The assertion set α for $!\,\mathrm{RR}(x) \,||\, \mathrm{HO}(\mathrm{LW}(x))$ (from Fig. 1) comprises the original assertion as well as the elements $\mathit{uid} : !(!\,\mathrm{RR}(x) \,||\, \mathrm{HO}(\mathrm{LW}(x)))$, $\mathit{uid} : !\,\mathrm{RR}(x)$, $\mathit{uid} : \mathrm{RR}(x)$, $\mathit{uid} : \mathrm{HO}(\mathrm{LW}(x))$, and $\mathit{uid} : \mathrm{LW}(x)$. ◁

A *state* σ is a finite mapping from locations $\mathbb{L}$ to values $\mathbb{V}$. $\mathcal{S}$ denotes the set of all conceivable states, and $\sigma[\ell \mapsto v]$ denotes the state that maps $\ell \in \mathbb{L}$ to $v \in \mathbb{V}$ and all other locations $\ell' \neq \ell$ to $\sigma[\ell']$. For a given set of locations $\mathcal{L} \subseteq \mathbb{L}$ the projection $\sigma|_{\mathcal{L}}$ of σ to $\mathcal{L}$ is the state that maps locations $\ell \in \mathcal{L}$ to $\sigma[\ell]$ and is undefined for all other locations.

A *history* χ is a tuple $\langle \delta, \theta \rangle$[2] of sets of assertion expressions which represents past evaluations of assertions by recording assertion expressions that evaluate to true at some point during the execution. $\chi.\delta$ (of type *AExpr*) represents the *immediate* past reflecting only the most recent event. $\chi.\theta$ is an assertion set representing the distant past, cumulating all tagged assertions that evaluated to true at some point in the past of the current execution trace. $\mathcal{H}$ denotes the set of all conceivable histories.

A *configuration* κ is a tuple $\langle \sigma, \chi, \alpha \rangle$ comprising a state $\sigma \in \mathcal{S}$, a history $\chi \in \mathcal{H}$, and an assertion set $\alpha \in \mathcal{A}$. $\mathcal{C}$ denotes the set of all configurations.

Evaluating Assertions. Assertions are evaluated over a given configuration. We introduce a reduction relation $\rightarrow_a \subseteq \mathcal{C} \times (\mathbb{N} \times AExpr) \times (\mathbb{N} \times AExpr)$ for assertions. We use $(\rightarrow_a)^*$ to denote the reflexive transitive closure of $\rightarrow_a$.

1. Assertion expressions (or sub-expressions) that do *not* contain the operators LR, LW, RR, RW, and HO are evaluated over σ according to the semantics of the C++ language. Therefore, $\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : v$ if *expr* evaluates to $v \in \mathbb{V}$ in state σ.
2. Expressions involving the access operators LR, LW, RR, or RW are evaluated according to the immediate history $\chi.\delta$:

$$\frac{\mathit{expr} \in \chi.\delta}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : \mathsf{T}} \quad \mathit{expr} \in \left\{ \begin{array}{l} \mathrm{LR}(\ell),\ \mathrm{LW}(\ell), \\ \mathrm{RR}(\ell),\ \mathrm{RW}(\ell) \end{array} \right\} \tag{2}$$

   and similarly $\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : \mathsf{F}$ if $\mathit{expr} \notin \chi.\delta$.
3. The operator HO maps its parameter *expr* to true if *expr* evaluates to true in the current configuration, or if *expr* was true at some point in the past.

$$\frac{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \quad (\rightarrow_a)^* \quad \mathit{uid} : \mathsf{T}}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathrm{HO}(\mathit{expr}) \rightarrow_a \mathit{uid} : \mathsf{T}} \tag{3}$$

$$\frac{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \quad (\rightarrow_a)^* \quad \mathit{uid} : \mathsf{F}}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathrm{HO}(\mathit{expr}) \rightarrow_a \mathit{uid} : b} \quad b \stackrel{\text{def}}{=} \begin{cases} \mathsf{T} \text{ if } ((\mathit{uid} : \mathit{expr}) \in \chi.\theta) \\ \mathsf{F} \text{ otherwise} \end{cases} \tag{4}$$

   Note that the parameters of HO must not be reduced by $\rightarrow_a$ or evaluated over σ unless this step yields T. This is necessary to avoid mixing values of σ and values of past states during the evaluation.

Example 3. The configuration $\kappa \stackrel{\text{def}}{=} \langle \sigma, \langle \{\mathrm{LR}(x)\}, \{\mathit{uid} : \mathrm{LW}(x)\} \rangle, \alpha \rangle$ (with α as in Example 2) reflects a recent read access as well as a previous write access to x by the asserting thread. Thus, $\rightarrow_a$ yields T for $\mathrm{HO}(\mathrm{LW}(x))$ (by rule 4, since $\kappa \vdash \mathit{uid} : \mathrm{LW}(x) \rightarrow_a \mathit{uid} : \mathsf{F}$ and $\mathit{uid} : \mathrm{LW}(x) \in \chi.\theta$) and $!\mathsf{F}$ for $!\,\mathrm{RR}(x)$ (since $\mathrm{RR}(x) \notin \chi.\delta$). The assertion $!\,\mathrm{RR}(x) || \mathrm{HO}(\mathrm{LW}(x))$ does not fail at this point. ◁

[2] Our notation is inspired by the Dirac delta function δ and the Heaviside function θ.

$$\frac{}{\langle (W_{tid}\,\ell\, v) :: ex,\ \langle \sigma, \chi, \alpha \rangle\rangle \rightarrow \langle ex,\ \langle \sigma[\ell \mapsto v], \langle \{\mathrm{LW}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle}\ \ tid = n \qquad (6)$$

$$\frac{}{\langle (R_{tid}\,\ell) :: ex,\ \langle \sigma, \chi, \alpha \rangle\rangle \rightarrow \langle ex,\ \langle \sigma, \langle \{\mathrm{LR}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle}\ \ tid = n \qquad (7)$$

$$\frac{}{\langle (W_{tid}\,\ell\, v) :: ex,\ \langle \sigma, \chi, \alpha \rangle\rangle \rightarrow \langle ex,\ \langle \sigma[\ell \mapsto v], \langle \{\mathrm{RW}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle}\ \ tid \neq n \qquad (8)$$

$$\frac{}{\langle (R_{tid}\,\ell) :: ex,\ \langle \sigma, \chi, \alpha \rangle\rangle \rightarrow \langle ex,\ \langle \sigma, \langle \{\mathrm{RR}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle}\ \ tid \neq n \qquad (9)$$

Fig. 3. Reduction rules for read and write accesses

3.2 Operational Semantics for Events

Given a set of events $\mathbb{E}$, its Kleene closure $\mathbb{E}^*$ is the set of all sequences of events in $\mathbb{E}$, including the empty sequence ϵ. We use the ML-like notation :: for sequence concatenation. An execution of a thread with $tid = n$ is a sequence $ex \in \mathbb{E}^*$ of events such that for every sub-sequence $e_i :: e_{i+1}$ of ex we have $e_i \xrightarrow{\mathrm{obs}_n} e_{i+1}$. The semantics of an execution is determined by the (reflexive) reduction relation $\rightarrow \subseteq (\mathbb{E}^* \times \mathcal{C}) \times (\mathbb{E}^* \times \mathcal{C})$ which characterises the impact of events on configurations. In the following, we define this reduction relation $\rightarrow$ based on $\rightarrow_a$.

A common side effect of all events is the modification of the history. Let $\langle \sigma, \chi, \alpha\rangle$ be the current configuration and let χ and χ' denote the history before and after an event, respectively. For all events, $\chi'.\theta$ is $\chi.\theta$ augmented with all assertion expressions in α that evaluate to true in the previous configuration:

$$\chi'.\theta \overset{\mathrm{def}}{=} \chi.\theta \cup \{(uid : expr) \in \alpha \mid \langle \sigma, \chi, \alpha\rangle \vdash (uid : expr)\ (\rightarrow_a)^*\ uid : \mathsf{T}\} \qquad (5)$$

The reduction rules presented in the following refer to $\chi'.\theta$ as defined in (5). From now on, we use n to denote the identifier of the current (asserting) thread.

1. Fig. 3 shows the reduction rules for memory events. We distinguish between events generated by thread n and events generated by other threads in order to determine their effect on the immediate past $\chi.\delta$.
2. Fig. 4 shows the reduction rules for scope events. Upon entry of a scope, the respective assertion is added to the assertion set. Note that Rule (11) does not allow us to exit a scope if the corresponding assertion (identified by *eid*) failed. Finally, a thread only observes the scope events generated by itself.
3. Skip and fence events modify the history in accordance with (5). In addition, the memory model (see §5) may impose ordering constraints on fence events.

$$\frac{}{\langle e_i :: ex,\ \langle \sigma, \chi, \alpha\rangle\rangle \rightarrow \langle ex,\ \langle \sigma, \langle \emptyset, \chi'.\theta\rangle, \alpha\rangle\rangle}\ \ e_i \in (\mathbb{F} \cup \{\mathtt{skip}\}) \qquad (12)$$

$$\frac{}{\langle\langle \mathit{uid}, \mathit{tid}, \text{ENTRY}, \phi_{uid}\rangle :: \mathit{ex},\ \langle \sigma, \chi, \alpha\rangle\rangle \ \rightarrow\ \langle \mathit{ex},\ \langle \sigma, \langle \emptyset, \chi'.\theta\rangle, \alpha \cup \alpha|_{uid}\rangle\rangle} \quad \mathit{tid} = n \tag{10}$$

$$\frac{(\mathit{eid}:\neg\phi_{eid}) \notin \chi.\theta \ \wedge\ \langle \sigma, \chi, \alpha\rangle \vdash \mathit{eid}:\phi_{eid}\,(\rightarrow_a)^*\ \mathit{eid}:\mathsf{T}}{\langle\langle \mathit{uid}, \mathit{tid}, \text{EXIT}, \mathit{eid}\rangle :: \mathit{ex},\ \langle \sigma, \chi, \alpha\rangle\rangle \ \rightarrow\ \langle \mathit{ex},\ \langle \sigma, \langle \emptyset, \chi'.\theta\rangle, \alpha \setminus \alpha|_{eid}\rangle\rangle} \quad \mathit{tid} = n \tag{11}$$

Fig. 4. Reduction rules for scope events

Assertion Failures. In case of a failure of an active assertion ϕ_{uid}, the configuration can be reduced to a (canonical) error configuration **error**. We allow this rule to be applied *non-deterministically* at any point after an assertion failure (but at latest upon exit of the corresponding scope, see Rule (11)). This leaves some freedom for the implementation as to when a failed assertion is reported.

$$\frac{(\mathit{uid}:\neg\phi_{uid}) \in \chi.\theta \ \vee\ (\mathit{uid}:\phi_{uid}) \in \alpha \ \wedge\ \kappa \vdash (\mathit{uid}:\phi_{uid})\,(\rightarrow_a)^*\,(\mathit{uid}:\mathsf{F})}{\langle \mathit{ex},\ \kappa\rangle \ \rightarrow\ \mathbf{error}} \tag{13}$$

Example 4. Consider an execution $W_n\,x\,1 :: R_{tid}\,x$ (where $tid \neq n$) starting in the configuration $\kappa \overset{\text{def}}{=} \langle \sigma, \langle\{\text{LR}(x)\}, \{\mathit{uid} : \text{LW}(x)\}\rangle, \alpha\rangle$ introduced in Example 3. By rule 6, we derive $\langle R_{tid}\,x, \langle \sigma[x \mapsto 1], \langle\{\text{LW}(x)\}, \{\mathit{uid}:\text{LW}(x)\}, \alpha\rangle\rangle$ from $\langle W_n\,x\,1 :: R_{tid}\,x, \kappa\rangle$. Note that $\mathit{uid}:\text{LR}(x)$ is *not* added to $\chi'.\theta$ by rule 5, since it is not an element of the assertion set α. ◁

4 Optimisations

In this section, we formally prove that only a fraction of the events in an execution is necessary to evaluate an assertion. Since logging information can be expensive, there is a significant optimisation opportunity in filtering the executions before they are evaluated. In particular, this means that throughout a scope we can dismiss the events that are irrelevant to the corresponding assertion, as long as assertion failures are preserved:

Definition 2. *Let* $\text{ex}_1, \text{ex}_2 \in \mathbb{E}^*$ *be two executions delimited by a scope with assertion* ϕ_{uid} *that contains no other scope events. Then,* ex_1 *and* ex_2 *are* parallel assertion equivalent *with respect to* ϕ_{uid} *iff in all configurations* $\langle \sigma, \chi, \alpha|_{\text{uid}}\rangle$ *it holds that* $(\langle \text{ex}_1,\ \langle \sigma, \chi, \alpha|_{\text{uid}}\rangle\rangle\,(\rightarrow)^*\,\mathbf{error}) \Leftrightarrow (\langle \text{ex}_2,\ \langle \sigma, \chi, \alpha|_{\text{uid}}\rangle\rangle\,(\rightarrow)^*\,\mathbf{error})$.

Nested Assertion Scopes. For the purpose of checking whether a specific assertion ϕ_{uid} is violated by an execution, we can treat other scope events similar to `skip`. Scope events occurring in threads other than the current one have no impact on the evaluation of ϕ_{uid} (c.f. Rules 10, 11). A nested scope event only affects the execution if its respective assertion fails. Therefore, it is legal to process assertion scopes independently as long as we report the first assertion that fails (upon exit of the corresponding scope). This allows us to define parallel assertion equivalence (Definition 2) with respect to a *single* parallel assertion.

In the following, we formally define the projection of configurations and executions to a given assertion and prove that projection preserves assertion failures.

Definition 3. *We define the projection of a configuration* $\langle \sigma, \chi, \alpha \rangle$ *to a given assertion* ϕ_{uid} *as* $\langle \sigma, \chi, \alpha \rangle|_{\phi_{\text{uid}}} \overset{\text{def}}{=} \langle \sigma|_{\text{lvalues}(\phi_{\text{uid}})}, \langle \chi.\delta \cap \text{accessops}(\phi_{\text{uid}}), \chi.\theta \rangle, \alpha \rangle$.

Theorem 1. *Projecting a configuration* $\langle \sigma, \chi, \alpha \rangle$ *to an assertion* ϕ_{uid} *has no impact on the evaluation of* ϕ_{uid}*:*

$$\begin{aligned} &\forall \kappa \in \mathcal{C} \,.\, \forall \text{expr}_1 \in \alpha|_{\text{uid}}, \text{expr}_2 \in \text{AExpr} \,. \\ &\quad (\kappa \vdash \text{expr}_1 \ (\rightarrow_a)^* \ \text{expr}_2) \Leftrightarrow (\kappa|_{\phi_{\text{uid}}} \vdash \text{expr}_1 \ (\rightarrow_a)^* \ \text{expr}_2) \end{aligned}$$

Proof. By induction on the structure and height of derivations generated by the reduction rules in §3.1. ■

Definition 4. *The projection of an execution* $\text{ex} \in \mathbb{E}^*$ *to an assertion* ϕ_{uid} *is defined inductively by* $\epsilon|_{\phi_{\text{uid}}} \overset{\text{def}}{=} \epsilon$ *and* $(e :: \text{ex})|_{\phi_{\text{uid}}} \overset{\text{def}}{=} e|_{\phi_{\text{uid}}} :: (\text{ex}|_{\phi_{\text{uid}}})$*, where*

$$e|_{\phi_{\text{uid}}} \overset{\text{def}}{=} \begin{cases} e & \textit{if} \ (e = W_{\text{tid}}\, \ell\, v) \wedge \\ & \quad \begin{pmatrix} (\ell \in \text{lvalues}(\phi_{\text{uid}})) & \vee \\ (\text{tid} = n \wedge \text{LW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) & \vee \\ (\text{tid} \neq n \wedge \text{RW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) & \end{pmatrix} \\ & \textit{or} \ (e = R_{\text{tid}}\, \ell v) \wedge \\ & \quad \begin{pmatrix} (\text{tid} = n \wedge \text{LR}(\ell) \in \text{accessops}(\phi_{\text{uid}})) & \vee \\ (\text{tid} \neq n \wedge \text{RR}(\ell) \in \text{accessops}(\phi_{\text{uid}})) & \end{pmatrix} \\ \texttt{skip} & \textit{otherwise} \end{cases}$$

Theorem 2. *Let* $\text{ex} \in \mathbb{E}^*$ *be an execution that is delimited by a scope associated with* ϕ_{uid} *and does not contain any other assertion scopes. Then* ex *and* $\text{ex}|_{\phi_{\text{uid}}}$ *are parallel assertion equivalent with respect to the assertion* ϕ_{uid}*.*

The proof of Theorem 2, led by induction over the length of *ex*, shows that events can be treated as `skip` as long they have no impact on the evaluation of ϕ_{uid} (in accordance with Theorem 1).

Theorems 1 and 2 enable the following optimisation. For each scope instance, no events need to be logged before its respective entry event. Upon reaching a scope entry event with assertion ϕ_{uid}, our implementation records only the relevant fraction $\sigma|_{\phi_{uid}}$ of the state. After that, it is sufficient to log all events $e|_{\phi_{uid}}$ (in observation order) until the corresponding scope exit event is reached.

Relaxed Order Observations. Under certain conditions it is sufficient to approximate the observation order $\xrightarrow{\text{obs}_n}$ by a partial order. In particular, we show that for a certain class of assertions all that matters is the order of events with respect to the scope events $\mathbb{S}$.

Theorem 3. *Let* ex *be an execution that is delimited by a scope associated with* ϕ_{uid} *and does not contain any other assertion scopes, and let* $\pi(\mathrm{ex})$ *be an arbitrary permutation of* ex*. If one of the following conditions holds for* ϕ_{uid}*, then* ex *and* $\pi(\mathrm{ex})$ *are parallel assertion equivalent with respect to* ϕ_{uid}*:*

i ϕ_{uid} *does not contain the operator* HO *and* $\mathrm{lvalues}(\phi_{\mathrm{uid}}) = \emptyset$*.*
ii ϕ_{uid} *does not contain the operator* HO*,* $\mathrm{lvalues}(\phi_{\mathrm{uid}}) = \{\ell\}$ *(for some* $\ell \in \mathbb{L}$*), and* $\mathrm{accessops}(\phi_{\mathrm{uid}}) = \emptyset$*.*

Proof. We prove that the order of the sequence is irrelevant by showing that the following logical equivalence holds:

$$(\langle ex, \langle \sigma, \chi, \alpha|_{uid}\rangle\rangle \,(\rightarrow)^{*}\, \mathbf{error}) \quad \Leftrightarrow$$
$$\exists e \in ex \,.\, \forall \langle \sigma', \chi', \alpha|_{uid}\rangle \,.\, \langle e :: \epsilon, \langle \sigma', \chi', \alpha|_{uid}\rangle\rangle \,(\rightarrow)^{*}\, \mathbf{error}$$

Note that the implication holds trivially in one direction ($\Leftarrow$). For the other direction, we need to show that ϕ_{uid} does not depend on the configuration before the execution of $e \in ex$. Let $\langle \sigma'', \chi'', \alpha|_{uid}\rangle$ be the configuration after the execution of e. By Theorem 1 we have $\langle \sigma'', \chi'', \alpha|_{uid}\rangle \,(\rightarrow_a)^{*}\, \mathbf{error}$ if $\langle \sigma'', \chi'', \alpha|_{uid}\rangle|_{\phi_{uid}} \,(\rightarrow_a)^{*}\, \mathbf{error}$. In case (i), the domain of $\sigma|_{\phi_{uid}}$ is empty and ϕ_{uid} refers only to $\chi''.\delta$. In case (ii), $\sigma|_{\phi_{uid}}$ is only defined for ℓ, and $\chi''.\delta \cap \mathit{accessops}(\phi_{uid}) = \emptyset$. Accordingly, ϕ_{uid} depends on only a single item in the configuration, which can only be updated *atomically* by the events in *ex*. ■

The conditions for ϕ_{uid} in Theorem 3 are not tight. Other criteria (based on partial order reduction, for instance) may allow for more aggressive optimisations. Note that in the most extreme case — if we can establish statically that the assertion holds — it is not necessary to log any events at all. In general, an approach based in this insight is obviously impractical. However, since observing events and orderings between events is expensive, an improved relaxation function which limits the number of required observations can significantly reduce the work required by the checker.

5 Memory Models and Fences

On a platform that guarantees sequential consistency (SC) the operations of each individual thread are globally observed in a sequential order consistent with the program order. For performance reasons, modern processors do not provide such a guarantee. They do, however, provide *fence* instructions, which enable the programmer (or compiler) to enforce a global ordering between events.

Different platforms provide different types of fences, and their semantics depends on the specific architecture (c.f. [2,6]). In general, we distinguish between *non-cumulative* and *cumulative* fences. Intuitively, non-cumulative fences prevent thread-local reordering of events across the fence event. Cumulative barriers also affect the order of events of other threads (e.g., by flushing store buffers or caches). One non-cumulative fence operations is `mfence` on Intel processors:

it “guarantees that every load and store instruction that precedes in program order the `mfence` instruction is globally visible before any load or store instruction that follows the `mfence` instruction is globally visible.” [1, pg. 4-23][3]

Formally, two memory events $e_1, e_2 \in \mathbb{M}$ occurring before and after an `mfence` event in program order, respectively, will be observed in this order by all threads:

$$\exists n \, . \, \left(e_1 \xrightarrow{\mathrm{po}_n} \texttt{mfence} \xrightarrow{\mathrm{po}_n} e_2 \right) \quad \Rightarrow \quad \forall n \, . \, e_1 \xrightarrow{\mathrm{obs}_n} e_2 \tag{14}$$

The `hwsync` instruction of the Power architecture provides a similar guarantee, but is also cumulative in the sense that it separates the memory events *observed* by the thread before and after executing `hwsync`.

Example 5. The assertion in the program in Fig. 5 holds on any platform that guarantees sequential consistency, but may fail on a platform that permits reordering of write events (as shown in the diagram in Fig. 5). A fence between the events $W_1\,x\,1$ and $W_1\,y\,1$ (indicated by a dashed arrow) rules out the failing execution and restores sequential consistency for this program. ◁

The side effect described in Example 5 is not always desirable. If the fence instruction is part of the instrumentation code required to log the write events, then this modification effectively eliminates an erroneous execution. Logging mechanisms requiring stronger guarantees (as provided by the atomic compare-and-swap instruction `lock cmpxchg` on Intel architectures, for instance, which is frequently used to implement locks) exacerbate the probe effect.

A complete formalisation of fences exceeds the scope of our paper; we refer the reader to [2] instead. The following section presents our implementation, which makes use of fences, and discusses their side effects.

6 Implementation

We implemented a runtime checker for parallel assertions called PASSERT [12] as an extension of the LLVM compiler [9]. During compilation, PASSERT instruments read and write accesses in a program annotated with assertions with calls to logging functions. The log is then analysed for assertion violations by a checker (either during or after the execution).

The instrumentation results in a number of side effects. Firstly, logging events takes time, and hence changes the timing behaviour of the program under test. More subtly, the instrumentation adds locks and fences which may rule out executions that are otherwise legal the underlying memory model. In the following, we discuss two key optimisations that dramatically reduced these effects based on the results from §4.

Filtering Optimisation. To counteract the probe effect, we implemented an event filter that conservatively approximates the set of events required by Theorem 2.

Firstly, we use an alias analysis to narrow down which memory locations may affect the evaluation of an assertion, which significantly reduces the number of

[3] The concept of global visibility coincides with our notion of observability.

```
          P1                        P2

thru {
  x = 1;
  y = 1;                      if (y && !x)
} passert(!RW(x) &&             z = 1;
          !RW(y) &&
          !RW(z)  );
```

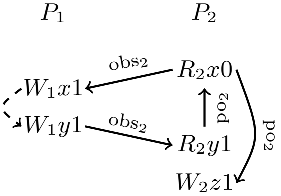


Fig. 5. Restoring sequential consistency using fences (x and y are initially 0)

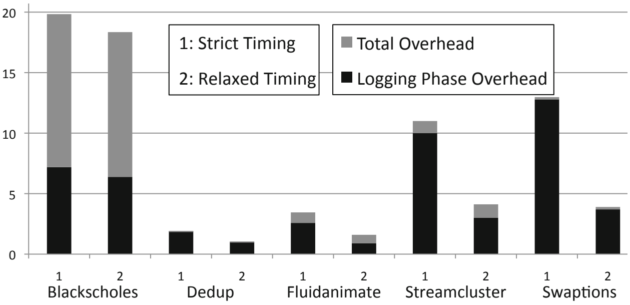


Fig. 6. Runtime overhead for PARSEC benchmarks

instructions that need to be instrumented and therefore eliminates the associated side effect. Secondly, we use dynamic filtering to determine at runtime which locations are not referenced by any live assertions, and hence need not be logged. This further reduces – but does not entirely eliminate – the probe effect.

Filtering effectively *enables* runtime checking. Before this optimisation, most instrumented programs simply ran out of memory, while all our benchmarks completed successfully after filtering.

Time-stamping Mechanisms. Assertion evaluation requires recording the timing of remote events relative to the asserting thread. For events generated by different threads this requires a certain amount of synchronisation. Our implementation uses a global time-stamp for this purpose.

A time-stamping mechanism must correctly associate events with timestamps reflecting the observation order. In a weak memory model, this may need to be enforced through the use of locks and memory fences. Consequently, stronger ordering requirements exacerbate the probe effect, which in turn may rule out otherwise legal executions. In Fig. 5, for example, a time-stamping mechanism inserting a fence between $W_1\,x\,1$ and $W_1\,y\,1$ effectively prevents reordering of these events, thus hiding the bug (as explained in Example 5).

By Theorem 3, the order of events within a scope can be safely ignored in certain cases (such as the program in Example 5). Our experience suggests that this relaxation is admissible for many assertions: previous work [12] showed that of the 14 out of 17 real world bugs presented in [17] can be captured using parallel assertions; all of those assertions are amenable to relaxed timing. This observation led us to implement two different time-stamping mechanisms:

- *Strict time-stamping* uses locks to guarantee that a shared counter is incremented atomically with the execution of an event, hence providing a total order over all time-stamped events.
- *Smeared time-stamps* yield a partial order sufficiently accurate to evaluate the assertions characterised in Theorem 3. Scope events atomically increment a global counter. All other events e are logged using a preceding and a successive read to the global counter `ts`. $R_{tid}\,\texttt{ts} \xrightarrow{\text{po}_{tid}} e \xrightarrow{\text{po}_{tid}} R_{tid}\,\texttt{ts}$ is enforced using (non-cumulative) fences if necessary, thus avoiding the use of locks.

Smeared time-stamping is sufficient to determine whether an event happened within a certain scope. A potentially ambiguity may arise in the rare case that different time-stamps are recorded before and after the event. If this difference affects the correctness of an assertion, we report a potential error (though we have not observed this in practice).

In order to measure the run-time overhead, we evaluated the runtime overhead of passert by annotating a set of PARSEC benchmarks with parallel assertions. We did not discover any new bugs in this widely used benchmark suite. Strict timing had an overhead of 6.6×, which can be reduced to 3.5× through the use of smeared timestamps (Fig. 6).

7 Related Work

A number different assertion formalisms for parallel programs have been proposed and implemented. JMPaX [14], for instance, checks linear temporal logic properties for Java programs. Phalanx [15] allows the checking of expressive heap assertions in Java programs. SharC [3] enables the static and dynamic verification of rules for sharing individual objects in C. These formalisms address different properties than parallel assertions — a more detailed discussion is given in [13], where we initially introduced parallel assertions.

There is a wide variety of work on debugging programs in the presence of weak memory models. One approach is to minimize the probe effect through hardware based logging. [16] allows the observation of events with minimal (2%) perturbation to the program execution on both TSO(x86) and SC systems. Predictive analyses (e.g. [5]), on the other hand, enable the prediction of possible assertion violations in a weak memory model based on an SC execution. Trace based analysis can also be used to automatically fix bugs. Liu et al. [10], for example, provide a formal semantics for LLVM bytecode under weak memory models, and show how to add fences to prevent erroneous traces. A model checker can exhaustively explore all possible interleavings under a given memory model for all possible inputs (e.g. [8]), and hence capture all bugs albeit at a high computational cost. All of these techniques are orthogonal to our work. Parallel assertions could be implemented as an extension to any of these systems.

8 Conclusion

We provide a formal definition of *parallel assertions*, a novel assertion language for detecting intricate concurrency bugs, and an operational semantics enabling their evaluation on architectures with weak memory models. Unlike the original semantics [13], which assumes sequential consistency, our new semantics is less restrictive and enables the detection of a highly relevant class of bugs introduced by the complexity of modern multi-processor architectures. Secondly, our novel semantics enables two key optimisations which, as demonstrated in §6, are crucial to making run-time checking of parallel assertions feasible.

Acknowledgements. The authors thank Lennart Beringer and the anonymous reviewers for their helpful suggestions and comments.

References

1. Intel 64 and IA-32 architectures software developer's manual (March 2012)
2. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). Formal Methods in System Design 40(2) (2012)
3. Anderson, Z., Gay, D., Ennals, R., Brewer, E.: SharC: checking data sharing strategies for multithreaded C. In: PLDI. ACM (2008)
4. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI. ACM (2008)
5. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA. ACM (2011)
6. Chong, N., Ishtiaq, S.: Reasoning about the ARM weakly consistent memory model. In: MSPC. ACM (2008)
7. International Standard 14882 (Programming Languages) C++, Final Committee Draft N3092 (March 2010), ISO/IEC
8. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings (preliminary version). SIGARCH Comput. Archit. News 36(5) (2009)
9. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization. IEEE (2004)
10. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. ACM (2012)
11. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS. ACM (2008)
12. Schwartz-Narbonne, D., Liu, F., August, D., Malik, S.: PASSERT: A Tool for Debugging Parallel Programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 751–757. Springer, Heidelberg (2012)
13. Schwartz-Narbonne, D., Liu, F., Pondicherry, T., August, D.I., Malik, S.: Parallel assertions for debugging parallel programs. In: MEMOCODE. IEEE (2011)
14. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: ESEC/FSE. ACM (2003)
15. Vechev, M., Yahav, E., Yorsh, G.: Phalanx: parallel checking of expressive heap assertions. In: ISMM. ACM (2010)
16. Xu, M., Bodik, R., Hill, M.D.: A hardware memory race recorder for deterministic replay. IEEE Micro 27(1) (2007)
17. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. In: ISCA. ACM (2009)

Improved Multi-Core Nested Depth-First Search

Sami Evangelista[1], Alfons Laarman[2], Laure Petrucci[1], and Jaco van de Pol[2]

[1] LIPN, CNRS UMR 7030 — Université Paris 13, France
[2] Formal Methods and Tools, University of Twente, The Netherlands

Abstract. This paper presents CNDFS, a tight integration of two earlier multi-core nested depth-first search (NDFS) algorithms for LTL model checking. CNDFS combines the different strengths and avoids some weaknesses of its predecessors. We compare CNDFS to an earlier ad-hoc combination of those two algorithms and show several benefits: It has shorter and simpler code and a simpler correctness proof. It exhibits more robust performance with similar scalability, while at the same time reducing memory requirements.

The algorithm has been implemented in the multi-core backend of the LTSMIN model checker, which is now benchmarked for the first time on a 48 core machine (previously 16). The experiments demonstrate better scalability than other parallel LTL model checking algorithms, but we also investigate apparent bottlenecks. Finally, we noticed that the multi-core NDFS algorithms produce shorter counterexamples, surprisingly often shorter than their BFS-based counterparts.

1 Introduction

Model checking is a resource demanding task that can be performed by a systematic exploration of a huge directed graph representing the dynamic behaviour of the analysed system. Although memory is usually the major bottleneck, execution times can also often exceed acceptable limits. For instance the exploration of a 10^9 states graph at a high exploration rate of 10^5 states per second would take more than a day. This remains acceptable but becomes problematic when increasing the number of system configurations and properties analysed. Hence, model checking has gained a renewed interest with the advent of multi-core architectures that can help tackle this time explosion.

Some properties like safety properties rely on a complete enumeration of system states and can thus be easily parallelised since they do not ask for a specific search order. However, the problem is harder when it comes to the verification of Linear Time temporal Logic (LTL) properties. LTL model checking can be reduced to a cycle detection problem and state-of-the-art algorithms [8,9,11] proceed depth-first since cycles are more easily discovered using this search order. However, this characteristic also makes them unsuitable for parallel architectures since DFS is inherently sequential [20].

One approach to address this issue is to sacrifice the optimal linear complexity provided by DFS algorithms and switch to BFS-like algorithms, which are highly scalable both theoretically and experimentally. We compare our approach to the best representative of that family. More recently, two algorithms (LNDFS from [13] and ENDFS from [10]) adapted the well known Nested DFS (NDFS) algorithm [8] to multi-core architectures. They share the principle of launching multiple instances of NDFS that synchronise

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 269–283, 2012.

themselves to avoid useless state revisits. Although they are heuristic algorithms in the sense that, in the worst case, they reduce to spawn multiple unsynchronised instances of NDFS, the experiments reported in [13,14] show good practical speedups.

The contribution of this paper is an improvement to both the LNDFS and ENDFS algorithms, called CNDFS. This new algorithm is both much simpler and uses less memory, making it more compatible with lossy compression techniques such as tree compression [17] that can compress large states down to two integers. We also pursue a thorough experimental evaluation of this algorithm on the models of the BEEM database [18] with an implementation of this algorithm on top of the LTSMIN toolset [16]. The outcome of these experiments is threefold. Firstly, CNDFS exhibits a similar speedup to its predecessors, but achieves this more robustly, with smoother speedup lines, while using less memory. Second, it combines nicely with heuristics limiting the amount of redundant work performed by individual threads. Finally, in the presence of bugs, it reports counterexamples that are usually much shorter than those reported by NDFS and, more importantly, this length tends to decrease as more working threads get involved in the verification. This property is quite appreciable from a user perspective as it eases the task of error correction.

The outline of this paper is the following. In Section 2 we formally express the LTL model checking problem and review existing (sequential and parallel) algorithms that address it. CNDFS, our new algorithm, is introduced and formally proven in Section 3. Our experimental evaluation of this algorithm is summarised in Section 4. Finally, Section 5 concludes our paper and explores some research perspectives to this work.

2 Background

We give in this section the few ingredients that are required for the understanding of this paper and briefly review existing works in the field of explicit parallel LTL model checking based on the automata theoretic approach.

2.1 The Automata Theoretic Approach to LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [22] that proceeds in several steps. In this paper we focus only on the last step of the process that can be reduced to a graph problem: given a graph representing the synchronised product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. In this paper we will only reason on *automaton graphs* that result from the product of a Büchi property automaton and a system graph describing the dynamic behaviour of the modelled system.

Definition 1 (Automaton graph). *An* automaton graph *is a tuple* $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$*, where* $\mathcal{S}$ *is a finite set of states;* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *is a set of transitions;* $\mathcal{A} \subseteq \mathcal{S}$ *is the set of accepting states; and* $s_0 \in \mathcal{S}$ *is an initial state.*

Notations. Let $(\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$ be an automaton graph. For $s \in \mathcal{S}$ the set of its *successor states* is denoted by $succ(s) = \{s' \in \mathcal{S} \mid (s, s') \in \mathcal{T}\}$. $(s, s') \in \mathcal{T}$ is also denoted by $s \rightarrow s'$.

$s \rightarrow^+ s'$ ($s \rightarrow^* s'$) denotes the (reflexive) transitive closure of $\mathcal{T}$, i.e. the fact that s' is *reachable* from s. A *path* is a state sequence $s_1, \ldots, s_n$ with $s_i \rightarrow s_{i+1}, \forall i \in \{1, \ldots, n-1\}$, a *cycle* is a path $s_1, \ldots, s_n$ with $s_1 = s_n$ and a cycle $C \equiv s_1, \ldots, s_n$ is an *accepting cycle* if $C \cap \mathcal{A} \neq \emptyset$. An *accepting run* is an accepting cycle reachable from the initial state: $s_0, \ldots, s_i, \ldots, s_n$ where $s_i = s_n$. The *LTL model checking problem* consists of finding an accepting run in an automaton graph. An LTL model checking algorithm proceeds on-the-fly if it can report an accepting run without visiting all transitions.

2.2 Sequential LTL Model Checking Algorithms

NDFS [8] was the first LTL model checking algorithm proposed. It enjoys several nice properties: an optimal linear complexity, the on-the-fly discovery of accepting cycles and a low memory consumption (2 bits per state). Two variations of Tarjan's algorithm for SCC decomposition [9,11] have also been proposed with similar characteristics but we focus here on NDFS as our new algorithm is a direct descendant of this one.

Alg. 1 NDFS [8] as presented in [21].

```
$dfsBlue(s_0)$
procedure $dfsBlue(s)$ is
    $s.cyan$ := true
    for all $s'$ in $succ(s)$ do
        if $\neg s'.blue$ then $dfsBlue(s')$
    if $s \in \mathcal{A}$ then $dfsRed(s)$
    $s.cyan$ := false
    $s.blue$ := true
procedure $dfsRed(s)$ is
    $s.red$ := true
    for all $s'$ in $succ(s)$ do
        if $s'.cyan$ then exit(cycle)
        if $\neg s'.red$ then $dfsRed(s')$
```

The pseudo-code of this algorithm is given by Alg. 1. The algorithm performs a first level DFS (the blue DFS) to discover accepting states. When such a state is backtracked from, a second level DFS (the red DFS) is launched to see whether this accepting state (now called the *seed*) is reachable from itself and is thus part of an accepting cycle. It is sufficient to find a path back to the stack of the blue DFS [21], hence the *cyan* colour in Alg. 1. Correctness depends on the fact that different invocations of the red DFS happen in post-order. The algorithm works in linear time: each state is visited at most twice, since the result of a red DFS can be reused in subsequent red DFSs; states retain their red colour.

2.3 Parallel LTL Model Checking Algorithms for Shared-Memory Architectures

In the field of parallel LTL model checking, the first algorithms designed targeted distributed memory architectures like clusters of machines. This family of algorithms includes MAP [6], OWCTY [7] and BLEDGE [2]. It is however well known that this kind of message passing algorithm can be easily ported to shared-memory architectures like multi-core computers although the specificities of these architectures must be considered to achieve good scalability [4]. Their common characteristic is to rely on some form of breadth-first search (BFS) of the graph that has the advantage of being easily parallelised, unlike depth-first search (DFS) [20]. They hence deliver excellent speedups but sacrifice optimality and the ability to report accepting cycles on-the-fly. A combination of OWCTY and MAP (OWCTY+MAP [3]) restores "on-the-flyness", is linear-time for the class of weak LTL properties, and maintains scalability.

SWARM verification [12] consists of spawning multiple unsynchronised instances of NDFS each exploring the graph in a random way. Accepting cycles are expected to be reported faster thanks to randomised parallel search, but in the absence of such cycles parallelisation does not help. This pragmatic strategy however targets graphs that are too large in any case to be explored in reasonable time. The purpose is then to maximise the graph coverage in a given time frame and thereby increase confidence in the model.

Two recent multi-core algorithms follow the principle of the SWARM technique but deviate from it in that working threads executing NDFS are synchronised through the sharing of some state attributes. In the first one, LNDFS [13], workers share the outcome of the red (nested) search which can then also be used to prune the blue search. Since the blue flags are not shared among threads, the red searches are still invoked in the appropriate DFS postorder. The ENDFS algorithm [10] also allows the sharing of blue flags, but a sequential emergency procedure is triggered if the appropriate invocation order of the red DFS is not respected. Moreover, to maintain correctness, information on a red DFS in progress cannot be transmitted in "real time" to other threads: the states visited by a red DFS are only marked globally red after it has returned.

A thorough experimental comparison of ENDFS and LNDFS [14] led to the main conclusion that ENDFS and LNDFS complement each other on a variety of models: the larger amount of information shared by ENDFS can potentially yield a better work distribution, but LNDFS is to be preferred when ENDFS threads often launch unfruitful emergency procedures. Since this emergency procedure launches the sequential NDFS algorithm, large portions of the graph may then be revisited, in the worst case by all workers. Hence, a combination of ENDFS and LNDFS was proposed [14] to remedy the downsides of the two algorithms. The principle of that parallel algorithm (called NMCNDFS) is to run ENDFS but replace its sequential emergency procedure by a parallel LNDFS. Experiments show that this combination pays off: NMCNDFS is always at least as fast as ENDFS or LNDFS.

While NMCNDFS combines the strengths of both earlier algorithms in terms of performance, it also conjoins their memory usage. LNDFS requires $2P+\log_2(P)+1$ bits per state (2 local colours for all P workers, a synchronisation counter and a global red bit) and ENDFS $4P+3$ (2 local colours plus another 2 for the repair procedure and 3 global bits: $\{dangerous, red, blue\}$). Next to more than doubling the memory usage, the conglomerated algorithm is long and complex.

3 A New <u>C</u>ombination of Multi-Core <u>NDFS</u>

To mitigate the downsides of NMCNDFS, we present a new algorithm, CNDFS, shown in Alg. 2. Like the previous multi-core algorithms, it is based on the principle of SWARM worker threads (indicated by subscript p here), sharing information via colours stored in the visited states, here: *blue* and *red*. After randomly ($shuffle_p^{blue}$) visiting all successors (l.13–l.15), a state is marked blue at l.16 (meaning "globally visited") and causing the (other) blue DFS workers to lose the strict postorder property.

If the state s is accepting, as usual, a red DFS is launched at l.19 to find a cycle. At this point, state s is called "the seed". All states visited by $dfsRed_p$ are collected in $\mathcal{R}_p$. If no cycle is found in the red DFS, we can prove that none exists for the seed

Alg. 2 CNDFS, a new multi-core algorithm for LTL model checking

```
 1: procedure $mcNdfs(s_0, P)$ is
 2:     $dfsBlue_1(s_0) \,||\, \ldots \,||\, dfsBlue_P(s_0)$
 3:     report no-cycle
 4: procedure $dfsRed_p(s)$ is
 5:     $\mathcal{R}_p := \mathcal{R}_p \cup \{s\}$
 6:     for all $s'$ in $shuffle_p^{red}(succ(s))$ do
 7:         if $s'.cyan[p]$ then
 8:             report cycle and terminate
 9:         if $s' \notin \mathcal{R}_p \wedge \neg s'.red$ then
10:             $dfsRed_p(s')$
11: procedure $dfsBlue_p(s)$ is
12:     $s.cyan[p]$ := true
13:     for all $s'$ in $shuffle_p^{blue}(succ(s))$ do
14:         if $\neg s'.cyan[p] \wedge \neg s'.blue$ then
15:             $dfsBlue_p(s')$
16:     $s.blue$ := true
17:     if $s \in \mathcal{A}$ then
18:         $\mathcal{R}_p := \emptyset$
19:         $dfsRed_p(s)$
20:         await $\forall s' \in \mathcal{R}_p \cap \mathcal{A} : s \neq s' \Rightarrow s'.red$
21:         for all $s'$ in $\mathcal{R}_p$ do $s'.red$ := true
22:     $s.cyan[p]$ := false
```

(Prop. 1). Still, because the red DFS was not necessarily called in postorder, other (non-seed, non-red) accepting states may be encountered for which we know nothing, except the fact that they are out of order and reachable from the seed. These are handled after completion of the red DFS at l.20 by simply waiting for them to become red.

Our proof shows that in this scenario there is always another worker which can colour such a state red (Prop. 3). The intuition behind this is that there has to be another worker to cause the out-of-order red search in the first place (by colouring blue) and, in the second place, this worker can continue its execution because cyclic waiting configurations can only happen for accepting cycles. These accepting cycles would however be encountered first, causing termination and a cycle report (l.8). After completion of the waiting procedure, CNDFS marks all states in $\mathcal{R}_p$ globally red, pruning other red DFSs.

The crude waiting strategy requires some justification. After reassessing the ingredients of LNDFS and ENDFS, we found that ENDFS is most effective at parallelising the blue DFS. This is absolutely necessary since the number of blue states (all reachable states) typically exceeds the number of red states (visited by the red DFS). In ENDFS, however, sharing the blue colour often led to the expensive (memory and performance wise) sequential repair procedure [10]. We were unable to construct a correct algorithm that colours both blue and red while backtracking from the respective DFS procedures. Therefore, we now want to investigate whether the intermediate solution, using a wait statement as a compromise, leaves enough parallelism to maintain scalability.

CNDFS only uses $N+2$ bits per state plus the sizes of $\mathcal{R}$. In the theoretical worst case (an accepting initial state), each worker p could collect all states in $\mathcal{R}_p$. In our vast set of experiments (cf. Sec. 4), however, we found that the set rarely contains more than one state and never more than thousands, which is still negligible compared to $|\mathcal{S}|$. Our experiments also confirmed that memory usage is close to the expected amount.

Correctness Proving correctness comprises two parts: proving the consistency of the algorithm, i.e. CNDFS reports a cycle *iff* an accepting cycle is reachable from s_0, and *termination*. The former turned out to be easier than for our previous parallel NDFS algorithms. The wait condition in combination with the late red colouring forces the accepting states to be processed in postorder. Stated differently: a worker makes the effects of its $dfsRed_p(s)$ globally visible (via the red colouring), only after all smaller (in postorder) accepting states t have been processed by some $dfsRed_{p'}(t)$. This is expressed by Lemma 3. In Theorem 1, we finally show that, if the algorithm terminates without reporting a cycle, all accepting states must be red and consequently cannot lie on a cycle. Proof of termination was already discussed briefly and is detailed in Prop. 3.

In the following proofs, the graph colouring and the process counter of Alg. 2 are viewed as state properties of the execution. When writing $\mathit{dfsBlue}_p(s)@19$, we refer to the point in the execution at which a worker p is about to call *dfsRed* on a state s at l.19, within the execution of $\mathit{dfsBlue}_p(s)$. Graph colourings are denoted as follows: $s \in \mathit{Red}$ means that the *red* flag of s is set to true and similarly $s \in \mathit{Blue}$ means that the *blue* flag is set. For local flags we use $s \in \mathit{Cyan}_p$. Also, we use the modal operator $s \in \Box X$, to express $\forall s' \in \mathit{succ}(s){:}\, s' \in X$. We show that our propositions hold in the initial state $(\forall s \in \mathcal{S} : s \notin \mathit{Red} \wedge s \notin \mathit{Blue} \wedge \forall p \in \{1 \dots P\} : s \notin \mathit{Cyan}_p)$ and inductively that they are maintained by execution of each statement in the algorithm, considering only lines that can influence the truth value of the proposition. Here an important assumption is that all lines of Alg. 2 are executed atomically.

Lemma 1. *Red states have red successors: $\mathit{Red} \subseteq \Box \mathit{Red}$.*

Proof. Initially, there are no red states, hence the lemma holds.

States are coloured red when $\mathit{dfsBlue}_p@21$ and are never uncoloured red. The set of states $\mathcal{R}_p$ that is coloured at l.21 contains all states reachable from the seed s, but not yet red, since $\mathit{dfsRed}_p(s)$ performed a DFS from s over all non-red states. For the red states reachable from s, the induction hypothesis can be applied, hence there are no non-red states reachable from s that are not in $\mathcal{R}_p$. □

Lemma 2. *At l.20, the set $\mathcal{R}_p$ invariably contains (1) the seed s, (2) all non-red states reachable from s and also (3) all states in the set are reachable from the seed s: $\mathit{dfsBlue}_p(s)@20 \Rightarrow (s \in \mathcal{R}_p \wedge (\forall s' \notin \mathit{Red} : s \rightarrow^* s' \Rightarrow s' \in \mathcal{R}_p) \wedge (\forall s'' \in \mathcal{R}_p \Rightarrow s \rightarrow^* s''))$.*

Proof. At l.5, we have $s \in \mathcal{R}_p$. For the rest, see proof of Lemma 1. □

Lemma 3. *The only accepting state that can be coloured red at l.21 (for the first time) is the current seed s itself: $\mathit{dfsBlue}_p(s)@21 \Rightarrow (\mathcal{R}_p \cap \mathcal{A}) \setminus \mathit{Red} \subseteq \{s\}$.*

Proof. Assume $\mathit{dfsBlue}_p(s)@21$ and $\exists a \in (\mathcal{A} \setminus \{s\}) : a \in \mathcal{R}_p$. We show that $a \in \mathit{Red}$.

By Lemma 2, $\mathcal{R}_p$ contains at least s and the non-red states reachable from s. After l.20, all non-seed accepting states in $\mathcal{R}_p$ are red: $(\mathcal{R}_p \cap (\mathcal{A} \setminus \{s\})) \subseteq \mathit{Red}$. Since, $a \in \mathcal{R}_p \cap (\mathcal{A} \setminus \{s\})$, we have: $a \in \mathit{Red}$. □

Proposition 1. *The initial invocation of $\mathit{dfsRed}_p(s)$ at l.19 of Alg. 2 reports a cycle if and only if the seed s belongs to a cycle.*

Proof. $\Leftrightarrow$ is split into two cases: Case $\Rightarrow$: Every state $s' \in \mathit{Cyan}_p$ can reach the seed from $\mathit{dfsBlue}_p(s)@19$ by properties of the DFS stack. Similarly, when $\mathit{dfsRed}_p(s'')@8$, s'' is reachable from the seed s. Therefore, there is a cycle: $s'' \rightarrow s' \rightarrow^* s \rightarrow^* s''$.

Case $\Leftarrow$: assume $\mathit{dfsRed}_p(s)$ at l.19 finishes normally (without cycle report), while s lies on a cycle C. We show this leads to a contradiction. Since *dfsRed* avoids only red states (l.9), there would have to be some $r \in C \cap \mathit{Red}$ obstructing the search. The state r can only be coloured red at l.21 by a worker. W.l.o.g. we investigate the first worker $\mathit{dfsRed}_{p'}$ to have coloured r red. p' started for an $s' \in \mathcal{A}$ ($\mathit{dfsBlue}_{p'}(s')@$l.19).

Since r is not yet red, by Lemma 1 $C \cap Red = \emptyset$. Before r is coloured red, it is first stored in $\mathcal{R}_{p'}$. By Lemma 2, we also have $C \subseteq \mathcal{R}_{p'}$. Either $s' \in C$, then the cycle through s' would have been detected since $s' \in Cyan_{p'}$. Or else $s' \notin C$, and then we have $\{s\} \subseteq (\mathcal{R}_{p'} \setminus Red)$ when $dfsBlue_{p'}(s')$@21, contradicting Lemma 3. □

Proposition 2. *Red states never lie on an accepting cycle.*

Proof. Initially, there are no red states, hence the proposition holds.

When $dfsBlue_p(s)$@21, the set of states $\mathcal{R}_p$ is coloured red. The only accepting state to be colored red is the seed s (Lemma 3). By Prop. 1, this state s does not lie on an accepting cycle. Hence, Prop. 2 is preserved. □

Lemma 4. *Blue states have blue or cyan successors: $Blue \subseteq \bigcup_p \Box(Blue \cup Cyan_p)$.*

Proof. Initially there are no blue states, hence the lemma holds.

Only at l.16, states are coloured blue, after each successor t has been skipped at l.14 ($t \in Cyan \cup Blue$), or processed by $dfsBlue_p$ at l.15 (leading to $t \in Blue$). States can be uncoloured cyan (l.22), but only after they have been coloured blue (l.16). □

Lemma 5. *A blue accepting state, that is not also $Cyan_p$ for some worker p, must be red: $\forall a \in (Blue \cap \mathcal{A}) : (\forall p \in \{1 \dots P\} : a \notin Cyan_p) \Rightarrow a \in Red$.*

Proof. Assume $s \in (\mathcal{A} \cap Blue)$ and $\forall p \in \{1 \dots P\} : s \notin Cyan_p$. We show that $s \in Red$.

State s can only be coloured blue when $dfsBlue_p(s)$@16. There, it still retains its cyan colouring from l.12, it only loses this colour at l.22. But, since $s \in \mathcal{A}$, l.21 was reached and there $a \in \mathcal{R}_p$ by Lemma 2. Hence, $s \in Red$ at l.22. □

Proposition 3. *Algorithm 1 always terminates with a report.*

Proof. The individual DFSs cannot proceed indefinitely due to a growing set of red and blue states. So eventually a cycle (l.8) or no cycle is reported (l.3). However, progress may also halt due to the wait statement at l.20. We now assume towards a contradiction that a worker p is waiting indefinitely for a state $a \in \mathcal{A}$ to become red: $dfsBlue_p(s)$@20, $s \neq a$ and $a \in \mathcal{R}_p$. We will show that either a will be coloured red eventually, or a cycle would have been detected, contradicting the assumption that p keeps waiting.

By Lemma 2, a is reachable from s: $s \rightarrow^+ a$. And by l.16, $s \in Blue$. Induction on the path $s \rightarrow^* a$, using Lemma 4, tells us that: either all states are blue (1) or there is a cyan state on this path (2):

1. $a \in Blue \wedge \forall p \in \{1 \dots P\} : a \notin Cyan_p$: by Lemma 5, $a \in Red$, which contradicts the assumption that p is waiting for a to become red.
2. $\exists c \in Cyan_{p'} : s \rightarrow^+ c \rightarrow^* a$, then depending on the identity of worker p', we have:
   A) $p = p'$: but then $dfsRed_p(s)$ would have terminated on cycle detection ($C \equiv s \rightarrow^+ c \rightarrow^+ s$), except when $dfsRed_p$ did not reach c in presence of a red state lying on C. However, this would contradict Prop. 2.
   B) $p \neq p'$: we show that either p' is executing or going to execute $dfsRed_{p'}(a)$. To eventually colour state a red, worker p' must not end up itself in a waiting state: $dfsBlue_{p'}(a')$@20. First, consider the case $a' \neq a$. We also have $a' \in \mathcal{R}_p$: If

$a' \in \mathit{Red}$, then by Prop. 2 all its reachable states are red and it cannot be waiting for a non-red reachable accepting state (Lemma 2). Therefore, $a' \notin \mathit{Red}$ and since also $s \rightarrow^+ c \rightarrow^* a'$ (stack Cyan_p), we have: $a' \in \mathcal{R}_p$ (Lemma 2). Therefore, we can assume w.l.o.g. that $a = a'$ and only consider $\mathit{dfsBlue}_{p'}(a)$ @20. We can repeat the reasoning process of this proof, with $p \equiv p'$ and $s \equiv a$. But since there are finitely many workers, the chain of processes waiting for each other eventually terminates, except the hypothetical configuration of a cyclic waiting dependency, which we consider finally.

To exclude cyclic dependencies, assume $n \geq 2$ workers are simultaneously waiting for each other's seed to be coloured red at l.20. We have: $\mathit{dfsBlue}_1(s_1)$ @20 $\wedge \ldots \wedge$ $\mathit{dfsBlue}_n(s_n)$ @20 $\wedge s_2 \in \mathcal{R}_1 \wedge \ldots \wedge s_1 \in \mathcal{R}_n$. This is only possible if $s_1 \rightarrow^+ s_n \wedge \ldots \wedge s_n \rightarrow^+ s_1$, hence there is a cycle: $s_1 \rightarrow^+ \ldots \rightarrow^+ s_n \rightarrow^+ s_1$. However, this contradicts that the red DFSs (which terminate anyway) would have detected this cycle (Prop. 1). □

Theorem 1. *Alg. 2 reports an accepting cycle if and only if one is reachable from* s_0.

Proof. By Prop. 3, the algorithm is guaranteed to terminate with some report, forming the basis for two cases: Case ⇒: $\mathit{dfsRed}_p(s)$ @8 implies a cycle (Prop. 1).

Case ⇐: At l.3, we have $s_0 \in \mathit{Blue}$ and $\mathit{Cyan} = \emptyset$ by properties of DFS. Now, by Lemma 4, we have: $\forall s \in \mathcal{G} : s_0 \rightarrow^* s \Rightarrow s \in \mathit{Blue}$. Hence, all reachable accepting states must be red by Lemma 5 and do not lie on cycles by Prop. 2. □

4 Experimental Evaluation

Our previously reported experiments [15,14,13] were performed on 16-core machines. Meanwhile, in accordance with Moore's law applied to parallelism, we obtained access to a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168). The added parallelism puts extra stress on the scalability of our algorithms and therefore also forces a repeat of some of our previous reachability experiments [15]. We investigated the cause for the performance difference between various algorithms: NMCNDFS [14], CNDFS (this paper), OWCTY+MAP [5] (the best representant of parallel BFS-based algorithms [13]) and reachability from [15]. Work duplication due to overlapping stacks can cause slowdowns for all multi-core NDFS variants, as can long await cycles in CNDFS. We introduced counters to measure and study these effects. Initially, we focus on models without cycles, the hardest case for these algorithms. Later we move on to show that CNDFS exhibits the same on-the-fly performance as existing multi-core NDFS variants [14].

We have used models from the BEEM database [18].[1] From each type of model, we selected the variants with more than 9 million states. Our CNDFS algorithm is implemented in the multi-core backend of the LTSMIN model checking tool set [16], based on a dedicated scalable lock-free hash table and an off-the-shelf load balancer [15]. For a fair comparison with previous algorithms, we also implemented some NDFS optimizations [13, Sec. 4.4], *all-red* and *early cycle detection*. All-red colours a state s red, if all its successors are red after l.15 of Alg. 2; correctness follows from Prop. 2. Early cycle detection detects certain accepting cycles already in the blue search.

[1] All results are available at `http://fmt.cs.utwente.nl/tools/ltsmin/atva-2012/`.

LTSMIN 1.9[2] was compiled with GCC 4.4.2 (with optimisation -O2) and ran with: `dve22lts-mc --threads=N -s28 --state=table --strategy=name`, where `name` can be `cndfs` or `endfs,lndfs`, representing the different algorithms [14]. We used DiVinE 2.5.2 [5] as OWCTY+MAP implementation, compiled and run with equivalent parameters. Since LTSMIN reuses its next-state function, both tools are comparable [15].

4.1 Models without Accepting Cycles

In [14], we showed that NMCNDFS was the best scaling LTL model checking algorithm on 16 core machines. Hence, we started comparing plain CNDFS and NMCNDFS. Table 1 shows the average runtime of both algorithms over five runs on all benchmarks, for 1, 8, 16 and 48 cores. The performance of CNDFS is on par with that of NMCNDFS, which is impressive considering the crude waiting strategy of the algorithm.

We confirmed that the time spent at the **await** statement (l.20 in Alg. 2) is indeed less than 0.01 sec on runs with 48 cores for all models in the BEEM database. This is caused by the all-red extension, which greatly reduces work in the red DFS. Without all-red, we observed high waiting times causing speeddowns with more than 8 cores.

Additionally, we made a comparison of *absolute speedups* so as to investigate the properties of the different algorithms (Fig. 1–6). For CNDFS and NMCNDFS, we included the standard deviation of the 5 runs as error bars. As the base case for the speedup of the LTL algorithms we used CNDFS: $S_n = T_1^{\text{CNDFS}}/T_n^{algo}$, for reachability we used its own base case. We included reachability from [15] to serve as a reference point for CNDFS. We were primarily interested to see whether the scalability of CNDFS keeps up with our parallel reachability implementation. After all, sequential NDFS visits each state at most twice; once in the blue DFS and possibly once in the red DFS.

Table 1. Runtimes (sec) with NMCNDFS and CNDFS for all models

	States	NMCNDFS				CNDFS			
		1	8	16	48	1	8	16	48
anderson.6.prop2	2.9E+7	144.0	46.5	31.3	23.7	146.6	47.2	31.7	23.6
anderson.6.prop4	3.6E+7	172.9	54.1	35.8	27.1	172.9	54.3	36.2	27.3
bakery.9.prop2	1.1E+8	378.9	62.4	35.5	18.9	368.9	64.6	36.9	19.9
bopdp.4.prop3	2.4E+7	74.7	11.1	6.4	3.3	74.9	11.0	6.4	3.3
elevator.5.prop3	2.1E+8	1,387.0	272.7	154.6	67.3	1,390.8	273.3	154.2	71.2
elevator2.3.prop4	1.5E+7	134.6	25.7	15.5	8.7	136.9	25.5	15.8	8.7
lamport.7.prop4	7.4E+7	299.2	61.9	35.5	23.5	297.7	60.8	35.9	22.9
leader_election.6.prop2	3.6E+7	1,495.2	189.5	194.5	31.9	1,501.9	190.1	94.5	32.2
leader_filters.6.prop2	2.1E+8	444.2	59.5	30.4	12.4	439.0	59.5	31.0	12.8
leader_filters.7.prop2	2.6E+7	73.5	9.7	6.4	2.3	73.3	9.4	5.0	2.3
lup.4.prop2	9.1E+6	19.6	4.7	2.9	2.2	19.5	4.7	2.9	2.1
mcs.5.prop4	1.2E+8	538.3	147.0	89.9	58.2	540.3	146.5	90.2	57.1
peterson.5.prop4	2.6E+8	1,186.0	229.4	135.3	84.9	1,146.5	226.2	133.0	83.6
rether.7.prop5	9.5E+6	43.0	6.2	3.8	2.7	43.6	6.3	3.9	2.6
synapse.7.prop3	1.5E+7	37.3	5.6	3.3	2.0	37.1	5.5	3.3	1.9

[2] http://fmt.cs.utwente.nl/tools/ltsmin/ LTSmin version 2.0.

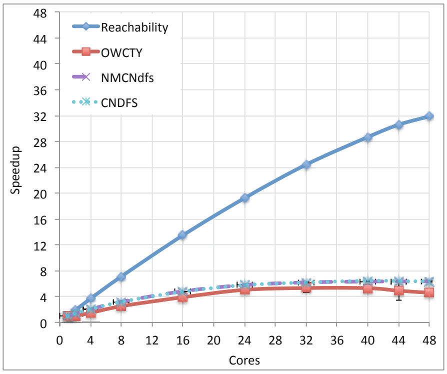


Fig. 1. Speedups of `anderson.6.prop4`

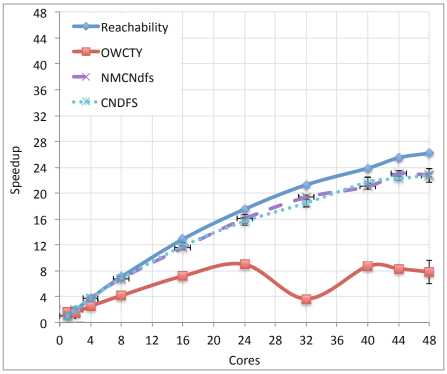


Fig. 2. Speedups of `bobp.4.prop3`

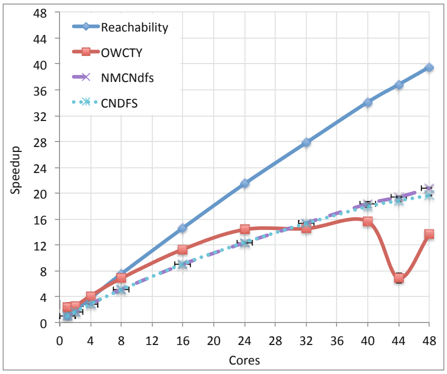


Fig. 3. Speedups of `elevator.5.prop3`

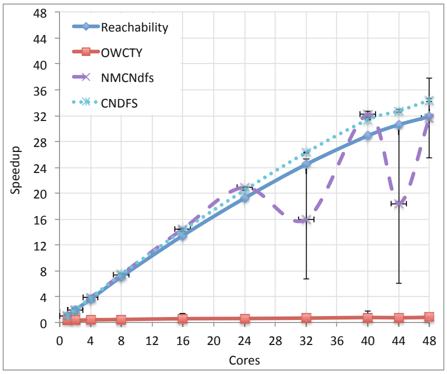


Fig. 4. Speedups of `leader_flt.6.prop2`

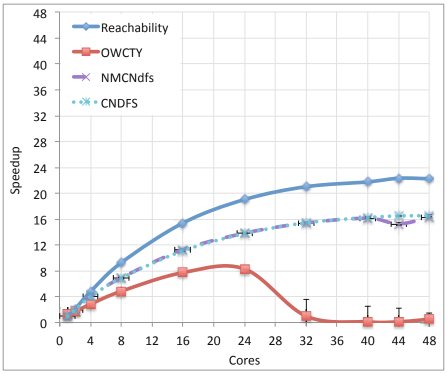


Fig. 5. Speedups of `rether.7.prop5`

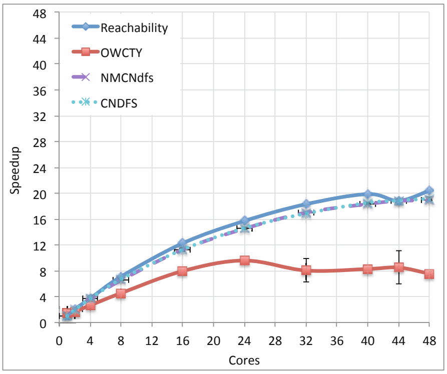


Fig. 6. Speedups of `synapse.7.prop3`

We notice that NMCNDFS and CNDFS are always faster than OWCTY+MAP. The error bars show less robust runtimes for NMCNDFS as they fluctuate greatly (e.g. `leader_filters`). Upon investigation it turned out that NMCNDFS sometimes launches a repair search even though we also fitted its ENDFS search with all-red. When only few workers enter this repair search, it cannot be parallelized. In these cases, CNDFS turns to waiting, a much better strategy, since in total it waits less than 0.01 sec. Also, reachability scales sometimes twice as good as CNDFS; `anderson` even scales 5 times better.

We investigated why the speedup of CNDFS differs from reachability. We measured the total amount of work performed by all workers. In particular, we counted for each benchmark the state count $|\mathcal{G}|$, and the numbers B_n and R_n, the total number of blue and red colourings in a run with n cores. Next, we estimate the duplicate work compared to reachability as $D_n := (R_n + B_n)/|\mathcal{G}|$. We view the reachability speedups S_n^{reach} as ideal (under the plausible assumption that maximal speedup is limited mostly by the memory bandwidth). Hence we can calculate the expected speedup $E_n^{alg} := S_n^{reach}/D_n^{alg}$ for $alg \in \{fsh, cndfs\}$ where *fsh* is CNDFS with heuristics (see below).

Table 2 compares these estimated speedups E_{48} with the actual speedups S_{48}. Note that the estimated speedups for CNDFS E_{48}^{cndfs} correspond nicely with the measured speedups S_{48}^{cndfs} for many benchmarks. Hence, we conclude that the variation in speedup is mainly caused by the degree of work duplication.

To combat work duplication, we reuse the "fresh successor heuristics" [14]. If possible, this randomly selects a successor that has not yet been visited before. It is available in the LTSMIN toolset (`--permutation=dynamic`). As a consequence, workers tend to be directed towards different regions of the state space, reducing work duplication.

These results are also shown in Table 2: D_{48}^{fsh}, E_{48}^{fsh} and S_{48}^{fsh} together with the measured amount of blue and red colourings: B_{48}^{fsh} and R_{48}^{fsh}. The heuristic approach shows quite some improvement, sometimes halving work duplication and doubling speedup

Table 2. Expected and actual speedups for CNDFS according to speedup model

	$\lvert\mathcal{G}\rvert$	B_{48}^{fsh}	R_{48}^{fsh}	S_{48}^{reach}	D_{48}^{fsh}	E_{48}^{fsh}	S_{48}^{fsh}	D_{48}^{cndfs}	E_{48}^{cndfs}	S_{48}^{cndfs}
anderson.6.prop2	3E+7	1E+8	4E+3	30.6	3.6	8.6	6.4	4.7	6.6	4.6
anderson.6.prop4	4E+7	1E+8	3E+3	31.9	3.1	10.2	6.4	4.0	8.0	5.0
bakery.9.prop2	1E+8	2E+8	4E+5	28.0	1.4	20.5	19.2	1.6	17.2	14.3
bopdp.4.prop3	2E+7	3E+7	6E+5	26.2	1.3	20.0	22.8	1.8	14.6	15.5
elevator.5.prop3	2E+8	4E+8	2E+3	39.5	1.9	21.0	19.5	3.2	12.5	9.0
elevator2.3.prop4	1E+7	3E+7	2E+6	33.2	2.0	16.3	15.8	5.3	6.3	8.0
lamport.7.prop4	7E+7	1E+8	6E+4	30.5	1.7	17.6	13.3	1.9	15.8	10.4
leader_el.6.prop2	4E+7	4E+7	4E+4	40.5	1.0	40.4	46.6	1.0	40.3	39.5
leader_filt.6.prop2	2E+8	2E+8	7E+5	31.9	1.0	31.6	34.4	1.0	30.7	29.9
leader_filt.7.prop2	3E+7	3E+7	1E+5	27.6	1.0	27.4	31.9	1.0	26.9	27.8
lup.4.prop2	9E+6	2E+7	4E+3	17.7	2.5	7.1	9.7	4.6	3.8	6.3
mcs.5.prop4	1E+8	3E+8	1E+4	34.4	2.2	15.7	9.5	2.7	12.6	7.3
peterson.5.prop4	3E+8	4E+8	8E+5	34.1	1.6	20.9	13.9	1.9	18.3	11.0
rether.7.prop5	1E+7	2E+7	1E+5	22.3	1.9	11.9	16.5	2.4	9.2	14.3
synapse.7.prop3	2E+7	2E+7	1E+2	20.4	1.1	17.9	19.2	1.2	17.0	18.6

(see `elevator`). Still we see duplications as high as 3.6 (see `anderson`). Note that the earlier speedups in Fig. 1–6 already include the benchmarks with this heuristic.

We expect that in the near future, the number of cores in many-core systems will still grow. Will this increase work duplication and put a limit on speedup of CNDFS? To give an indication, we plotted the increase of work duplication with a growing number of cores with fresh successor heuristics (Fig. 7). The increase is sub-linear, so we expect that speedups will be maintained on larger many-core systems with similar architecture and scaling bandwidth characteristics.

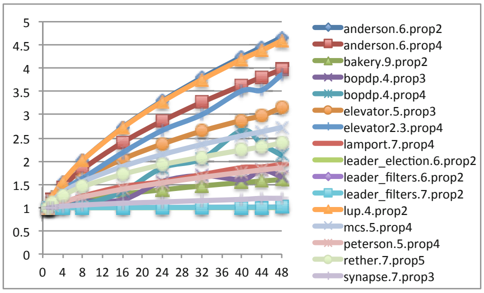


Fig. 7. Work duplication per core per model

Finally, we note that the size of the input has a small yet significant effect on the amount of work duplication; models with higher state count have less duplication.

4.2 Models with Accepting Cycles

In [14], we experimented thoroughly to investigate the “on-the-flyness” of SWARM NDFS and LNDFS. We noticed that the benefits of independent SWARM verification is limited, on average only yielding a speedup of 2-8 on 16 core machines. LNDFS however yielded speedups from 4 to 14. Combined with the fresh successor heuristic

Table 3. On-the-fly behavior of parallel LTL algorithms

		1 core	48 core				OWCTY+MAP	
		NDFS	LNDFS		CNDFS		1 core	48 core
	model	Rand.	Rand.	Fsh	Rand.	Fsh.	Static	Rand.
Runtimes (sec)	anderson.8.prop3	36.4	4.0	1.2	4.1	0.2	2858.8	1433.2
	bakery.7.prop3	3.2	0.4	0.2	0.3	0.2	2.2	5.2
	bakery.8.prop4	15.7	0.6	0.3	0.6	0.3	73.4	14.3
	elevator2.3.prop3	8.4	1.4	0.2	1.4	0.2	432.3	192.5
	extinction.4.prop2	2.2	0.1	0.1	0.1	0.1	1.8	1.7
	peterson.6.prop4	29.1	0.6	0.5	0.9	0.5	668.4	705.7
	szymanski.5.prop4	1.7	1.4	0.1	1.3	0.2	2.1	376.4
Speedups	anderson.8.prop3		9.1	31.1	8.8	175.0		2.0
	bakery.7.prop3		8.7	18.3	10.9	21.2		0.4
	bakery.8.prop4		28.3	51.1	26.2	48.9		5.1
	elevator2.3.prop3		6.0	51.5	5.9	52.1		2.2
	extinction.4.prop2		30.4	32.1	18.5	28.8		1.0
	peterson.6.prop4		46.1	59.8	33.0	62.4		0.9
	szymanski.5.prop4		1.2	12.0	1.3	10.9		0.0

speedups became often superlinear. This is not surprising [19], because we verified that in those cases there are many cycles, distributed evenly over the state space.

We performed the same experiments again with CNDFS on a 48 core machine. The results in Table 3 show that CNDFS exhibits the same desirable on-the-fly behaviour as LNDFS, scaling up to 48 cores. We conclude that our new multi-core CNDFS algorithm scales well also for models with bugs.

For completeness, we also included the runtimes and speedups with OWCTY+MAP in the table. While the heuristic on-the-fly behavior seems to work well for some models, for others it does not. It must however be mentioned that the on-the-fly capabilities of this algorithm have recently been improved by changing its exploration order to be more DFS-like [1]. In [1], performance is reported on par with the LNDFS algorithm. Unfortunately, we do not have the means (a GPGPU) to reproduce any results here.

4.3 Counterexample Length

Lengthy counterexamples are hard to study even with good model checking tools. Therefore, finding short counterexamples is quite an important property of model checking algorithms. Strict BFS algorithms deliver minimal counterexamples, while DFS algorithms can yield very long ones. Once the strict BFS/DFS order is loosened, these properties can be expected to fade. This is exactly what both OWCTY+MAP and CNDFS do. We studied the length of the counterexamples that these algorithms produce.

For this purpose, 45 models with counterexamples were selected from the BEEM database, all algorithms run 5 times, and computed the average counterexample length and standard deviation. The results are summarised in scatter plots with bars representing the standard deviation. Fig. 8 compares randomised sequential NDFS (vertical axis) against sequential OWCTY+MAP (horizontal axis). Fig. 9 compares the results of CNDFS with fresh successor heuristic (fsh) against OWCTY+MAP on 48 cores.

In the sequential case, most bars are above the equilibrium so, as expected, NDFS produces long counterexamples of variable size compared to OWCTY+MAP (which we could not randomise). The parallelism of a 48 core run, however, greatly stabilises and

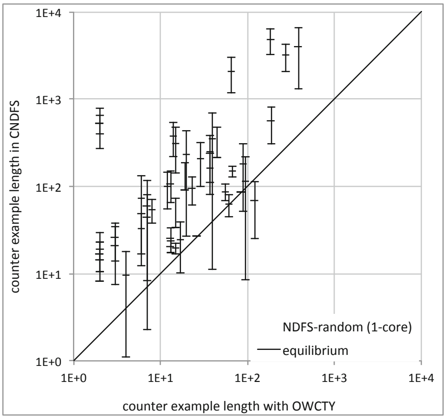


Fig. 8. NDFS vs OWCTY+MAP (1 core)

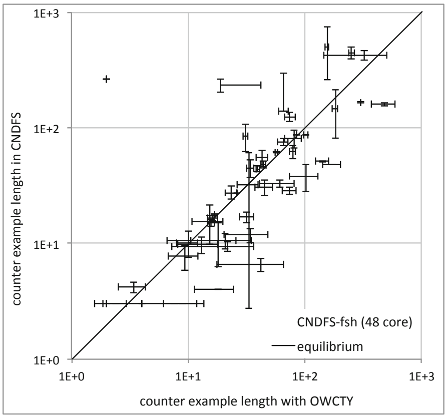


Fig. 9. Fsh vs OWCTY+MAP (48 cores)

reduces counterexample lengths for CNDFS, while the randomness added by parallelism introduces variable results for OWCTY+MAP (horizontal bars). In many cases, CNDFS counterexamples become shorter than those of OWCTY+MAP, a surprising result considering the BFS-like order of this algorithm. The one extreme outlier in this case is the `plc.4` model. All our NDFS algorithms consistently find a counterexample of length 216, while OWCTY+MAP finds one of length 2!

5 Conclusion

We presented CNDFS, a new multi-core NDFS algorithm. It can detect accepting cycles on-the-fly, and its worst case execution time is linear in the size of the input graph. We showed that CNDFS is considerably simpler than its predecessor NMCNDFS, because of the deep integration of ENDFS and LNDFS. Experiments show that CNDFS delivers performance and scalability similar to its predecessors, but achieves this more robustly. Hence CNDFS is currently the fastest multi-core LTL model checking algorithm in practice. Moreover, CNDFS halves the memory requirements per state per worker thread; an important factor since the total number of cores keeps growing.

Experiments revealed that the main bottleneck for perfect scalability of CNDFS is currently the work duplication due to overlapping stacks. Forcing workers to favour "fresh" successor states already decreases duplication. The same experiments indicate that work duplication grows only linearly in the number of cores, and decreases for larger input sizes. From this we conjecture that CNDFS will scale even beyond 48 cores.

CNDFS shares global information only during or even after backtracking, which leads to potential work duplication. In the worst case, every worker could visit the whole graph, blocking any speedup. During our extensive experiments with the entire BEEM database we have not found such cases. However, we did observe work duplication of factor 3 on 48 cores, so there is room for improvement.

Designing a provably scalable, linear-time algorithm remains an open question. Such an algorithm should cause negligible duplicate work and avoid synchronisation by await statements. So far, we have not been able to come up with a correct algorithm without await statements or a repair procedure. An improvement might be to invent a smart work stealing scheme, in which workers can cooperate instead of waiting.

Finally, we demonstrated that counterexamples in CNDFS become shorter with more parallelism, even shorter than counterexamples in parallel BFS-based OWCTY+MAP. This is an interesting and desirable property for a model checking algorithm. It is intriguing that our parallel DFS based algorithm shows good scalability and short counterexamples, usually attributed to BFS algorithms, while still maintaining the linear-time and on-the-fly properties expected from a DFS algorithm.

References

1. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. Journal of Parallel and Distributed Computing (2011)
2. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: ASE 2003, pp. 106–115. IEEE Computer Society (2003)

3. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 407–425. Springer, Heidelberg (2009)
4. Barnat, J., Brim, L., Ročkai, P.: Scalable shared memory LTL model checking. STTT 12(2), 139–153 (2010)
5. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: PDMC 2010, pp. 4–7. IEEE (2010)
6. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
7. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection (Set Based Approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
8. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
9. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
10. Evangelista, S., Petrucci, L., Youcef, S.: Parallel Nested Depth-First Searches for LTL Model Checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 381–396. Springer, Heidelberg (2011)
11. Geldenhuys, J., Valmari, A.: Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
12. Groce, A., Holzmann, G.J., Joshi, R.: Swarm Verification Techniques. Transactions on Software Engineering 37(6), 845–857 (2011)
13. Laarman, A.W., Langerak, R., van de Pol, J.C., Weber, M., Wijs, A.: Multi-Core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
14. Laarman, A.W., van de Pol, J.C.: Variations on multi-core nested depth-first search. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 13–28 (2011)
15. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Sharygina, N., Bloem, R. (eds.) FMCAD 2010, Lugano, Switzerland, USA. IEEE Computer Society (October 2010)
16. Laarman, A., van de Pol, J., Weber, M.: Multi-core lTSmin: Marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
17. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011)
18. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
19. Rao, V.N., Kumar, V.: Superlinear Speedup in Parallel State-space Search. In: Kumar, S., Nori, K.V. (eds.) FSTTCS 1988. LNCS, vol. 338, pp. 161–174. Springer, Heidelberg (1988)
20. Reif, J.H.: Depth-first Search is Inherently Sequential. Information Processing Letters 20(5), 229–234 (1985)
21. 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)
22. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344. IEEE Computer Society (1986)

An Experiment on Parallel Model Checking of a CTL Fragment*

Rodrigo T. Saad, Silvano Dal Zilio, and Bernard Berthomieu

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse France
Univ de Toulouse, LAAS, F-31400 Toulouse, France
{rsaad,dalzilio,Bernard.Berthomieu}@laas.fr

Abstract. We propose a parallel algorithm for local, on the fly, model checking of a fragment of CTL that is well-suited for modern, multi-core architectures. This model-checking algorithm benefits from a parallel state space construction algorithm, which we described in a previous work, and shares the same basic set of principles: there are no assumptions on the models that can be analyzed; no restrictions on the way states are distributed; and no restrictions on the way work is shared among processors. We evaluate the performance of different versions of our algorithm and compare our results with those obtained using other parallel model checking tools. One of the most novel contributions of this work is to study a space-efficient variant for CTL model-checking that does not require to store the whole transition graph but that operates, instead, on a reverse spanning tree.

1 Introduction

Several model-checking methods address the state-explosion problem from a purely algorithmic perspective, for instance with the use of abstractions on the set of states (such as stubborn sets or symmetries) or symbolic techniques. Despite the fact that considerable progress has been made at the theoretical level, there are still classes of systems that cannot benefit from these advanced methods, like for example models that rely on real time constraints or on dynamic priorities. In this case, it is interesting to take advantage of the computation power—and increased amount of primary memory—provided by multi-processor and multi-core computers in order to handle very large state spaces.

In this paper, we propose a parallel algorithm for local, on the fly, model checking of a fragment of CTL that is well-suited for modern, multi-core architectures. We target shared-memory computers with a moderate number of cores (say 4 to 64) operating on a large shared memory space (typically from 16GB to 1TB of RAM). This description fits many available mid-range servers, but is also quite close to tomorrow's mainstream desktop computers.

Our model-checking algorithm takes advantage of a parallel state space construction algorithm defined in previous work [11] and shares the same principles.

* This work was partially supported by JU Artemisia project CESAR, AESE project Topcased and Région Midi-Pyrénées.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 284–299, 2012.

First, we make no specific assumptions on the models that can be analyzed (we only assume that we know how to compute the successors of a given state and test them for equality). Second, we put no restrictions on the way states are distributed: in our solution, every process keeps a local share of the global state space and we do not rely on an a-priori static partition of the states. Finally, we put no restrictions on the way work is shared among processors; this means that our algorithm plays nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

In this paper, we extend this state space construction algorithm by adding model checking capabilities. While the leading parallel model-checking tools are based on LTL model-checking, (see Sect. 5) we advocate the use of CTL. The choice of CTL derives from a number of desirable properties that we want for our parallel algorithm.

First, it must take advantage of parallelism and be compatible with our parallel state space generator. For this reason, the logic should preferably be *branching time* rather than linear time since model checking algorithms for linear time logics are strongly tied to depth-first-search (dfs) exploration techniques; and that dfs algorithms are "inherently sequential" (they belong to the class of P-complete problems [8,3]). Parallel techniques for LTL model checking have been quite investigated however [1,6] so we will compare our approach with these.

Secondly, we want an *on-the-fly* algorithm; only states essential for answering the model-checking problem should be enumerated. For this reason, model checking should be *local* rather than global; properties will be interpreted at the initial state rather than at all states.

Next, because the full state space may have to be enumerated (e.g. when checking a property that is true), we want it to be space-efficient. Hence, we shall accept that a small amount of information is recomputed every time it is needed rather than kept in storage.

For all these reasons, we decided to select a fragment of Computation Tree Logic (CTL) with state subformulas restricted to atomic propositions. This fragment strictly includes the logic used by popular tools like Uppaal. Though obviously less expressive than CTL, it implements a good trade-off between expressiveness and cost of verification when used to check large state spaces. While not yet implemented in our tool, it is possible to adapt our algorithm to model-check full CTL; more expressive logics will be considered in further work.

Contributions. We follow the classical approach of Clarke and Emerson [2] for CTL model-checking. During model-checking, we label each state of the system with the subformulas that are true at this given state. Labels are computed iteratively until we reach a fix-point, that is until we cannot add new labels. We consider two variants of this algorithm that differ by the amount of information on the transition relation that is stored. Both variants have two passes: a *forward pass* performs a constrained exploration of the state graph in which we start labeling each state with local information; a second, *backward pass*, propagates information towards the root of the state graph and checks if the resulting graph admits an infinite path.

In the first version of the algorithm, that we call *RG* (for reverse graph), we assume that, for every reachable state, we have a constant time access to the list of all its "parents". In other words, we store the reverse transition relation of the state space. Algorithm RG is simply a parallel version of the algorithm in [2] that uses our parallel state space construction method. Our experimental results show that, even with this simple approach, we obtain a very good parallel implementation (that is with a good speedup) and a very good model-checking tool (that is with a good execution time when compared with other tools on a similar setup).

In the second version, we assume that we have direct access to only one of the parents, meaning that we may have to recompute some transitions dynamically. We call this second version *RPG*, for *reverse parental graph.* The advantage of the RPG version is to save memory space. Indeed, if we use the symbol S to denote the number of reachable states, the RG algorithm has a space complexity in the order of $O(S^2)$ in the worst-case, while it is of the order of $O(S)$ for the RPG version. We show, in our benchmarks, that the RPG version allows bigger examples to be computed without sacrificing execution time.

Outline of the paper. In the next section, we summarize the parallel state space generation algorithm [11] that is used in our work. The model checking algorithms are defined in Sect. 3 using pseudo-code, while our parallel implementation is described in Sect. 4. Before concluding, we discuss the related work and compare our performances with the DiVinE[1] tool, a state of the art parallel model checker for LTL.

2 Parallel State Space Generation

State space generation is often a preliminary step for model checking behavioral formulas. This is a very basic operation: take a state that has not been explored; compute its successors and check if they have already been found before; iterate until there are no more new states to explore. Hence, a key point for performance is to use an efficient data structure for storing the set of generated states and for testing membership in this set. In [11], we propose an algorithm for parallel state space construction based on an original concurrent data structure, called a *localization table* (*LT*), that aims at improving spatial and temporal balance.

This approach is close in spirit to algorithms based on distributed hash tables, with the distinction that states are dynamically assigned to processors; i.e. we do not rely on an a-priori static partition of the state space. In our solution, every process keeps a share of the global state space in a local data structure. Data distribution and coordination between processes is made through the *LT*, that is a lockless, thread-safe data structure. The localization table is used to dynamically assign newly discovered states and can be queried to return the identity of the processor that owns a given state. With this approach, we are able to consolidate a network of local state repositories into an (abstract) distributed one without sacrificing memory affinity—data that are "logically connected" and physically close to each other—and without incurring performance costs associated to the use of locks to ensure data consistency.

The performance of our state space construction algorithm was evaluated on different benchmarks and compared with the results obtained using other solutions proposed in the literature. A first implementation of our algorithm showed promising results as we observed performances that are consistently better—both in terms of absolute speedup and memory footprint—than other parallel algorithms. For example, this algorithm does consistently better than algorithms based on the use of static partitioning or a similar approach based on the concurrent hash map implementation provided in the Intel Threading Building Blocks (TBB) library.

State space generation has a direct impact on the performance of the model-checking algorithm. For one thing, state space generation alone is enough to model-check reachability properties (of the form $A\Box(\phi)$). Moreover, for more complicated properties (see our benchmark results in Sect. 4), the time needed to explore the state space still makes up a big part of the model checking time.

3 Parallel Model Checking for a CTL Fragment

We build our model checking algorithm on top of the parallel state space generation algorithm of [11], described in the previous section. Our other design choices follow from our goal to target models with very large state spaces. More particularly, we choose to restrict ourselves to a fragment of CTL and to disallow the nesting of operators; that is, every subformula—denoted $\phi, \psi, \ldots$—is a (boolean composition of) atomic propositions.

The logic used for model-checking essentially relies on three operators: *Exist Until* (EU), $E\,(\psi \cup \phi)$, that is true if there exists a trace (a path) in the state space such that ψ has to hold until, at some position, ϕ holds; Always Until (AU), $A\,(\psi \cup \phi)$, that is true if the "until condition" holds on every trace; and finally the *leadsto* formula, $\psi \leadsto \phi$, that is true if, for every trace, whenever ψ holds then necessarily ϕ will hold later. The last property can be expressed as $A\Box(\neg\,\psi \vee A\Diamond(\phi))$ in CTL. From the interpretation given in Table 1, we see that these operators define an expressive fragment of CTL (and also LTL).

Model-checking procedures for these operators will be described in Sections 3.2 to 3.4. In our implementation, we consider two variants—RG and RPG—of the algorithms. Both versions are based on two elementary phases: (1) a forward constrained exploration of the state graph using the state space construction discussed in Sect 2; followed by (2) a backward traversal and label propagation phase ensuring that the resulting graph is acyclic.

The backward traversal phase is only needed for AU and leadsto formulas, since checking EU formulas amounts to performing a constrained exploration of the state space (for instance, the formula $A\Box(\phi)$ is true if no state satisfies $\neg\phi$, which can be checked during the exploration phase). Consequently, our algorithm is not completely on-the-fly for these cases because the presence of a cycle is detected after the (constrained) state space is constructed, delaying the discovery of an invalid path. The last column of Table 1 indicates, for each formula, whether the backward phase is necessary.

Formulas	Interpretation in CTL	Forward	Backward
$E\ (\psi \cup \phi)$	$E\ (\psi \cup \phi)$	x	
$A\ (\psi \cup \phi)$	$A\ (\psi \cup \phi)$	x	x
$E\Diamond(\phi)$	E (True $\cup\, \phi$)	x	
$A\Diamond(\phi)$	A (True $\cup\, \phi$)	x	x
$E\Box(\phi)$	$\neg A\Diamond(\neg\phi)$	x	x
$A\Box(\phi)$	$\neg E\Diamond(\neg\phi)$	x	
$\psi \rightsquigarrow \phi$	$A\Box(\neg\psi \vee A\Diamond\phi)$	x	x
$A\Box A\Diamond(\phi)$	$true \rightsquigarrow \phi$	x	x

Fig. 1. List of Supported Formulas

3.1 Concepts and Notations

We assume that we perform model-checking on a Kripke System $KS(S, R, s_0)$. We will use, interchangeably, the notation KS for the Kripke structure (S, R, s_0) and G for the directed graph $G(S, R)$, also called the *state graph*. In the RPG version of our algorithm, we make use of the *Parental Graph* of a Kripke System, that is a reverse spanning tree of the (currently computed) state graph.

Definition 1 (Parental Graph). *The directed graph, $PG(V_p, E_p)$, is a parental graph of $G(V, E)$ if: (1) PG if a subgraph of G that has the same vertices, that is $V_p = V$ and $E_p \subseteq E$, and (2) for every vertex $v \in V$, if v is not the root of G then v has an in-degree of one in PG.*

A simple way to obtain a parental graph, PG, when exploring the state graph, G, is to keep for every state, s, a vertex to the state in G that was used to generate s (and forget the others). The parental graph has nice properties. If PG is a parental graph of G and G is acyclic then so is PG. Moreover, the set of leaves of PG subsumes that of G; a leaf of G is necessarily a leaf of PG.

In the remainder of the text, the expression $|S|$ is used to denote the cardinality of S (the number of reachable states), while $|R|$ is the number of transitions. We assume that every state $s \in S$ is labeled with a value, denoted suc(s), that records the out-degree of s in KS. The value of suc(s) is set during the forward exploration phase. Initially, suc(s) is the cardinality of the set of successors of s in KS, that is suc(s) $= |\{s' \mid s\ R\ s'\}|$. We decrement this label during the backward traversal of the state graph; when the value of suc(s) reaches zero, we say that s is *cleared* from the state graph. In our pseudo-code, we use the expression suc(s).dec() to decrement the value of the label suc for the state s in KS, and the expression suc(s).set(i) to set the label of s to some integer value i.

When we deal with the reverse parental graph version of our algorithm, we assume that we implicitly work with one particular parental graph of KS, denoted PKS. In this case, we assume that every state $s \in S$ is also labeled with a value, denoted sons(s), that records the out-degree of s in KS. We also label each state $s \in S$ with a state, denoted father(s), that is the (unique) predecessor of s in PKS. (The label father(s) makes sense only if s is not the initial state, s_0, of KS.) Initially, the value of sons(s) is set to zero. The label will be incremented during the forward exploration, when we build PKS (that is, we select the transitions from KS that will be stored in PKS). This operation is denoted sons(s).inc() in our pseudo-code. We will decrement the value of sons(s) during the backward traversal phase.

3.2 Checking EU Properties

Checking EU properties for the initial state is standard, except that we perform the forward phase concurrently, on all states; this can be done on the fly in a single, forward pass. To check the formula $E\,(\psi \cup \phi)$, we explore the state space stopping along a path when a state is found such that (1) ϕ holds, or (2) $\neg\psi \wedge \neg\phi$ holds. In the first case, the algorithm reports success. In the second case the path cannot lead to a state making the property true; exploration continues over the set of open paths until (1) or (2) holds. The property is false if (1) never holds or true otherwise. The check function is the same for the two versions of our algorithm, whether based on the reverse graph or the reverse parental graph data structure.

```
function BOOL check_a($\psi$ : pred, $\phi$ : pred, $s_0$ : state)
    Stack A ← new Stack($\emptyset$) ;
    // Start with the forward exploration
    if forward_check_a($\psi$ , $\phi$, $s_0$, A) then // If all forward constraints are respected
        return backward_check_a($s_0$, A)        //start the backward phase
    else return FALSE // We found a problem during the forward exploration
```

Listing 1.1. Algorithm for the formula $A\,(\psi \cup \phi)$—function check_a

3.3 Checking AU Properties

For checking the formula $A\,(\psi \cup \phi)$, as for EU properties, we stop exploring a path when we find a state such that (1) ϕ holds or (2) $\neg\psi \wedge \neg\phi$ holds. If an occurrence of case (2) is found, we know at once that the property is false. Otherwise, we start the backward traversal phase in order to detect cycles. Indeed, the property $A\,(\psi \cup \phi)$ is false if there is an infinite path of states (starting from s_0) that all obey ψ. We call this second phase the *clearing phase*, because it consists of recursively removing leaf nodes from the graph. This process ends either when the only remaining state is the initial state (meaning that the property is true), or when no states with out-degree zero can be found (in which case we know that there is a cycle). The validity of this method follows from the fact that a finite Directed Acyclic Graph (DAG) has at least one leaf.

We give the pseudo-code for checking $A\,(\psi \cup \phi)$ in Listing 1.1. The inputs are the atomic properties ψ and ϕ and the initial state s_0. The algorithm makes use of a stack A to collect the states at which ϕ holds during the forward exploration phase. The procedure uses two auxiliary functions, forward_check_a and backward_check_a, that depends on the version of the algorithm. We start by defining these helper functions for the Reverse Graph version.

Algorithm for the Reverse Graph version (RG). We give the pseudo-code for the function forward_check_a (for the RG version) in Listing 1.2. The last parameter of this function, A, is a stack that is used to collect the leaf nodes of the state graph, that is the states where ϕ holds. These states are the starting points in our backward traversal of the graph.

During the forward exploration phase (function forward_check_a) each state s is labelled with its number of successors in the initial state graph (the Kripke structure). During the backward traversal phase (function backward_check_a), this label is decremented each time a successor of s is removed; decrementations are done in parallel. Intuitively, a state can be removed as soon as it is cleared. We never actually remove a state from the graph. Instead, when a processor changes the label of a state s to 0, we also decrement the labels of all the parents of s in the graph. This motivates the choice to store the reverse of the transition function in the data structure.

In the function backward_check_a, see Listing 1.2, we start by clearing all the states in A which are, by construction, the states s such that suc(s) is nil. When a state is cleared, we decrement the labels of all its parents ($suc(s').dec()$) and check which ones can be cleared ($suc(s') == 0$). The algorithm stops if the initial state, s_0, can be cleared or if there are no more states to update.

```
 1  function BOOL forward_check_a($\psi$ : pred, $\phi$ : pred, $s_0$:state, A:Stack)
 2      Set S ← new Set($s_0$) ; Stack W ←new Stack($s_0$) ;
 3      while (W is not empty) do
 4          s ← W.pop();
 5          if (s $\models \phi$) then
 6              suc(s).set(0) ; // We clear state s from KS
 7              A.push(s)
 8          elsif (s $\models \psi$) then // We tag s with its number of successors
 9              suc(s).set(number of successors of s in KS) ;
10              if (suc(s) = 0) // Check if s is not a dead state
11                  return FALSE
12              forall s' successor of s in KS do // and continue the exploration
13                  if (s' $\notin$ S) then
14                      S ← S ∪ {s'} ; // s' is a new state
15                      W.push(s')
16          else return FALSE
17      return TRUE
18
19  function BOOL backward_check_a($s_0$ : state, A : Stack)
20      while (A is not empty) do
21          s ← A.pop() ;
22          if (s = $s_0$) then // The property is true if
23              return TRUE     // we reach the initial state
24          forall s' parent of s in KS do      // Otherwise we check if the
25              suc(s').dec() ;                 // predecessors of s can be cleared
26              if (suc(s') = 0) then A.push(s')
27      return FALSE
```

Listing 1.2. Forward and backward exploration for $A(\psi \cup \phi)$ with Reverse Graph

Algorithm for the Reverse Parental Graph version (RPG). The function for the RPG version is only slightly more complicated, because we need to recompute some successors in the transition relation: we can only access one of the parents of a state in constant time (which we call the *father* of the state). The pseudo-code for the forward exploration phase (function forward_check_a) is essentially the same as in Listing 1.2; this is why it is omitted here. Compared to the RG version, we only need to add two additional statements when adding a new state (around line 14 in Listing 1.2): assuming that the state s

is generated from a state s', we set the value of the father for the newly generated state ($father(s).set(s')$) and increment the number of sons of the father $sons(s).inc()$). This information is used during the backward traversal to track non cleared leaves.

We give the pseudo-code for the backward traversal phase in Listing 1.3. During this phase, we follow the parental graph structure to "propagate" the cleared states toward the root of the state graph. The algorithm alternates between two behaviors, *clearing* and *collecting*. The *clearing* behavior is similar to the pseudo-code for the RG algorithm, with the difference that we decrement only the father of a state and not all the predecessors. When there are no more labels to decrement—and if the root state is not yet cleared—the algorithm starts looking for states that can be cleared. For this, we test all the states s such that $sons(s) == 0$; that is, such that all the sons of s have been cleared (in the parental graph). In this case, to check if s can be cleared, we have to recompute all its successors in KS and check whether they have also been cleared (if their suc label is zero).

```
function BOOL backward_check_a($s_0$ : state , A : Stack)
   over ← FALSE
   while (not over)
      while (A is not empty) do
         //Clearing
         s ← A.pop() ;
         if (s = $s_0$) then // The property is true if
            return TRUE     // we reach the initial state
         s' ← father(s) ;     // Otherwise we check if
         sons(s').dec() ;   // the father of s can be cleared
         suc(s').dec() ;
         if (suc(s') = 0) then A.push(s')
      //Collecting: if we have no more states to clear in A we try to find
      // candidates among the states with no children in PKS
      forall s such that sons(s) = 0 and suc(s) $\neq$ 0 in KS do
         if test(s) then
            suc(s).set(0) ;
            A.push(s)
      if (A is empty) then
         over ← TRUE //No good candidate was found, end backward search
   return FALSE

function BOOL test(s : state)
   forall s' successor of s in KS do
      if suc(s') $\neq$ 0 then
         return FALSE // at least one successor is not cleared
   return TRUE
```

Listing 1.3. Backward exploration for $A\,(\psi \cup \phi)$ with Reverse Parental Graph

The advantage of this strategy is that we do not have to consider all the states in the graph but just a subset of them. Indeed, we know that if KS is a acyclic (is a DAG) then PG has at least one leaf that is also a leaf in G [9]. Hence, this subset is enough to test the presence of a cycle. Conversely, the drawback of this approach is that we may try to clear the same vertex several times, which may be time consuming.

3.4 Checking Leadsto Properties

To check the formula $\psi \leadsto \phi$, we need to prove that no cycle can be reached from a state where ψ holds, without first reaching a state where ϕ holds. Indeed, otherwise, we can find an infinite path where ϕ never holds after an occurrence of ψ. Figure 2 shows an example of graph for which the formula is valid.

This observation underlines the link between checking the formula $\psi \leadsto \phi$ locally—for the initial state—and checking the validity of $A\Diamond(\phi)$ globally—at every state where ψ holds. As a consequence, we can use an approach similar to the one used for AU properties in the previous section. The main difference is that, instead of clearing the initial state, we have to clear all the states where ψ holds. Hence, the pseudo-code for the leadsto formulas is similar to that of AU formulas (this is why it is omitted here), the main difference is in the termination condition: the function returns true if all the states where ψ holds are cleared.

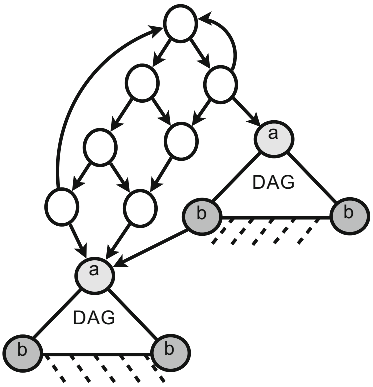


Fig. 2. Formula $a \leadsto b$.

3.5 Correctness and Complexity of Our Algorithms

Proofs of correctness (termination, completeness and soundness) and a precise study of the complexity of our algorithms can be found in [9]. We just discuss here the worst-case complexity in the sequential case, and for formulas $A\,(\psi \cup \phi)$. The results for this case can be generalized to our whole logic. (Inside asymptotic notations, we use the symbols S and R when we really mean $|S|$ and $|R|$.)

The algorithm given in Sect. 3.3 may inspect every state in the Kripke Structure KS and, for every transition, it may update one label. Therefore, its worst-case time complexity is in the order of $O(S + R)$ for the RG algorithm. The complexity is higher in the RPG version since, for each altered state, we may have to recompute the successors for all the reachable states s such that sons(s) is nil. Hence, solving a simple recurrence, we can prove that the time complexity is in the order of $O(S \cdot (R - S))$ for the RPG version. Since the number of transitions in KS is bounded by $|S|^2$, we obtain a complexity in the order of $O(S^2)$ for the RG version and of $O(S^3)$ for the RPG variant. Concerning the space complexity, the RPG version is in the order of $O(S)$, while the RG version is linear in the size of the graph, that is in the order of $O(S + R)$ (or $O(S^2)$).

We show in our experiments that the decision to favor "space-efficiency" (in the case of the RPG version) is quite interesting. In particular, on some examples, the RPG version may run faster than the RG version because it needs to "write less information" in main memory, an effect that is not visible if we only look at the theoretical complexity. More importantly, memory is one of the key resources used during model-checking. Indeed, it is common to exhaust the available memory during verification.

4 Implementation and Experimental Results

In the code presented in Sect. 3, no underlying computational model was make precise. The code can be easily adapted to a Parallel RAM model, following a Single Program Multiple Data (SPMD) programming style. In this section, we discuss the details surrounding the parallel implementation of our algorithms, then report on a set of experiments performed to evaluate their effectiveness.

4.1 Parallel Implementation of our Algorithm

In a SPMD context, all processing units will execute the same functions (those in Listings 1.1–1.3). Following this approach, both the (forward) exploration phase and the (backward) cycle detection phase are easily parallelized. For the model-checking functions themselves—for instance check_a—we only need to synchronize the termination of the forward exploration with the start of the backward label propagation. At each point, a processing unit can terminate the model-checking process if it can prove (or disprove) the validity of the formula before the end of the exploration phase. Actually, most of the burden of parallelizing our algorithm is hidden inside the use of our specialized, lock-free data structures.

We consider a shared memory architecture where all processing units share the state space (using the mixed approach presented in [11]) and where the working stacks are partially distributed (such as the stacks W and A used in our pseudo-code). For most of our pseudo-code, it is enough to rely on atomic compare and swap primitives to protect from parallel data races and other synchronization issues; typically, compare-and-swap primitives will be used when we need to test the value of a label or when we need to update the label of a state (for instance with expressions like sons(s).dec()). Together with the compare-and-swap primitive, we use our combination of distributed, local hash tables with a concurrent localization table to store and manage the state space.

For the RG version of the algorithm, we can ensure the consistency of our algorithm by protecting all the operations that manipulate a state label. (We made sure, in our pseudo-code, that every operation only affects one state at a time.) The parallel version of RPG is a bit more involved. This problem is related to the behavior of the *collecting* operations of the backward exploration (see the comment on line 14 of Listing 1.3)—and in particular the function test—that needs to check all the successors of a state to see if they are cleared. First, this code is not atomic and it is not practical to put it inside a critical section (it would require a mutex for every state). If two processors collect the same state, then the father of this state could be decremented twice, during the *clearing* operations. Second, *collecting* must be performed after all processors have finished the *clearing* operations, otherwise the algorithm may end prematurely (see [9] for a complete explanation.) We solve the concurrency issues for the RPG version through the synchronization of all processors before both *clearing* and *collecting*. Then, we take advantage of our distributed local state repositories to avoid problems due to concurrent access; each processor can perform the *collecting* operations only over the states that it owns. Finally, we use a work-stealing strategy (see [11]) to balance the work-load between the different phases of our algorithm; for instance, whenever a thread has no more states to clear, it tries to "steal" non-cleared states from other processors.

4.2 Experimental Results

We have implemented the parallel versions of our model-checking algorithm and evaluated their performances on several benchmarks. Experimental results presented in this section were obtained on a Sun Fire X4600 M2 server configured with 8 dual core opteron processors and 208 GB of RAM memory, running the Solaris 10 operating system. (The complete set of experiments can be found in [9].) We give results obtained on 8 classical models, including a Token Ring protocol and the Peg-Solitaire board-game, with a mix of valid and invalid properties. We experimented with all the formulas: reachability ($E\Diamond\phi$), safety ($A\Box\phi$ and $E\Box\phi$), liveness ($A\Diamond\phi$) and leadsto ($\psi \leadsto \phi$).

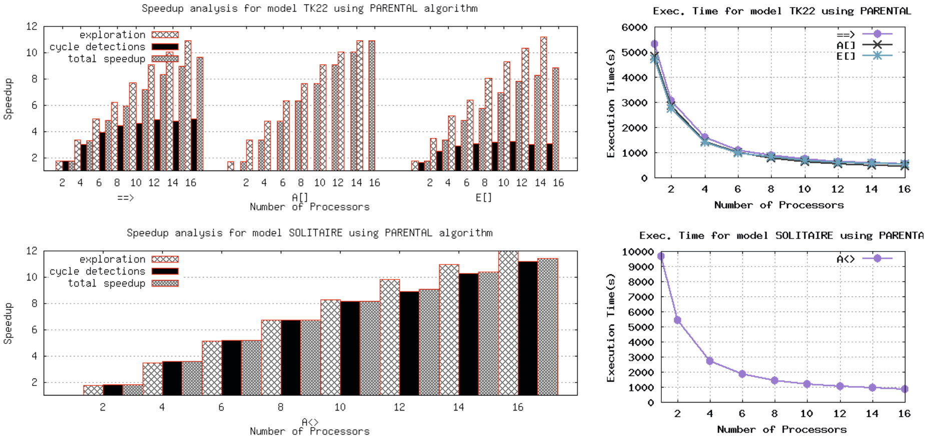
Speedup Analysis: We study the relative speedup and the execution time for our algorithms. In addition, we also give the separate speedup obtained in each phase of the algorithm—during the exploration (forward) and cycle detection (backward) phases—in order to better analyze the advantages of our approach.

Figure 3 shows speedup analysis for the RPG version of our algorithm. We only show the results for two models—a Token Ring with 22 bases (TK22) and a Solitaire game with 33 pegs—since they are representative of the results obtained with our complete benchmark. These models have different execution profiles which impacts significantly the overall performance. The main difference is the time spent in the backward traversal phase. Figure 4 shows a series of bar charts putting in evidence the time required for each phase of the algorithm (exploration and cycle detection). In addition, we compare our approach (RG and RPG) with a third algorithm (NO_GRAPH) that uses the same code as RG but recomputes predecessor states instead of storing them (this is possible only because, for this particular benchmark, we know how to compute the predecessors of a state). We have observed two main categories of behaviors in this analysis.

negligible backward traversal: the time spent in the backward exploration phase is negligible compared to the overall execution time (e.g. model TK22 in Fig. 3, 4). This is the case, for instance, if the property is false and the cycle detection phase terminates early. In this category of experiments, there are no significant differences between RG and RPG, mainly because the gain in performance during the forward exploration phase outweighs the extra work performed during the cycle detection phase;

complete backward traversal: the cycle detection phase needs to run through all the state space (e.g. model SOLITAIRE in Fig. 3, 4). We observe a significant difference in performance between the RG and RPG versions in this case. The extra work performed by the RPG version becomes the dominant factor.

These experiments confirm that RPG is a good choice when we are limited by the memory space: although it may require more computations (in our examples, we may loose a factor of 5 in execution time), it can be applied on models that



a) Speedup b) Execution Time

Fig. 3. Speedup and execution time analysis for Token Ring and Solitaire models

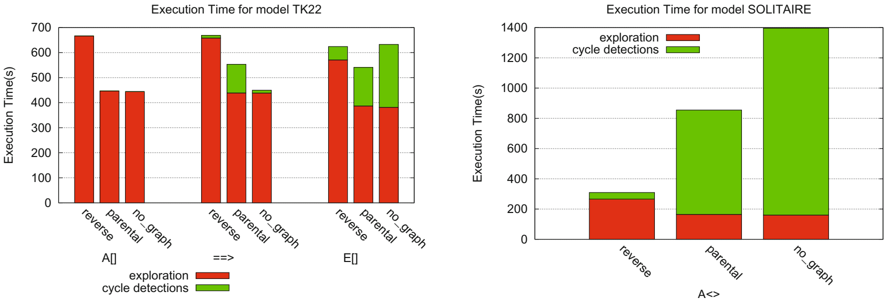


Fig. 4. Comparison with a Standard Algorithm

are not tractable with the RG version because of the space needed to store the transitions. For instance, for the Peg Solitaire model with 37 pegs (that has 3.10^9 states and 3.10^{10} transitions), the RPG only needs 15 GB of memory while, with the RG version, we would need 240 GB of memory just to store the transitions.

5 Related Work and Comparisons with Other Tools

Several works address the problem of designing efficient, parallel model-checking algorithms. Most of the proposals follow an "automata-theoretic approach" for LTL model checking. In this context, the difficulty is to adapt the cycle detection algorithms (Tarjan or Nested-DFS), which are inherently sequential. Two works stand out: one with a mature implementation, DiVinE [1], with the **owcty + map** algorithm; another with a prototype, named LTSmin, with the **mc-ndfs** algorithm [6]. They mostly differ by the algorithm used to detect cycles.

DiVinE combines two algorithms, **owcty** and **map**, that result in "a parallel on-the-fly linear algorithm for model checking weak LTL properties" (weak LTL properties are those expressible by an automata that has no cycle with both accepting and no-accepting states on its path). If the LTL property does not meet this requirement, the algorithm complexity may be quadratic. The *multi-core nested DFS* (**mc-ndfs**) algorithm [6] is a recent extension of the **swarm** [4] distributed algorithm to a multi-core setting. The authors in [6] propose a multi-core version with the distinction that the storage state space is shared among all workers in conjunction with some synchronization mechanisms for the nested search. Even if, in the worst-case, all the processors may duplicate the same work, this approach has a linear complexity (given a fixed number of processors).

In contrast with the number of solutions proposed for parallel LTL model checking, just two specifically target CTL model checking on shared memory machines: Inggs and Barringer work [5] supports CTL*, while the work of van de Pol and Weber [7] supports the μ-calculus.

Comparison with DiVinE. We now compare our algorithms with DiVinE [1], which is the state of the art tool for parallel model checking of LTL. The results given here have been obtained with DiVinE 2.5.2, considering only the best results given by the owcty or map, separately. This benchmark (experimental data and examples are available in report [10]) is based on the set of models borrowed from DiVinE on which, for a broader comparison, we check both valid and non valid properties. Figure 5 shows the exact set of models and formulas that are used. All experiments were carried out using 16 cores and with an initial hash table sized enough to store all states. The DiVinE experiments were executed with flag (-n) to remove counter-example generation.

Figure 6 shows for each model the execution time (T.) in seconds and the memory peak (M.) (in GB). Figure 7 summarizes these results using the normalized weighted sum of the memory footprint and the execution time, separated for valid and non valid formulas.

Algorithms owcty and map show better overall results when the formula is not valid (FALSE). By contrast, reverse holds the best execution time when the formula is valid. Regarding the RPG version of our algorithm, our results show that it holds the best memory footprint among all results, it uses on average 2 to 4 times less memory than map and owcty when the formula is valid. In addition, regardless of its “cubic” worst-case complexity, it shows good results when compared to map and owcty. For instance, it is able to verify a valid formula on average using 4 times less memory than owcty with a limited slow-down (≈ 1.8 times slower).

To conclude, for the set of models and formulas used in this benchmark, both RPG and RG delivered good results when compared to DiVinE. For instance, RG has a better performance in both time and memory usage when compared with DiVinE (map and owtcy). Finally, RPG proved to be the most *space conscious* algorithm—the one to choose for the biggest models—without sacrificing too much the execution time.

Model	Formula	Results
Anderson (AN) $18\cdot 10^6$ states	`F1:(-cs_0) ==>  (cs_0)`	*false*
	`F2:A[]<>(cs_0 or ... or cs_n)`	*true*
Lamport (LA) $38\cdot 10^6$ states.	`F1:(wait_0 and (- cs_0)) ==> (cs_0)`	*false*
	`F2:(- cs_0) ==> (cs_0)`	*false*
	`F3:A[]<>(cs_0 or ... or cs_n)`	*true*
Rether (RE) $4\cdot 10^6$ states	`F1:A[]<>(nrt_0)`	*true*
	`F2:A[]<>(rt_0)`	*false*
Szymanski (SZY) $2\cdot 10^6$ states .	`F1:(wait_0 and (- cs_0)) ==> (cs_0)`	*false*
	`F2:(- cs_0) ==> (cs_0)`	*false*
	`F3:A[]<>(cs_0 or ... or cs_0)`	*true*

Fig. 5. Formulas and Models for our Comparison

M	Formula	owcty		map		reverse (RG)		parental (RPG)	
		T.(s)	M.(Gb)	T.(s)	M.(Gb)	T.(s)	M.(Gb)	T.(s)	M.(Gb)
AN	F1: *false*	61.3	3.3	110.2	5.5	28.8	2.8	94.4	1.8
	F2: *true*	79.5	7.4	110.5	4.8	26.4	2.9	50.4	1.8
LA	F1: *false*	1.6	1.1	1.4	1.1	42.4	5.1	74.2	3.3
	F2: *false*	1.4	1.1	1.7	1.2	47.6	5.6	327.2	3.6
	F3: *true*	153.6	14.1	282.8	12.1	51.0	5.6	370.4	3.7
RE	F1: *true*	12.0	1.8	20.1	1.3	5.0	0.7	12.0	0.6
	F2: *false*	13.2	1.8	1.2	0.3	3.4	0.7	7.8	0.6
SZY	F1: *false*	8.5	0.9	7.0	0.5	2.2	0.3	1.4	0.2
	F2: *false*	9.8	0.9	6.6	.5	4.2	0.3	39.6	0.3
	F3: *true*	9.0	0.9	24.7	0.6	3.8	0.3	32.8	0.3

Fig. 6. Table of results

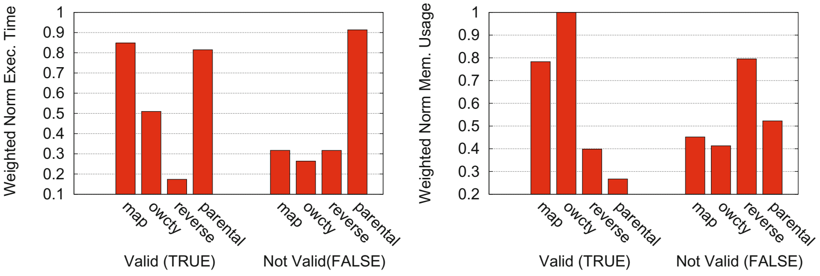


Fig. 7. Comparison with divine (reverse = RG, parental = RPG)

6 Conclusion

We have described ongoing works concerning parallel (enumerative) model-checking algorithms for finite state systems. We define two versions of a new model checking algorithm that support an expressive fragment of both CTL and LTL. These algorithms are based on the standard, semantic model-checking algorithm for CTL but specifically target parallel, shared memory machines. Our two versions differ by the amount of information they need to store: a Reverse Graph (RG) version that explicitly stores the complete transition relation in memory, and a Reverse Parental Graph (RPG) that relies on a spanning tree.

We use the reverse parental graph structure as a means to fight the state explosion problem. In this respect, this approach has a similar impact—on the space—than algorithmic techniques like *sleep sets* (used with partial-order methods), but with the difference that we do not take into account the structure of the model. Moreover, our approach is effective regardless of the formalism used to model the system. For instance, it is particularly useful in cases where it is not possible to compute the "inverse" of the transition relation.

Our prototype implementation shows promising results for both the RG and RPG versions of the algorithm. The choice of a "labeling algorithm" based on the out-degree number has proved to be a good match for shared memory machines and a work stealing strategy; we consistently obtained speedups close to linear with an average efficiency of 75%. Our experimental results also showed that the RPG version is able to outperform the RG version for some categories of models.

Using our work, one can easily obtain a parallel algorithm for checking any CTL formula Φ by running one instance of our algorithms (for the AU and EU formulas) for each subformula of Φ. But this approach, as such, is too naive. For future works, we are considering improvements of our algorithms that support full CTL formulas without having to manage several copies of our labels (sons and suc) in parallel, which could have an adverse effect on memory consumption.

References

1. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker. In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010), pp. 4–7. IEEE (2010)
2. Clarke, E.M., Emerson, A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
3. Greenlaw, R., James Hoover, H., Ruzzo, W.L.: Limits to Parallel Computation: P-Completeness Theory. Oxford University Press, USA (1995)
4. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Proc. of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering, pp. 1–6 (2008)
5. Inggs, C.P., Barringer, H.: CTL* model checking on a shared-memory architecture. Formal Methods in System Design 29(2), 135–155 (2006)
6. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
7. van de Pol, J., Weber, M.: A Multi-Core solver for parity games. In: Proc. of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008). ENTCS, vol. 220(2), pp. 19–34 (2008)
8. Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters 20(5), 229–234 (1985)
9. Saad, R.T.: Parallel Model Checking for Multiprocessor Architecture. PhD thesis, Institut National des Sciences Appliquées, Toulouse, France (December 2011)

10. Saad, R.T., Zilio, S.D., Berthomieu, B.: Parallel Model Checking with Lazy Cycle Detection—MCLCD. Technical Report 12139, LAAS-CNRS (2012), http://hal.archives-ouvertes.fr/hal-00669752/en
11. Saad, R.T., Zilio, S.D., Berthomieu, B.: Mixed Shared-Distributed hash tables approaches for parallel state space construction. In: Int. Symposium on Parallel and Distributed Computing (ISPDC 2011) (July 2011)

Variable Probabilistic Abstraction Refinement

Luis María Ferrer Fioriti[1], Ernst Moritz Hahn[1],
Holger Hermanns[1], and Björn Wachter[2]

[1] Saarland University, Germany
[2] University of Oxford, UK

Abstract. Predicate abstraction has proven powerful in the analysis of very large probabilistic systems, but has thus far been limited to the analysis of systems with a fixed number of distinct transition probabilities. This excludes a large variety of potential analysis cases, ranging from sensor networks to biochemical systems. In these systems, transition probabilities are often given as a function of state variables—leading to an arbitrary number of different probabilities.

This paper overcomes this shortcoming. It extends existing abstraction techniques to handle such *variable probabilities*. We first identify the most precise abstraction in this setting, the best transformer. For practicality purposes, we then devise another type of abstraction, mapping on extensions of constraint or interval Markov chains, which is less precise but better applicable in practice. Refinement techniques are employed in case a given abstraction yields too imprecise results. We demonstrate the practical applicability of our method on two case studies.

1 Introduction

Many systems, including network protocols, manufacturing systems and biological systems are characterised by *random* phenomena which can be modeled using probabilities. Since these system are distributed, they are inherently *concurrent*. Markov decision processes (MDPs) are often used as a semantic foundation in this context, because they account for both nondeterminism (used to model concurrency) and probabilism. Typically, one is interested in computing (maximal or minimal) reachability probabilities, e.g., the chance of having delivered three messages after ten transmission attempts, under best-case and worst-case assumptions concerning the environment. For finite MDPs, these probabilities can be computed by linear optimisation or using numerical techniques such as value iteration [6]. The latter is done e.g., in the popular probabilistic model checker Prism [13]. However, this approach suffers from the state explosion problem, even more than in non-probabilistic model checking, due to expensive numerical computations.

Abstraction-refinement methods have gained popularity as approaches to alleviate this problem. Early approaches (e.g. [1,3]) were restricted to finite models, since they unfold the state space of the model under analysis. More recently, symbolic abstractions that operate at the source-code level were introduced to

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 300–316, 2012.

support infinite or very large models. Predicate abstraction and counterexample-guided abstraction refinement (CEGAR) have been proposed [8] and implemented in the verification tool PASS [7] for concurrent probabilistic programs. Building on seminal work on game-based abstraction [14], abstraction refinement has also been applied to sequential probabilistic programs [11], probabilistic timed automata [15] and finite-state concurrent probabilistic programs [12].

There is one disturbing limitation to the otherwise promising techniques based on symbolic abstraction, which is common to all works in this field [8,14,11,15,12]. Namely, so far, the analysis is limited to models with a fixed set of transition probabilities, and effectively cannot handle probabilities expressed in terms of arithmetic expressions that involve program variables. For instance, it is not possible to have transition probabilities $\frac{1}{n}$ and $1 - \frac{1}{n}$ where n is an integer variable. Such *variable probabilities* induce a potentially unbounded number of transition probabilities corresponding to the underlying variable domain. Previous symbolic-abstraction techniques [14,19] produce intractably large, or even infinite abstract models in presence of variable probabilities, while existing interval abstractions of transition probabilities [9] only apply to finite explicit-state models [4,10].

However, variable probabilities naturally arise in most biological systems, in wireless sensor networks, manufacturing systems and many other application contexts. For instance, protocol standards such as IETF RFC 3927 (Zeroconf), or IEEE 802.11e (QoS for WLAN) make extensive use of random selections which are based on the value of certain state variables. With the further advancement of self-stabilising, self-healing, self-supportive or energy-harvesting systems, the use of situation-specific random sampling is likely to be the rule, not the exception. Guarantees for such systems are probabilistic in nature, but are only possible if such randomisation mechanisms are supported by analysis tools. This is what this paper aims to achieve for probabilistic abstraction refinement. We develop theory and tools for the automatic analysis of concurrent probabilistic systems with variable probabilistic selections. We introduce novel abstractions and refinement procedures for probabilistic programs based on probabilistic games.

To handle variable probabilities, our new abstractions explicitly summarise transition probabilities, which is more challenging than both (interval) abstraction of explicit-state models and non-variable symbolic abstraction, because computing abstract transition probabilites requires reasoning over the arithmetic theory of the program. We solve this problem by using optimisation techniques and explore the trade-off between precision versus performance by applying optimisation at different stages. Our symbolic abstraction retains the symbolic representation of transition probabilities, as in the program. While this is very precise, the analysis involves a costly fixed-point iteration with nested linear optimisation problems. We therefore develop an alternative interval abstraction technique, computing a probabilistic game with transition probabilities given as intervals, which may be less precise but requires to only solve an optimisation problem once. A prototype of the interval abstraction with abstraction refinement has been implemented in PASS and has been successfully applied to a number of non-trivial case studies.

Outline. In Section 2, we discuss probabilistic programs and their semantics. In Section 3, we introduce an abstract game model to safely overapproximate properties of the semantics. Computing abstractions in practice is treated in Section 4. Automatic refinement is discussed in Section 5. Experiments follow in Section 6 and Section 7 concludes the paper. An extended version containing the proofs and covering also abstractions for models in which the size of the probability distributions can be variable can be found at [5].

2 Preliminaries

Let S be a set. A *probability distribution* over S is a function $\mu\colon S \to [0,1]$ with $\sum_{s\in S} \mu(s) = 1$. The *support* of μ is given by $\mathit{Supp}(\mu) = \{s \in S \mid \mu(s) > 0\}$. We denote the set of all probability distributions over a given set S by $\mathit{Distr}(S)$. The domain of a partial function $f\colon A \rightharpoonup B$ is $\mathit{Dom}(f) \subseteq A$.

Definition 1. *A* probabilistic automaton *is a tuple* $(S, I, \mathit{Act}, \delta)$ *where*

- S *is a set of* states *of which* $I \subseteq S$ *with* $I \neq \emptyset$ *is the set of* initial states*,*
- Act *is a set of* actions*, and*
- $\delta\colon (S \times \mathit{Act}) \rightharpoonup \mathit{Distr}(S)$ *is the* transition function.

We denote by $en(s) \subseteq \mathit{Act}$ the actions that are enabled in s, i.e. $\alpha \in en(s)$ iff $\delta(s, \alpha)$ is defined. We require that $en(s) \neq \emptyset$ for all $s \in S$. Any automaton that does not satisfy this condition can be modified by introducing self-loops in the deadlock states. Intuitively, in each state s, we choose an action $\alpha \in en(s)$. In turn, the probability distribution $\mu = \delta(s, \alpha)$ provides a probabilistic choice over the states reached in the next step.

Definition 2. *A* probabilistic program *is a tuple* $\mathcal{P} = (\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ *where*

- $\mathtt{X} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ *is a set of* program variables*; for each variable* $\mathtt{x}_i \in \mathtt{X}$*, we denote by* $\mathtt{Dom}(\mathtt{x}_i)$ *the (possibly infinite)* variable domain *so that the state space of the program is the set* $\mathtt{Dom}(\mathtt{X}) \overset{\text{def}}{=} \mathtt{Dom}(\mathtt{x}_1) \times \cdots \times \mathtt{Dom}(\mathtt{x}_n)$*,*
- $\mathtt{I} \subseteq \mathtt{Dom}(\mathtt{X})$ *with* $\mathtt{I} \neq \emptyset$ *is an expression characterising the set of* initial states*,*
- $\mathtt{C}$ *is a set of* guarded commands *of the form* $\mathtt{c} = (\mathtt{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \ldots + \mathtt{p}_m : \mathtt{u}_m)$ *with a guard expression* $\mathtt{g} \subseteq \mathtt{Dom}(\mathtt{X})$ *and probabilistic choices* $1 \leq i \leq m$*; each choice* i *comes with a* weight function $\mathtt{p}_i$ *and an* update function $\mathtt{u}_i$*:*
    - *update function* $\mathtt{u}_i\colon \mathtt{Dom}(\mathtt{X}) \to \mathtt{Dom}(\mathtt{X})$ *assigns a successor to each state,*
    - *weight function* $\mathtt{p}_i\colon \mathtt{Dom}(\mathtt{X}) \to [0,1]$ *assigns a weight to each state* $\vec{\mathtt{x}}$*, such that the weights sum to one:* $\sum_{1\leq i\leq m} \mathtt{p}_i(\vec{\mathtt{x}}) = 1$.

  We require that $\bigcup_{\mathtt{c}\in\mathtt{C}} \mathtt{g}_\mathtt{c} = \mathtt{Dom}(\mathtt{X})$.

Definition 3. *The semantics* $\mathit{sem}(\mathcal{P})$ *of a probabilistic program* $\mathcal{P} = (\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ *is the probabilistic automaton* $(S, \mathtt{I}, \mathtt{C}, \delta)$ *with* $S \overset{\text{def}}{=} \mathtt{Dom}(\mathtt{X})$*. The transition function is defined on pairs of states* s *and commands* $\mathtt{c}$ *which can be executed in* s*, i.e.,* $\mathit{Dom}(\delta) \overset{\text{def}}{=} \{(s, \mathtt{c}) \in S \times \mathtt{C} \mid \mathtt{c} = (\mathtt{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \ldots + \mathtt{p}_m : \mathtt{u}_m) \wedge s \in \mathtt{g}\}$*, and the corresponding distribution is defined by the probabilistic choices, i.e., for each* $s' \in S$ *it is* $\delta(s, \mathtt{c})(s') \overset{\text{def}}{=} \sum_{\substack{1\leq i\leq m,\\ \mathtt{u}_i(s)=s'}} \mathtt{p}_i(s)$ *for* $\mathtt{c} = (\mathtt{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \ldots + \mathtt{p}_m : \mathtt{u}_m) \in \mathtt{C}$.

The guarded-command language in Definition 2 forms the core language of the probabilistic model checker PRISM [13], for which a vast collection of case studies exists, many of which feature variable probabilities. However, previous predicate abstraction techniques for probabilistic programs did not fully support variable probabilities. The probability associated with a probabilistic choice had to be a *constant.* To a limited degree, variable probabilities could be encoded by creating a copy of the same command for each induced distribution. However, variable probabilities may induce infinitely many distributions, as illustrated by the following Example 1. Even if the number of distributions is finite, the encoding creates extra work for the abstraction procedure with each copy of a command.

Example 1. Consider the program $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ in Fig. 1. The program contains two variables, `s` and `x` where $\mathtt{Dom}(\mathtt{s}) = \{0, 1, 2, 3\}$ and $\mathtt{Dom}(\mathtt{x}) = \mathbb{N}$. We have $\mathtt{I} = \{0\} \times \mathbb{N}$. For command $\mathtt{b} = (\mathtt{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \mathtt{p}_2 : \mathtt{u}_2)$ we have $\mathtt{g} = \{1\} \times \mathbb{N}$, $\mathtt{p}_1(s, x) = \frac{9x-8}{16x}$, $\mathtt{p}_2(s, x) = \frac{7x+8}{16x}$, $\mathtt{u}_1(s, x) = (2, x)$ and $\mathtt{u}_2(s, x) = (3, x)$. The other commands are formalised likewise.

This program could not have been handled by the previous approach, as command `b` would have to be encoded by a countably infinite number of guarded commands. Even if the domain of variable x was finite, the number of transition probabilities in the abstraction would have at least the size of the domain $\mathtt{Dom}(x)$.

Probabilistic Reachability. Given the automata semantics $\mathcal{M} = (S, I, Act, \delta)$ of a probabilistic program, we are interested in the probability to reach a set of goal states $F \subseteq S$. This probability depends on the resolution of nondeterminism in the automaton, i.e. choosing an action at a given state. A particular resolution of nondeterminism is a function from paths to a distribution over actions, and induces a probability measure over a set of paths. We are interested in the minimal and maximal probabilities to reach a set of goal states F, particularly starting in an initial state.

```
s : [0..3];   x : [1..inf);
[a] s=0 -> (s'=1) & (x'=1);
[b] s=1 -> ((9x-8)/(16x)):(s'=2)
         + ((7x+8)/(16x)):(s'=3);
[c] s=1 -> 1 : (x'=x+1);
init s=0 endinit
```

Fig. 1. Example for a probabilistic program

Reachability probabilities can be expressed and computed in terms of functions from states to probabilities $\nu\colon S \to [0, 1]$. We call these functions *valuations* and denote by $Asg(S)$ the set of all valuations for S.

The set $Asg(S)$ forms a complete lattice with the pointwise order $\leq$, i.e. for each pair $\nu_1, \nu_2 \in Asg(S)$ we let $\nu_1 \leq \nu_2 \overset{\text{def}}{\iff} \forall s \in S.\ \nu_1(s) \leq \nu_2(s)$. We call a monotone function $f\colon Asg(S) \to Asg(S)$ a *valuation transformer.* As valuations form a complete lattice, each transformer has a least (and a greatest) fixed point $\mathit{lfp}_{\leq}\, f$ $(\mathit{gfp}_{\leq}\, f)$.

Definition 4. *Given a probabilistic automaton* $\mathcal{M} = (S, I, Act, \delta)$ *and a set of* goal states $F \subseteq S$, *the* + valuation transformer *is the function* $\mathit{pre}^{+}_{\mathcal{M},F}\colon Asg(S) \to Asg(S)$. *For* $\nu \in Asg(S)$ *and* $s \in S$ *it is* $\mathit{pre}^{+}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} 1$ *for* $s \in F$ *and otherwise*

$$\mathit{pre}^{+}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} \sup_{\alpha \in en(s)} \sum_{s' \in S} \delta(s, \alpha)(s') \cdot \nu(s').$$

The $-$ valuation transformer is defined accordingly using the infimum. By $p^{\bullet}_{\mathcal{M},F} \overset{\text{def}}{=} \mathit{lfp}_{\leq}\ \mathit{pre}^{\bullet}_{\mathcal{M},F}$ *for* $\bullet \in \{+,-\}$ *we describe the least fixed points of these operators.*

The maximal (minimal) reachability probability equals $p^{+}_{\mathcal{M},F}$ ($p^{-}_{\mathcal{M},F}$). This allows to use an iterative algorithm to compute concrete probabilities in finite models [6]: We let $\nu^0(s) \overset{\text{def}}{=} 0$ and $\nu^{i+1}(s) \overset{\text{def}}{=} \mathit{pre}^{+}_{\mathcal{M},F}(\nu^i)(s)$ for $i \geq 0$. The sequence ν^i converges to the fixed point. We can thus use ν^n as an approximation for the fixed point, where n is chosen according to the precision to be obtained.

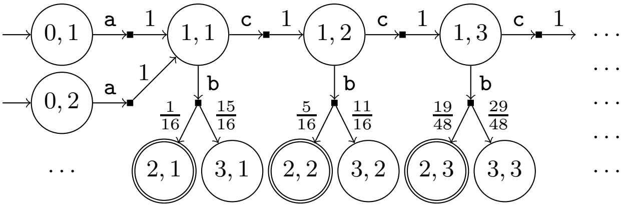


Fig. 2. Probabilistic automata semantics of the probabilistic program in Fig. 1

Example 2. Fig. 2 shows a fraction of the infinite semantics of the program of Fig. 1, where we mark states of the goal set $F = \{(s, x) \in \mathtt{Dom}(\mathtt{X}) \mid s = 2\}$. The supremum probability to reach F is $\frac{9}{16}$: After an initial transition to state $(1, 1)$ by command $\mathtt{a}$, in each state $(1, x)$ one can decide to either move to state $(1, x)$ with certainty or to move to F with probability $\frac{9x-8}{16x}$. It is $\lim_{x\to\infty} \frac{9x-8}{16x} = \frac{9}{16}$, which means that by first executing a sufficiently long sequence of command $\mathtt{c}$, one can reach F with a probability arbitrary close to $\frac{9}{16}$.

Abstract Interpretation. Variable probabilities induce models with unbounded numbers of transition probabilities. This asks for abstraction techniques that extend the existing frameworks [14,19]. In this section, we revisit the formal framework [19] to reason about abstractions for probabilistic programs building on the theory of abstract interpretation. We begin by introducing an abstract state space which partitions[1] the concrete state space.

Definition 5. *Given a probabilistic automaton* $(S, I, \mathit{Act}, \delta)$, *an* abstract state space *is a finite partition* $\mathcal{A} = \{\mathsf{z}_1, \ldots, \mathsf{z}_n\}$ *of* S, *i.e., for all* $1 \leq i \leq n$ *it is* $\mathsf{z}_i \subseteq S$, *for all* $1 \leq j \leq n$ *with* $i \neq j$ *we have* $\mathsf{z}_i \cap \mathsf{z}_j = \emptyset$, *and* $S = \bigcup_{i\in\{1,\ldots,n\}} \mathsf{z}_i$. *For* $\mathsf{z} \in \mathcal{A}$ *and* $s_1, s_2 \in \mathsf{z}$ *we need to have* $en(s_1) = en(s_2)$. *For the analysis with a goal set* F, *we require that for all* $\mathsf{z} \in \mathcal{A}$ *either* $\mathsf{z} \cap F = \emptyset$ *or* $\mathsf{z} \subseteq F$.

When we go from the domain $\mathit{Asg}(S)$ to $\mathit{Asg}(\mathcal{A})$, we summarise states *and* probabilities. Assume we already have a concrete valuation and want to compute an abstraction in $\mathit{Asg}(\mathcal{A})$ that represents a lower bound. The valuation we choose

[1] In the definition, we require that for a given abstract state all contained concrete states are able to perform the same set of commands. This assumption is not strictly needed but used to simplify notations later on, and because it is fulfilled automatically by our implementation of the abstraction methods.

maps a given abstract state z to the lower bound (infimum) of the probabilities over all states $s \in \mathsf{z}$ contained in z. This is captured by the function α^l:

$$\alpha^l\colon\ Asg(S) \to Asg(\mathcal{A}), \quad \nu \mapsto \nu^\sharp \text{ where } \nu^\sharp(\mathsf{z}) \stackrel{\text{def}}{=} \inf_{s\in\mathsf{z}} \nu(s) \text{ for all } \mathsf{z} \in \mathcal{A}.$$

The *upper-bound abstraction* function α^u is defined analogously with the difference that ν is mapped to $\nu^\sharp(\mathsf{z}) \stackrel{\text{def}}{=} \sup_{s\in\mathsf{z}} \nu(s)$ for all $\mathsf{z} \in \mathcal{A}$.

The abstraction functions have a counterpart, the *concretisation* function γ. It maps an abstract valuation back to a concrete one in the obvious way:

$$\gamma\colon\ Asg(\mathcal{A}) \to Asg(S), \quad \nu^\sharp \mapsto \nu \text{ where } \nu(s) \stackrel{\text{def}}{=} \nu^\sharp(\mathsf{z}) \text{ for all } s \in S \text{ with } s \in \mathsf{z}.$$

Intuitively, our choice of pairs (α^l, γ) and (α^u, γ) defines a way to map a valuation to "a lower resolution" and back—preserving either lower or upper bounds. The following theorem establishes that this mapping is canonical yielding the most precise mapping that still guarantees correct bounds. Precisely this notion is captured by the well-established concept of a Galois connection:

Proposition 1 (Galois Connections [18]). *Let α^l, α^u, and γ be as defined above. The pair (α^l, γ) is a Galois connection between the domains $(Asg(S), \geq)$ and $(Asg(\mathcal{A}), \geq)$, and (α^u, γ) is a Galois connection between the two domains $(Asg(S), \leq)$ and $(Asg(\mathcal{A}), \leq)$.*

Notably we have two Galois connections depending on whether we are aiming for lower bounds or for upper bounds. The order on the domains is an approximation order in the following sense: Lower bounds are expressed in the domain $(Asg(S), \geq)$. Because a lower bound is more precise than another one if it is greater, the order for lower bounds is the point-wise ordering $\geq$. Conversely, smaller upper bounds are more precise, therefore the order for upper bounds is $\leq$.

To obtain a working analysis on the abstract domain, we additionally require an abstract transformer with type $Asg(\mathcal{A}) \to Asg(\mathcal{A})$, so that the concrete fixed-point computation of transformer $pre^\bullet_{\mathcal{M},F}\colon Asg(S) \to Asg(S)$ can be replaced by an abstract fixed-point computation. Due to the complex interplay between non-determinism and probability, it is not immediately clear what is the best choice. Thankfully abstract interpretation provides canonical and most precise abstract transformers defined in terms of function composition: $\alpha^l \circ pre^\bullet_{\mathcal{M},F} \circ \gamma$ for the lower bound and $\alpha^u \circ pre^\bullet_{\mathcal{M},F} \circ \gamma$ for the upper bound [19].

While abstract interpretation specifies desirable abstract transformers, probabilistic games provide a representation for them, which stands out from other abstract models in that it admits computing effective lower and upper bounds on reachability probabilities [14]. In the next section, we discuss such games.

3 Probabilistic Games

In this section, we discuss several variants of turn-based probabilistic games, which are needed to obtain abstractions of probabilistic programs with variable probabilities. We begin by introducing probabilistic games which extend probabilistic automata by introducing a second instance of nondeterministic choice. The two kinds of nondeterminism are resolved by two *players*. The basic definition of a probabilistic game is as follows:

Definition 6. *A* probabilistic two-player game *is a tuple* $(S, I, Act_1, Act_2, \delta)$ *where*

- S *is a set of* states *of which* $I \subseteq S$ *with* $I \neq \emptyset$ *is the set of* initial states*,*
- *for* $i = 1, 2$*, it is* Act_i *the set of player-i actions, and*
- $\delta\colon (S \times Act_1 \times Act_2) \rightharpoonup Distr(S)$ *provides the* transition function*.*

Similarly to probabilistic automata, we introduce the notion of enabled actions. We have $\alpha_1 \in en_1(s) \subseteq Act_1$ if there exist $\alpha_2 \in Act_2$ such that $\delta(s, \alpha_1, \alpha_2)$ is defined and we have $\alpha_2 \in en_2(s, \alpha_1) \subseteq Act_2$ if $\delta(s, \alpha_1, \alpha_2)$ is defined. We also require $en_1(s) \neq \emptyset$ for all $s \in S$ and $en_2(s, \alpha_1) \neq \emptyset$ if $\alpha_1 \in en_1(s)$. In state s of a probabilistic game $(S, I, Act_1, Act_2, \delta)$, the first player chooses an action $\alpha_1 \in en_1(s)$. Afterwards, the second player chooses an action $\alpha_2 \in en_2(s, \alpha_1)$. From this, we obtain a distribution $\mu = \delta(s, \alpha_1, \alpha_2)$ which probabilistically determines the next state of the game.

We can define valuation transformers for probabilistic games, similar to those of probabilistic automata. As there are now two types of nondeterminism to be resolved, there are more possible resolutions, and thus valuation transformers.

Definition 7. *Given a probabilistic two-player game* $\mathcal{M} = (S, I, Act_1, Act_2, \delta)$ *and a set of goal states* $F \subseteq S$*, the* $+,-$ valuation transformer *is the function* $pre^{+,-}_{\mathcal{M},F}\colon Asg(S) \to Asg(S)$*. For* $\nu \in Asg(S)$ *and* $s \in S$ *it is* $pre^{+,-}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} 1$ *for* $s \in F$ *and otherwise*

$$pre^{+,-}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} \sup_{\alpha_1 \in en_1(s)} \inf_{\alpha_2 \in en_2(s,\alpha_1)} \sum_{s' \in S} \delta(s, \alpha_1, \alpha_2)(s') \cdot \nu(s').$$

Other transformers are defined accordingly. By $p^{\bullet_1,\bullet_2}_{\mathcal{M},F} = lfp_{\leq}\ pre^{\bullet_1,\bullet_2}$ *we denote the least fixed points of these operators.*

Similar solution techniques as for probabilistic automata exist [6].

Definition 8. *Let* $\mathcal{A}$ *be an abstract state space of* S *and let* $\mu \in Distr(S)$ *be a distribution. By* $lift_{\mathcal{A}}(\mu)\colon \mathcal{A} \to [0, 1]$ *we denote the* lifting *of* μ *where, for* $\mathsf{z} \in \mathcal{A}$*, we have* $lift_{\mathcal{A}}(\mu)(\mathsf{z}) \overset{\text{def}}{=} \sum_{s \in \mathsf{z}} \mu(s)$*. The lifting* $lift_{\mathcal{A}}(g)\colon Act \rightharpoonup Distr(\mathcal{A})$ *of a partial function* $g\colon Act \rightharpoonup Distr(S)$ *is defined such that* $lift_{\mathcal{A}}(g)(\alpha) \overset{\text{def}}{=} lift_{\mathcal{A}}(g(\alpha))$ *for all valid* $\alpha \in Act$*.*

For a probabilistic automaton (S, I, Act, δ) *and an abstract state space* $\mathcal{A}$*, we define* $Valid(\mathsf{z}) \overset{\text{def}}{=} \{lift_{\mathcal{A}}(\delta(s, \cdot)) \mid s \in \mathsf{z}\}$ *for all* $\mathsf{z} \in \mathcal{A}$*. The* game-based abstraction *is then the probabilistic game* $(\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ *where*

- $\mathcal{I} \overset{\text{def}}{=} \{\mathsf{z} \in \mathcal{A} \mid I \cap \mathsf{z} \neq \emptyset\}$,
- $Act_1 \overset{\text{def}}{=} \{f\colon Act \rightharpoonup Distr(\mathcal{A}) \mid \exists \mathsf{z} \in \mathcal{A}.\ f \in Valid(\mathsf{z})\}$,
- $Act_2 \overset{\text{def}}{=} \{\alpha \in Act \mid \exists \mathsf{z} \in \mathcal{A}.\ \exists f \in Valid(\mathsf{z}).\ \alpha \in Dom(f)\}$,
- $\delta'(\mathsf{z}, f, \alpha) \overset{\text{def}}{=} \begin{cases} f(\alpha) & ; \quad \mathsf{z} \in \mathcal{A}, f \in Valid(\mathsf{z}), \alpha \in Dom(f) \\ \mathit{undefined} & ; \quad \mathit{otherwise.} \end{cases}$

Definition 7 defines abstract reachability probabilities of probabilistic games. Previously [19], we had shown how this is possible for the restricted setting of static transition probabilities. Maximisation and minimisation at Act_1-choices exactly capture the lower-bound and upper-bound abstraction function, while Act_2-choices capture nondeterministic choices of the original model.

Here we extend this result to variable probabilities, which is possible as the property relies on characteristics of the Galois connection that have been established in Proposition 1. The following theorem states in more detail how the specific lower and upper bounds can be obtained from the game. Furthermore, it shows that the valuation transformer of Definition 7 is most precise, i.e., among the abstract transformers $Asg(\mathcal{A}) \to Asg(\mathcal{A})$ it gives the most precise abstract transformer that approximates the original probabilistic automaton.

Theorem 1. *Let $\mathcal{M} = (S, I, Act, \delta)$ be a probabilistic automaton and let $\mathcal{M}' = (\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ be a game-based abstraction of $\mathcal{M}$. Let $F' = \{\mathsf{z} \in \mathcal{A} \mid \mathsf{z} \cap F \neq \emptyset\}$. Then, for $\bullet \in \{+, -\}$, for $\mathsf{z} \in \mathcal{A}$ and $s \in \mathsf{z}$, we have:*

$$pre^{-,\bullet}_{\mathcal{M}',F'} = (\alpha^l \circ pre^{\bullet}_{\mathcal{M},F} \circ \gamma) \qquad \mathit{and} \qquad pre^{+,\bullet}_{\mathcal{M}',F'} = (\alpha^u \circ pre^{\bullet}_{\mathcal{M},F} \circ \gamma)$$

where $pre^{-,\bullet}_{\mathcal{M}',F'}$ and $pre^{+,\bullet}_{\mathcal{M}',F'}$ are the valuation transformers defined by the game.

Corollary 1. *Let $\mathcal{M} = (S, I, Act, \delta)$ be a probabilistic automaton and let $\mathcal{M}' = (\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ be a game-based abstraction of $\mathcal{M}$. Let $F' = \{\mathsf{z} \in \mathcal{A} \mid \mathsf{z} \cap F \neq \emptyset\}$. Then, for $\bullet \in \{+, -\}$, for $\mathsf{z} \in \mathcal{A}$ and $s \in \mathsf{z}$, we have*

$$p^{-,\bullet}_{\mathcal{M}',F'}(\mathsf{z}) \leq p^{\bullet}_{\mathcal{M},F}(s) \leq p^{+,\bullet}_{\mathcal{M}',F'}(\mathsf{z}).$$

Example 3. In Fig. 3, we abstract the semantics of the program in Fig. 1 to a probabilistic game with four abstract states. Each of them subsumes the concrete states with the same value of $\mathtt{s}$. Small circles correspond to choices of Act_1, whereas small squares correspond to choices of Act_2. Because of the infinite number of distributions, the abstraction is infinitely large.

As seen in Example 3, abstractions of probabilistic programs might still be infinite. The abstraction is guaranteed to be finite, only if probabilities are constants, i.e., do not depend on the program variables. The infiniteness of the game-based abstraction in Definition 8 lies in the number of player choices Act_1 and Act_2.

The key idea is to make the sets Act_1 and Act_2 finite, and move the infiniteness to the level of distributions. This is realised by generalising the transition function $\delta\colon (S \times Act_1 \times Act_2) \rightharpoonup Distr(S)$ to return a set of distributions rather than a single distribution. The set of distributions, in turn, can then be represented symbolically, for instance by intervals. The player that controls the abstraction is in charge of picking a distribution, as the different distributions arise from summarising different states. The underlying abstract model is defined as follows:

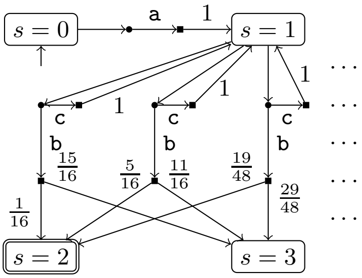


Fig. 3. Game-based abstraction of the semantics in Fig. 2 of the probabilistic program in Fig. 1

Definition 9. *A* constraint Markov game *is a tuple* $(S, I, Act_1, Act_2, \delta)$ *where* S, I, Act_1 *and* Act_2 *are as in Definition 6 and the* transition function *is of the form* $\delta\colon (S \times Act_1 \times Act_2) \rightharpoonup \mathfrak{C}(\mathbb{N} \times S)$ *where* $\mathfrak{C}(A) \overset{\text{def}}{=} 2^{Distr(A)} \setminus \emptyset$.

As in probabilistic two-player games, the two players choose their valid actions. However, after the first choice of player 2, there is an additional choice of this player on a distribution function $\mu \in \delta(s, \alpha_1, \alpha_2)$. This distribution is of $Distr(\mathbb{N} \times S)$, where the number $\mathbb{N}$ will be used to later on distinguish between different branches of a guarded command leading to the same abstract state. Constraint Markov games are a special case of probabilistic two-player games—as the choice of both α_2 and the distribution is controlled by player 2.

Definition 10. *Given a constraint Markov game* $\mathcal{M} = (S, I, Act_1, Act_2, \delta)$ *and a set of goal states* $F \subseteq S$*, the* $+,-$ valuation transformer *is defined as the function* $pre^{+,-}_{\mathcal{M},F}\colon Asg(S) \to Asg(S)$. *For* $\nu \in Asg(S)$ *and* $s \in S$ *it is* $pre^{+,-}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} 1$ *for* $s \in F$ *and otherwise*

$$pre^{+,-}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} \sup_{\alpha_1 \in en_1(s)} \inf_{\alpha_2 \in en_2(s,\alpha_1)} \inf_{\mu \in \delta(s,\alpha_1,\alpha_2)} \sum_{i \in \mathbb{N}} \sum_{s' \in S} \mu(i, s') \cdot \nu(s').$$

Other transformers are defined accordingly. By $p^{\bullet_1,\bullet_2}_{\mathcal{M},F} \overset{\text{def}}{=} lfp_{\leq}\ pre^{\bullet_1,\bullet_2}$ *we denote the least fixed points of these operators.*

We define a specific form of constraint Markov games.

Definition 11. *An* interval assignment *over a set* A *is a function* $\iota\colon A \to ([0,1] \times [0,1])$. *The set of all interval assignments over* A *is denoted by* $\mathfrak{I}(A)$. *An interval assignment* $\iota\colon A \to ([0,1] \times [0,1])$ *represents the set* $Distr(\iota)$ *of* valid distributions of ι, *where* $\mu \in Distr(\iota)$ *iff* $\mu \in Distr(A)$ *and for all* $a \in A$ *if* $\iota(a) = (l, u)$ *then* $\mu(a) \in [l, u]$. *We identify* ι *and* $Distr(\iota)$ *and write* $\mu \in \iota$ *if* $\mu \in Distr(\iota)$

An interval Markov game *is a tuple* $(S, I, Act_1, Act_2, \delta)$ *where* S, I, Act_1 *and* Act_2 *are as in Definition 6 and the* transition function *is of the form* $\delta\colon (S \times Act_1 \times Act_2) \rightharpoonup \mathfrak{I}(\mathbb{N} \times S)$.

Note that interval Markov games are special constraint Markov games and thus the definitions of the valuation transformers $pre^{-,-}_{\mathcal{M},F}, pre^{+,-}_{\mathcal{M},F}, \ldots : Asg(S) \to Asg(S)$ remain as for general constraint Markov games.

We will later on consider both abstractions using interval as well as general constraint Markov games. The interval Markov game abstraction will be coarser, but easier to compute than a constraint Markov game abstraction on the same abstract state space.

4 Computing Abstractions

In this section, we discuss how to obtain abstractions of probabilistic programs.

Let $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ be a program. A *predicate* is a subset $\mathsf{pred} \subseteq \mathtt{Dom}(\mathtt{X})$ of the state space, which can be represented as a Boolean expression over program variables. A *predicate set* is a finite set of predicates $\mathsf{Pred} = \{\mathsf{pred}_1, \ldots, \mathsf{pred}_n\}$. Let F be a set of goal states. We require that $\mathtt{I}, F \in \mathsf{Pred}$ and that for each $\mathsf{c} = (\mathsf{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \ldots + \mathtt{p}_m : \mathtt{u}_m) \in \mathtt{C}$ it is $\mathsf{g} \in \mathsf{Pred}$. By $v_1 \cdots v_n$, with $v_i \in \{0, 1\}$, we denote the abstract state $\bigcap_{1 \leq i \leq n} f(v_i, \mathsf{pred}_i)$, where $f(0, A)$ denotes the complement of the set A and $f(1, A)$ denotes A itself.

In the following, we will assume that predicates are described using a linear theory, which ensures that optimisation problems over the predicate theory remain tractable. Assuming that we already have computed a predicate set, the definition below shows how to obtain a constraint Markov game abstraction. As in menu-based abstraction [19] the first player has to choose a command that is enable in one of the concrete states. The same holds for constraint Markov game abstraction. However, while in menu-based abstraction the second player chooses a possible concretisation for this command, player two makes two choices in a constraint Markov game abstraction: first, it picks a valid branching structure without considering the probabilities (represented by the function f) and, second, it chooses a valid assignment for the probabilities (represented by δ').

Definition 12. *Given a probabilistic program* $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ *with a predicate set* $\mathsf{Pred} = \{\mathsf{pred}_1, \ldots, \mathsf{pred}_n\}$*, the* constraint Markov game abstraction *is defined as* $(\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ *where*

- $\mathcal{A} \overset{\text{def}}{=} \{v_1 \cdots v_n \mid (v_1, \ldots, v_n) \in \{0, 1\}^n \wedge v_1 \cdots v_n \neq \emptyset\}$,
- $\mathcal{I} \overset{\text{def}}{=} \{v_1 \cdots v_n \mid v_1 \cdots v_n \in \mathcal{A} \wedge v_i = 1$ *for* $\mathsf{pred}_i = \mathtt{I}\}$,
- $Act_1 \overset{\text{def}}{=} \mathtt{C}$,
- $Act_2 \overset{\text{def}}{=} \{(\mathsf{c}, f) \mid \mathsf{c}{=}(\mathsf{g} \to \mathtt{p}_1 : \mathtt{u}_1 + \ldots + \mathtt{p}_m : \mathtt{u}_m) \in Act_1 \wedge f \colon \{1, \ldots, m\} \to \mathcal{A}\}$,
- $Valid(v_1 \cdots v_n) = \{\mathsf{c} \mid \mathsf{c} \in Act_1 \wedge \exists i \in \{1 \ldots n\}.\ \mathsf{pred}_i = \mathsf{g} \wedge v_i = 1\}$,
- $Valid(\mathsf{z}, \mathsf{c}) \overset{\text{def}}{=} \{(\mathsf{c}, f) \mid (\mathsf{c}, f) \in Act_2 \wedge \exists s \in \mathsf{z}.\ \forall i \in \{1 \ldots m\}.\ \mathtt{u}_i(s) \in f(i)\}$,
- *for* $\mathsf{z} \in \mathcal{A}$, $\alpha_1 = \mathsf{c} \in Valid(\mathsf{z})$, $\alpha_2 = (\mathsf{c}, f) \in Valid(\mathsf{z}, \alpha_1)$ *let*

$$\delta'(\mathsf{z}, \alpha_1, \alpha_2) \overset{\text{def}}{=} \{\mu \mid \exists s \in wp(\mathsf{z}, \mathsf{c}, f).\ \forall i \in \{1 \ldots m\}.\ \mu(i, f(i)) = \mathtt{p}_i(s)\}$$

- $\delta'(\mathsf{z}, \cdot, \cdot)$ *is undefined if no such* α_1, α_2 *exist.*

Here, $wp(\mathsf{z}, \mathsf{c}, f) \overset{\text{def}}{=} \{s \in \mathsf{z} \mid \forall i, 1 \leq i \leq m.\ \mathtt{u}_i(s) \in f(i)\}$ *for* $\mathsf{c} \in Act_1$.

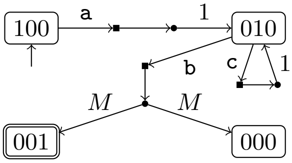


Fig. 4. Constraint Markov game abstraction of the probabilistic program of Fig. 1

The set $A = wp(\mathsf{z}, \mathsf{c}, f)$ can be efficiently computed and represented [19]. To be able to apply value iteration, we need to handle the sets $\delta'(\mathsf{z}, \alpha_1, \alpha_2)$ symbolically. For $\mathsf{z} \in \mathcal{A}$, $\alpha_1 = \mathsf{c} = (\mathsf{g} \to \mathsf{p}_1 : \mathsf{u}_1 + \ldots + \mathsf{p}_m : \mathsf{u}_m) \in \mathit{Valid}(\mathsf{z})$ and $\alpha_2 = (\mathsf{c}, f) \in \mathit{Valid}(\mathsf{z}, \alpha_1)$, the part $\inf_{\mu\in\delta(s,\alpha_2,\alpha_2)} \sum_{i\in\mathbb{N}} \sum_{s'\in S} \mu(i, s') \cdot \nu(s')$ of Definition 10 can be rewritten as $\inf_{x\in A} B(x)$ with $B(x) \overset{\text{def}}{=} \sum_{\mathsf{z}'\in\mathcal{A}} \sum_{\substack{1\le i\le m,\\ \mathsf{z}'=f(i)}} \mathsf{p}_i(x)\cdot\nu(\mathsf{z}')$. In case all p_i are fractions of linear functions over the program variables, i.e. there are $a_i, b_i \in \mathbb{R}$ such that $\mathsf{p}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \frac{a_0+a_1\mathtt{x}_1+\ldots+a_n\mathtt{x}_n}{b_0+b_1\mathtt{x}_1+\ldots+b_n\mathtt{x}_n}$ and the denominator is always positive, $B(x)$ can be written as $\frac{a}{b}$ where a and b are linear in the variables of $\mathtt{X}$. In turn, this optimisation problem can be handled by mixed integer quadratic programming [16]. We can ignore the restriction that the program variables are integers to obtain a quadratic program in which all variables are reals. The correctness property stated later will then still hold, but we will obtain a coarser overapproximation. Another way to compute the bounds is to specify some probability bound to hold and then to prove or to disprove it using a constraint solver. The bounds can then be refined as far as necessary by decreasing (increasing) the upper (lower) bound, in a similar way as a binary search. Notice however that a value iteration using this model will still be quite costly, because we have to solve optimisation problems for each state in each step of the value iteration algorithm.

Example 4. In Fig. 4 we show the constraint Markov game abstraction of the semantics of the program in Fig. 1. We used the predicates $\mathsf{pred}_1 = (\mathsf{s} = 0)$, $\mathsf{pred}_2 = (\mathsf{s} = 1)$, $\mathsf{pred}_3 = (\mathsf{s} = 2)$, which induce the same state space as for the game-based abstraction of Fig. 3. We have $M = \{\mu \in \mathit{Distr}(\mathcal{A}) \mid \exists x \in \mathbb{N} \cap [1, \infty).\ \mu(001) = \frac{9x-8}{16x} \wedge \mu(000) = \frac{7x+8}{16x}\}$. We can obtain the bound $[\frac{1}{16}, \frac{9}{16}]$ for the maximal reachability probability.

We define another abstraction, based on interval Markov games. This new abstraction reduces the information contained in each abstract state in order to reduce the computation and memory complexity. The reduction of information comes at the cost of introducing distributions that are not present in the original model. Roughly, the abstraction consist in maintaining the branching structure of distributions as in the previous abstraction except for the probabilities. Only upper and lower bounds are maintained.

Definition 13. *Given a probabilistic program* $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ *with a predicate set* $\mathsf{Pred} = \{\mathsf{pred}_1, \ldots, \mathsf{pred}_n\}$*, the* interval Markov game abstraction *is the tuple* $(\mathcal{A}, \mathcal{I}, \mathit{Act}_1, \mathit{Act}_2, \delta')$ *where* $\mathcal{A}$*,* $\mathcal{I}$*,* Act_1*,* Act_2 *and* $\mathit{Valid}(\cdot)$ *are as in Definition 12 and for* $\mathsf{z} \in \mathcal{A}$*,* $\alpha_1 = \mathsf{c}$*,* $\alpha_2 = (\mathsf{c}, f) \in \mathit{Valid}(\mathsf{z}, \alpha_1)$ *and* $\mathsf{z}' \in \mathcal{A}$*, with*

$$M(\mathsf{z}, \alpha_1, \alpha_2) \overset{\text{def}}{=} \{\mu \mid \exists s \in \mathit{wp}(\mathsf{z}, \mathsf{c}, f).\ \forall i \in \{1 \ldots m\}.\ \mu(i, f(i)) = \mathsf{p}_i(s)\}$$

$$\textit{we let } \delta'(\mathsf{z}, \alpha_1, \alpha_2)(i, \mathsf{z}') \overset{\text{def}}{=} \left(\inf_{\mu \in M(\mathsf{z}, \alpha_1, \alpha_2)} \mu(i, \mathsf{z}'), \sup_{\mu \in M(\mathsf{z}, \alpha_1, \alpha_2)} \mu(i, \mathsf{z}') \right),$$

and $\delta'(\mathsf{z}, \cdot, \cdot)$ *is undefined if no such* α_1, α_2 *exist.*

All ingredients of an interval Markov game abstraction are finite. The lower bound of $\delta'(\mathsf{z}, \mathsf{c}, (\mathsf{c}, f))(i, \mathsf{z}')$ can be expressed using state variables as $\inf_{s \in wp(\mathsf{z}, \mathsf{c}, f)} \mathsf{p}_i(s)$, and correspondingly for the upper one. Thus, as for constraint Markov games, we have to solve optimisation problems. In contrast to the previous abstraction, optimisation problems are simpler. In addition, we have to compute the interval bounds only once and thus do not have to solve an optimisation problem in each step of the value iteration. Instead, we use an adaption of an existing algorithm for *interval Markov chains* [10] to compute these values: The optimisation over the interval part is described in the literature [10]. The optimisations over the other part of the player-2 choices and for the player-1 choices is simple, because there are only finitely many choices.

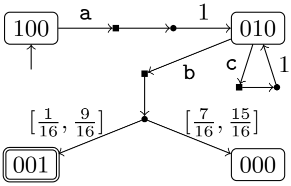

100
a
1
010
b
c
1
$[\frac{1}{16}, \frac{9}{16}]$
$[\frac{7}{16}, \frac{15}{16}]$
001
000

Fig. 5. Interval Markov game abstraction of the probabilistic program of Fig. 1

Example 5. Fig. 5 shows an interval Markov game abstraction of the program in Fig. 1. We choose the same predicates as in Example 4. In this example, we obtain the same bound $[\frac{1}{16}, \frac{9}{16}]$ for the maximal reachability probability.

Theorem 2. *Consider a probabilistic program* $\mathcal{P} = (\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$ *and a constraint or interval Markov game abstraction* $\mathcal{M} = (\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ *of* $\mathcal{P}$*, and let* $F \subseteq Dom(\mathtt{X})$ *be a set of goal states. Let* $F' = \{\mathsf{z} \in \mathcal{A} \mid \mathsf{z} \cap F \neq \emptyset\}$*. Then for* $\bullet \in \{+, -\}$*, for* $\mathsf{z} \in \mathcal{A}$ *and* $s \in \mathsf{z}$ *it is*

$$p^{\bullet,-}_{\mathcal{M},F'}(\mathsf{z}) \leq p^{\bullet}_{sem(\mathcal{P}),F}(s) \leq p^{\bullet,+}_{\mathcal{M},F'}(\mathsf{z}).$$

5 Abstraction Refinement

In Section 4, we discussed how to compute an abstraction for a probabilistic program, given a fixed set of predicates. However, the probability bounds computed by a given abstraction might be too coarse. This section describes an abstraction refinement technique to obtain tighter probability bounds, which introduces additional predicates, to separate concrete states with a behaviour too different to be subsumed. In the following, we focus on bounds for maximal reachability probability, as the approach is analogous for the minimal case.

Definition 14. *A* player-1 strategy *for a constraint or interval Markov game* $\mathcal{M} = (S, I, Act_1, Act_2, \delta)$ *is a function* $\sigma_1 \colon S \to Act_1$ *such that* $\sigma_1(s) \in en_1(s)$ *for each state* $s \in S$. *A* player-2 action strategy *is a function* $\sigma_2 \colon (S \times Act_1) \rightharpoonup Act_2$ *that is defined for all* $(s, \alpha_1) \in S \times Act_1$ *such that* $\alpha_1 \in en(s)$. *We require that* $\sigma_2(s, \alpha_1) \in en_2(s, \alpha_1)$. *A* player-2 constraint strategy *is a function* $\sigma_{2'} \colon (S \times Act_1 \times Act_2) \rightharpoonup Distr(\mathbb{N} \times S)$ *which is defined for all* $(s, \alpha_1, \alpha_2) \in S \times Act_1 \times Act_2$ *for which* $\delta(s, \alpha_1, \alpha_2)$ *is defined. We require that* $\sigma_{2'}(s, \alpha_1, \alpha_2) \in \delta(s, \alpha_1, \alpha_2)$. *By* $\Sigma^1_{\mathcal{M}}$ *we denote the set of all player-1 strategies of* $\mathcal{M}$, *by* $\Sigma^2_{\mathcal{M}}$ *all player-2 action strategies and by* $\Sigma^{2'}_{\mathcal{M}}$ *all player-2 constraint strategies.*

Definition 15. *For* $F \subseteq S$ *and* $\nu \in Asg(S)$, *let* $pre^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F} : Asg(S) \to Asg(S)$ *be defined such that* $pre^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} 1$ *if* $s \in F$ *and otherwise we have* $pre^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F}(\nu)(s) \overset{\text{def}}{=} \sum_{s' \in S} \sum_{i \in \mathbb{N}} \delta^{\sigma_1,\sigma_2,\sigma_{2'}}(s)(i, s') \cdot \nu(s')$. *There,* $\delta^{\sigma_1,\sigma_2,\sigma_{2'}}$ *is the natural composition of the strategies* σ_1, σ_2 *and* $\sigma_{2'}$. *Also, let* $p^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F}$ *be the least fixed point of* $pre^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F}$.

An optimal $+,-$ *player-1 strategy* $\sigma^{1,+,-}_{\mathcal{M},F}$ *is an element of*

$$\arg \max_{\sigma_1 \in \Sigma^1_{\mathcal{M}}} \min_{\sigma_2 \in \Sigma^2_{\mathcal{M}}} \min_{\sigma_{2'} \in \Sigma^{2'}_{\mathcal{M}}} p^{\sigma_1,\sigma_2,\sigma_{2'}}_{\mathcal{M},F}$$

and likewise the optimal player-2 strategies and the other cases (as the order of min *and* max *does not play a role), where the infimum and supremum are to be understood componentwise.*

As constraint and interval Markov games are special cases of Markov games, optimal values $p^{\bullet_1,\bullet_2}$ can be obtained by corresponding optimal strategies $\sigma^{1,\bullet_1}$, $\sigma^{2,\bullet_2}$,$\sigma^{2',\bullet_2}$ [6]. In the above we assume compactness of the sets $\delta(s, \alpha_1, \alpha_2)$. For interval Markov games, this assumption holds automatically.

When considering pairs of optimal strategies $\sigma^{1,+,-}_{\mathcal{M},F}$, $\sigma^{1,+,+}_{\mathcal{M},F}$ for player 1 against different objectives of player 2 we assume that their decisions agree if possible [18]. We make a corresponding assumption for player 2. Based on it we define predicates called *splitters* that remove the choices that make the strategies differ from the abstract model.

Definition 16. *Let* $(\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ *be a constraint or interval Markov game abstraction of a probabilistic program* $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$, $\mathsf{z} \in \mathcal{A}$ *and a command* $\mathsf{c} \in \mathtt{C}$ *with* $(\mathsf{c}, f^-) \overset{\text{def}}{=} \sigma^{2,+,-}_{\mathcal{M},F}(\mathsf{z}, \mathsf{c}) \neq \sigma^{2,+,+}_{\mathcal{M},F}(\mathsf{z}, \mathsf{c}) \overset{\text{def}}{=} (\mathsf{c}, f^+)$. *The* wp-based splitter *of* (z, c) *is* $\{wp(\mathsf{z}, \mathsf{c}, f^-), wp(\mathsf{z}, \mathsf{c}, f^+)\}$.

For models with constant probabilities, wp-based splitting has been described before [19]. It was enough to guarantee progress of the refinement approach since $\delta'(\mathsf{z}, \alpha_1, \alpha_2)$ was a singleton. We introduce a new complementary method to split abstract states when the imprecision is due to the constraint optimisation.

Definition 17. *Let* $(\mathcal{A}, \mathcal{I}, Act_1, Act_2, \delta')$ *be a constraint or interval Markov game abstraction of a probabilistic program* $(\mathtt{X}, \mathtt{Dom}, \mathtt{I}, \mathtt{C})$, $\mathsf{z} \in \mathcal{A}$, *a command* $\mathsf{c} \in \mathtt{C}$ *and*

$f\colon \{1,\dots,m\} \to \mathcal{A}$ *with* $\mu^- \overset{\text{def}}{=} \sigma^{2',+,-}_{\mathcal{M},F}(\mathsf{z},\mathsf{c},(\mathsf{c},f)) \neq \sigma^{2',+,+}_{\mathcal{M},F}(\mathsf{z},\mathsf{c},(\mathsf{c},f)) \overset{\text{def}}{=} \mu^+$. *The* constraint-based splitter *of* $(\mathsf{z},\mathsf{c},f)$ *is* $\{\{s\in S \mid \mathsf{p}_{i_{\max}}(s) \le \frac{U^+(i_{\max})+U^-(i_{\max})}{2}\}\}$ *where* $U^\bullet(i) \overset{\text{def}}{=} \sum_{\mathsf{z}\in\mathcal{A}} \mu^\bullet(i,\mathsf{z})$ *and* $i_{\max} \overset{\text{def}}{=} \arg\max_{i\in\mathbb{N}} |U^+(i)-U^-(i)|$.

To refine, we choose some splitters and add them to the set of predicates. Afterwards, we recompute a refined version of the abstraction with the new predicate set. There are some heuristics [19] to choose splitters that are likely to improve the bounds.

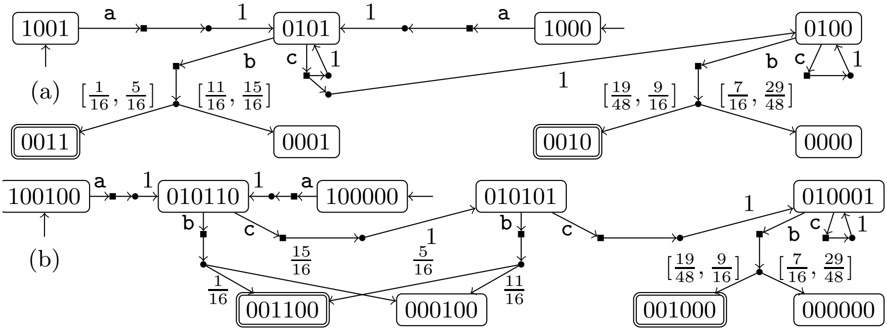


Fig. 6. Refinements of the model of Fig. 5

Example 6. Consider the interval Markov game abstraction in Fig. 5. There is no splitter based on *wp*. Notice that this implies that if we had constant p_i, we would have already an abstraction yielding exact reachability bounds. We can however obtain the predicate pred_4 equivalent to $(x \le 2)$ from a splitter based on the imprecision of the interval $[\frac{1}{16}, \frac{9}{16}]$. Adding it leads to the model depicted in Fig. 6 (a). We do not get an improvement of the reachability bounds directly. However, we can now obtain a *wp*-based splitter from state 0101 and command c with predicates $\mathsf{pred}_5 = (s = 1 \wedge x \le 1)$ and $\mathsf{pred}_6 = (s = 1 \wedge x \not\le 1)$. The refinement in part (b) yields the improved bound $[\frac{19}{48}, \frac{9}{16}]$.

6 Experiments

We implemented the abstraction and refinement methods described in this paper in our tool Pass [7]. Empirical evaluations of the techniques have been carried out on two different case studies with varying parameters and properties. The first case study corresponds to the von Neumann NAND multiplexer [17] which is used to construct reliable devices from unreliable gates. The second case study is based on the Rubinstein's Alternating Offers Protocol [2]. For both cases, we ran the abstraction refinement loop until precise bounds were achieved. Some characteristic observations for these experiments are summarised in Table 1. Column `total states` contains the numbers of reachable states as reported by Prism. Column `abs states` lists the numbers of abstract states handled

Table 1. Empirical results

Model	param	property	total states	abs states	preds	ref (int ref)	time
NAND	20/3	p0	78322	45496	107	21	1m30s
		p4		61644	132	10	1m10s
		p8		70968	129	17	7m49s
		p12		70968	135	23	9m32s
		p20		45496	136	52	16m05s
	20 / 9	p0	308162	45469	112	28	1m59s
		p4		61644	121	13 (5)	5m24s
		p8		70968	122	29	13m31s
		p12		70968	138	15	13m10s
		p20		45469	117	34	8m7s
Alternating Offers	K 1 / 8	pval	504	101	87	6	16s
	K 2 / 8		504	108	90	15 (9)	32s
	Cinc 10 / 100		922	351	85	16 (9)	22s
	Cinc 100 / 10		504	101	94	17 (4)	24s
	T 2000		3848	133	68	4	6s
	B_RP 5000		?	351	90	21 (11)	22s

in the final Pass refinement step, i.e. when the precise bound was obtained. Column `preds` refers to the total number of predicates used to construct the final abstraction. Column `ref` reports the total number of refinement steps, and column `int ref` refers to the number of refinement steps in which our new refinement approach for intervals was used. Column `time` reports the total time spent until Pass built a precise abstraction. As can be seen, the abstract state numbers are generally below the total state numbers, and the time consumptions are acceptable, in view of the overall work carried out. In the last model Prism was not able to build the model due to memory exhaustion. The problem seems to be the large amount of terminal nodes introduced due to variable probabilities in conjunction with the construction of the transition matrix for the possible state space. In the NAND case study most of the time in Pass was spent on internal BDD garbage collection and node creation. This is rooted in the lack of automatic reordering of variables after each refinement step. In our experiments we have observed that the number of abstract states in the final refinement step is in the order of the number of states with non-trivial probability ($\notin \{0, 1\}$) in the original model. Considerably smaller state spaces can be obtained if one does not run the refinement loop until an exact probability results, but only until a safe bound is established.

7 Conclusion

We have introduced new abstraction and refinement methods for probabilistic systems with variable probabilities, which, unlike previous symbolic abstractions, explicitly abstract transition probabilities. While our current experimental results are already very promising, being able to deal with variable probabilities, opens up a whole spectrum of potential applications and case studies, which we would like to study, ranging from sensor network protocols to biochemical reaction cascades. The experiments also provide directions for further performance improvements of the prototype implementation in this direction.

The presented abstractions extend menu-based abstraction [19], which is distinct from the game-based abstraction defined in [14]. We conjecture that our notion of interval abstraction could be fruitfully combined with the latter. However, the challenge would be to prevent a combinatorial explosion that results from keeping track of the interaction of different concurrent commands, and resulting interval bounds – a problem that does not occur with the present abstraction.

Acknowledgements. This work has been 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), by the DFG/NWO Bilateral Research Programme ROCKS and by the European Union 7th Framework Programme under grant agreement number 295261 (MEALS).

References

1. de Alfaro, L., Roy, P.: Magnifying-Lens Abstraction for Markov Decision Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 325–338. Springer, Heidelberg (2007)
2. Ballarini, P., Fisher, M., Wooldridge, M.: Automated game analysis via probabilistic model checking: a case study. ENTCS 149(2), 125–137 (2006)
3. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability Analysis of Probabilistic Systems by Successive Refinements. In: de Luca, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
4. 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)
5. Ferrer Fioriti, L.M., Hahn, E.M., Hermanns, H., Wachter, B.: Variable probabilistic abstraction refinement. Tech. Rep. 87, SFB/TR 14 AVACS (2012)
6. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer (1996)
7. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: Abstraction Refinement for Infinite Probabilistic Models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)
8. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
9. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277. IEEE Computer Society (1991)
10. Katoen, J.P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for probabilistic systems. Journal on Logic and Algebraic Programming, 1–55 (2012)
11. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: Abstraction Refinement for Probabilistic Software. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
12. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods in System Design 36(3), 246–280 (2010)
13. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)

14. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: QEST, pp. 157–166 (2006)
15. Kwiatkowska, M.Z., Norman, G., Parker, D.: A Framework for Verification of Software with Time and Probabilities. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 25–45. Springer, Heidelberg (2010)
16. Lazimy, R.: Mixed-integer quadratic programming. Mathematical Programming 22, 332–349 (1982)
17. Norman, G., Parker, D., Kwiatkowska, M.Z., Shukla, S.K.: Evaluating the reliability of NAND multiplexing with PRISM. TCAD 24(10), 1629–1637 (2005)
18. Wachter, B.: Refined Probabilistic Abstraction. Ph.D. thesis, Saarland Univ. (2010)
19. Wachter, B., Zhang, L.: Best Probabilistic Transformers. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 362–379. Springer, Heidelberg (2010)

Pareto Curves for Probabilistic Model Checking

Vojtěch Forejt[1], Marta Kwiatkowska[1], and David Parker[2]

[1] Department of Computer Science, University of Oxford, UK
[2] School of Computer Science, University of Birmingham, UK

Abstract. Multi-objective probabilistic model checking provides a way to verify several, possibly conflicting, quantitative properties of a stochastic system. It has useful applications in controller synthesis and compositional probabilistic verification. However, existing methods are based on linear programming, which limits the scale of systems that can be analysed and makes verification of time-bounded properties very difficult. We present a novel approach that addresses both of these shortcomings, based on the generation of successive approximations of the Pareto curve for a multi-objective model checking problem. We illustrate dramatic improvements in efficiency on a large set of benchmarks and show how the ability to visualise Pareto curves significantly enhances the quality of results obtained from current probabilistic verification tools.

1 Introduction

Probabilistic model checking is an automated technique for verifying systems that exhibit stochastic behaviour. This arises due to, for example, failures in physical components, unreliable communication media or randomisation. Systems are typically modelled as Markov chains or Markov decision processes (MDPs), and probabilistic temporal logics are used to specify quantitative properties to be verified such as "the probability of a message packet being lost is less than 0.05" or "the expected energy consumption is at most 100 mJ".

It is often necessary to incorporate *nondeterminism* into system models, to represent, for example, the actions of an external controller or the order in which a scheduler chooses to interleave system components running in parallel. In these cases, systems are usually modelled as MDPs. Each possible way of resolving the nondeterminism in an MDP is represented by an *adversary* (also known as a strategy or policy). Properties to be verified against the MDP quantify over its adversaries, e.g., "the probability of a message packet being lost is less than 0.05 *for all possible adversaries*". It is also common to use numerical queries, e.g., "what is the *maximum* expected energy consumption?".

Model checking reduces to an *optimisation* problem, namely determining the maximum (or minimum) probability (or expected cost/reward) achievable by any adversary. For the most common classes of property (the probability of reaching a set of target states or the expected cumulated reward), model checking can be reduced to solving a linear programming (LP) problem. In practice, however, most probabilistic verification tools use (approximate) iterative numerical methods, such as *value iteration* [19], since they scale to much larger systems and are

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 317–332, 2012.

amenable to symbolic (BDD-based) implementations. Value iteration can also be used for *time-bounded* (finite-horizon) properties, which is impractical with LP. Another alternative is policy iteration, but this is also impractical for time-bounded properties, and preliminary investigations in [10] showed no particular improvement over value iteration in the context of probabilistic verification.

There has recently been increased interest in *multi-objective* probabilistic model checking for MDPs [5,9,11,4], which can be used to analyse trade-offs between several, possibly conflicting, quantitative properties. Consider, for example, two events of interest, A and B, and let p_A^σ and p_B^σ be the probability that each occurs under an adversary σ of an MDP. In this paper, we study several kinds of multi-objective properties. *Achievability* queries ask, e.g., "is there an adversary σ satisfying the predicate $\boldsymbol{\psi} = p_A^\sigma \geqslant x \wedge p_B^\sigma \geqslant y$?" and *numerical* queries ask, e.g., "what is maximum value of x such that $\boldsymbol{\psi}$ is achievable?". We also consider the *Pareto curve* of *undominated* solution points: for this example, the set of pairs (x, y) such that $\boldsymbol{\psi}$ is achievable but any increase in either x or y would necessitate a decrease in the other.

Multi-objective techniques have natural applications to *controller synthesis* for MDPs (e.g., "how can we maximise the probability of successful message transmission, whilst keeping the expected energy usage below 100 mJ?"). They also form the basis of recent *compositional verification* techniques [15], which decompose model checking into separate tasks for each system component using assume-guarantee reasoning (e.g., "what is the maximum probability of a global system error, under the assumption that component 1 fails with probability at most 0.02?"). This approach has been successfully used to verify probabilistic systems too large to analyse without compositional techniques.

Existing multi-objective model checking methods [5,9,11,4] rely on a reduction to LP. The linear program solved, although of a rather different form to the standard (single objective) case, is still linear in the size of the MDP, yielding polynomial time complexity. As discussed above, though, LP-based probabilistic verification has several important weaknesses. In this paper, we present a novel approach to multi-objective model checking of probabilistic reachability and expected total reward properties. Our method is based on the generation of successive, increasingly precise approximations to the Pareto curve by optimising weighted sums of objectives using value iteration. On a large selection of benchmarks, we demonstrate the following benefits:

(i) dramatic improvements in run-time *efficiency*, by factors of up to 150;
(ii) significant *scalability* improvements: over an order of magnitude model size;
(iii) the usefulness of visualising *Pareto curves* for verification problems;
(iv) solution of *time-bounded* probabilistic reachability and cumulative reward.

The last of these also paves the way for the development of multi-objective techniques for richer, *timed* classes of models such as continuous-time MDPs.

An extended version of this paper, including proofs, is also available [12].

Related work. Multi-objective optimisation has been extensively studied in areas such as operations research, economics and stochastic control [6], including its application to MDPs [1]. Many general approaches exist, based on, for example, *normalising* multiple objectives into a single weighted objective; *constrained* approaches optimising one objective while bounding the others; heuristic search using, e.g., *evolutionary algorithms* [7]; application of satisfiability/constraint solvers [17]; and stochastic search with restarts [16]. Several methods, including e.g. [16], iterate over weighted sums of objectives, as we do; the main difference is that our approach is tailored to the convex, linear problems derived from MDPs.

Multi-objective optimisation is routinely used in areas such as embedded systems design and a variety of general-purpose optimisation tools exist, e.g., PISA [2], ModeFRONTIER and libraries for MATLAB. Such tools tend to be targeted at much more complex (e.g., non-convex and non-linear) design spaces than the ones that we focus on in this paper. They are also typically used for *static* design problems, rather than our *dynamic* models of system behaviour.

A rigorous complexity analysis of multi-objective optimisation, in particular for approximating *Pareto curves*, was undertaken in the influential work of [18]. More recently [8], some of these results were improved for the simpler case of *convex* multi-objective problems but no practical investigations are undertaken.

Most relevant to the current work is the application of multi-objective optimisation to *probabilistic verification* [5,9,11,4]. In [5,9,4], discounted total reward, probabilistic ω-regular and long-run average properties are studied, respectively. In each case, algorithms are given using a reduction to LP, also showing the existence of methods to approximate Pareto curves using the results of [18], but implementations are not considered. The work of [11] adds expected total reward properties and provides an implementation, based on LP. As discussed above, the performance and scalability of our approach is significantly better, as we show in Section 5. None of the above consider time-bounded properties. In principle, for discrete-time models like MDPs, these can be reduced to unbounded properties using a finite counter but this is generally impractical in terms of scalability.

2 Background

Geometry. For a vector $\boldsymbol{x} \in \mathbb{R}^n$, we use x_i to denote its i-th component and say $\boldsymbol{x}$ is a *weight vector* if $x_i \geqslant 0$ for all i and $\sum_{i=1}^{n} x_i = 1$. The *Euclidean inner product* of $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ is defined as $\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i \cdot y_i$. For a set of vectors $X = \{\boldsymbol{x}_1, \dots \boldsymbol{x}_k\} \subseteq \mathbb{R}^n$, a *convex combination* is $\sum_{j=1}^{k} w_j \cdot \boldsymbol{x}_j$ for some weight vector $\boldsymbol{w} \in \mathbb{R}^k$. We use $down(X)$ to denote the *downward closure* of the convex hull of X, i.e. the set of vectors $\boldsymbol{z} \in \mathbb{R}^n$ that satisfy $\boldsymbol{z} \leqslant \boldsymbol{y}$ for some convex combination $\boldsymbol{y}$ of X. Given a convex set Y, we say that a point $\boldsymbol{y} \in Y$ is on the *boundary* of Y if, for any $\varepsilon > 0$, there is a point $\boldsymbol{z} \notin Y$ such that the Euclidean distance between $\boldsymbol{y}$ and $\boldsymbol{z}$ is at most ε. From the *separating hyperplane* and *supporting hyperplane* theorems, we have the following.

Proposition 1 ([3]). *Let $Y \subseteq \mathbb{R}^n$ be a downward closed set of points. For any $\boldsymbol{p} \in \mathbb{R}^n$ not in Y, there is a weight vector $\boldsymbol{w} \in \mathbb{R}^n$ such that $\boldsymbol{w} \cdot \boldsymbol{p} > \boldsymbol{w} \cdot \boldsymbol{y}$ for all $\boldsymbol{y} \in Y$. Also, for any $\boldsymbol{q}$ on the boundary of Y, there is a weight vector $\boldsymbol{w} \in \mathbb{R}^n$ such that $\boldsymbol{w} \cdot \boldsymbol{q} \geqslant \boldsymbol{w} \cdot \boldsymbol{y}$ for all $\boldsymbol{y} \in Y$. We say that $\boldsymbol{w}$* separates $\boldsymbol{q}$ from *$down(Y)$.*

Markov decision processes (MDPs). MDPs are commonly used to model systems with probabilistic and nondeterministic behaviour. Denoting by $\mathit{Dist}(X)$ the set of probability distributions over a set X, an *MDP* takes the form $\mathcal{M} = (S, \bar{s}, \alpha, \delta)$, where S is a set of states, $\bar{s} \in S$ is an initial state, α is a set of actions and $\delta : S \times \alpha \to \mathit{Dist}(S)$ is a (partial) probabilistic transition function.

Each state s of an MDP $\mathcal{M}$ has an associated set $A(s)$ of *enabled* actions, given by $A(s) \stackrel{\text{def}}{=} \{a \in \alpha \,|\, \delta(s,a) \text{ is defined}\}$. If action $a \in A(s)$ is taken in state s, then the next state is determined randomly according to the distribution $\delta(s,a)$, i.e., a transition to state s' occurs with probability $\delta(s,a)(s')$. A *path* through $\mathcal{M}$ is a (finite or infinite) sequence $\pi = s_0 a_0 s_1 a_1 \ldots$ where $s_0 = \bar{s}$, and $a_i \in A(s_i)$ and $\delta(s_i, a_i)(s_{i+1}) > 0$ for all i. We denote by *IPaths* (*FPaths*) the set of all infinite (finite) paths and, for finite path π, $\mathit{last}(\pi)$ is its last state.

An *adversary* $\sigma : \mathit{FPaths} \to \mathit{Dist}(\alpha)$ (also called a strategy or policy) of $\mathcal{M}$ is a resolution of the choices of action in each state, based on its execution so far. In standard fashion [13], an adversary σ induces a probability measure $\mathit{Pr}^{\sigma}_{\mathcal{M}}$ over *IPaths*. An adversary σ is *deterministic* if $\sigma(\pi)$ is a Dirac distribution for all π (and *randomised* if not); it is *memoryless* if $\sigma(\pi)$ depends only on $\mathit{last}(\pi)$. The set of all adversaries for $\mathcal{M}$ is $\mathit{Adv}_{\mathcal{M}}$.

A *reward structure* for $\mathcal{M}$ is a function $\rho : S \times \alpha \to \mathbb{R}$ mapping actions to (positive or negative) reals. For an infinite path $\pi = s_0 a_0 s_1 a_1 \ldots$ and a number $k \in \mathbb{N} \cup \{\infty\}$ the *total reward in k steps* for π over ρ is $\rho[k](\pi) \stackrel{\text{def}}{=} \sum_{i=0}^{k-1} \rho(s_i, a_i)$.

Model checking MDPs. In this paper, we focus on two key classes of properties for MDPs: the *probability of reaching a target* and the *expected total reward*. In each case, we consider both time-bounded and unbounded variants. We will later discuss generalisation to more expressive properties. In this and the following sections, we assume a fixed MDP $\mathcal{M} = (S, \bar{s}, \alpha, \delta)$.

Definition 1 (Reachability predicate). *A* reachability predicate $[T]^{\leqslant k}_{\sim p}$ *comprises a set of target states $T \subseteq S$, a relational operator $\sim \in \{\geqslant, \leqslant\}$, a rational probability bound p and a time-bound $k \in \mathbb{N} \cup \{\infty\}$. It states that the probability of reaching T within k steps satisfies $\sim p$. Formally, satisfaction of $[T]^{\leqslant k}_{\sim p}$ by MDP $\mathcal{M}$, under adversary σ, denoted $\mathcal{M}, \sigma \models [T]^{\leqslant k}_{\sim p}$, is defined as follows:*

$$\mathcal{M}, \sigma \models [T]^{\leqslant k}_{\sim p} \Leftrightarrow \mathit{Pr}^{\sigma}_{\mathcal{M}}(\{s_0 a_0 s_1 a_1 \cdots \in \mathit{IPaths} \mid \exists i \leqslant k \,:\, s_i \in T\}) \sim p\,.$$

Definition 2 (Reward predicate). *A* reward predicate $[\rho]^{\leqslant k}_{\sim r}$ *comprises a reward structure $\rho : S \times \alpha \to \mathbb{R}$, a relational operator $\sim \in \{\geqslant, \leqslant\}$, a rational reward bound r and a time bound $k \in \mathbb{N} \cup \{\infty\}$. It states that the* expected total reward *cumulated within k steps satisfies $\sim r$. Formally, satisfaction of $[\rho]^{\leqslant k}_{\sim r}$ by $\mathcal{M}$, under adversary σ, denoted $\mathcal{M}, \sigma \models [\rho]^{\leqslant k}_{\sim r}$, is defined as follows:*

$$\mathcal{M}, \sigma \models [\rho]^{\leqslant k}_{\sim r} \;\Leftrightarrow\; \mathit{ExpTot}^{\sigma,k}_{\mathcal{M}}(\rho) \sim r \quad \textit{where} \quad \mathit{ExpTot}^{\sigma,k}_{\mathcal{M}}(\rho) \;\stackrel{\text{def}}{=}\; \int_{\pi} \rho[k](\pi)\, d\mathit{Pr}^{\sigma}_{\mathcal{M}}.$$

For the *unbounded* forms of the notation above ($k = \infty$), we will often omit k, writing e.g. $[\rho]_{\sim r}$ instead of $[\rho]^{\leqslant \infty}_{\sim r}$ or $\mathit{ExpTot}^{\sigma}_{\mathcal{M}}(\rho)$ instead of $\mathit{ExpTot}^{\sigma,\infty}_{\mathcal{M}}(\rho)$.

For this paper, we also need to consider *weighted sums* of rewards.

Definition 3 (Weighted reward sum). *Given a weight vector $\boldsymbol{w} \in \mathbb{R}^n$ and vectors of time bounds $\boldsymbol{k} = (k_1, \ldots, k_n) \in (\mathbb{N} \cup \{\infty\})^n$ and reward structures $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_n)$ for MDP $\mathcal{M}$, the* weighted reward sum $\boldsymbol{w}\cdot\boldsymbol{\rho}[\boldsymbol{k}]$ *over a path π is defined as $\boldsymbol{w}\cdot\boldsymbol{\rho}[\boldsymbol{k}](\pi) \overset{\text{def}}{=} \sum_{i=1}^n w_i \rho_i[k](\pi)$. The* expected total weighted sum *is then: $ExpTot_{\mathcal{M}}^{\sigma,\boldsymbol{k}}(\boldsymbol{w}\cdot\boldsymbol{\rho}) \overset{\text{def}}{=} \int_\pi \boldsymbol{w}\cdot\boldsymbol{\rho}[\boldsymbol{k}](\pi)\, dPr_{\mathcal{M}}^{\sigma}$. For any adversary σ, we have: $ExpTot_{\mathcal{M}}^{\sigma,\boldsymbol{k}}(\boldsymbol{w}\cdot\boldsymbol{\rho}) = \sum_{i=1}^n w_i ExpTot_{\mathcal{M}}^{\sigma,k_i}(\rho_i)$.*

Notice that satisfaction of reachability and reward predicates is defined above with respect to a specific adversary σ of an MDP $\mathcal{M}$. When performing model checking on the MDP, the most common approach is to verify that such a predicate is satisfied *for all* adversaries $\sigma \in Adv_{\mathcal{M}}$. An alternative, often described as *controller synthesis*, is to ask the dual question: whether *there exists* an adversary σ satisfying the predicate. In either case, model checking reduces to computing the maximum or minimum reachability probability or expected reward. For the unbounded cases, this can be done by solving an LP problem, using policy iteration, or with value iteration, an approximate iterative numerical method [19]. For time-bounded properties, only value iteration is applicable.

3 Multi-objective Queries

We now describe how to formalise multi-objective queries for MDPs. In the following section, we will present novel, efficient algorithms for their verification. We formulate our queries in a similar style to the one taken in [11], but with two key additions. Firstly, we include the ability to specify *time-bounded* reachability and reward properties. Secondly, we consider *Pareto curves*.

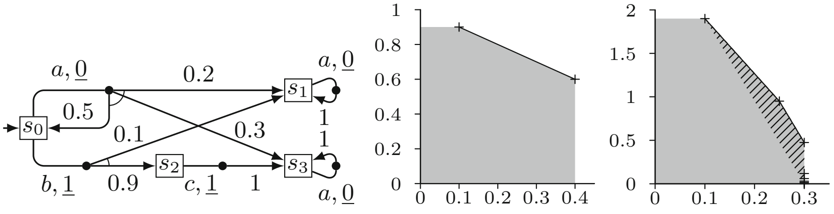
The essence of multi-objective properties for MDPs is that they require multiple predicates to be satisfied concurrently for the same adversary.

Definition 4 (Multi-objective predicate). *A* multi-objective predicate *is a vector $\boldsymbol{\psi}=(\psi_1, \ldots, \psi_n)$ of reachability or reward predicates. We say that $\boldsymbol{\psi}$ is* satisfied *by MDP $\mathcal{M}$ under adversary σ, denoted $\mathcal{M}, \sigma \models \boldsymbol{\psi}$, if $\mathcal{M}, \sigma \models \psi_i$* for all $1 \leqslant i \leqslant n$. *We call $\boldsymbol{\psi}$ a* basic multi-objective predicate *if it is of the form $([\rho_1]_{\geqslant r_1}^{\leqslant k_1}, \ldots, [\rho_n]_{\geqslant r_n}^{\leqslant k_n})$, i.e. it comprises only lower-bounded reward predicates.*

We define three ways to formulate multi-objective queries for an MDP: *achievability queries*, which check for the existence of an adversary satisfying a multi-objective predicate $\boldsymbol{\psi}$; *numerical queries*, which maximise or minimise a reachability/reward objective over the set of adversaries satisfying $\boldsymbol{\psi}$; and *Pareto queries*, which determine the Pareto curve for a set of objectives.

Definition 5 (Achievability query). *For MDP $\mathcal{M}$ and multi-objective predicate $\boldsymbol{\psi}$, an* achievability query *asks if $\boldsymbol{\psi}$ is* satisfiable *(or* achievable*), i.e. whether there exists an adversary $\sigma \in Adv_{\mathcal{M}}$ such that $\mathcal{M}, \sigma \models \boldsymbol{\psi}$.*

Definition 6 (Numerical query). *For MDP $\mathcal{M}$, a* numerical query *is of the form $num([o_1]_\star^{\leqslant k_1}, (\psi_2 \ldots, \psi_n))$, comprising an $n{-}1$-sized multi-objective predicate $(\psi_2 \ldots, \psi_n)$ and an* objective $[o_1]_\star^{\leqslant k_1}$, *where o_1 is a reward structure ρ_1 or target set T_1, $k_1 \in \mathbb{N} \cup \{\infty\}$ is a time bound and $\star \in \{\min, \max\}$. We define:*



(a) (b) (c)

Fig. 1. Example MDP (a), graphs for $([\{s_1\}]_{\geqslant x}, [\{s_3\}]_{\geqslant y})$ (b) and $([\{s_1\}]^{\leqslant 2}_{\geqslant x}, [\rho]_{\geqslant y})$ (c)

$$num([o_1]^{\leqslant k_1}_{\min}, (\psi_2, \ldots, \psi_n)) \overset{\text{def}}{=} \inf\{x \in \mathbb{R} \mid ([o_1]^{\leqslant k_1}_{\leqslant x}, \psi_2, \ldots, \psi_n) \textit{ is satisfiable}\}\,.$$
$$num([o_1]^{\leqslant k_1}_{\max}, (\psi_2, \ldots, \psi_n)) \overset{\text{def}}{=} \sup\{x \in \mathbb{R} \mid ([o_1]^{\leqslant k_1}_{\geqslant x}, \psi_2, \ldots, \psi_n) \textit{ is satisfiable}\}\,.$$

Definition 7 (Pareto query). *For MDP $\mathcal{M}$, a* Pareto query *takes the form $pareto([o_1]^{\leqslant k_1}_{\star_1}, \ldots, [o_n]^{\leqslant k_n}_{\star_n})$, where each $[o_i]^{\leqslant k_i}_{\star_i}$ is an objective as in Defn. 6. The set of* achievable values *is $A = \{\boldsymbol{x} \in \mathbb{R}^n \,|\, ([o_1]^{\leqslant k_1}_{\sim_1 x_1}, \ldots, [o_n]^{\leqslant k_n}_{\sim_n x_n})$ is satisfiable$\}$ where $\sim_i = \geqslant$ if $\star_i = \max$ and $\sim_i = \leqslant$ if $\star_i = \min$. We say, for points $\boldsymbol{x}, \boldsymbol{y} \in A$, that $\boldsymbol{x}$* dominates *$\boldsymbol{y}$ if $x_i \sim_i y_i$ for all i and $x_j \neq y_j$ for some j. Then:*

$$pareto([o_1]^{\leqslant k_1}_{\star_1}, \ldots, [o_n]^{\leqslant k_n}_{\star_n}) \overset{\text{def}}{=} \{\boldsymbol{x} \in A \mid \boldsymbol{x} \textit{ is not dominated by any } \boldsymbol{y} \in A\}\,.$$

Convexity. A fundamental property of the multi-objective optimisation problems solved in this paper (and on MDPs in general) is their *convexity*. More precisely, consider target sets $T_1, \ldots, T_n$, reward structures $\rho_1, \ldots, \rho_m$ and time-bounds $k_1, \ldots, k_n, l_1, \ldots, l_m \in \mathbb{N} \cup \{\infty\}$. Let $\boldsymbol{x}^\sigma \in \mathbb{R}^{n+m}$ be the vector defined such that $x_i = Pr^\sigma_{\mathcal{M}}(\Diamond^{\leqslant k_i} T_i)$ for $1 \leqslant i \leqslant n$ and $x_{n+j} = ExpTot^{\sigma,k_j}_{\mathcal{M}}(\rho_j)$ for $1 \leqslant j \leqslant m$, where $Pr^\sigma_{\mathcal{M}}(\Diamond^{\leqslant k_i} T_i)$ denotes the probability of reaching T_i in k_i steps under σ. Then, the set $\{\boldsymbol{x}^\sigma \,|\, \sigma \in Adv_{\mathcal{M}}\}$ forms a convex polytope [9,11][1]. As a direct consequence of this, the set of *achievable values* for a Pareto query is also convex.

Example 1. Fig. 1(a) shows an MDP with accompanying reward structure ρ indicated by underlined numbers. Consider first the multi-objective predicate $\boldsymbol{\psi} = ([\{s_1\}]_{\geqslant x}, [\{s_3\}]_{\geqslant y})$, which imposes lower bounds on the probabilities of reaching states s_1 and s_3. The grey area in Fig. 1(b) shows the values of x and y for which $\boldsymbol{\psi}$ is satisfiable. The two points on the graph marked as + correspond to the two possible *memoryless deterministic* adversaries in $\mathcal{M}$. The line joining them (their convex closure) represents the points for all possible adversaries. For this example, this line also constitutes the Pareto curve. Achievability queries on $\boldsymbol{\psi}$ for $(x, y) = (0.2, 0.7)$ and $(0.4, 0.7)$ return true and false, respectively. Numerical query $num([\{s_1\}]_{\max}, ([\{s_3\}]_{\geqslant 0.7}))$ returns 0.3.

Consider a second predicate $\boldsymbol{\psi}' = ([\{s_1\}]^{\leqslant 2}_{\geqslant x}, [\rho]_{\geqslant y})$, now with a time-bounded reachability and reward predicate. Fig. 1(c) depicts (by +) points for some of the *deterministic* adversaries of $\mathcal{M}$, of which there are infinitely many. Their convex combination, the dashed area, marks the points achievable by *all* (randomised)

[1] Strictly speaking, this requires finiteness of rewards, which we discuss below.

adversaries, and its downward closure, in grey, shows the values of x and y for which $\boldsymbol{\psi}'$ is satisfiable. The Pareto curve is the black line along the top edge.

Assumptions. For the purposes of model checking the queries described in this section, we need to impose certain restrictions on the use of rewards. For clarity, we describe these in terms of achievability queries but they apply to all three classes. We first need the following definition.

Definition 8 (Reward-finiteness). *Let $\mathcal{M}$ be an MDP and consider an achievability query $\boldsymbol{\psi}{=}([T_1]^{\leqslant k_1}_{\sim_1 p_1}, \ldots, [T_n]^{\leqslant k_n}_{\sim_n p_n}, [\rho_1]^{\leqslant l_1}_{\bowtie_1 r_1}, \ldots, [\rho_m]^{\leqslant l_m}_{\bowtie_m r_m})$ for $\mathcal{M}$. We say that $\boldsymbol{\psi}$ is* reward-finite *if, for each $1{\leqslant}j{\leqslant}m$ such that $l_j{=}\infty$ and $\bowtie_j\ =\ \geqslant$, we have:* $\sup\{ExpTot^{\sigma,l_j}_{\mathcal{M}}(\rho_j) \mid \mathcal{M},\sigma \models ([T_1]^{\leqslant k_1}_{\sim_1 p_1}, \ldots, [T_n]^{\leqslant k_n}_{\sim_n p_n})\} < \infty$ *and is* fully reward-finite *if, for each $1{\leqslant}j{\leqslant}m$ such that $l_j{=}\infty$ and $\bowtie_j\ =\ \geqslant$, we have:* $\sup\{ExpTot^{\sigma,l_j}_{\mathcal{M}}(\rho_j) \mid \sigma \in Adv_{\mathcal{M}}\} < \infty$.

Let $\mathcal{M}$ and $\boldsymbol{\psi}$ be as in Defn. 8. To model check $\boldsymbol{\psi}$ on $\mathcal{M}$, we require that: (i) each reward structure ρ_i assigns only non-negative values; (ii) $\boldsymbol{\psi}$ is reward-finite; and (iii) for indices $1{\leqslant}j{\leqslant}m$ such that $l_j{=}\infty$, either all $\bowtie_j$s are $\geqslant$ or all are $\leqslant$.

Condition (ii) imposes natural restrictions on finiteness of rewards. Notice that we only require finiteness for adversaries which satisfy the probabilistic predicates contained in $\boldsymbol{\psi}$. We adopt this approach from [11] where, in addition, algorithms are given to check that $\boldsymbol{\psi}$ is reward-finite and to construct a modified MDP that is equivalent (in terms of satisfiability of $\boldsymbol{\psi}$) but for which $\boldsymbol{\psi}$ is *fully* reward-finite. This can be checked by a simpler multi-objective query containing only probabilistic predicates. Thus, in the remainder of this paper, we assume that all queries are fully reward-finite.

Condition (iii) ensures that the algorithms we define in the next section do not need to compute *unbounded expected total rewards* for MDPs with *both positive and negative rewards*, which is unsound. For unbounded reachability predicates, again using methods from [11], we can easily invert their bounds (to match those of any reward predicates) by making a simple change to the MDP.

Extensions. We also remark that the class of multi-objective properties outlined in this section can be extended in several respects. In particular, as shown in [11], we can add support for probabilistic ω-regular (e.g. LTL) properties via reduction to probabilistic reachability on a product of the MDP and one or more deterministic Rabin automata. That work also allows arbitrary Boolean combinations of predicates, which are reduced to disjunctive normal form and treated separately. Both of these extensions can be adapted to our setting; the former we have implemented and used for our experiments in Section 5.

4 Multi-objective Probabilistic Model Checking

We now present efficient algorithms for checking the multi-objective queries defined in the previous section. Proofs of correctness can be found in [12].

Reduction to basic form. The first step when checking any type of query is to reduce the problem to one over a *basic* predicate on a modified MDP.

```
Input: MDP $\mathcal{M}$, multi-objective predicate $\boldsymbol{\psi} = ([\rho_1]^{\leqslant k_1}_{\geqslant r_1}, \dots, [\rho_n]^{\leqslant k_n}_{\geqslant r_n})$
Output: true if $\boldsymbol{\psi}$ is achievable, false if not
$X := \emptyset$; $\boldsymbol{\rho} = (\rho_1, \dots \rho_n)$; $\boldsymbol{k} = (k_1, \dots k_n)$; $\boldsymbol{r} = (r_1, \dots r_n)$;
do
    Find $\boldsymbol{w}$ separating $\boldsymbol{r}$ from $down(X)$;
    Find adversary $\sigma$ maximising $ExpTot^{\sigma,\boldsymbol{k}}_{\mathcal{M}}(\boldsymbol{w}\cdot\boldsymbol{\rho})$;
    $\boldsymbol{q} := (ExpTot^{\sigma,k_i}_{\mathcal{M}}(\rho_i))_{1\leqslant i\leqslant n}$;
    if $\boldsymbol{w} \cdot \boldsymbol{q} < \boldsymbol{w} \cdot \boldsymbol{r}$ then return false;
    $X := X \cup \{\boldsymbol{q}\}$;
while $\boldsymbol{r} \notin down(X)$;
return true;
```

Alg. 1. Basic algorithm for checking achievability queries

We do so by converting reachability predicates into reward predicates (by adding a one-off reward of 1 upon reaching the target) and then negating objectives for predicates with upper bounds. Formally, we do the following.

Proposition 2. *Let* $\mathcal{M}{=}(S, \bar{s}, \alpha, \delta)$ *be an MDP and* $\boldsymbol{\psi}{=}([T_1]^{\leqslant k_1}_{\sim_1 p_1}, \dots, [T_n]^{\leqslant k_n}_{\sim_n p_n}, [\rho_1]^{\leqslant l_1}_{\bowtie_1 r_1}, \dots, [\rho_m]^{\leqslant l_m}_{\bowtie_m r_m})$ *be a multi-objective predicate. Let* $\mathcal{M}'{=}(S', (\bar{s}, \emptyset), \alpha', \delta')$ *be the MDP defined as follows:* $S'{=}S \times 2^{\{1,\dots,n\}}$, $\alpha'{=}\alpha \times 2^{\{1,\dots,n\}}$ *and, for all* $s, s' \in S$, $a \in \alpha$ *and* $c \subseteq \{1, \dots, n\}$*:*

- $\delta'((s, c), (a, c'))((s', c \cup c')) = \delta(s, a)(s')$ *where* $c' = \{i \mid s \in T_i\} \setminus c$*;*
- $\delta'((s, c), a')((s', c')) = 0$ *for all other* c, c' *and* a'*.*

Now, let $\boldsymbol{\psi}'$ *be* $([\rho_{T_1}]^{\leqslant k_1+1}_{\geqslant p_1}, \dots, [\rho_{T_n}]^{\leqslant k_n+1}_{\geqslant p_n}, [\bar{\rho}_1]^{\leqslant l_1}_{\geqslant r_1}, \dots, [\bar{\rho}_m]^{\leqslant l_m}_{\geqslant r_m})$*, where: reward* $\rho_{T_i}((s, c), (a, c'))$ *is equal to* 1 *if* $i \in c'$ *and* $\sim_i = \geqslant$*, to* -1 *if* $\sim_i = \leqslant$*, and to* 0 *otherwise; and* $\bar{\rho}_i((s, c), (a, c'))$ *is equal to* $\rho_i(s, a)$ *if* $\bowtie_i = \geqslant$ *and to* $-\rho_i(s, a)$ *if* $\sim_i = \leqslant$*. Then* $\boldsymbol{\psi}$ *is satisfiable in* $\mathcal{M}$ *if and only if* $\boldsymbol{\psi}'$ *is satisfiable in* $\mathcal{M}'$*.*

Notice that the reduction described above results in reward structures with both positive and negative rewards. For time-bounded properties, this is not a concern. For unbounded ones, we must take care that they are all either non-negative or non-positive, as mentioned earlier in our discussion of condition (iii).

Achievability queries. We begin with achievability queries. We first give an outline of the overall algorithm; subsequently, we will describe in more detail how it is implemented in practice using value iteration.

By applying the reduction described above, we only need to consider the case of a *basic* multi-objective predicate $\boldsymbol{\psi} = ([\rho_1]^{\leqslant k_1}_{\geqslant r_1}, \dots, [\rho_n]^{\leqslant k_n}_{\geqslant r_n})$. Alg. 1 shows how to check if $\boldsymbol{\psi}$ is satisfiable. It works by generating a sequence of weight vectors $\boldsymbol{w}$ and optimising a $\boldsymbol{w}$-weighted sum of the n objectives. A resulting optimal adversary σ is then used to generate a point $\boldsymbol{q}$ which is guaranteed to be contained on the Pareto curve for $\boldsymbol{\psi}$, and a collection X of such points is assembled. Each new weight vector $\boldsymbol{w}$ is identified by finding a separating hyperplane between $down(X)$ and $\boldsymbol{r} = (r_1, \dots, r_n)$. Once $\boldsymbol{r}$ is found to be contained in $down(X)$, we

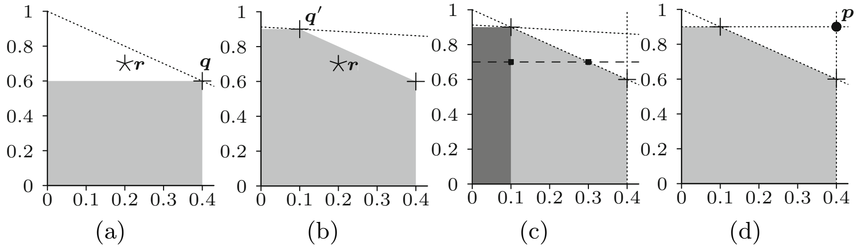


Fig. 2. Example executions of Algorithms 1, 3 and 4 (see Examples 2, 3 and 4).

know that ψ is achievable. If, on the other hand, $\boldsymbol{w} \cdot \boldsymbol{q} < \boldsymbol{w} \cdot \boldsymbol{r}$, then we know that ψ cannot possibly be achievable.

Correctness of Alg. 1 is proved in [12]. Termination is guaranteed by the fact that each iteration of the loop identifies a point $\boldsymbol{q}$ on a unique face of the Pareto curve. In the worst case, the number of faces is exponential in $|\mathcal{M}|$, k and n [9]; however, our experimental results (in Section 5) show the number of steps is usually small on practical examples. The individual model checking problems solved in each step (lines 4-5 of Alg. 1) require time polynomial in $|\mathcal{M}|$. We describe their practical implementation in the next section.

Example 2. We illustrate the execution of Alg. 1 on the MDP from Example 1 and the achievability query $([\{s_1\}]_{\geqslant 0.2}, [\{s_3\}]_{\geqslant 0.7})$. Let us assume that we have already applied Proposition 2 so that we have an equivalent reward predicate $([\rho_1]_{\geqslant 0.2}, [\rho_2]_{\geqslant 0.7})$ (the full reduction is in [12]). As a first (arbitrary) weight vector, we pick $\boldsymbol{w}$=(0.5, 0.5) and then maximise $\boldsymbol{w} \cdot \boldsymbol{\rho}$. The resulting optimal adversary σ (which chooses a in s_0) gives $\boldsymbol{q} = (\mathit{ExpTot}^{\sigma}_{\mathcal{M}}(\rho_1), \mathit{ExpTot}^{\sigma}_{\mathcal{M}}(\rho_2))$ $= (0.4, 0.6)$ and we have $X = \{\boldsymbol{q}\}$. Fig. 2(a) shows the point $\boldsymbol{q}$ (as +) and the target point $\boldsymbol{r} = (0.2, 0.7)$ (as ⋆). The dotted line represents the hyperplane with orientation $\boldsymbol{w}$ passing through $\boldsymbol{q}$ (i.e. the points $\boldsymbol{x}$ for which $\boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w} \cdot \boldsymbol{q} = 0.5$), the points above which correspond to unachievable value pairs. The grey region is $\mathit{down}(X)$, in which all points are achievable. Since $\boldsymbol{r} \notin \mathit{down}(X)$, we continue.

Next, we pick a weight $\boldsymbol{w}' = (0.1, 0.8)$. Maximising $\boldsymbol{w}' \cdot \boldsymbol{\rho}$ results in adversary σ' (which chooses b in s_0) giving $\boldsymbol{q}' = (0.1, 0.9)$, which we add to X. Fig. 2(b) again shows both points in X, $\mathit{down}(X)$ and $\boldsymbol{r}$. It also plots points $\boldsymbol{x}$ for which $\boldsymbol{w}' \cdot \boldsymbol{x} = \boldsymbol{w}' \cdot \boldsymbol{q}' = 0.73$. Since $\boldsymbol{r}$ is now in $\mathit{down}(X)$, the algorithm returns true.

Value iteration. The most expensive part of Alg. 1 in practice is the combination of lines 4-5, which computes the maximum possible value for a weighted sum of reward objectives, determines a corresponding optimal adversary σ, and then finds the value for the n individual objectives under σ.

Alg. 2 shows how to perform all these tasks using a value iteration-style computation. One key difference between this algorithm and standard value iteration is that it needs to optimise a combination of unbounded and bounded properties. This is done in three phases (lines 3-8, 9-13 and 14-20). The first two correspond to the unbounded part; the third to the bounded part.

```
    Input: MDP $\mathcal{M}{=}(S, \bar{s}, \alpha, \delta)$, weight vect. $\boldsymbol{w}$, reward structures $\boldsymbol{\rho}{=}(\rho_1, \ldots, \rho_n)$,
           vector of time bounds $\boldsymbol{k} \in (\mathbb{N} \cup \{\infty\})^n$, convergence threshold $\varepsilon$
    Output: Adv. $\sigma$ maximising $ExpTot^{\sigma,\boldsymbol{k}}_{\mathcal{M}}(\boldsymbol{w}{\cdot}\boldsymbol{\rho})$, $\boldsymbol{q} = (ExpTot^{\sigma,k_i}_{\mathcal{M}}(\rho_i))_{1\leqslant i\leqslant n}$
 1  $\boldsymbol{x} := \boldsymbol{0}$; $\boldsymbol{x}^1 := \boldsymbol{0}$; $\ldots$; $\boldsymbol{x}^n := \boldsymbol{0}$; $\boldsymbol{y} = \boldsymbol{0}$; $\boldsymbol{y}^1 := \boldsymbol{0}$; $\ldots$; $\boldsymbol{y}^n := \boldsymbol{0}$;
 2  $\sigma^\infty(s) = \bot$ for all $s \in S$;
 3  do
 4      foreach $s \in S$ do
 5          $y_s := \max_{a\in A(s)}(\sum_{\{i|k_i=\infty\}} w_i{\cdot}\rho_i(s,a) + \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'})$;
 6          $\sigma^\infty(s) := \arg\max_{a\in A(s)}(\sum_{\{i|k_i=\infty\}} w_i{\cdot}\rho_i(s,a) + \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'})$;
 7      $\delta := \max_{s\in S}(y_s - x_s)$; $\boldsymbol{x} := \boldsymbol{y}$;
 8  while $\delta > \varepsilon$;
 9  do
10      foreach $s \in S$ and $i \in \{1,\ldots,n\}$ where $k_i = \infty$ do
11          $y^i_s := \rho_i(s, \sigma^\infty(s)) + \sum_{s'\in S} \delta(s, \sigma^\infty(s))(s') \cdot x^i_{s'}$;
12      $\delta := \max^n_{i=1} \max_{s\in S}(y^i_s - x^i_s)$; $\boldsymbol{x}^1 := \boldsymbol{y}^1$; $\ldots$; $\boldsymbol{x}^n := \boldsymbol{y}^n$;
13  while $\delta > \varepsilon$;
14  for $j = \max\{k_\ell < \infty \mid \ell \in \{1,\ldots,n\}\}$ down to 1 do
15      foreach $s \in S$ do
16          $y_s := \max_{a\in A(s)}(\sum_{\{i|k_i\geqslant j\}} w_i{\cdot}\rho_i(s,a) + \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'})$;
17          $\sigma^j(s) := \arg\max_{a\in A(s)}(\sum_{\{i|k_i\geqslant j\}} w_i{\cdot}\rho_i(s,a) + \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'})$;
18          foreach $i \in \{1,\ldots,n\}$ where $k_i \geqslant j$ do
19              $y^i_s := \rho_i(s, \sigma^j(s)) + \sum_{s'\in S} \delta(s, \sigma^j(s))(s') \cdot x^i_{s'}$;
20      $\boldsymbol{x} := \boldsymbol{y}$; $\boldsymbol{x}^1 := \boldsymbol{y}^1$; $\ldots$; $\boldsymbol{x}^n := \boldsymbol{y}^n$;
21  foreach $i \in \{1,\ldots,n\}$ do $q_i := y^i_{\bar{s}}$;
22  $\sigma$ behaves as $\sigma^j$ in $j$-th step when $j < \max_{i\in\{1,\ldots,n\}} k_i$, and as $\sigma^\infty$ afterwards;
23  return $\sigma, \boldsymbol{q}$
```

Alg. 2. Value iteration-based algorithm for lines 4-5 of Alg. 1

Another important difference is that the algorithm performs the optimisation of the weighted sum $\boldsymbol{w}{\cdot}\boldsymbol{\rho}[\boldsymbol{k}]$ and the computation of the vector of individual objective values $\boldsymbol{q} = (ExpTot^{\sigma,k_i}_{\mathcal{M}}(\rho_i))_{1\leqslant i\leqslant n}$ simultaneously: the former in phases 1 and 3; the latter in phases 2 and 3. Consider first the optimisation of $\boldsymbol{w}{\cdot}\boldsymbol{\rho}[\boldsymbol{k}]$. The values, for all states $s \in S$, are computed as a sequence of increasingly precise approximations, stored in a pair of vectors, $\boldsymbol{x}$ and $\boldsymbol{y}$. Each new approximation is stored in $\boldsymbol{y}$ (line 5); then, $\boldsymbol{x}$ and $\boldsymbol{y}$ are compared for convergence and $\boldsymbol{x}$ is set to $\boldsymbol{y}$ (line 7) before proceeding to the next iteration. Computation of the bounded part of $\boldsymbol{w}{\cdot}\boldsymbol{\rho}[\boldsymbol{k}]$ continues in phase 3 in similar fashion (although no convergence check is needed). During optimisation of $\boldsymbol{w}{\cdot}\boldsymbol{\rho}[\boldsymbol{k}]$, a corresponding optimal adversary is also determined, with the unbounded and bounded fragments stored in σ^∞ and σ^j, respectively. The choices made by this adversary are used to compute the value q_i for each of the n individual objectives, the values for which are also stored in pairs of vectors ($\boldsymbol{x}^i, \boldsymbol{y}^i$ for each q_i).

In practice, storing multiple $|S|$-sized vectors ($\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x}^i$, and $\boldsymbol{y}^i$) is relatively expensive. We discuss later how the algorithm's memory usage can be improved. We include an example of the execution of Alg. 2 in [12].

```
   **Input**: MDP $\mathcal{M}$, objective $[\rho_1]^{\leqslant k_1}_{\max}$, predicate $([\rho_2]^{\leqslant k_2}_{\geqslant r_2}, \dots, [\rho_n]^{\leqslant k_n}_{\geqslant r_n})$
   **Output**: Value of $num([\rho_1]^{\leqslant k_1}_{\max}, ([\rho_2]^{\leqslant k_2}_{\geqslant r_2}, \dots, [\rho_n]^{\leqslant k_n}_{\geqslant r_n}))$
1  $X{:=}\emptyset$; $\boldsymbol{\rho}{:=}(\rho_1, \dots \rho_n)$; $\boldsymbol{k}{:=}(k_1, \dots k_n)$; $\boldsymbol{r}{:=}(\min_{\sigma \in Adv_M} ExpTot^{\sigma,k_1}_{\mathcal{M}}(\rho_1), r_2, \dots r_n)$;
2  **do**
3      Find $\boldsymbol{w}$ separating $\boldsymbol{r}$ from $down(X)$ such that $w_1 > 0$;
4      Find adversary $\sigma$ maximising $ExpTot^{\sigma,\boldsymbol{k}}_{\mathcal{M}}(\boldsymbol{w} \cdot \boldsymbol{\rho})$;
5      $\boldsymbol{q} := (ExpTot^{\sigma,k_i}_{\mathcal{M}}(\rho_i))_{1 \leqslant i \leqslant n}$;
6      **if** $\boldsymbol{w} \cdot \boldsymbol{q} < \boldsymbol{w} \cdot \boldsymbol{r}$ **then return** $\bot$;
7      $X := X \cup \{\boldsymbol{q}\}$;   $r_1 := \max\{r_1, \max\{r' \mid (r', r_2, \dots, r_n) \in down(X)\}\}$;
8  **while** $\boldsymbol{r} \notin down(X)$ *or* $\boldsymbol{w} \cdot \boldsymbol{q} > \boldsymbol{w} \cdot \boldsymbol{r}$;
9  **return** $r_1$;
```

Alg. 3. Algorithm for checking numerical queries

Numerical queries. We now turn our attention to numerical queries. Alg. 3 shows how Alg. 1 can be adapted to check these. Like Alg. 1, it generates points $\boldsymbol{q}$ on the Pareto curve from a sequence of weight functions $\boldsymbol{w}$. For the objective ρ_1 that is being optimised, we generate a sequence of lower bounds r_1 that are used in the same fashion as Alg. 1. Initially, we take r_1 to be the minimum possible value for ρ_1, which can be computed with a separate instance of value iteration. New (non-decreasing) values for r_1 are generated at each step based on the set of points X determined so far. The numerical computation for each step (lines 4-5 of Alg. 3) can again be carried out with Alg. 2. Correctness of Alg. 3 is proved in [12]. The bound on the number of steps needed is as for Alg. 1.

Example 3. We demonstrate Alg. 3 on the MDP from Example 1 and numerical query $([\{s_1\}]_{\max}, ([\{s_3\}]_{\geqslant 0.7}))$. Initially, $r_1{=}0.1$ and, with $\boldsymbol{w}{=}(0.1, 0.8)$, we get $\boldsymbol{q}{=}(0.1, 0.9)$. The resulting area $down(X)$ is shown as dark grey in Fig. 2(c). Next, r_1 remains as 0.1 and, with $\boldsymbol{w}{=}(1, 0)$, we get $\boldsymbol{q}{=}(0.4, 0.6)$. Adding this to X, $down(X)$ is enlarged by the light grey area. Finally, r_1 is set to 0.3, choosing $\boldsymbol{w}{=}(0.5, 0.5)$ yields $\boldsymbol{q}{=}(0.4, 0.6)$ again, and the loop ends. Fig. 2(c) also shows the points $\boldsymbol{q}$ and $\boldsymbol{r}$ (as + and ▪). The final value returned is $r_1{=}0.3$.

Pareto curves. Next, we discuss Pareto queries. Generating and visualising Pareto curves (or their approximations) provides a much clearer view of the trade-offs between objectives. Our algorithm is implemented as a simple modification of our previous algorithms, and is presented as Alg. 4. For simplicity, we focus on the 2-objective case, which is most practical for visualisation. Our implementation, described later, also supports the 3-objective case and, in theory, this can be extended to an arbitrary number of objectives.

Alg. 4, like the earlier ones, builds a set X of points on a Pareto curve P using weights $\boldsymbol{w}$. Since P is convex, the surface of points X represents a *lower* approximation of P. Our algorithm also constructs an *upper* approximation Y using the generated weights $\boldsymbol{w}$. As illustrated in Example 2, for each point $\boldsymbol{q} \in X$, there is a corresponding hyperplane passing through $\boldsymbol{q}$ and with orientation $\boldsymbol{w}$, above which no values are achievable. Hence these represent upper bounds on P and we store, in Y, any weight $\boldsymbol{w}$ that resulted in each point $\boldsymbol{q} \in X$.

```
Input: MDP $\mathcal{M}$, reward structures $\boldsymbol{\rho} = (\rho_1, \rho_2)$, time bounds $(k_1, k_2)$, $\varepsilon_p \in \mathbb{R}_{>0}$
Output: An $\varepsilon_p$-approximation of a Pareto curve
$X := \emptyset$; $Y : \mathbb{R}^2 \to 2^{\mathbb{R}^2}$, initially $Y(x) = \emptyset$ for all $x$; $\boldsymbol{w} = (1, 0)$;
Find adversary $\sigma$ maximising $ExpTot^{\sigma,\boldsymbol{k}}_{\mathcal{M}}(\boldsymbol{w}\cdot\boldsymbol{\rho})$;
$\boldsymbol{q} := (ExpTot^{\sigma,k_1}_{\mathcal{M}}(\rho_1), ExpTot^{\sigma,k_2}_{\mathcal{M}}(\rho_2))$;
$X := X \cup \{\boldsymbol{q}\}$; $Y(\boldsymbol{q}) := Y(\boldsymbol{q}) \cup \{\boldsymbol{w}\}$; $\boldsymbol{w} = (0, 1)$;
do
    Find adversary $\sigma$ maximising $ExpTot^{\sigma,\boldsymbol{k}}_{\mathcal{M}}(\boldsymbol{w}\cdot\boldsymbol{\rho})$;
    $\boldsymbol{q} := (ExpTot^{\sigma,k_1}_{\mathcal{M}}(\rho_1), ExpTot^{\sigma,k_2}_{\mathcal{M}}(\rho_2))$;
    $X := X \cup \{\boldsymbol{q}\}$; $Y(\boldsymbol{q}) := Y(\boldsymbol{q}) \cup \{\boldsymbol{w}\}$; $\boldsymbol{w} = \bot$;
    Order $X$ to a sequence $\boldsymbol{x}^1, \dots, \boldsymbol{x}^m$ such that $\forall\ i$: $x^i_1 \leqslant x^{i+1}_1$ and $x^i_2 \geqslant x^{i+1}_2$;
    foreach $i \in \{1, \dots m-1\}$ do
        Let $\boldsymbol{u}$ be the element of $Y(\boldsymbol{x}^i)$ with maximal $u_1$;
        Let $\boldsymbol{u'}$ be the element of $Y(\boldsymbol{x}^{i+1})$ with minimal $u'_1$.;
        Find a point $\boldsymbol{p}$ such that $\boldsymbol{u}\cdot\boldsymbol{p} = \boldsymbol{u}\cdot\boldsymbol{x}^i$ and $\boldsymbol{u'}\cdot\boldsymbol{p} = \boldsymbol{u'}\cdot\boldsymbol{x}^{i+1}$;
        if distance of $\boldsymbol{p}$ from $down(X)$ is $\geqslant \varepsilon_p$ then
            Find $\boldsymbol{w}$ separating $down(X)$ from $\boldsymbol{p}$, maximising $\boldsymbol{w}\cdot\boldsymbol{p} - \max_{\boldsymbol{x}\in down(X)} \boldsymbol{w}\cdot\boldsymbol{x}$;
while $\boldsymbol{w} \neq \bot$;
return $X$
```

Alg. 4. Algorithm for Pareto curve approximation for 2 objectives

The sequence of weights $\boldsymbol{w}$ is generated as follows. We construct an initial curve using weights $(1, 0)$ and $(0, 1)$. Then, we repeatedly: (i) sort the points in X; (ii) for each successive pair $\boldsymbol{x}^i, \boldsymbol{x}^{i+1}$ in X, find the lowest point $\boldsymbol{p}$ on the intersection of the hyperplanes stored in Y for $\boldsymbol{x}^i$ and $\boldsymbol{x}^{i+1}$; (iii) choose $\boldsymbol{w}$ as a separating hyperplane between $down(X)$ and $\boldsymbol{p}$. The algorithm continues until the maximum distance between the two approximations falls below some threshold ε_p. In principal, the algorithm can enumerate all faces of P. The reason for constructing an ε_p-approximation is two-fold: firstly, the number of faces is potentially large, whereas an approximation may suffice; secondly, computation of individual points (using value iteration), is already approximate.

Example 4. We illustrate Alg. 4 on the MDP from Example 1 with objectives $([\{s_1\}]_{\max}, [\{s_3\}]_{\max})$. The first two weight vectors $\boldsymbol{w}$ are $(1, 0)$ and $(0, 1)$, yielding points $\boldsymbol{q}$ of $(0.4, 0.6)$ and $(0.1, 0.9)$, repectively (see Fig. 2(d)). The hyperplanes attached to each point are also shown, by dotted lines, as is their intersection $\boldsymbol{p} = (0.4, 0.9)$. We choose separating hyperplane $\boldsymbol{w} = (0.5, 0.5)$, indicated by the sloped dotted line. The algorithm then finds the intersection $(0.1, 0.9)$ of this with the horizontal line and, since this point is already in $down(X)$, terminates.

Adversary generation. Finally, we describe how to generate *optimal adversaries* for our multi-objective queries. We explain this for achievability queries, but it can easily be adapted to the other types too. Unlike standard (single-objective) MDP model checking, where deterministic adversaries always suffice to optimise reachability/reward objectives, multi-objective optimisation requires randomised adversaries. Alg. 1, when finding that bounds $\boldsymbol{r}$ are achievable, generates points $\boldsymbol{q}^1, \ldots, \boldsymbol{q}^m$ on the Pareto curve. Each corresponding call to Alg. 2 returns a (deterministic) adversary, say $\sigma_{\boldsymbol{q}^j}$ for the current point $\boldsymbol{q}^j$. The final adversary σ_{opt} is constructed from these and a weight vector $\boldsymbol{u} \in \mathbb{R}^m$ satisfying $r_i \leqslant \sum_{j=1}^{m} u_i \cdot q_i^j$ for all $1 \leqslant i \leqslant n$: it simply makes an initial one-off random choice of adversary $\sigma_{\boldsymbol{q}^j}$ to mimic (each with probability u_j).

5 Implementation and Results

We implemented our multi-objective model checking techniques in PRISM [14], also adding the automaton construction of [11] to support ω-regular properties. Value iteration is built on top of PRISM's "sparse" engine. It would also be straightforward to adapt its symbolic (MTBDD) engine, which can improve scalability on models exhibiting regularity; but, for the current set of experiments, the sparse engine suffices to illustrate the benefits offered by our approach.

Heuristics and optimisations. Our core algorithms are based on generating weight vectors $\boldsymbol{w}$ representing separating hyperplanes (e.g. at line 3 of Alg. 1). The choice of each $\boldsymbol{w}$ is not unique and affects the number of steps needed by the algorithm. Based on our results, the following is an effective heuristic. For the first n vectors (assuming n objectives), choose $\boldsymbol{w}$ with $w_i = 1$ for some i. Next, given point $\boldsymbol{r}$ and set of points X, choose $\boldsymbol{w}$ to maximise $\min_{\boldsymbol{x} \in X}(\boldsymbol{w} \cdot \boldsymbol{q} - \boldsymbol{w} \cdot \boldsymbol{x})$, i.e., pick the hyperplane with maximal Euclidean distance d from $\boldsymbol{q}$. This is done by solving the LP: "maximise d subject to $\sum_{i=1}^{n} w_i = 1$ and $w_i \cdot (q_i - x_i) \geqslant d$ for all $\boldsymbol{x} \in X$". In practice, these problems are small and fast to solve.

We also apply various optimisations to the basic value iteration algorithms of Section 4. For unbounded properties, Gauss-Seidel value iteration [19] can be used to increase performance. Furthermore, we can significantly reduce the number of vectors stored with slight changes to Alg. 2; details are in [12].

Experimental results. We evaluated our techniques on benchmarks from several sources.[2] First, we used multi-objective problems resulting from the assume-guarantee framework of [15]. Second, we verified multi-objective properties on existing PRISM models: (i) a task-graph *scheduler* problem, minimising expected job completion time and expected energy consumption; (ii) a *team-formation* protocol, maximising the probability of completing two (separate) tasks and the expected size of the team that does so; (iii) a dynamic power management (*dpm*) controller, minimising over k steps both expected energy consumption and expected average queue size. Experiments were run on a 2.66GHz PC with 8GB of RAM. We used $\varepsilon = 10^{-6}$ for value iteration (this is the default in PRISM; smaller values led to very similar results) and $\varepsilon_p = 10^{-4}$ for Pareto curve generation.

The results are shown in Table 1: assume-guarantee problems at the top; the others below. For each model, we give the size (number of states), and details of the objectives in the query used. The middle part of the table compares the performance of our value iteration-based technique with the LP-based implementation of [11] on numerical queries. In our experiments, performance for

[2] All models/properties used are at www.prismmodelchecker.org/files/atva12mo/

Table 1. Experimental results for our implementation and a comparison with [11].

Case study [parameters]		Num. states	Objectives		Numerical query				Pareto query	
			Num.	Types	LP ([11])		Val. iter.		Val. iter.	
					LP size	Time (s)	Pt.s	Time (s)	Pt.s	Time (s)
consensus (2 proc.s) [R K]	3 2	691	2	$[T_1]_{\max}$ $[T_2]_{\max}$	1026	0.57	3	**0.02**	3	0.04
	4 2	1517			2288	0.67	3	**0.03**	3	0.05
	5 2	3169			4812	0.94	3	**0.05**	3	0.06
consensus (3 proc.s) [R K]	3 2	17455	2	$[T_1]_{\max}$ $[T_2]_{\max}$	40386	9.85	3	**0.22**	3	0.27
	4 2	61017			140676	144.06	3	**0.87**	3	1.06
	5 2	181129			*mem-out*		3	**2.83**	3	3.44
zeroconf [K]	4	5449	2	$[T_1]_{\max}$ $[T_2]_{\max}$	12916	1.25	2	**0.13**	4	0.60
	6	10543			24639	7.07	4	**0.46**	4	0.79
	8	17221			40833	19.6	4	**0.76**	4	1.13
zeroconf-tb [K T]	2 14	29572	2	$[T_1]_{\max}$ $[T_2]_{\max}$	61816	5.25	3	**1.69**	2	0.85
	4 10	19670			46659	5.01	2	**0.32**	3	0.84
	4 14	42968			103964	11.01	2	**0.63**	3	1.77
team-form. [N]	3	12475	2	$[T_1]_{\max}$ $[\rho_2]_{\max}$	14935	1.37	4	**0.21**	7	0.24
	4	96665			115289	11.57	4	**1.08**	7	1.72
	5	907993			*mem-out*		2	**5.66**	6	12.66
team-form. [N]	3	12475	3	$[T_1]_{\max}$ $[T_2]_{\max}$ $[\rho_2]_{\max}$	14935	1.37	3	**0.18**	57	1.39
	4	96665			115289	10.55	5	**1.77**	61	14.55
	5	907993			*mem-out*		2	**9.49**	57	141.76
scheduler [K]	5	31965	2	$[\rho_1]_{\min}$ $[\rho_2]_{\min}$	57954	59.15	8	**6.10**	10	8.08
	25	633735			*mem-out*		8	**526.56**	11	776.44
	50	2457510			*mem-out*		8	**3938.94**	10	5361.86
dpm [k]	100	636	2	$[\rho_1]_{\min}^{\leqslant k}$ $[\rho_2]_{\min}^{\leqslant k}$	n/a	n/a	3	**4.50**	6	0.12
	200	636			n/a	n/a	3	**4.30**	11	0.32
	300	636			n/a	n/a	3	**4.59**	9	0.36

achievability queries was very similar, so we omit them. The right part of the table shows times to compute Pareto curves for the same objectives on each model (which cannot be done with the implementation of [11]). For value iteration-based algorithms, we show the number of points (steps of the algorithm) needed; for LP-based, we show the size of the linear program solved.

Comparing the value-iteration and LP-based approaches, we see huge gains in run-time for our methods (up to approx. 150 times faster). There are also significant improvements in scalability: the biggest models solved with value iteration are about 20 times bigger than those for LP. One factor in the low run-times for our technique is that the algorithms generally require a fairly small number of steps, even when generating the Pareto curve.

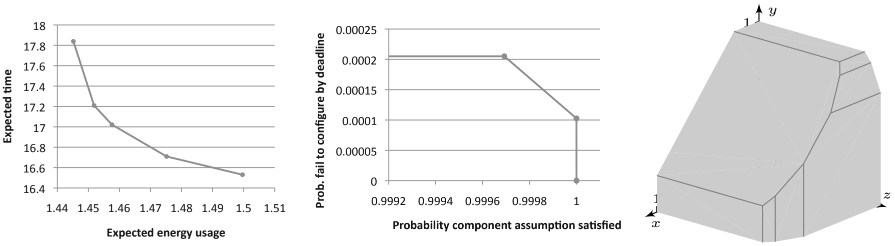


Fig. 3. Pareto curves from: (a) task-graph scheduler, K=2; (b) Zeroconf protocol, K=2, T=10; (c) team formation protocol, N=3 (axes x/y/z = Probability of completing task 1/probability of completing task 2/expected size of successful team)

Pareto curves. Finally, we show in Fig. 3 the Pareto curves generated for some of our examples. Plot (a) shows, for a task-graph scheduling problem, how different schedulers vary in terms of completion time and energy usage. Plot (b) is from an instance of assume-guarantee verification; the plot shows how it is possible to bound the probability of an error in the overall system (y-axis) for various different reliability levels of one of the components (x-axis). Plot (c) shows a 3-objective Pareto curve evaluating strategies in a team-formation protocol (see Fig. 3 caption for objectives). In each case, the plots give a clear, visual illustration of the trade-off between competing objectives. The curves could also be used to quickly answer any additional achievability or numerical queries for those objectives, without running any further model checking.

6 Conclusions

We have presented novel techniques for multi-objective model checking of MDPs, using a value iteration-based computation to build successive approximations of the Pareto curve. Compared to existing approaches, this gives significant gains in efficiency and scalability, and enables verification of time-bounded properties. Furthermore, we showed the benefits of visualising the Pareto curve for several probabilistic model checking case studies. Future directions include extending our techniques to timed probabilistic models such as CTMDPs and PTAs.

Acknowledgments. The authors are part supported by ERC Advanced Grant VERIWARE and EPSRC grant EP/F001096/1. Vojtěch Forejt is also supported by a Royal Society Newton Fellowship.

References

1. Altman, E.: Constrained Markov Decision Processes. Chapman & Hall/CRC (1999)
2. Bleuler, S., Laumanns, M., Thiele, L., Zitzler, E.: PISA – A Platform and Programming Language Independent Interface for Search Algorithms. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 494–508. Springer, Heidelberg (2003)
3. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge Univ. Press (2004)
4. Brázdil, T., Brožek, V., Chatterjee, K., Forejt, V., Kučera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. In: LICS 2011 (2011)
5. Chatterjee, K., Majumdar, R., Henzinger, T.: Markov Decision Processes with Multiple Objectives. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 325–336. Springer, Heidelberg (2006)
6. Clímaco, J. (ed.): Multicriteria Analysis. Springer (1997)
7. Coello, C., Lamont, G., Veldhuizen, D.v.: Evolutionary Algorithms for Solving Multi-Objective Problems. Springer (2007)
8. Diakonikolas, I., Yannakakis, M.: Succinct approximate convex Pareto curves. In: Proc. SODA 2008, pp. 74–83. SIAM (2008)
9. Etessami, K., Kwiatkowska, M., Vardi, M., Yannakakis, M.: Multi-objective model checking of Markov decision processes. LMCS 4(4), 1–21 (2008)

10. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated Verification Techniques for Probabilistic Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
11. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative Multi-objective Verification for Probabilistic Systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011)
12. Forejt, V., Kwiatkowska, M., Parker, D.: http://arxiv.org/abs/1206.6295
13. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer (1976)
14. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
15. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-Guarantee Verification for Probabilistic Systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
16. Legriel, J., Cotton, S., Maler, O.: On universal search strategies for multi-criteria optimization using weighted sums. In: Proc. CEC 2011, pp. 2351–2358 (2011)
17. Legriel, J., Le Guernic, C., Cotton, S., Maler, O.: Approximating the Pareto Front of Multi-criteria Optimization Problems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 69–83. Springer, Heidelberg (2010)
18. Papadimitriou, C., Yannakakis, M.: On the approximability of trade-offs and optimal access of web sources. In: Proc. FOCS 2000, pp. 86–92 (2000)
19. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons (1994)

Verification of Partial-Information Probabilistic Systems Using Counterexample-Guided Refinements

Sergio Giro[1,*] and Markus N. Rabe[2]

[1] Department of Computer Science, University of Oxford
[2] Department of Computer Science, Saarland University

Abstract. The verification of partial-information probabilistic systems has been shown to be undecidable in general. In this paper, we present a technique based on inspection of counterexamples that can be helpful to analyse such systems in particular cases. The starting point is the observation that the system under complete information provides safe bounds for the extremal probabilities of the system under partial information. Using classical (total information) model checkers, we can determine optimal schedulers that represent safe bounds but which may be spurious, in the sense that they use more information than is available under the partial information assumptions. The main contribution of this paper is a refinement technique that, given such a scheduler, transforms the model to exclude the scheduler and with it a whole class of schedulers that use the same unavailable information when making a decision. With this technique, we can use classical total information probabilistic model checkers to analyse a probabilistic partial information model with increasing precision. We show that, for the case of infimum reachability probabilities, the total information probabilities in the refined systems converge to the partial information probabilities in the original model.

1 Introduction

Verification algorithms for formalisms like Markov Decision Processes and their variants have been studied extensively in the last 20 years, given their wide range of applications. In these systems, there are two kinds of choices: non-deterministic and probabilistic (with probability values specified in the model). Non-deterministic choices are resolved using the so-called schedulers: by restricting a system to the choices of the scheduler, the restricted system collapses to a Markov chain, and probability values for properties can be calculated. Worst-case probability values are then defined by considering the maximum/minimum

[*] This work was supported by DARPA and the Air Force Research Laboratory under contract FA8650-10-C-7077 (PRISMATIC). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFRL or the U.S. Government.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 333–348, 2012.

probability over all schedulers. In consequence, a counterexample is a scheduler that yields greater/less probability than allowed by the specification.

Background. In recent years, considerable attention has been paid to probabilistic systems in which the non-deterministic choices are resolved according to partial information (see [2,3,5,9] and references therein). The formalisms in these works can be seen as generalized versions of Decentralized Partial Observation MDPs. Using these formalisms we can model, for instance, a game in which players keep some information hidden.

The quantitative model checking problem was shown to be undecidable in general for partial information MDPs [10]. Some techniques are available for finite-horizon properties or to obtain over-approximations [11]. To the best of our knowledge, the technique for quantitative analysis in this paper is the first one in which the amount of information available is gradually refined. As a work in a similar direction, the first abstraction refinement technique for *non-probabilistic* partial-information games was proposed quite recently [6]. A recent work concerning *qualitative* properties in a setting similar to ours is [2].

Contribution. We present an iterative technique that allows to improve the accuracy of the values obtained using total information analysis on partial information systems. In order to do this, we present an algorithm to check whether a total information scheduler complies with the partial information assumptions. We also present a transformation (called *refinement*) that, given a scheduler that does not comply with the assumptions, modifies the model to exclude this scheduler and a whole class of schedulers that use the same unavailable information when making a decision. This transformation can be carried out using different criteria. For the case of infimum reachability probabilities, we show a criterion under which, by successively applying the refinements, the total information probabilities in the refined systems converge to the partial information probabilities in the original model.

Introductory Example. We illustrate the problem we address, and the usefulness of our technique, using the players A and B in Fig. 1 (the automaton $A \parallel B$ will be used later). To simplify, we assume that they play a turn-based game (the systems we consider in the rest of the paper allow for non-deterministic arbitrary interleaving). When the game starts, player A tosses a coin whose sides are labelled with 0 and 1. Then, B tosses a similar coin keeping the outcome hidden. In the next turn, A tries to guess if both outcomes agree: in the state $a0$, the guess of A is that an agreement happened, and his outcome has been 0 (the meaning of the other states is similar). After the guess, a synchronized transition (depicted with a dashed line) takes both A and B to the initial state, where another round starts. Player B wins if A fails to guess at least once. The problem under consideration is to calculate the minimum probability that B wins. Intuitively we can think that player A wants to prevent the system from reaching one of the states in which B wins. An example strategy for the first round for player A would be "if A's outcome is 0, then A guesses an agreement. Otherwise, it guesses a disagreement". In this scheduler/strategy, B wins iff its outcome is 1. Hence, for this scheduler/strategy, the probability that B wins is the probability that B's outcome is 1, that is, 1/2. It is easy to see that, for every scheduler, B wins in the first round with probability 1/2. In subsequent rounds, player A might try different schedulers, but since all of them lead B to win the round with probability 1/2, the probability that B has won after round N is $1 - (1/2)^N$. Hence, the probability that B wins the game in some round is 1. The minimum probability that B wins, quantifying over all schedulers, is then 1.

This result is not, however, the one yielded by standard tools for probabilistic model checking such as Prism [12] or LiQuor [4]. Such tools verify this model by constructing the parallel composition $A \parallel B$ shown in Fig. 1 (double-framed boxes indicate the states in which B wins) and considering all schedulers for the composed model under total information. The problem with this approach (sometimes called the *compose-and-schedule* approach) is that there exist some unrealistic schedulers as the one shown in Fig. 2: in this scheduler, the probability of reaching a state in which B wins is 0. The scheduler simply guesses an agreement in case an agreement happened, and a disagreement otherwise. This is unrealistic, as in the original model A is unable to see the outcome of B.

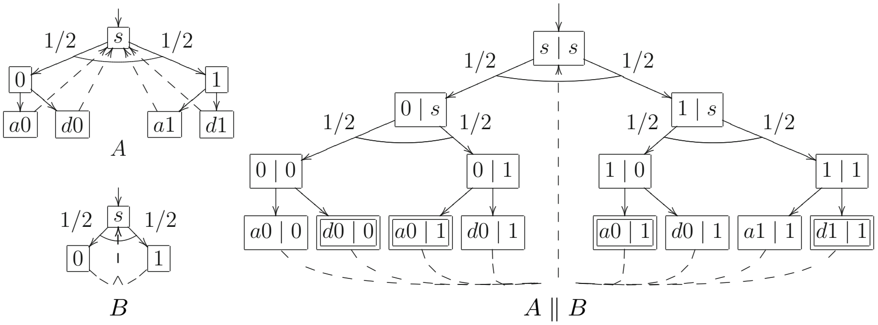


Fig. 1. Player A tries to guess if the choices agree

The problem illustrated by the example led to the definition of partial-information schedulers or distributed schedulers (in which a scheduler of the compound system is obtained by composing schedulers for each player). However, the verification of properties under partial-information schedulers was

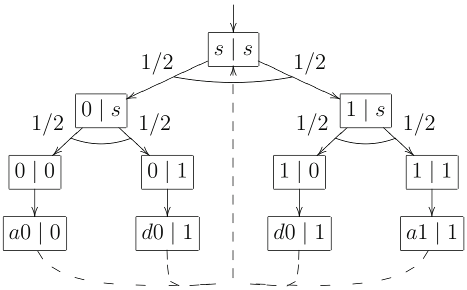
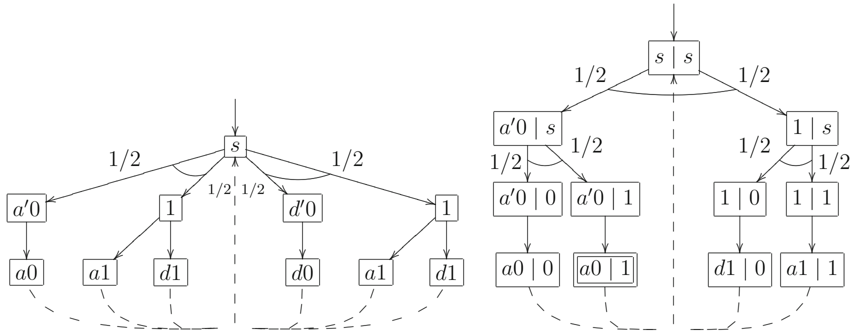


Fig. 2. An unrealistic scheduler for the parallel composition

proven to be difficult, and several hardness and undecidability results are known (see [10,2], just to name a few).

The result considering all schedulers can be seen as a safe (although overly pessimistic) bound on the minimum/maximum probability. In this paper, we present a technique to obtain tighter safe bounds. This technique works through a series of *refinements*: it starts by verifying the system as if total information were available, using standard algorithms for the total-information case. If the system is deemed correct, then it is also correct under partial information, as the set of schedulers under partial information is a subset of the ones under total information. If the system is deemed as incorrect, it can be checked whether the counterexample obtained is valid under partial information: that is, if all choices are resolved using only available information. If the scheduler is indeed valid, then we can conclude that the system under consideration is incorrect, and we can use the counterexample obtained as witness. For the case in which the counterexample is not valid under partial information (that is, the case in which there is a decision that is resolved according to information not available) we present a transformation that produces a system in which the spurious counterexample is less likely to occur in a new analysis under total information. We can analyse the resulting system by repeating this refinement each time we get a spurious counterexample, in the hope that eventually we find the system correct or we get a real counterexample. We show that, for infimum reachability probabilities, the refinements can be carried out in such a way that the results converge to the actual value for all systems.



(a) Refined model of A

(b) A scheduler after the first refinement

Fig. 3.

We can explain our counterexample-based transformation of the system by following our previous example: we first detect that, in the counterexample in Fig. 2, the player A performs a choice using unavailable information while in state 0, by noticing that its choices differ for the state $(0, 0)$ and the state $(0, 1)$ (the player also cheats in state 1, but can tackle one state at a time). The transformation forces (the refined model of) A to choose beforehand what the move will be in state 0, this choice being resolved during the coin toss. If the state reached is 0, player A must adhere to its previous decision. The refined model of A is shown in Fig. 3(a). Roughly speaking, the non-determinism at state 0 has been "pulled backwards". If we apply the compose-and-schedule approach, the compound system still has some unrealistic counterexamples, as A can still cheat in state 1. One of such unrealistic counterexamples is shown in Fig. 3(b). However, the minimum probability that B wins is now 1 for all schedulers (as eventually A passes through state 0, in which A cannot cheat). Since our transformation ensures that 1 is a lower bound for the minimum probability, we know that the result is 1 and the verification finishes.

This verification, calculating the exact result after one refinement, can be contrasted with the naïve approach of computing the minimum probability p_N that B wins before round N, increasing N successively. These probabilities can be computed by considering each of the schedulers for A up to round N. The value of p_N is $1-(1/2)^N$, and so this approximation never reaches the actual value 1. In addition, as general schedulers depend on the local history of A, computing p_N involves computations for 2^{2^N} different schedulers.

2 The Model

In this section we introduce Markov Decision Processes, together with a notion of parallel composition suited to define partial-information schedulers.

Markov Decision Processes. Let $\text{Dist}(A)$ denote the set of discrete probability distributions over the set A. The support of A is denoted by $\text{supp}(A)$.

A Markov Decision Process (Mdp) $\mathcal{M}$ is a quadruple (S, Q, Σ, T) where S is a finite set of states, $Q \subseteq S$ is a set of initial states, Σ is the finite alphabet of the system, each element in Σ is called a label, and $T \subseteq S \times \Sigma \times \text{Dist}(S)$ is a transition structure: if $\mu = (s, \alpha, d) \in T$ then there is a transition μ with label α enabled in s, and the probability of reaching t from s using this transition is $d(t)$. When no confusion arises, we write $\mu(t)$ instead of $d(t)$. We write $\text{en}(s)$ for the set of transitions (t, α, d) with $t = s$. The label of μ is written $\text{label}(\mu)$. We assume that subindices and superindices map naturally from Mdps to their constituents and so, for instance, S_p is the set of states of $\mathcal{M}_p$.

A path in an Mdp is a (possibly infinite) sequence $\rho = s^0.\mu^1.s^1.\cdots..\mu^n.s^n$, where $\mu^i \in \text{en}(s^{i-1})$ and $\mu^i(s^i) > 0$ for all i. If ρ is finite, the last state of ρ is denoted by $\text{last}(\rho)$, and the length is denoted by $\text{len}(\rho)$ (a path having a single state has length 0). Given two paths ρ, σ such that $\text{last}(\rho)$ is equal to the first state of σ, we denote by $\rho\cdot\sigma$ the concatenation of the two paths. The set of finite paths of an $\mathcal{M}$ is denoted by $\text{Paths}_{\mathcal{M}}$. We write $s \xrightarrow{\mu} t$ to denote $\mu \in \text{en}(s) \wedge \mu(t) > 0$. Overloading the notation, we write $s \xrightarrow{\alpha} t$ iff $\exists \mu : \text{label}(\mu) = \alpha \wedge s \xrightarrow{\mu} t$. Given a set of target states U, the set $\text{reach}(U)$ comprises all infinite paths w such that at least one state in w is in U.

The standard semantics of Mdps is given by schedulers. A (total information) scheduler η for an Mdp is given by a state $\text{init}_\eta \in Q$ and a function $\eta\colon \text{Paths}_{\mathcal{M}} \to T$ such that, if $\text{en}(\text{last}(\rho)) \neq \emptyset$, then $\eta(\rho) \in \text{en}(\text{last}(\rho))$. In words, the scheduler chooses an enabled transition based on the previous path.

For simplicity, in case $\mathrm{en}(\mathrm{last}(\rho)) = \emptyset$ we define $\eta(\rho) = \varsigma_{\mathrm{last}(\rho)}$, where $\varsigma_{\mathrm{last}(\rho)}$ is a fictitious transition representing the fact that, after ρ, the system remains in $\mathrm{last}(\rho)$ forever. Accordingly, we define $\varsigma_s(s) = 1$ for all s. The set of all schedulers for $\mathcal{M}$ is denoted by $\mathrm{Sched}(\mathcal{M})$.

The set $\mathrm{Paths}(\eta)$ contains all the paths $s^0.\mu^1.s^1.\cdots.\mu^n.s^n$ such that $s^0 = \mathrm{init}_\eta$, $\eta(s^0.\mu^1.s^1.\cdots..\mu^{i-1}.s^{i-1}) = \mu^i$ and $s^{i-1}\xrightarrow{\mu^i}s^i$ for all i. We say that two schedulers η, ζ are *equivalent* (denoted $\eta \equiv \zeta$) iff $\mathrm{Paths}(\eta) = \mathrm{Paths}(\zeta)$ (note that this implies $\eta(\rho) = \zeta(\rho)$ for all $\rho \in \mathrm{Paths}(\eta)$).

The probability $\mathrm{Pr}^\eta(\rho)$ of the path ρ under η is $\prod_{i=1}^n \mu^i(s^i)$ if $\rho \in \mathrm{Paths}(\eta)$. If $\rho \notin \mathrm{Paths}(\eta)$, then the probability is 0.

We are interested on the probability of (sets of) infinite paths. Given a finite path ρ, the probability of the set $\rho^\uparrow$ comprising all the infinite paths that have ρ as a prefix is defined by $\mathrm{Pr}^\eta(\rho^\uparrow) = \mathrm{Pr}^\eta(\rho)$. In the usual way (that is, by resorting to the Carathéodory extension theorem) it can be shown that the definition on the sets of the form $\rho^\uparrow$ can be extended to the σ-algebra generated by the sets of the form $\rho^\uparrow$. Since the measure of any set in the σ-algebra is determined by the measure in the sets $\rho^\uparrow$, it follows that for all measurable sets $\mathcal{H}$: $\eta \equiv \zeta \implies \mathrm{Pr}^\eta(\mathcal{H}) = \mathrm{Pr}^\zeta(\mathcal{H})$.

Composition of variable-based MDPs**.** The systems we compose exchange information through the use of shared variables. In consequence, we assume that the state of an MDP is given by a valuation on a set of variables V. The variables in the set $W \subseteq V$ are the *write* variables. The set of all valuations on a set V is denoted by $\mathcal{V}[V]$. The value of variable v in state s is denoted by $s(v)$. Given a state s and a set $V' \subseteq V$, we define the *restriction* $[s]_{V'}$ as the valuation on V' such that $[s]_{V'}(v) = s(v)$ for all $v \in V'$. Given an MDP $\mathcal{M}_p$ whose set of variables is V_p, we write $[s]_p$ for $[s]_{V_p}$ and $[s]_p^W$ for $[s]_{W_p}$.

The set of states of a variable-based MDP $\mathcal{M}_p$ is the set $\mathcal{V}[V_p]$. We assume, however, that the transitions of variable-based MDPs are of the form (s, α, d) where d is a distribution on $\mathcal{V}[W_p]$ (instead of of $\mathcal{V}[V_p]$). In order to comply with the definition of MDP given in the previous subsection, we can lift d to $\mathcal{V}[V_p]$ in the obvious way by defining $d'(t) = 0$, if $t(v) \neq s(v)$ for some $v \notin W_p$; and $d'(t) = d([t]_p^W)$, otherwise. In what follows, when writing $\mu(t)$, we mean $d(t)$ if $t \in \mathcal{V}[W_p]$, or $d'(t)$ if $t \in \mathcal{V}[V_p]$.

We say that the MDPs $M_1, \cdots, M_N$, are compatible if $\forall_{p\neq q}\colon W_p \cap W_q = \emptyset$. Given a set of compatible MDPs $\{M_1, \cdots M_N\}$, let $\mathcal{M}(\alpha)$ be the subset comprising the modules such that $\alpha \in \Sigma_p$, $W(\alpha)$ be $\cup_{M_p\in\mathcal{M}(\alpha)} W_p$ and $\neg W(\alpha)$ be $V \setminus W(\alpha)$. We define the parallel composition $\mathcal{M} = \|_{p=1}^N M_p$ as the MDP (S, Q, Σ, T) such that:

- S is the set of valuations on $\bigcup_p V_p$
- Q is the set of states s such that $[s]_p^W \in Q_p$ for all p
- $\Sigma = \bigcup_{p=1}^N \Sigma_p$
- $\mu = (s, \alpha, d) \in T$ iff for all $M_p \in \mathcal{M}(\alpha)$ there exists $\mu_p \in \mathrm{en}_p([s]_p)$ such that $\mathrm{label}(\mu_p) = \alpha$, $\mu(t) = \prod_{M_p\in\mathcal{M}(\alpha)} \mu_p([t]_p^W)$ if $[s]_{\neg W(\alpha)} = [t]_{\neg W(\alpha)}$ and $\mu(t) = 0$ whenever $[s]_{\neg W(\alpha)} \neq [t]_{\neg W(\alpha)}$.

Given a transition $\mu = (s, \alpha, d)$ in $\mathcal{M}$, we define $[\mu]_p = ([s]_p, \alpha, d')$, where $d'(t') = \sum_{\{t|[t]_p^W = t'\}} d(t)$ for all $t' \in \mathcal{V}[W_p]$. Note that $\mu \in \mathrm{en}(s) \Longrightarrow [\mu]_p \in \mathrm{en}([s]_p)$.

Control functions. The definitions introduced so far map easily to modelling languages with shared variables such as Prism, but in order to consider partial information we also need to introduce a concept resembling input/output as in Probabilistic I/O Automata (PIOA [3]): the *control function*.

The technique we present considers a composition $\|_{p=1}^{N} M_p$ together with a function control: $\Sigma \to \{M_p\}_p \cup \{\bot\}$. If $\mathrm{control}(\alpha) = M_p$, the intended meaning is that M_p decides to execute the transitions with this label, while the other modules M_q with $\alpha \in \Sigma_q$ react to this transition. We formalize this meaning when introducing partial-information schedulers. Labels with $\mathrm{control}(\alpha) = \bot$ are not controlled by any module (it can be thought that they are controlled by an external entity that has full knowledge). For the previous definition to make sense, we require $\mathrm{control}(\alpha) \neq \bot \implies \alpha \in \Sigma_{\mathrm{control}(\alpha)}$. We impose the following condition, analogous to the input-enabledness condition for Input/Output Automata [3]:

$$\alpha \in \Sigma_p \wedge \mathrm{control}(\alpha) \neq \bot \wedge \mathrm{control}(\alpha) \neq M_p \implies \forall s_p \in S_p : \exists t_p : s_p \xrightarrow{\alpha} t_p \; . \quad (1)$$

In other words, whenever the module $\mathrm{control}(\alpha)$ chooses to execute a transition, it will not be blocked because of other modules not having an α transition enabled. Given a transition μ we write $\mathrm{control}(\mu)$ instead of $\mathrm{control}(\mathrm{label}(\mu))$.

The control/reaction mechanism is similar to the PIOA, but our definition for Mdps is simpler, as it does not need input and output schedulers as in Switched PIOA [3] nor tokens [3] (or interleaving schedulers [9]) for interleaving non-determinism.

If the model is specified in a language that does not allow control specifications (such as Prism), we can take $\mathrm{control}(\alpha) = M_p$ if M_p is the only module with $\alpha \in \Sigma_p$, and $\mathrm{control}(\alpha) = \bot$ if $\alpha \in \Sigma_p \cap \Sigma_q$ for some $p \neq q$.

Partial-information schedulers for Mdps. Our partial-information schedulers require the choices of a module to be the same in all paths in which the information observed by such module is the same. Given a path ρ the information available to M_p is called the *projection* of ρ over M_p (denoted $[\rho]_p$). In order to define projections, we say that a transition μ *affects* module M_p iff $\mathrm{label}(\mu) \in \Sigma_p$ or $s \xrightarrow{\mu} t$ for some s, t such that $[s]_p \neq [t]_p$. Projections are defined inductively as follows: the projection of the path s^0 is $[s^0]_p$. The projection $[\rho.\mu.s]_p$ is defined as $[\rho]_p.[s]_p$ if μ affects M_p or $[\rho.\mu.s]_p = [\rho]_p$, otherwise. Alternatively, projections might have been defined to include also information about the transitions. Our definition is simpler and, in addition, it is easy to emulate the (apparently) more general definition by keeping the last transition executed as part of the state of the module.

Definition 1 (Partial-information scheduler). *Given* $\mathcal{M} = \|_{p=1}^{N} M_p$*, the set* $\mathrm{PISched}(\mathcal{M})$ *comprises all schedulers* η *such that for all modules* M_p*, paths* $\rho, \sigma \in \mathrm{Paths}(\eta)$ *with* $[\rho]_p = [\sigma]_p$ *the two following conditions hold:*

$$\text{label}(\eta(\rho)) = \text{label}(\eta(\sigma)) \in \Sigma_p \implies [\eta(\rho)]_p = [\eta(\sigma)]_p \tag{2}$$

$$\text{control}(\eta(\rho)) = \text{control}(\eta(\sigma)) = M_p \implies \text{label}(\eta(\rho)) = \text{label}(\eta(\sigma)) \; . \tag{3}$$

To understand (2), recall that in a state s there might be different transitions enabled, say μ, λ, with $\text{label}(\mu) = \text{label}(\lambda) = \alpha$. Hence, (2) ensures that, if the module M_p synchronizes in a transition labelled with α after both ρ and σ, then the particular transition used in the synchronization must be the same after both ρ and σ. In Eq. (3) we require that, if M_p is the module that chooses the label in both ρ and σ, then the chosen label must be the same.

Given an Mdp $\mathcal{M} = \|_{p=1}^{N} M_p$, a set $\mathcal{H}$ of infinite paths and $\mathcal{B} \in [0,1]$, we are interested in the problem of deciding if $\text{Pr}^{\eta}(\mathcal{H}) \leq \mathcal{B}$ for all $\eta \in \text{PISched}(\mathcal{M})$. The inequality holds iff $\sup_{\eta \in \text{PISched}} \text{Pr}^{\eta}(\mathcal{H}) \leq \mathcal{B}$. This problem has been proven undecidable, even in the particular case in which $\mathcal{H}$ is the set of paths that reach a particular state [10]. In contrast, it is well-known that the model-checking problem under total information is solvable in polynomial time on the size of the model for logics such as Pctl* and Ltl [7].

3 Refinement

In this section we show how a system can be verified under partial information by using total-information techniques on successive refinements of the original system. Our technique starts by verifying the system $\mathcal{M}$ under total information, that is, by calculating $\mathcal{S}_{\text{Tot}} = \sup_{\eta \in \text{Sched}} \text{Pr}^{\eta}(\mathcal{H})$. If $\mathcal{S}_{\text{Tot}}$ is less than or equal to the allowed probability $\mathcal{B}$ then the inclusion PISched $\subseteq$ Sched implies $\mathcal{S}_{\text{Par}} = \sup_{\eta \in \text{PISched}} \text{Pr}^{\eta}(\mathcal{H}) \leq \mathcal{B}$, and hence the system is correct under partial information. In case that $\mathcal{S}_{\text{Tot}} > \mathcal{B}$, model-checking algorithms can provide a representation of a scheduler η such that $\text{Pr}^{\eta}(\mathcal{H}) > \mathcal{B}$ (see [7]). If $\eta \in$ PISched, then $\mathcal{M}$ is not correct under partial information, and η is a counterexample. If $\eta \notin$ PISched, then we perform a transformation on the system that prevents the particular counterexample and with it a class of schedulers that violate the partial information assumption in a similar way.

We start the description of our technique by showing how to check if a scheduler η is in PISched.

3.1 Detection of Partial-Information Counterexamples

Under total information, for minimum/maximum reachability it suffices to consider only *globally Markovian* (Gm) schedulers. A scheduler η is Gm iff $\eta(\rho) = \eta(\sigma)$ for all $\rho, \sigma \in \text{Paths}(\eta)$ with $\text{last}(\rho) = \text{last}(\sigma)$. Moreover, the verification of general Ltl and Pctl* formulae is carried out by reducing the original problem to problems for which Gm schedulers are sufficient [7].

We address the problem of checking whether $\eta \in$ PISched for a given Gm scheduler η. The method here resembles the well-known technique of *self-composition* [1]. We denote by $\eta^{>0}(s)$ the value of $\eta(\rho)$ for all $\rho \in \text{Paths}(\eta)$ with

$\mathrm{last}(\rho) = s$. Note that a Gm scheduler is completely determined by the value of $\eta^{>0}(s)$ in the states s reachable in η, in the sense that if two schedulers η, ζ reach the same states and $\eta^{>0}(s) = \zeta^{>0}(s)$ for all reachable states, then $\eta \equiv \zeta$.

We reduce the problem to that of checking whether η has *conflicting paths.* Given a Gm η, we say that two states s, t with $[s]_p = [t]_p$ are *η-conflicting* for the module M_p iff either of the following holds

$$\mathrm{label}(\eta^{>0}(s)) = \mathrm{label}(\eta^{>0}(t)) \in \Sigma_p \wedge [\eta^{>0}(s)]_p \neq [\eta^{>0}(t)]_p \quad \text{or} \tag{4}$$

$$\mathrm{control}(\eta^{>0}(s)) = \mathrm{control}(\eta^{>0}(t)) = M_p \wedge \mathrm{label}(\eta^{>0}(s)) \neq \mathrm{label}(\eta^{>0}(t)) \; . \tag{5}$$

We say that two paths $\rho, \sigma \in \mathrm{Paths}(\eta)$ are *η-conflicting* for M_p if $[\rho]_p = [\sigma]_p$, $\mathrm{last}(\rho) = s$, $\mathrm{last}(\sigma) = t$ and the states s, t are η-conflicting. From the definition of η-conflicting and the definition of PISched, we have the following equivalence for all Gm η:

$$\eta \in \mathrm{PISched} \iff \eta \text{ has no conflicting paths} \; . \tag{6}$$

Now we show how to check the (in)existence of conflicting paths for M_p. For all $s, t \in S$, $r_p \in S_p$, we define the relation $s \overset{r_p}{\rightsquigarrow} t$, that holds iff there exist paths ρ, σ such that $\rho \cdot \sigma \in \mathrm{Paths}(\eta)$, $\mathrm{last}(\rho) = s$, $\mathrm{last}(\sigma) = t$ and $[\sigma]_p = r_p$. This relation can be extended naturally to the sequences $\pi_p = r_p^1 \cdots r_p^k$, so we can write $s \overset{\pi_p}{\rightsquigarrow} t$. By the definition of $\rightsquigarrow$, if $s \overset{\pi_p}{\rightsquigarrow} t$ then there exists a path π from s to t in η such that $[\pi]_p = \pi_p$. Consider the non-deterministic finite automaton $\mathrm{Nfa}_p(\eta)$ that represents the relation $\overset{r_p}{\rightsquigarrow}$. In this automaton, each word starting in s and ending in t corresponds to a π_p such that $s \overset{\pi_p}{\rightsquigarrow} t$.

The problem of checking whether η has conflicting paths is now that of checking whether $\mathrm{init}_\eta \overset{\pi_p}{\rightsquigarrow} s$ and $\mathrm{init}_\eta \overset{\pi_p}{\rightsquigarrow} t$ for some η-conflicting s, t. This can be done by constructing the synchronous product automaton

$$\mathrm{Nfa}_p^2(\eta) = \mathrm{Nfa}_p(\eta) \times \mathrm{Nfa}_p(\eta) \tag{7}$$

and checking whether it has a path from $(\mathrm{init}_\eta, \mathrm{init}_\eta)$ to some η-conflicting (s, t).

Superfluous conflicts. We show that some conflicts can be ignored. For instance, consider that we are calculating $\sup_\eta \mathrm{Pr}^\eta(\mathrm{reach}(U))$, and we find that s, t are η-conflicting, and the probability $\mathrm{Pr}^\eta_s(\mathrm{reach}(U))$ to reach U starting from s is 0. We say that this conflict is *superfluous.* Intuitively, one might think that η could be changed so as to not cheat in s, and the probabilities would not decrease, as the probability from s was already 0 in η: if all the conflicts are superfluous, then, we should be able to construct a partial-information scheduler yielding the same probabilities. This intuitive reasoning can be proven, and we state it formally in the theorem below. For minimum probabilities, we say that the conflict is superfluous if $\mathrm{Pr}^\eta_s(\mathrm{reach}(U)) = 1$.

Theorem 1. *Let η be such that $\mathrm{Pr}^\eta(\mathrm{reach}(U)) = \sup_{\eta'} \mathrm{Pr}^{\eta'}(\mathrm{reach}(U))$ (or, resp., $\mathrm{Pr}^\eta(\mathrm{reach}(U)) = \inf_{\eta'} \mathrm{Pr}^{\eta'}(\mathrm{reach}(U))$). If all conflicts in η are superfluous, then there exists $\eta^* \in \mathrm{PISched}$ such that $\mathrm{Pr}^{\eta^*}(\mathrm{reach}(U)) = \mathrm{Pr}^\eta(\mathrm{reach}(U))$.*

3.2 Refining a System for a Conflict

According to Eq. (6), if a counterexample η does not comply with the partial-information constraints, there exist two η-conflicting states s and t for a module M_p. Since these states are conflicting, we have that $[s]_p = [t]_p$ and either (4) or (5) holds. In words, two different transitions μ, λ are chosen in M_p while, because of the partial-information constraints the choices must coincide. Next, we show how to refine the module M_p. The refinement is modular, in the sense that only M_p is affected.

As illustrated in the example in the introduction, the idea is to split $[s]_p$ in such a way that μ and λ are not enabled in the same state. We present a general way to split states, that will be useful to develop a splitting criterion that ensures convergence for the case of the infimum probabilities (see Thm. 3).

In order to ensure input-enabledness, the transitions enabled in each of the split states must comply with a certain condition: given a state s_p of M_p, a *minimal choice set* is a set F of transitions such that, for each label α not controlled by M_p there is transition labelled with α in F and, if there is a controlled transition enabled in s_p, then F has at least one controlled transition. Intuitively, when a module restricts its choices to a minimal choice set F, it fixes the reactions for all non controlled transitions, and fixes the controlled transition to execute (if any). In the following, let $\mathcal{T}_1, \cdots, \mathcal{T}_Z$ be sets of transitions such that for every minimal choice set F there exists i such that $F \subseteq \mathcal{T}_i$. These sets are called *choice sets*. As an example, the simplest splitting criterion is, given two states s, t being η-conflicting for M_p, take $\mathcal{T}_1 = \mathrm{en}([s]_p) \setminus \{[\eta(s)]_p\}$ and $\mathcal{T}_2 = \mathrm{en}([s]_p) \setminus \{[\eta(t)]_p\}$ as choice sets.

The overall idea of the transformation is then simple: the state s_p being split is replaced by Z states $s_p^1, \cdots, s_p^Z$ in such a way that the transitions enabled in s^i correspond to $\mathcal{T}_i$. We construct the refined module M_q in the following definition.

Definition 2. *Given a module M_p in $\mathcal{M}$, $u \in S_p$, and choice sets $\{\mathcal{T}_i\}_{i=1}^Z$, the refinement $M_q(\eta, u, \{\mathcal{T}_i\}_{i=1}^Z)$ is defined as:*

- *$V_q = V_p \cup \{x\}$ where $x \notin V_p$ is a fresh variable name. The domain of x is $\{1, \cdots, Z\}$. In addition, $W_q' = W_p \cup \{x\}$*
- *S_q is the set of valuations on V_q*
- $Q_q = \{v \mid [v]_p = u \wedge [v]_p \in Q_p\} \,\cup\, \{v \mid [v]_p \neq u \wedge v(x) = 1 \wedge [v]_p \in Q_p\}$
- $\Sigma_q = \Sigma_p \cup \{\alpha \mid \exists u' : [u']_p^W \xrightarrow{\alpha} [u]_p^W\}$
- *for all $v \in S_q$, we have $\mu_q \in \mathrm{en}_q(r)$ iff some of the following conditions hold:*
  - $\mathrm{label}(\mu_q) \in \Sigma_q \setminus \Sigma_p$ *and* $\mu_q(v') = 1$ *for some* v' *such that* $[v']_p^W = [v]_p^W$.
  - $\mathrm{label}(\mu_q) \in \Sigma_p$,

$$\forall v', v'' \in \mathrm{supp}(\mu_q) : v'(x) = v''(x) \tag{8}$$

and there exists $\mu_p \in \mathrm{en}_p([v]_p)$ such that $\mu_p([u]_p^W) > 0$ and

$$\forall v' : \mu_q(v') = \mu_p([v']_p) \qquad \textit{and} \tag{9}$$

$$[v]_p \neq u \;\vee\; (v(x) = i \wedge \mu_p \in \mathcal{T}_i\ )\ . \tag{10}$$

- $\text{label}(\mu_q) \in \Sigma_p$, *and*

$$\forall v' \in \text{supp}(\mu) : v'(x) = 1 \tag{11}$$

and there exists $\mu_p \in \text{en}_p([v]_p)$ *such that* $\mu_p([u]_p^W) = 0$ *and (9) and (10) hold.*

Next we explain this transformation informally. If $u \in Q_p$, then in Q_q we have Z states corresponding to u_p, the variable x having different values in each state (note that these two states constitute the set of all states v such that $[v]_p = u$). For all the other initial states, we just set the value of x arbitrarily to 1 (we could have chosen any value in $\{1, \cdots, Z\}$), since the transitions that lead to u will update the variable regardless of the initial value.

Notice that the alphabet of the module is extended in order to synchronize in all transitions that might change the variables of M_p, and lead it to u.

The transitions in M_q that synchronize on the newly added labels only change the value of x in a non-probabilistic fashion. The transitions in M_q having labels that were already in Σ_p correspond to transitions μ_p that were already in M_p. We have two cases: first we consider the case in which μ_p reaches u. Each of the corresponding transitions μ_q changes the value of x in a non-probabilistic fashion (condition (8)) so that there are Z transitions corresponding to μ_p: each one setting x to a value in $\{1, \cdots, Z\}$. Wrt. the other variables, the transitions μ_q behave in the same way as in M_p (condition (9)). In addition, condition (10) ensures that, if μ_q is enabled in a state corresponding to u, then μ_q corresponds to a transition in M_p that is consistent with the value of x. In case μ_p does reach not u, condition (11) specifies that the transition μ_q set the value of x to 1 (for the same reason that some initial states were set to such an arbitrary value). Note that the transitions μ_q must also respect the probabilities in μ_p, and be consistent with the value of x (as we require the conditions (9) and (10)).

If the module M_p in $\mathcal{M} = \|_r M_r$ is refined to $M_q(u, \{\mathcal{T}_i\}_{i=1}^Z)$, the refined system $\mathcal{M}(u, \{\mathcal{T}_i\}_{i=1}^Z)$ is $M_q(u, \{\mathcal{T}_i\}_{i=1}^Z) \;\|_{r \neq p} M_r$. $\text{control}(\alpha)$ remains unchanged.

Given a set of infinite paths $\mathcal{H}$ in an MDP $\mathcal{M}$ with variables V, we can obtain the corresponding set in $\mathcal{M}(u, \{\mathcal{T}_i\}_{i=1}^Z)$:

$$\mathcal{H}' = \{s'^0.\mu'^1.s'^1.\cdots \mid [s'^0]_V.[\mu'^1]_V.[s'^1]_V \cdots \in \mathcal{H}\} \ .$$

The following theorem ensures that we can apply refinements without changing the partial-information extremal probability values.

Theorem 2. $\sup_{\eta \in \text{PISched}(\mathcal{M})} \Pr^\eta(\mathcal{H}) = \sup_{\eta \in \text{PISched}(\mathcal{M}(u, \{\mathcal{T}_i\}_{i=1}^Z))} \Pr^\eta(\mathcal{H}')$

3.3 Convergence

In the beginning of this section we introduced the problem of computing $\mathcal{S}_{\text{Par}}$, in order to know whether $\mathcal{S}_{\text{Par}} \leq \mathcal{B}$. Depending on the particular system and property being checked, and on how the choice sets are chosen, the probabilities in the refined systems might actually converge or not to $\mathcal{S}_{\mathrm{Par}}$. The next subsection shows that, when calculating infimum values for reachability properties, there exists a sequence of choice sets that ensures that the probabilities converge.

Suppose the variables in the original system are $x_1, \cdots, x_N$, and that given a system with N variables, the new variable introduced in Definition 2 is x_{N+1}. Given $\mathcal{M}$, we write $N(\mathcal{M})$ for the number of variables in $\mathcal{M}$. Let $V \downarrow i$ be the set of variables $\{x_1, \cdots, x_i\}$. An i-state is an element of $\mathcal{V}[V{\downarrow}i]$. We say that two transitions are an i-choice in state s_p if they are enabled in s_p and $1 \leq i \leq N$ is the minimum value such that $[\mu]_{W_p \cap V{\downarrow}i} \neq [\lambda]_{W_p \cap V{\downarrow}i}$. Exceptionally, if $\mathrm{label}(\mu) \neq \mathrm{label}(\lambda)$ and $\mathrm{control}(\mathrm{label}(\mu)) = \mathrm{control}(\mathrm{label}(\lambda)) = M_p$, we say that μ, λ are a 0-choice. When refining a system with N variables, we eliminate (at least) one i-choice, where trivially $i \leq N$, and for each transition leading to the refined state we create new $(N+1)$-choices: the different transitions μ_q corresponding to a transition μ_p in Definition 2 can be distinguished only by the values they assign to x_{N+1}.

Consider a scheduler $\eta \notin \mathrm{PISched}$. By Eq. (6), it has at least two conflicting states s, t. We say that μ, λ, is an i-conflict in η iff there exist s, s', s_p, such that $[s]_p = [s']_p = s_p$, $[\eta(s)]_p = \mu$, $[\eta(t)]_p = \lambda$, and μ, λ are an i-choice in s_p. We say that μ, λ is a minimum conflict if it is an i-conflict with minimum i, quantifying over all conflicts in η.

Informally, the following theorem states that, if every time we search for a minimum conflict, and we refine all the states having such a conflict, then the infimum probabilities under total information (in the refined systems) converge to the infimum probability under partial information (in the original system).

Theorem 3. *Given an* MDP $\mathcal{M}$ *and a set of states* U*, consider the sequence of* MDP*s defined inductively as follows:* $\mathcal{M}_0$ *is* $\|_p M_p$*. Given* $\mathcal{M}_k$ *and* $\eta_k \notin \mathrm{PISched}(\mathcal{M}_k)$*, we define* $\mathcal{M}_{k+1}$ *by defining intermediate systems* $\mathcal{M}_{k,l}$*. Let* $S_{\min}$ *be the states with a minimum conflict* μ*,* λ *for* M_p *in* η_k*. The system* $\mathcal{M}_{k,0}$ *is* $\mathcal{M}_k$*; given* $\mathcal{M}_{k,l}$*, if there exists* s^{l+1} *such that* $[s^{l+1}]_{V{\downarrow}N(\mathcal{M}_k)} \in S_{\min}$ *then we let* $\mathcal{M}_{k,l+1} = \mathcal{M}_{k,l}([s^{l+1}]_p, \{\mathcal{T}_1, \mathcal{T}_2\})$ *, where* $\mathcal{T}_1 = \mathrm{en}([s^{l+1}]_p) \setminus \{\mu' \mid [\mu']_{V_p{\downarrow}N(\mathcal{M}_k)} = \mu\}$ *and* $\mathcal{T}_2 = \mathrm{en}([s^{l+1}]_p) \setminus \{\lambda' \mid [\lambda']_{V_p{\downarrow}N(\mathcal{M}_k)} = \lambda\}$*. If no such* s_p^{l+1} *exists, we let* $\mathcal{M}_{k+1} = \mathcal{M}_{k,l}$*. There exists* l *such that* $\mathcal{M}_{k+1} = \mathcal{M}_{k,l}$*. Moreover, we have*

$$\lim_{k \to \infty} \inf_{\eta \in \mathrm{Sched}} \mathrm{Pr}^{\eta}_{\mathcal{M}_k}(\mathrm{reach}(U_k)) = \inf_{\eta \in \mathrm{PISched}} \mathrm{Pr}^{\eta}_{\mathcal{M}}(\mathrm{reach}(U)),$$

where U_k *is the set of all states* s *in* $\mathcal{M}_k$ *such that* $[s]_{V{\downarrow}N(\mathcal{M})} \in U$*.*

For simplicity, in the theorem above we assume that we are unlucky and we never find a partial information counterexample. In case we do, we can simply make $\mathcal{M}_{k+1} = \mathcal{M}_k$. We also preserve indices across refinements: if M_p is refined in $\mathcal{M}_{k,l}$, then M_p is the refined module in $\mathcal{M}_{k,l+1}$.

There is no similar convergence for upper bounds of supremum probabilities: together with the computable lower bounds $\lim_{n \to \infty} \mathrm{Pr}^{\eta}(\mathrm{reach}(U_n))$ (where $\mathrm{reach}(U_n)$ is the set of paths reaching U before n steps) such upper bounds would turn the approximation problem for the supremum decidable, and it is not [10].

4 Experimental Results

In spite of Thm. 3, we do not have an upper bound on the number of refinements needed to get a probability ϵ-close to the actual one. In consequence, we implemented a preliminary prototype of the refinement technique in Prism, to check whether some improvements in worst-case probabilities can be found in practically acceptable times.

Variants implemented. We found that, in some cases, the criterion for selecting conflicts in Thm. 3 (which we refer to as “minimum variable conflict” or MVC) does not improve the probabilities quickly enough. An explanation for this behaviour can be seen in a small example: in a system with N variables, suppose that in a module we have $s \xrightarrow{\mu} t \xrightarrow{\lambda} r$, and in r there is a non-deterministic choice that the fictitious counterexamples resolve according to a hidden coin that is tossed by another module in a transition than synchronizes with μ. In the first refinement, we pull the non-determinism backwards to s'. For the scheduler not to cheat any longer, we need to pull the non-determinism to s but, in the first variant, this only occurs after all the N-conflicts have been refined, as the choices introduced in s' are $(N+1)$-choices.

We implemented a second variant (which we refer to as “shortest projection conflict” or SPC) that performs a breadth-first search on the automaton $\mathrm{Nfa}_p^2(\eta)$ in Eq. (7), in order to obtain the shorter projection ρ_p for which the partial information restrictions are violated. In the previous example, if $s.\mu.t.\lambda.r$ is the smallest conflicting path in the first counterexample, then $s.\mu.t$ is the smallest conflicting path in the second counterexample, and the cheating is eliminated in the second refinement. In addition, in this variant we split each state into several states: if the conflict concerns controlled transitions, then each choice set has a single controlled transition (and also has all non-controlled transitions); if the conflict concerns a reaction to a certain label α, then each choice set has a single transition labelled with α (and also all controlled transitions, and transitions with labels other than α). This is easy to implement and yields better results in practice. Examples can be constructed to show that SPC does not converge in all cases.

Table 1. Experimental results

					SPC				MVC		
	N	k	$\lvert S\rvert$ initial	Initial prob.	$\lvert S\rvert$ final	Final prob.	Time (s)	Superf. conflicts	$\lvert S\rvert$ final	Final prob.	Time (s)
Light bulb	3	4	712	0.44	2376731	0.30 *	2973	10	8506371	0.44	12418
	3	5	1042	0.62	5774983	0.50	16762	0	7592182	0.62	11317
	4	4	2069	0.09	893665	0.05 *	1561	80	2015056	0.09	19983
	4	5	3453	0.23	7708257	0.15	26363	156	8294102	0.23	14182
Crowds	3	-	109	1.00	6393	0.40 *	6	0	17705	0.40 *	54
	4	-	237	1.00	163634	0.30 *	530	0	308510	0.55	20050
Afs	20	7	6600	1.00	6978	0.23 *	197	0	6978	0.23 *	198
	25	8	10125	1.00	10719	0.22 *	710	0	10719	0.22 *	728

Experiments. We analysed three systems: (1) the crowds protocol [14]: a group of N entities tries to deliver a message to an evil party in an anonymous way. We ask for the maximum probability that the evil party guesses the sender correctly. (2) the protocol for anonymous fair service (Afs) in [11] (precisely, the variant called Afs1 in [11]), which involves two clients and a server. We ask for the worst-case probability that one of the clients gets k services more than the other one before the first N services happen; (3) the problem of the light bulb and the prisoners [15]: there is a group of N prisoners. In each round, a prisoner is selected at random and taken to a room where he can switch the light bulb. The only information available to each prisoner is the state of the light bulb when he enters the room. The prisoners win the game if one of them can, at some point, guess that all of them have been in the room (they lose in case of a wrong guess). We ask for the maximum probability that the prisoners lose the game in k steps. For (2), we reused in [10]; the other models were written specifically, as there are no existing Prism models with partial information constraints.

The results are shown in Table 1. The experiments were ran on a six-core Intel Xeon at 2.80 GHz with 32Gb of memory, in order to be able to analyse the problem of the light bulb and the prisoners for as many refinements as possible. The table shows the number of states in the initial system, and in the final system after the last refinement, and similarly for the probability values. We set a time-out of 8hs. for experiments: the cases where the last scheduler had no conflicts (or all of them were superfluous) are marked with a $*$. In these cases we also indicate the number of superfluous conflicts in the last scheduler, as a measure of the usefulness of Thm. 3. For timed-out experiments, the time reported is the time spent to compute the last probability computed.

The two criteria have similar performance for Afs, but SPC outperforms MVC in the other systems. Even when SPC cannot find the realistic probability, in the light bulb case study, it is able to obtain improvements for the probabilities. It is also remarkable that, for Afs, we ran the experiments for the same parameters as in [11] ($N = 20$, $k = 1..20$, not shown in the Table 1) and the probabilities obtained using our technique coincide with the ones in [11]: these were known to be safe bounds, but thanks to our technique now we know that they were in fact exact values.

5 Concluding Remarks

Decidability. Although Thm. 3 ensures that the successive values converge to the infimum reachability probability, we found no way to check if a given approximation is within a certain error threshold: although the bounds get tighter, there is no way to know when we are close enough to the actual value. These approximations are still valuable, as it is not known whether the approximation problem is decidable for infimum probabilities. It is undecidable for the supremum [10], but the infimum/supremum problems are not trivially dual to each other: in fact, for probabilistic finite automata (which are a particular case of the models in [3,10] and here) the approximation problem is undecidable for the supremum [13] and decidable for the infimum [8].

Complexity. Our result of convergence (Thm. 3) concerns the computation of the infimum reachability probability. The problem of, given a system such that the infimum is 0 or 1, determining which of the cases holds, can be shown to be NP-hard using a similar argument as in [9, Thm. 5.5]. This NP-hardness result implies that, unless $P = NP$, for each error threshold ϵ there is no polynomial algorithm to approximate the value within relative error ϵ (these problems are often called *inapproximable*). We do not have any bounds on the amount of refinements needed to get an approximation: however. Given this hardness result, the best we can expect is that our technique is useful in practical cases, as shown in Sec. 4.

Further work. We are particularly interested in linking this approach with existing ones. For instance, we plan to study if it can be applied to the game-like setting in [6], or whether it can be adapted for qualitative properties as studied in [2]. In addition, we plan to consider conflict selection criteria other than "minimum-variable-first" and "shortest-projection-first", to find better results in practical cases. Given that our results are appealing to quantitative verification, a natural step forward would be to extend the results here to consider discounts and rewards.

References

1. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW, pp. 100–114. IEEE Computer Society (2004)
2. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative Analysis of Partially-Observable Markov Decision Processes. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010)
3. Cheung, L., Lynch, N.A., Segala, R., Vaandrager, F.W.: Switched pioa: Parallel composition via distributed scheduling. TCS 365(1-2), 83–108 (2006)
4. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: QEST, pp. 131–132. IEEE CS (2006)
5. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional Methods for Probabilistic Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 351–365. Springer, Heidelberg (2001)
6. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In: FSTTCS. LIPIcs, vol. 2, pp. 175–186 (2008)
7. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated Verification Techniques for Probabilistic Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
8. Giro, S.: An algorithmic approximation of the infimum reachability probability for probabilistic finite automata. CoRR, abs/1009.3822 (2010)
9. Giro, S.: On the automatic verification of distributed probabilistic automata with partial information. PhD thesis, FaMAF – Universidad Nacional de Córdoba (2010), http://cs.famaf.unc.edu.ar/~sgiro/thesis.pdf
10. Giro, S., D'Argenio, P.R.: Quantitative Model Checking Revisited: Neither Decidable Nor Approximable. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 179–194. Springer, Heidelberg (2007)

11. Giro, S., D'Argenio, P.R.: On the verification of probabilistic i/o automata with unspecified rates. In: SAC, pp. 582–586. ACM (2009)
12. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
13. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artif. Intell. 147(1-2), 5–34 (2003)
14. Reiter, M.K., Rubin, A.D.: Anonymous web transactions with crowds. Commun. ACM 42(2), 32–38 (1999)
15. van Ditmarsch, H.P., van Eijck, J., Wu, W.: One hundred prisoners and a lightbulb - logic and computation. In: KR (2010)

The COMICS Tool – Computing Minimal Counterexamples for DTMCs

Nils Jansen[1], Erika Ábrahám[1], Matthias Volk[1], Ralf Wimmer[2], Joost-Pieter Katoen[1], and Bernd Becker[2]

[1] RWTH Aachen University, Germany
[2] Albert-Ludwigs-University Freiburg, Germany

Abstract. This paper presents the tool `COMICS` 1.0, which performs model checking and generates counterexamples for DTMCs. For an input DTMC, `COMICS` computes an abstract system that carries the model checking information and uses this result to compute a *critical subsystem*, which induces a counterexample. This abstract subsystem can be refined and concretized *hierarchically*. The tool comes with a command line version as well as a graphical user interface that allows the user to interactively influence the refinement process of the counterexample.

1 Introduction

Discrete-time Markov chains (DTMCs) are widely used to model safety-critical systems with uncertainties. Model checking *probabilistic computation tree logic* (PCTL) properties can be performed by prominent tools like Prism [1] and Mrmc [2]. Unfortunately, the implemented numerical methods do not provide diagnostic information in form of *counterexamples*, which are very important for debugging and are also needed for CEGAR frameworks [3].

Although different approaches [4,5,6] were proposed for probabilistic counterexamples, there is still a lack of efficient and user-friendly *tools*. To fill this gap, we developed the tool `COMICS`, supporting SCC-based model checking [7] and, in case the property is violated, the *automatic* generation of *abstract counterexamples* [5], which can be subsequently refined either automatically or user-guided.

While most approaches represent probabilistic counterexamples as sets of paths, we use (hierarchically abstracted) subgraphs of the input DTMC, so-called *critical subsystems*. The user can refine abstract critical subsystems *hierarchically* by choosing system parts of interest which are to be concretized and further examined. All computation steps of the hierarchical counterexample refinement can be *guided and revised*. Though refinement can be done until a fully concrete counterexample is gained, it seems likely that the user can gain sufficient debugging information from abstract systems considering real-world examples with millions of states. The tool's graphical user interface (GUI) permits *visualization, reviewing and creation* of test cases.

The only other available tool we are aware of is DiPro [8], which supports both DTMCs and CTMCs but no abstract counterexamples, which is crucial for

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 349–353, 2012.

the handling of large systems. It also does not allow the user to influence the search by using his or her expertise. Comparative experiments show that we can compute reasonably smaller counterexamples in shorter time with our tool.

In Section 2 we give a brief introduction to the methods implemented in our tool. We describe the features and architecture and report on benchmarks in Section 3. We conclude the paper in Section 4. The tool, a detailed manual, and a number of benchmarks are available at the COMICS website[1].

2 Foundations

In this section we briefly explain the algorithms implemented in COMICS (see [5] for more details). We use the standard definitions for DTMCs and PCTL.

Model checking time-unbounded PCTL properties for DTMCs can be reduced to the following problem: Given a DTMC M with one *initial state* s_{I} and a set of *target states* T, decide whether the probability to reach T from s_{I} is below an upper bound[2] $\lambda \in [0, 1] \subset \mathbb{R}$. In case this bound is violated, a *counterexample* can be given as a set of finite paths of M leading from s_{I} to T with a cumulated probability mass greater than λ.

In [7] we proposed a model checking approach for DTMCs based on *hierarchical abstraction*. The result is an abstract DTMC, which represents the total probabilities of reaching target states from the initial state by single transition probabilities. The abstraction is hierarchically refinable, where the refinement of an abstract state might again contain abstract states. Based on this approach, in [5] we presented a method to compute and represent counterexamples as *critical subsystems*, consisting of subsets of the original DTMC's states and transitions such that the probability of reaching target states from the initial state within the subsystem still exceeds the probability bound λ. We compute these subsystems using path searches on the abstract DTMCs: either the *global search* (GS), which searches for most probable paths from s_{I} to T, or the *local search* (LS), which connects fragments of already found paths to extend the current subsystem. Abstract subsystems can be refined by *selecting* and *concretizing* abstract states and performing path search again to reduce the number of concretized states and transitions in the subsystem.

3 The COMICS Tool

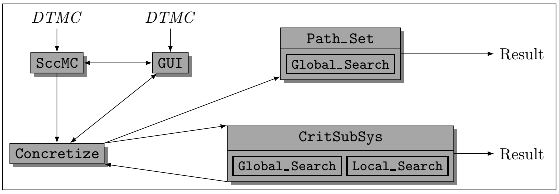
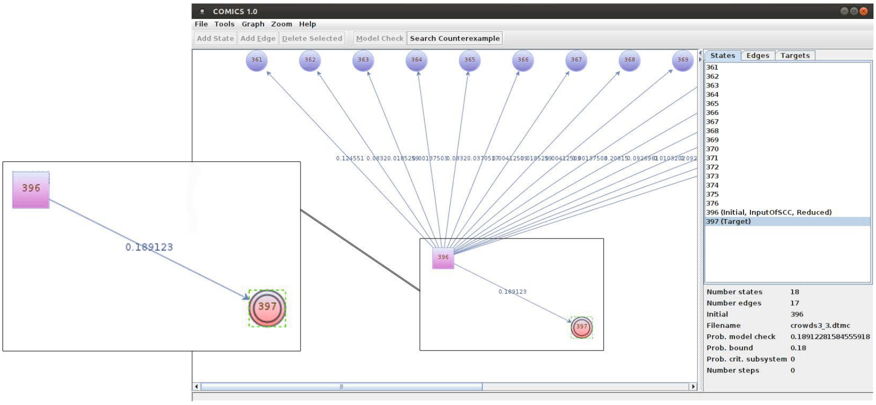


Fig. 1. Architecture of COMICS

COMICS can be used either as a command-line tool or with a GUI, the latter allowing the user to actively influence the process of finding a counterexample. The program consists of approximately 20 000 lines of code in five

[1] http://www-i2.informatik.rwth-aachen.de/i2/comics/

[2] We only consider unbounded PCTL properties.



361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
396 (Initial, InputOfSCC, Reduced)
397 (Target)

Number states	18
Number edges	17
Initial	396
Filename	crowds3_3.dtmc
Prob. model check	0.18912281584555918
Prob. bound	0.18
Prob. crit. subsystem	0
Number steps	0

Fig. 2. Screenshot of COMICS's GUI with an instance of the crowds protocol

main components (see Fig. 1). The GUI is implemented in Java, all other components in C++. The user may select *exact* or *floating point* arithmetics for the computations.

SccMC performs *model checking* for an input DTMC and returns an abstract DTMC to Concretize or to GUI. Concretize *selects and concretizes* some states, either automatically or user-guided via the GUI. CritSubSys can be invoked on the modified system to compute a critical subsystem using GS or LS. The result is given back to Concretize for further refinement or returned as the result. *Heuristics* for the number of states to concretize in a single step as well as for the choice of states are offered. It is also possible to predefine the number of concretization steps. Counterexample representations as *sets of paths* and as *critical subsystems* are offered. The first case yields a *minimal counterexample* [4]. The GUI provides a *graph editor* for specifying and modifying DTMCs. A large number of *layout algorithms* increase the usability even for large graphs. Both concrete and abstract graphs can be *stored*, *loaded*, *abstracted*, and *concretized* by the user. As the most important feature, the user is able to *control the hierarchical concretization* of a counterexample. If an input graph seems too large to display, the tool offers to operate without the graphical representation. In this case the abstract graph can be computed and refined in order to reduce the size. Fig. 2 shows one abstracted instance of the *crowds protocol* benchmark [9], where the probability of reaching the unique target state is displayed in the information panel on the right as well as on the edge leading from the initial state to the target state. The initial state is abstract and can therefore be expanded.

Fig. 3 provides a comparison with DiPro [8]. We applied our tool using GS, LS and the k-shortest path (kSP) approach [4] to the *crowds protocol* and the *probabilistic contract signing protocol* [10] for different probability thresholds all smaller than the model checking result (total prob.). We measured the size of the counterexample (states), the probability of reaching target states (prob.) and the computation time excluding the initial model checking. TO denotes timeout, MO out of memory and ERR wrong result. On the crowds protocol, GS performs

		crowds						contract signing			
states		3515		18817	198199	485941	1058353	33790	156670	737278	1654782
transitions		6035		32677	198199	857221	1872313	34813	157693	753663	1671165
total prob.		0.2346		0.4270	0.7173	0.809	0.8731	0.5156	0.5156	0.5039	0.5039
prob. threshold		0.15	0.23	0.25	0.35	0.4	0.4	0.5	0.5	0.5	0.5
`GS`	# states	629	1071	2036	5198	5248	5250	6827	37601	140034	369448
	prob.	0.1501	0.2301	0.25	0.3503	0.4002	0.4001	0.5	0.5	0.5	0.5
	time (s)	0.02	0.38	0.38	7.97	16.36	18.78	0.36	2.98	238.82	605.81
`LS`	# states	182	900	943	4180	6368	TO	6657	37377	MO	MO
	prob.	0.1501	0.2302	0.2501	0.3501	0.4		0.5	0.5		
	time (s)	0.14	1.11	6.1	619.06	2455.46		8	54.58		
k`SP`	# states	1071	TO	TO	TO	TO	TO	6827	37601	140034	369444
	prob.	0.15						0.5	0.5	0.5	0.5
	time (s)	6.58						1.93	0.13	0.69	1.49
DiPro	# states	938	2901	3227	9005	ERR	ERR	13311	74751	MO	MO
	prob.	0.1675	0.2334	0.254	0.3533			0.5	0.5		
	time (s)	2.02	7.06	7.87	44.34			1210	7114		

Fig. 3. Results for crowds and contract signing (TO $> 2h$)

best, while `LS` computes in general smaller counterexamples. k`SP` is the fastest method for contract signing, however, the representation of the result consists of a huge number of paths instead of a small subsystem of the input DTMC.

4 Conclusion and Future Work

We presented version 1.0 of our tool `COMICS` which generates abstract, hierarchically refinable counterexamples for DTMCs. In the future, we will integrate the computation of *minimal* critical subsystems [6] and the adaption of our approaches to *symbolic data structures*. We are also working on an *incremental version of the Dijkstra algorithm* for path search and on *compositional counterexamples*.

References

1. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
2. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. 68(2), 90–104 (2011)
3. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
4. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. on Software Engineering 35(2), 241–257 (2009)
5. Jansen, N., Ábrahám, E., Katelaan, J., Wimmer, R., Katoen, J.-P., Becker, B.: Hierarchical Counterexamples for Discrete-Time Markov Chains. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 443–452. Springer, Heidelberg (2011)
6. Wimmer, R., Jansen, N., Ábrahám, E., Becker, B., Katoen, J.-P.: Minimal Critical Subsystems for Discrete-Time Markov Models. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 299–314. Springer, Heidelberg (2012)

7. Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J.-P., Becker, B.: DTMC model checking by SCC reduction. In: Proc. of QEST, pp. 37–46. IEEE CS (2010)
8. Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: DiPro - A Tool for Probabilistic Counterexample Generation. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 183–187. Springer, Heidelberg (2011)
9. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for web transactions. ACM Trans. on Information and System Security 1(1), 66–92 (1998)
10. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. Journal of Computer Security 14(6), 561–589 (2006)

Computing Minimal Separating DFAs and Regular Invariants Using SAT and SMT Solvers

Daniel Neider

Lehrstuhl für Informatik 7, RWTH Aachen University, Germany

Abstract. We develop a generic technique to compute minimal separating DFAs (deterministic finite automata) and regular invariants. Our technique works by expressing the desired properties of a solution in terms of logical formulae and using SAT or SMT solvers to find solutions. We apply our technique to three concrete problems: computing minimal separating DFAs (e.g., used in compositional verification), regular model checking, and synthesizing loop invariants of integer programs that are expressible in Presburger arithmetic.

1 Introduction

In this paper, we present a generic technique based on SAT or SMT solvers to compute minimal separating DFAs (deterministic finite automata) and regular invariants.

A separating DFA is a DFA that separates two disjoint regular languages, i.e., whose accepted language contains the first language and is disjoint to the second. A minimal separating DFA is a separating DFA that has the least number of states among all separating DFAs. Many well known problems can be reduced to the problem of finding a minimal separating DFA. For instance, in [5] minimal separating DFAs are used in the context of compositional verification. Two other examples are the minimization of incomplete specified DFAs [13] and the computation of minimal DFAs that are consistent with a set of positively and negatively classified samples [3]. Note that finding a minimal separating DFA is in fact a non-trivial task since the corresponding decision problem "Given two disjoint regular languages. Does a separating DFA with $k \in \mathbb{N}$ states exist?" is NP-hard (cf. [14] where this is shown for two disjoint finite languages).

Regular invariants are defined in terms of binary relations T specified by finite-state transducers. Intuitively, a regular invariant is a regular language L (alternatively a DFA) that is invariant (or closed) under T, i.e., for all $(u, v) \in T$ with $u \in L$ also $v \in L$ is satisfied. Regular invariants occur, e.g., in regular model checking [4]. There, one way to prove a program correct is to find a regular invariant with respect to the transitions of the program that overapproximates the set of initial configurations and has an empty intersection with the set of bad configurations. In this sense, regular invariants extend the concept of separating DFAs.

The main contribution of this work is a technique to compute minimal separating DFAs and regular invariants. Our technique works as follows. It takes a

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 354–369, 2012.

problem instance, translates it into a logical formula of size polynomial in the input, and uses a SAT or SMT solver to find a solution. More precisely, our technique creates a logical formula φ_n that depends on the regular languages given as input as well as a natural number $n \geq 1$. Moreover, φ_n has the following two properties. First, φ_n is satisfiable if and only if there exists a DFA with n states that possesses the desired properties. Second, a model of φ_n allows to derive a DFA with these properties. Starting with $n = 1$ we increase n until φ_n becomes satisfiable. This guarantees to find a suitable DFA (if one exists).

We implement the formula φ_n in two logics: propositional logic over Boolean variables and the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ where we allow constants $c \in \mathbb{N}$ and uninterpreted functions. We choose these logics because highly-optimized off-the-shelf solvers are available in both cases, e.g., the MiniSat SAT solver in the first case and Microsoft's Z3 SMT solver in the latter. Since the decision problem variant of computing minimal separating DFAs and regular invariants is NP-hard, using SAT and SMT solvers is a natural approach.

We apply our technique to three concrete problems: computing minimal separating DFAs (Section 3), regular model checking (Section 4.1), and synthesizing loop invariants of integer programs that are expressible in Presburger arithmetic (Section 4.2). The latter two can be subsumed under the topic regular invariants. Each mentioned section begins with an introduction to the setting, then describes the application of our technique, and finally concludes with related work and experiments.

2 Preliminaries

Words, Languages and Finite automata. An *alphabet* Σ is a finite set of *symbols*. A *word* $w = a_1 \dots a_n$ is a finite, possibly empty, sequence of symbols $a_i \in \Sigma$. If a sequence is empty, we call it the *empty word*, denoted by ε. The *concatenation* of two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ is the word $uv = a_1 \dots a_n b_1 \dots b_m$. Σ^* is the set of all finite words over Σ. A subset $L \subseteq \Sigma^*$ is called a *language*.

A *(nondeterministic) finite automaton (NFA)* is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ where Q is a finite, non-empty set of states, Σ is the input alphabet, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. The size of an NFA is the number $|Q|$ of its states. For $w = a_1 \dots a_n \in \Sigma^*$, a *run* of $\mathcal{A}$ on w from a state q is a sequence $q_1, \dots, q_{n+1}$ such that $q_1 = q$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for $1 \leq i \leq n$; we also write $\mathcal{A} \colon q \xrightarrow{w} q_{n+1}$ for short. A word w is *accepted* by $\mathcal{A}$ if $\mathcal{A} \colon q_0 \xrightarrow{w} q$ with $q \in F$. The *language accepted by* $\mathcal{A}$ is the set $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \colon q_0 \xrightarrow{w} q, q \in F\}$. A language $L \subseteq \Sigma^*$ is called *regular* if there exists an NFA $\mathcal{A}$ such that $L = L(\mathcal{A})$.

A *deterministic finite automaton (DFA)* is an NFA where for every $p \in Q$ and $a \in \Sigma$ a unique $q \in Q$ exists such that $(p, a, q) \in \Delta$. In this case, Δ defines a function $\delta \colon Q \times \Sigma \to Q$, and we replace Δ by δ in the definition of a DFA.

Logics, Formulae, and Satisfiability. In this paper, we consider three logics: propositional logic over Boolean variables, quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with uninterpreted functions, and Presburger arithmetic. We use the first two logics to implement our technique in Section 3.1 and Section 3.2, respectively. The third logic is considered in Section 4.2 in the context of synthesizing loop invariants of integer programs.

In propositional logic, a *formula* φ—often called SAT formula—is built from Boolean variables $x_1, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$ with their usual meaning. The set $\mathrm{Var}(\varphi)$ is the set of variables occurring in φ; if the variables occurring in φ are of special interest, we also write $\varphi(x_1, \ldots, x_n)$. A *model of* φ is a mapping $\mathfrak{M} \colon \mathrm{Var}(\varphi) \rightarrow \{0, 1\}$ (0 representing `false`, 1 representing `true`) such that φ evaluates to `true` if all variables x_i in φ are substituted by $\mathfrak{M}(x_i)$; we then write $\mathfrak{M} \models \varphi$. A formula φ is said to be in *conjunctive normal form* if $\varphi = \bigwedge_{i=1}^{r} c_i$ where $c_i = \bigvee_{j=1}^{s_i} l_{ij}$ is a *clause* and l_{ij} is either a Boolean variable or its negation. Satisfiability of a formula can be decided by a SAT solver, and if the formula is satisfiable, then the solver can provide a model. Note that most SAT solver require the input to be in conjunctive normal form.

The second logic we consider is the usual quantifier-free logic over the structure $(\mathbb{N}, <, =)$. We additionally allow constants $c \in \mathbb{N}$ and enrich this logic with uninterpreted functions. Each such function is of the form $f \colon \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N}$ and represents an unknown function of which only the name f and its signature is known. In our case, it turns out that functions of the form $\mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \{\mathtt{true}, \mathtt{false}\}$, which can easily be simulated, are also helpful and used later on. The concept of a model $\mathfrak{M}$ is lifted to this kind of formulae in the natural way. If f is an uninterpreted function occurring in a formula φ and $\mathfrak{M} \models \varphi$, then we write $f_{\mathfrak{M}}$ to denote the function defined by the model $\mathfrak{M}$. Satisfiability of this kind of formulae can be decided with today's *satisfiability modulo theory (SMT)* solvers, and, hence, we bravely refer to them as SMT formulae.

Presburger arithmetic is the first order logic over the structure $(\mathbb{Z}, +, \leq, 0)$. For convenience, we add syntactic sugar to this logic by also allowing to use the relations $<$, $>$ and arbitrary constants $c \in \mathbb{Z}$. Presburger formulae have the nice property that they define regular languages. More formally, each Presburger formula $\varphi(x_1, \ldots, x_n)$ can be translated into a DFA $\mathcal{A}^{\varphi}$ working over the alphabet $\{0, 1\}^n$ (see, e.g., [11]). Intuitively, the automaton $\mathcal{A}^{\varphi}$ accepts exactly the set of binary strings encoding integer values that satisfy φ. Note, however, that not every regular language represents a Presburger formula. Nonetheless, in [11] the problem to decide whether a regular language represents a Presburger formula is shown to be decidable in polynomial time. Moreover, in this case, a formula defining the language can be computed in polynomial time.

3 Minimal Separating DFAs

The first application of our technique is to compute minimal separating DFAs. To this end, let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages over a fixed alphabet Σ. A DFA $\mathcal{A}$ with $L_1 \subseteq L(\mathcal{A})$ and $L_2 \cap L(\mathcal{A}) = \emptyset$ is said to be a *separating DFA* since it separates both languages. A minimal separating DFA is a separating DFA of minimal size. Note that minimal separating DFAs are not unique for fixed L_1, L_2 since their behavior on words not belonging to L_1 or

L_2 is unspecified. In fact, computing a minimal separating DFA for two disjoint regular languages is computationally hard (cf. [14]).

Minimal separating DFAs are helpful in various contexts. For instance, a direct application to compositional verification is described in [5]. Moreover, the well-known task of minimizing incompletely specified DFAs [13] can also be phrased in terms of minimal separating DFAs. To this end, an incompletely specified DFA is defined as a DFA $\mathcal{B} = (Q, \Sigma, q_0, \delta, \mathrm{Acc}, \mathrm{Rej}, \mathrm{Unspec})$ whose states are partitioned into accepting, rejecting, and unspecified states. The task of minimizing an incompletely specified DFA now is to compute a DFA of minimal size that accepts all words that lead to an accepting state in $\mathcal{B}$ and rejects all words that lead to a rejecting state in $\mathcal{B}$. This, however, is the same as computing a minimal separating DFA for the languages $L_1 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Acc}\}$ and $L_2 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Rej}\}$. Finally, let us mention Biermann's task of computing a minimal DFA that agrees with a finite set of positively and negative classified samples [3]. Here, L_1 and L_2 are both finite languages.

Let us now describe our technique to compute minimal separating DFAs. For the rest of this section, fix an alphabet Σ, and let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages. Let us assume that L_1 and L_2 are given by two NFAs $\mathcal{A}_1 = (Q_1, \Sigma, q_0^1, \Delta_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, q_0^2, \Delta_2, F_2)$, respectively.

We search for a separating DFA by creating a formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ that depends on the NFAs $\mathcal{A}_1$, $\mathcal{A}_2$, a natural number $n \geq 1$, and has the following properties:

- $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable if and only if there exists a separating DFA $\mathcal{A}$ with n states, i.e., $L_1 \subseteq L(\mathcal{A})$ and $L_2 \cap L(\mathcal{A}) = \emptyset$.
- If $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$, then $\mathfrak{M}$ can be used to derive a DFA separating L_1 and L_2.

Clearly, if $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable for a given value of n, then we have found a separating DFA. However, if the formula is unsatisfiable, then the parameter n has been chosen too small and we need to increment it. Since a separating DFA always exist, e.g., the DFA accepting exactly the language L_1, this process terminates eventually. The algorithm in pseudo code is shown in Algorithm 1. Note that an even faster approach is to use a binary search to find the minimal value of n rather than incrementing n by one in each iteration. Providing an intelligent starting value of n is difficult, but the size of $\mathcal{A}_1$ is a natural choice.

We can now state the main result of this section.

Theorem 1. *Let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages. If a minimal separating DFA has k states, then Algorithm 1 terminates after k iterations and returns a minimal separating DFA.*

Proof. Let us first state that a DFA separating L_1 and L_2 always exist, e.g., the DFA accepting exactly the language L_1. Then, the proof of Theorem 1 is a straight-forward application of the fact that $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ has indeed the desired properties (cf. Lemma 2 on page 360): if a minimal separating DFA has k states, then $\varphi_l^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable for all $l \geq k$, and we find the minimal value k since we increase n by one in every iteration. □

Algorithm 1: Computing minimal separating DFAs.

```
Input: two disjoint regular languages $L_1, L_2 \subseteq \Sigma^*$ given as NFAs $\mathcal{A}_1, \mathcal{A}_2$.
$n := 0$;
repeat
    $n := n + 1$;
    Construct and solve $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$;
until $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is satisfiable;
Construct and return $\mathcal{A}_{\mathfrak{M}}$;
```

In the following two subsections, we utilize two different logics to implement the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$. The first is the propositional logic over Boolean variables, which we consider in Section 3.1. In Section 3.2, we consider the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with uninterpreted functions. However, before we continue to implement $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$, let us briefly discuss related work.

Related work. Chen et al. [5] propose an algorithm to compute minimal separating DFAs, which is based on algorithmic learning. Internally, they use an algorithm to minimize incompletely specified DFAs as a subroutine. In fact, every algorithm that minimizes incompletely specified DFAs (e.g., [13]) can be used to compute minimal separating DFAs. Note, however, that these algorithms typically require deterministic automata as input whereas we allow NFAs.

With the emerge of fast and efficient SAT and SMT solvers and since the problem itself is computationally hard (cf. [14]), it seems justified to promote a SAT and SMT based approach in this context. Due to the lack of publicly available tools we did not yet compare our technique to other approaches.

3.1 Finding Separating DFAs Using SAT Formulae

In the following, we describe a formula in the propositional logic over Boolean variables that, if satisfiable, encodes a DFA with a fixed number $n \geq 1$ of states that separates L_1 and L_2. The automaton will have the state set $Q = \{q_0, \ldots, q_{n-1}\}$, and the initial state $q_0 \in Q$. To encode a DFA, we make the following simple observation: if the set of states and the initial state are fixed (e.g., as above), then each DFA is completely defined by its transition function and final states. Our encoding exploits this fact and uses Boolean variables $d_{p,a,q}$ and f_q with $p, q \in Q$ and $a \in \Sigma$. If $d_{p,a,q}$ is assigned to `true`, it means that $\delta(p, a) = q$. Similarly, if f_q is assigned to `true`, then state q is a final state.

To make sure that the variables $d_{p,a,q}$ indeed encode a deterministic transition function, we impose the following constraints.

$$\neg d_{p,a,q} \vee \neg d_{p,a,q'} \qquad p, q, q' \in Q,\ q \neq q',\ a \in \Sigma \tag{1}$$

$$\bigvee_{q \in Q} d_{p,a,q} \qquad p \in Q,\ a \in \Sigma \tag{2}$$

Constraints of type (1) make sure that the variables $d_{p,a,q}$ in fact encode a well-defined function, i.e., that for every state $p \in Q$ and input $a \in \Sigma$ there is at most one $q \in Q$ such that $d_{p,a,q}$ is set to `true`. Constraints of type (2), on the other hand, ascertain that the transition function is total. The latter constraints are not needed in general and might be skipped.

Let $\varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f})$ be the conjunction of the constraints of type (1) and (2) where $\overline{d}$ is the vector of all variables $d_{p,a,q}$ and $\overline{f}$ the vector of all variables f_q. From a model $\mathfrak{M}$ of the formula $\varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f})$ it is straight-forward to derive a DFA $\mathcal{A}_{\mathfrak{M}} = (\{q_0, \ldots, q_{n-1}\}, \Sigma, q_0, \delta, F)$: $\delta(p, a) = q$ for the unique q such that $\mathfrak{M}(d_{p,a,q}) = 1$, and $F = \{q \mid \mathfrak{M}(f_q) = 1\}$.

Until now, $L(\mathcal{A}_{\mathfrak{M}})$ is unspecified. Thus, to guarantee that $\mathcal{A}_{\mathfrak{M}}$ is a separating DFA, we need to impose further constraints on the formula φ_n^{DEA}. We do so by introducing two auxiliary formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ that express the following:

- If $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq}$, then $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.
- If $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}} \wedge \varphi_n^{\neg \mathcal{A}_2}$, then $L(\mathcal{A}_2) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.

Clearly, if $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq} \wedge \varphi_n^{\neg \mathcal{A}_2}$, then $\mathcal{A}_{\mathfrak{M}}$ is DFA separating L_1 and L_2.

The idea behind the formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ is to impose restrictions on the variables $d_{p,a,q}$ and f_q, which determine the automaton $\mathcal{A}_{\mathfrak{M}}$. Keeping this in mind, it is easier to explain the formulae by directly referring to the automaton $\mathcal{A}_{\mathfrak{M}}$ rather than to describe their influence on the variables $d_{p,a,q}$ and f_q. Note, however, that we thereby implicitly assume that the corresponding formula is satisfiable and that $\mathfrak{M}$ is a model.

The idea of the formula $\varphi_n^{\mathcal{A}_1 \subseteq}$ is to keep track of the parallel behavior of the automaton $\mathcal{A}_1$ and $\mathcal{A}_{\mathfrak{M}}$. For that, we use new auxiliary variables $x_{q,q'}$ where $q \in Q$ and $q' \in Q_1$. Intuitively, we want to establish for all models $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}} \wedge \varphi_n^{\mathcal{A}_1 \subseteq}$ that if some input $w \in \Sigma^*$ induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$, then $x_{q,q'}$ is assigned to `true`. This is done by the following constraints.

$$x_{q_0, q_0^1} \tag{3}$$

$$(x_{p,p'} \wedge d_{p,a,q}) \rightarrow x_{q,q'} \qquad p, q \in Q,\ p', q' \in Q_1,\ a \in \Sigma,\ (p', a, q') \in \Delta_1 \tag{4}$$

The constraint (3) requires the variable x_{q_0, q_0^1} to be assigned to `true` because ε induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{\varepsilon} q_0^1$. The constraints of type (4) describe how to propagate the values of the variables $x_{q,q'}$ step-by-step: if there exists a word u such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} p$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{u} p'$, i.e., $x_{p,p'}$ is assigned to `true`, and there are transitions $\delta(p, a) = q$ in $\mathcal{A}_{\mathfrak{M}}$ and $(p', a, q') \in \Delta_1$, then $x_{q,q'}$ has to be assigned to `true`, too.

Using the variables $x_{q,q'}$ we can now express that $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$ as done below by the constraints of type (5). These constraints state that if a word leads to an accepting state in $\mathcal{A}_1$, then it also leads to an accepting state in $\mathcal{A}_{\mathfrak{M}}$.

$$x_{q,q'} \rightarrow f_q \qquad q \in Q,\ q' \in F_1 \tag{5}$$

Let $\varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$ be the conjunction of the constraints of type (3) to (5) where $\overline{d}$, $\overline{f}$ are as described above, and $\overline{x}$ is the vector of new variables $x_{q,q'}$ with $q \in Q$ and $q' \in Q_1$. Then, we obtain the following lemma.

Lemma 1. *If $\mathfrak{M} \models \varphi_n^{DEA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$, then $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.*

Proof. The proof follows the same line of arguments that we used in our intuitive explanation above. Let $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x})$.

First, let us show by induction that if there exists a word w such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$, then $x_{q,q'}$ is assigned to `true`. For $w = \varepsilon$ the claim holds since $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{\varepsilon} q_0^1$, and constraint (3) forces x_{q_0,q_0^1} to be set to `true`. For $w = ua$ assume that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} p \xrightarrow{a} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{u} p' \xrightarrow{a} q'$. In particular, this means that there are transitions $\delta(p, a) = q$ in $\mathcal{A}_{\mathfrak{M}}$, i.e. $\mathfrak{M}(d_{p,a,q}) = 1$, and $(p', a, q') \in \Delta_1$. Moreover, the induction hypothesis yields that $x_{p,p'}$ is assigned to `true`. Then, however, the constraints of type (4) force $x_{q,q'}$ to be set to `true`, too.

Now, let $w \in L(\mathcal{A}_1)$. Then, there exists $q \in Q$ and $q' \in F_1$ such that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{w} q$ and $\mathcal{A}_1 \colon q_0^1 \xrightarrow{w} q'$. Hence, $\mathfrak{M}(x_{q,q'}) = 1$. In this case, the constraints of type (5) assure that $\mathfrak{M}(f_q) = 1$, and, hence, $w \in L(\mathcal{A}_{\mathfrak{M}})$. □

The formula $\varphi_n^{\neg \mathcal{A}_2}$ works analogous to $\varphi_n^{\mathcal{A}_1 \subseteq}$. We introduce new auxiliary variables $y_{q,q'}$ with $q \in Q, q' \in Q_2$ and modify the constraints of type (3) and (4) accordingly. Note that we need to change the constraints of type (5) slightly such that they now express that whenever a word is accepted by $\mathcal{A}_2$, then it is rejected by $\mathcal{A}_{\mathfrak{M}}$. The constraints on the variables $y_{q,q'}$ are listed below.

$$y_{q_0,q_0^2} \tag{6}$$

$$(y_{p,p'} \wedge d_{p,a,q}) \rightarrow y_{q,q'} \qquad p, q \in Q,\ p', q' \in Q_2,\ a \in \Sigma,\ (p', a, q') \in \Delta_2 \tag{7}$$

$$y_{q,q'} \rightarrow \neg f_q \qquad q \in Q,\ q' \in F_2 \tag{8}$$

Let $\varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y})$ be the conjunction of the constraints of type (6) to (8) where $\overline{d}$, $\overline{f}$ are as described above, and $\overline{y}$ is the vector of new variables $y_{q,q'}$ with $q \in Q$ and $q' \in Q_2$. Analogous to Lemma 1 we obtain $L(\mathcal{A}_2) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$ if $\mathfrak{M} \models \varphi_n^{\mathrm{DEA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y})$.

We can now combine all subformulae and obtain the following result.

Lemma 2. *Let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages, $n \in \mathbb{N}$ and*

$$\varphi_n^{\mathcal{A}_1, \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{x}, \overline{y}) = \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}_1 \subseteq}(\overline{d}, \overline{f}, \overline{x}) \wedge \varphi_n^{\neg \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{y}).$$

Then, $\varphi_n^{\mathcal{A}_1, \mathcal{A}_2}(\overline{d}, \overline{f}, \overline{x}, \overline{y})$ is satisfiable if and only if there exists a DFA with n states that separates L_1 and L_2.

Proof. The direction from left to right is a straight-forward application of the properties of the formulae $\varphi_n^{\mathcal{A}_1 \subseteq}$ and $\varphi_n^{\neg \mathcal{A}_2}$ (cf. Lemma 1). Let $\varphi_n^{\mathcal{A}_1, \mathcal{A}_2}$ be satisfiable and $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1, \mathcal{A}_2}$. Then, $\mathcal{A}_{\mathfrak{M}}$ is an automaton with n states that separates L_1 and L_2.

For the reverse direction, let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA with n states that separates L_1 and L_2. From $\mathcal{A}$ we can derive a model $\mathfrak{M}$ as follows: we set $\mathfrak{M}(d_{p,a,q}) = 1$ if and only if $\delta(p, a) = q$, and $\mathfrak{M}(f_q) = 1$ if and only if $q \in F$. Values for the variables $x_{q,q'}$ and $y_{q,q'}$ can be derived by looking at the states reachable in the products of $\mathcal{A}$ and $\mathcal{A}_1$ as well as $\mathcal{A}$ and $\mathcal{A}_2$, respectively. □

Finally, let us note that $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ can easily be turned into conjunctive normal form by applying that $A \rightarrow B$ is logically equivalent to $\neg A \vee B$ and De Morgan's law. In conjunctive normal form, $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ ranges over $\mathcal{O}(n^2|\Sigma| + n(|Q_1| + |Q_2|))$ variables and comprises $\mathcal{O}(n^2(|\Delta_1| + |\Delta_2|) + n(|F_1| + |F_2|))$ clauses.

3.2 Finding Separating DFAs Using SMT Formulae

In this section, we implement the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ in the quantifier-free logic over the structure $(\mathbb{N}, <, =)$ with constants $c \in \mathbb{N}$ and uninterpreted functions. To this end, let us assume that all automata have a special form: the set of states is $Q = \{0, 1, \ldots, n-1\}$, the initial state is 0, and the alphabet is $\Sigma = \{0, 1, \ldots, m-1\}$. We can easily achieve this form by renaming the states of the automaton and symbols of the alphabet.

The formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$ is based on the very same idea as in Section 3.1, but uses two functions $d \colon \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and $f \colon \mathbb{N} \rightarrow \{0, 1\}$ to encode the automaton we are searching for; the function d encodes the transitions whereas f encodes the set of final states. By means of these functions, it is easy to derive an automaton $\mathcal{A}_{\mathfrak{M}} = (\{0, \ldots, n-1\}, \Sigma, 0, \delta, F)$ from a model $\mathfrak{M} \models \varphi_n^{\mathcal{A}_1,\mathcal{A}_2}$: we define $\delta(i, a) = d_{\mathfrak{M}}(i, a)$ for $i \in Q$, $a \in \Sigma$ and $i \in F \Leftrightarrow f_{\mathfrak{M}}(i) = 1$. To ensure that δ is well-defined, we will make sure that $d(i, a) < n$ is satisfied for $0 \leq i < n$ and $0 \leq a < m$.

To express that $\mathcal{A}_{\mathfrak{M}}$ separates L_1 and L_2, we additionally utilize two auxiliary functions $x \colon \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}$ and $y \colon \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}$, which have the same meaning as the equally named variables in Section 3.1. By means of the functions d, f, x, y, we can now rephrase the constraints of Section 3.1 as follows.

$$\begin{aligned}
d(i,a) &< n && i \in Q,\ a \in \Sigma \\
x(0,0) &\wedge y(0,0) && \\
x(i,i') &\rightarrow x(d(i,a),j') && i \in Q,\ i',j' \in Q_1,\ a \in \Sigma,\ (i',a,j') \in \Delta_1 \\
x(i,i') &\rightarrow f(i) && i \in Q,\ i' \in F_1 \\
y(i,i') &\rightarrow y(d(i,a),j') && i \in Q,\ i',j' \in Q_2,\ a \in \Sigma,\ (i',a,j') \in \Delta_2 \\
y(i,i') &\rightarrow \neg f(i) && i \in Q,\ i' \in F_2
\end{aligned}$$

Let $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ be the conjunction of these constraints. Then, we obtain the following lemma. The proof is analogous to the proof of Lemma 2 and, therefore, skipped.

Lemma 3. *Let $L_1, L_2 \subseteq \Sigma^*$ be two disjoint regular languages and $n \in \mathbb{N}$. Then, $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ is satisfiable if and only if there exists a DFA $\mathcal{A}$ with n that separates L_1 and L_2.*

Finally, let us briefly remark that the formula $\varphi_n^{\mathcal{A}_1,\mathcal{A}_2}(d, f, x, y)$ comprises $\mathcal{O}(n|\Sigma| + n(|\Delta_1| + |\Delta_2|) + n(|F_1| + |F_2|))$ constraints.

4 Regular Invariants

Let us now extend the technique of the previous section to the task of synthesizing regular invariants. Regular invariants are defined in terms of finite-state transducers. A *finite-state transducer* is basically an NFA (or DFA) $\mathcal{A}$ that works over the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ and reads pairs (u, v) of words. We write $\mathcal{A}\colon p \xrightarrow{(u,v)} q$ if there exists a sequence of transitions in $\mathcal{A}$ that leads from state p to q and where the labels along the transitions yield the pair (u, v). Rather than a language, a finite-state transducer accepts (or defines) a relation $R(\mathcal{A}) \subseteq \Sigma^* \times \Sigma^*$ where $R(\mathcal{A}) = \{(u, v) \mid \mathcal{A}\colon q_0 \xrightarrow{(u,v)} q, q \in F\}$. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *rational* if there exists a finite-state transducer $\mathcal{A}$ such that $R = R(\mathcal{A})$.

For a relation $R \subseteq \Sigma^* \times \Sigma^*$ we denote the reflexive and transitive closure by R^*. For a language $L \subseteq \Sigma^*$ and a relation $R \subseteq \Sigma^* \times \Sigma^*$ let $R(L) = \{v \in \Sigma^* \mid \exists u \in A\colon (u, v) \in R\}$ be the image of L under R. Finally, we call a (regular) language $L \subseteq \Sigma^*$ a *(regular) invariant* if $R(L) \subseteq L$.

Synthesizing regular invariants (that of course possess additional properties) is an important task in various applications. In the next two subsections we show how our technique can be applied to two prominent settings: regular model checking (in Section 4.1) and synthesis of loop invariants (in Section 4.2).

4.1 Regular Model Checking

In the regular model checking framework [4], configurations of a program (or system) are modeled as finite words over a fixed alphabet Σ. The program itself is a tuple $\mathcal{P} = (I, T)$ consisting of a regular set $I \subseteq \Sigma^*$ of *initial configurations* and a rational relation $T \subseteq \Sigma^* \times \Sigma^*$ defining the *transitions*, i.e., the successor relation on the configurations. Regular model checking—more precise, the regular model checking problem—is the decision problem

"Given a program $\mathcal{P} = (I, T)$ and a regular set $B \subseteq \Sigma^*$ of *bad configurations.*
Does $T^*(I) \cap B = \emptyset$ hold?".

In other words, the model checking problem asks whether there is a path in $\mathcal{P}$ that leads from an initial configuration into the set of bad configurations. In this case, the program is erroneous. Note that the model checking problem is in general undecidable (cf. [4]).

Many tools for regular model checking such as T(O)RMC [10], which is based on LASH, or LEVER [16] try to compute a regular set that overapproximates the reachable configurations, is an invariant, and has an empty intersection with the set of bad configurations. More formally, these tools search for a regular set $P \subseteq \Sigma^*$ with $I \subseteq P$, $B \cap P = \emptyset$, and $T(P) \subseteq P$. We call such a set a *proof* that the program $\mathcal{P}$ is correct (with respect to B) since it proves that there does not exist a path from any initial configuration to a bad one. In other words, a proof is a regular invariant with the additional properties $I \subseteq P$ and $B \cap P = \emptyset$.

We can extend our technique from the previous section to compute proofs. To this end, let us assume that I and B are given as two NFAs $\mathcal{A}^I$ and $\mathcal{A}^B$ and that T is given as a finite-state transducer $\mathcal{A}^T = (Q^T, (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}), q_0^T, \Delta^T, F^T)$.

Analogous to Section 3, we create a formula $\psi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{A}^T}$ that is satisfiable if and only if there exists a DFA $\mathcal{A}$ with n states such that $L(\mathcal{A})$ is a proof. In the following, we show how this is done for SAT formulae. The adaptation for SMT formulae can be done in a straight-forward manner.

Following the definition of a proof from above, we define $\psi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{A}^T}$ as

$$\psi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{A}^T} := \varphi_n^{\text{DEA}} \wedge \varphi_n^{\mathcal{A}^I \subseteq} \wedge \varphi_n^{\neg \mathcal{A}^B} \wedge \varphi_n^{\text{inv } \mathcal{A}^T}$$

where the subformulae $\varphi_n^{\mathcal{A}^I \subseteq}$ and $\varphi_n^{\neg \mathcal{A}^B}$ are as in Section 3.1, and $\varphi_n^{\text{inv } \mathcal{A}^T}$ ensures that if $\mathfrak{M} \models \varphi_n^{\text{DEA}} \wedge \varphi_n^{\text{inv } \mathcal{A}^T}$, then $T(L(\mathcal{A}_\mathfrak{M})) \subseteq L(\mathcal{A}_\mathfrak{M})$.

In other words, the formula $\varphi_n^{\text{inv } \mathcal{A}^T}$ needs to make sure that $L(\mathcal{A}_\mathfrak{M})$ is an invariant. To this end, we consider the parallel behavior of $\mathcal{A}_\mathfrak{M}$ and $\mathcal{A}^T$ analogous to Section 3.1. This time, however, the situation is more involved since $\mathcal{A}^T$ works on pairs (u, v) of words.

We need to establish that if (u, v) is accepted by $\mathcal{A}^T$ and $u \in L(\mathcal{A}_\mathfrak{M})$, then $v \in L(\mathcal{A}_\mathfrak{M})$ holds, too. In other words, this means that if $\mathcal{A}^T$ reaches a final state after reading (u, v) and $\mathcal{A}_\mathfrak{M}$ reaches a final state after reading u, then $\mathcal{A}_\mathfrak{M}$ must also reach a final state after reading v. This condition can be expressed using auxiliary variables $z_{q,q',q''}$ with $q, q'' \in Q$ and $q' \in Q^T$. Their meaning is that if $\mathcal{A}^T : q_0^T \xrightarrow{(u,v)} q'$, $\mathcal{A}_\mathfrak{M} : q_0 \xrightarrow{u} q$, and $\mathcal{A}_\mathfrak{M} : q_0 \xrightarrow{v} q''$, then $z_{q,q',q''}$ is set to `true`. The following constraints define the formula $\varphi_n^{\text{inv } \mathcal{A}^T}$.

$$z_{q_0,q_0^B,q_0} \tag{9}$$

$$(z_{p,p',p''} \wedge d_{p,a,q} \wedge d_{p'',b,q''}) \rightarrow z_{q,q',q''} \qquad p, p'', q, q'' \in Q,\ a, b \in \Sigma,\ p', q' \in Q^T,\ (p', (a, b), q') \in \Delta^T \tag{10}$$

$$(z_{p,p',p''} \wedge d_{p,a,q}) \rightarrow z_{q,q',p''} \qquad p, p'', q \in Q,\ a \in \Sigma,\ p', q' \in Q^T,\ (p', (a, \varepsilon), q') \in \Delta^T \tag{11}$$

$$(z_{p,p',p''} \wedge d_{p'',b,q''}) \rightarrow z_{p,q',q''} \qquad p, p'', q'' \in Q,\ b \in \Sigma,\ p', q' \in Q^T,\ (p', (\varepsilon, b), q') \in \Delta^T \tag{12}$$

$$(z_{q,q',q''} \wedge f_q) \rightarrow f_{q''} \qquad q, q'' \in Q,\ q' \in F^T \tag{13}$$

Let $\varphi_n^{\text{inv } \mathcal{A}^T}(\overline{d}, \overline{f}, \overline{z})$ be the conjunction of the constraints of type (9) to (13) where $\overline{d}$, $\overline{f}$ are as described in Section 3.1, and $\overline{z}$ is the vector of new variables $z_{q,q',q''}$. Then, we obtain $T(L(\mathcal{A}_\mathfrak{M})) \subseteq L(\mathcal{A}_\mathfrak{M})$ in analogy to Lemma 1.

We can now combine all subformulae and obtain an algorithm to compute proofs in regular model checking. The algorithm is shown in Algorithm 2. Let us remark that $\psi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{A}^T}$ ranges over $\mathcal{O}(n^2|\Sigma| + n(|Q_I| + |Q_B| + n|Q_T|))$ variables and comprises $\mathcal{O}(n^2(|\Delta^I| + |\Delta^B| + n^2|\Delta^T|) + n(|F^I| + |F^B| + n|F^T|))$ clauses.

Using Algorithm 2 we can establish Theorem 2. The proof of Theorem 2 is done analogously to Theorem 1 and, hence, skipped.

Algorithm 2: Computing proofs using SAT or SMT solvers.

```
Input: a program $\mathcal{P} = (I, T)$ and a regular set $B$ of bad configurations.
$n := 0$;
repeat
    $n := n + 1$;
    Construct and solve $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$;
until $\psi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ is satisfiable;
Construct and return $\mathcal{A}_{\mathfrak{M}}$;
```

Theorem 2. *Let $\mathcal{P} = (I, T)$ be a program and B a regular set of bad configurations. If a proof that $\mathcal{P}$ is correct exists, then Algorithm 2 terminates and returns a proof (of minimal size).*

Note that we cannot guarantee that Algorithm 2 terminates since there might not exist a proof at all. Hence, Algorithm 2 is necessarily a semi algorithm (the regular model checking problem is undecidable). This, however, is of course also true for all other algorithms.

Related Work and Experiments. There are various tools for regular model checking, which have been successfully applied in practice. Prominent examples are T(O)RMC [10] (based on LASH), LEVER [16], and FAST [1].

T(O)RMC computes overapproximations of the set of reachable configurations using various extrapolation techniques. However, T(O)RMC does not consider the set of bad configurations directly and, hence, cannot guarantee to find an overapproximation having an empty intersection with the set of bad configurations. In such a situation, T(O)RMC has to be restarted with different settings until a proof has been found. This is a resource consuming process, and choosing "good" settings requires in-depth knowledge about both T(O)RMC and the particular application domain. By contrast, our technique is generic and does not need any particular knowledge about the domain. Moreover, T(O)RMC requires DFAs as input whereas our technique can also handle NFAs. Thus, the input for our technique can be exponentially smaller than an equivalent input for T(O)RMC.

LEVER computes proofs using automata learning techniques. Thereby, it does not learn a proof directly, but a set of configurations that are enriched with additional information. This is necessary to be able to answer queries posed by the learning algorithm. However, the problem with this approach is that even if a proof exists it can no longer be guaranteed that these augmented sets are regular. In this case, LEVER cannot find a proof and might not terminate. This is a drawback compared to our technique, which always finds a proof if one exists. Note, however, that learning based techniques treat the input automata as black boxes whereas we assume them to be known as white boxes.

Fast computes the exact set of reachable configurations by means of acceleration techniques. Also here, this set is not necessarily regular (in general not even computable) although a proof might exist. In such situations, Fast fails and does not terminate.

At this point let us note that our approach is different from bounded model checking [2] although it seems quite similar. The idea of bounded model checking is to search for a bad execution whose length is bounded by some integer k. The value of k is increased until a bug is found or some a priori defined bound is reached. We, on the other hand, always consider the whole program, but parametrize the size of a proof we are looking for.

We implemented a proof-of-concept based on the MiniSat SAT solver and Microsoft's Z3 SMT solver. To evaluate our technique, we ran experiments suggested in [9]. These experiments comprise a token-ring protocol and programs relying on so-called "arithmetic relations". The results are so far promising and show that our technique performs well not only on toy examples but also on non-trivial medium size examples. This holds especially in situation where it is a priori known that proofs are small. In our experiments, we could not find any significant difference between the SAT and SMT based approach.

Finally, let us mention that our proof-of-concept is not meant to compete with the above-named tools, mainly for two reasons. First, it is by far not as optimized as the mature tools, and further optimizations remain to be investigated. Second, it is a generic technique not solely dedicated to regular model checking.

4.2 Synthesizing Loop Invariants in Presburger Arithmetic

In this section, we consider imperative programs that work over integer variables. An example of such a program is depicted in Figure 1.

Our goal is to compute (or synthesize) loop invariants expressible in Presburger arithmetic for annotated loops. Thereby, we assume the following setting. Program states are described by Presburger formulae $\varphi(\overline{x})$ that range over all program variables $\overline{x} = (x_1, \dots, x_n) \in \mathbb{Z}^n$. The annotated loop is given as a precondition $\varphi_{\text{pre}}(\overline{x})$ of the loop, a postcondition $\varphi_{\text{post}}(\overline{x})$, and a formula $\varphi_{\text{loop}}(\overline{x}, \overline{x}')$ that describes the effect of the loop on the program variables. Thereby, $\overline{x}$ corresponds to the variables before the loop body is executed whereas $\overline{x}'$ corresponds to the (altered) program variables after the loop body has been executed.

```
Input: x

r = 0;
y = x;
while(y > 0) {
    r = r + 3;
    y = y - 1;
}
```

Fig. 1. An example program over integer variables

$$\begin{aligned} G(x,y,r) &\coloneqq y > 0 \\ \varphi_{\text{pre}}(x,y,r) &\coloneqq r = 0 \land y = x \\ \varphi_{\text{post}}(x,y,r) &\coloneqq y = 0 \land \\ &\qquad \exists t \colon x + x = t \land x + t = r \\ \varphi_{\text{loop}}(x,y,r,x',y',r') &\coloneqq x' = x \land y' = y - 1 \land \\ &\qquad r' = r + 3 \end{aligned}$$

Fig. 2. Presburger formulae describing the loop of the program in Figure 1

Finally, let $G(\overline{x})$ be the loop guard. Figure 2 shows an example of formulae that describe the loop of the program in Figure 1.

Intuitively, a loop invariant is a statement about the states of a program that is true before and after every iteration of the loop. Formally, we define a *loop invariant* as a set Inv of program states that satisfies the following properties:

- $\varphi_{\text{pre}}(\overline{x}) \rightarrow \overline{x} \in \mathit{Inv}$,
- $(\overline{x} \in \mathit{Inv} \wedge \neg G(\overline{x})) \rightarrow \varphi_{\text{post}}(\overline{x})$, and
- $(\overline{x} \in \mathit{Inv} \wedge G(\overline{x}) \wedge \varphi_{\text{loop}}(\overline{x}, \overline{x}')) \rightarrow \overline{x}' \in \mathit{Inv}$.

In the example of Figure 1 and 2, the set of program states satisfying the condition $3(x - y) = r$ is a loop invariant. In fact, this loop invariant is exactly what our technique (described below) computes in this example.

Since we are interested in loop invariants that can be expressed in Presburger arithmetic, we want to compute *regular loop invariants*, i.e., loop invariants that are regular languages, and translate them into Presburger formulae. To this end, we turn the formulae φ_{pre}, φ_{post}, and G into DFAs $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{\varphi_{\text{post}}}$, and $\mathcal{A}^{G}$ working over the alphabet $\Sigma = \{0, 1\}^n$ and φ_{loop} into a finite-state transducer $\mathcal{A}^{\varphi_{\text{loop}}}$ working over the alphabet $\Sigma \times \Sigma$. Then, we can reformulate the definition of loop invariants from above as follows, where now a loop invariant is a set $\mathit{Inv} \subseteq \Sigma^*$ that matches the encoding of program states used by $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{G}$, etc.

- $L(\mathcal{A}^{\varphi_{\text{pre}}}) \subseteq \mathit{Inv}$,
- $(\Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))) \cap \mathit{Inv} = \emptyset$ (which is true if and only if $(\mathit{Inv} \cap (\Sigma^* \setminus L(\mathcal{A}^G))) \subseteq L(\mathcal{A}^{\varphi_{\text{post}}})$), and
- $(R(\mathcal{A}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))(\mathit{Inv}) \subseteq \mathit{Inv}$.

If phrased this way, and if we set $I = L(\mathcal{A}^{\varphi_{\text{pre}}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))$, and $T = (R(\mathcal{A}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))$, then computing (or synthesizing) regular loop invariants boils down to computing proofs in the sense of Section 4.1. This leads to the main result of this section.

Theorem 3. *Let Presburger formulae φ_{pre}, φ_{post}, φ_{loop}, and G for a loop of an integer program be given, and let $I = L(\mathcal{A}^{\varphi_{pre}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{post}}))$, and $T = (R(\mathcal{A}^{\varphi_{loop}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))$. Then, Algorithm 2 with input $\mathcal{P} = (I, T)$ and B terminates and returns a regular loop invariant if one exists.*

Once Algorithm 2 returns a regular loop invariant, we can try to translate it into a Presburger formula according to [11]. However, even if this is not possible, a (regular) loop invariant is enough for the verification of programs as it proves that the postcondition holds once the loop terminates. Nonetheless, it would be interesting to investigate whether we can impose further constraints on the formula $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{A}^T}$ (corresponding to the characterization in [11]) that guarantee that a computed loop invariant can be translated into a Presburger formula.

Related Work and Experiments. Many techniques to compute loop invariants (for integer programs) have been proposed, e.g., abstract interpretation (used in [8]) or template-based approaches (used in [15]) to name just two examples. The idea of the latter is to search for loop invariants only in a restricted (often "simple") domain such as linear inequalities or polyhedra. In this sense, we use Presburger formulae, or DFAs, as templates. As to our knowledge not much research has been devoted to this type of templates. Moreover, the idea of using tools for regular model checking to synthesize loop invariants is novel.

To try our approach, we extended the SAT-based proof-of-concept of Section 4.1 using Mona [7] to translate Presburger formulae into DFAs. We considered example programs (mostly fragments taken from real world software) that are delivered with the InvGen toolkit [6]. About 10% of these examples (nine in total) were suitable for our setting, i.e., they contained a single loop, the effect of the loop could be expressed in Presburger arithmetic, etc.

Table 1. Experimental results of InvGen's "C test suite" examples

Experiment	Size	Presburger
down	3	yes
gulwani_cegar2	3	yes
half	7	
ken-imp	3	
NetBSD_g_Ctoc	2	yes
simple	2	yes
simple_if	2	yes
split	6	
up-nd	2	yes

Table 1 shows the results of our experiments. The column "Size" shows how many states a resulting DFA comprises. The column "Presburger" indicates whether a loop invariant could be translated into a Presburger formula. Since we did not use software for this task, a blank entry indicates that a DFA did not obviously encode a formula. All experiments were done on an Intel Q9550 CPU running Linux, each in less that 30 seconds with at most 300 MB of RAM used. As Table 1 depicts, our implementation found loop invariants for all examples. In comparison, InvGen also found loop invariants for all examples and used roughly the same amount of time and memory. This shows that our technique can match InvGen (where it is applicable).

However, we did not benchmark our technique with template-based tools other than InvGen. Such a comparison seems to be unsatisfactory for two reasons. First, there are very few (and perhaps uninteresting) examples that have two equally complex solutions for either template and would allow a fair comparison. Second, our implementation is a proof-of-concept, and it is doubtful whether it can compete with optimized tools on their type of templates.

5 Conclusion

We presented a generic technique to compute minimal separating DFAs and, based on that, regular invariants. The main idea is to express the desired properties of a DFA in terms of a logical formula and to use a SAT or SMT solver to find a solution. We applied this technique to the task of computing minimal separating DFAs directly as well as to regular model checking and to the synthesis of loop invariants of integer programs. Our experiments showed that SAT and SMT solvers are useful tools for these tasks.

The way we compute automata allows us to compute minimal DFAs (although that might not be needed in every situation). Moreover, we can easily adapt our technique to also compute minimal NFAs. On the one hand, this has the advantage that NFAs as solution can be exponentially smaller than DFAs, and, thus, φ_n might be satisfiable for much smaller n. The disadvantage, on the other hand, is that encoding NFAs enlarges φ_n significantly.

Finally, let us note that our work is meant as a showcase how to use SAT and SMT solvers to compute (minimal) DFAs that possess certain properties with respect to other given regular languages. In this sense, one can think of our technique as a generic toolkit from which an algorithm for a concrete problem can be instantiated. We hope that such a toolkit comes in handy for other researches and may be applied in different fields, too. For instance, our technique can also be used to compute winning strategies for games on automatic graphs [12].

Acknowledgment. I thank Madhusudan Parthasarathy and Christof Löding very much for drawing my attention to this topic and for their support.

References

1. Bardin, S., Leroux, J., Point, G.: FAST Extended Release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
3. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers C-21(6), 592–597 (1972)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
5. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning Minimal Separating DFA's for Compositional Verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
6. Gupta, A., Rybalchenko, A.: InvGen: An Efficient Invariant Generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
7. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic Second-order Logic in Practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
8. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
9. Legay, A.: Generic Techniques for the Verification of Infinite-State Systems. Ph.D. thesis, Universite de Liege (2007)

10. Legay, A.: T(O)RMC: A Tool for (ω)-Regular Model Checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 548–551. Springer, Heidelberg (2008)
11. Leroux, J.: A polynomial time presburger criterion and synthesis for number decision diagrams. In: LICS, pp. 147–156. IEEE Computer Society (2005)
12. Neider, D.: Reachability Games on Automatic Graphs. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 222–230. Springer, Heidelberg (2011)
13. Pena, J.M., Oliveira, A.L.: A new algorithm for the reduction of incompletely specified finite state machines. In: ICCAD, pp. 482–489 (1998)
14. Pfleeger, C.: State reduction in incompletely specified finite-state machines. IEEE Transactions on Computers C-22(12), 1099–1102 (1973)
15. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
16. Vardhan, A., Viswanathan, M.: LEVER: A Tool for Learning Based Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 471–474. Springer, Heidelberg (2006)

ALLQBF Solving by Computational Learning*

Bernd Becker[1], Rüdiger Ehlers[2], Matthew Lewis[1], and Paolo Marin[1]

[1] Albert-Ludwigs-Universität Freiburg
[2] Universität des Saarlandes

Abstract. In the last years, search-based QBF solvers have become essential for many applications in the formal methods domain. The exploitation of their reasoning efficiency has however been restricted to applications in which a "satisfiable/unsatisfiable" answer or **one** model of an open quantified Boolean formula suffices as an outcome, whereas applications in which a compact representation of **all** models is required could not be tackled with QBF solvers so far.

In this paper, we describe how computational learning provides a remedy to this problem. Our algorithms employ a search-based QBF solver and learn the set of all models of a given open QBF problem in a CNF (conjunctive normal form), DNF (disjunctive normal form), or CDNF (conjunction of DNFs) representation. We evaluate our approach experimentally using benchmarks from synthesis of finite-state systems from temporal logic and monitor computation.

Keywords: QBF, Computational learning, QBF model enumeration.

1 Introduction

Recent progress in quantified Boolean formula (QBF) and satisfiability (SAT) solving has strengthened the applicability of such solvers in many areas of formal methods. For example, in *bounded model checking* [1], the question whether some property holds along a run of a given system with some bounded length is encoded into a SAT formula, and then subsequently solved. In case the formula is found to be satisfiable, from a corresponding assignment to the variables, we can reconstruct a run that violates the specification. When generalizing from SAT to QBF solving, we can use the universal quantifiers to apply a more concise problem encoding or ask more complex questions such as: "do there exist values for some parameters in a system such that for every input of some fixed length to the system, we do not reach some error state?". In case of a positive answer, it is desirable to obtain values for the parameters. This is called *open QBF* solving, as here, we leave some variables in the QBF instance unquantified and ask for an assignment to these variables that witness the satisfiability of the QBF formula. Such an assignment is also called a *model* of the formula.

For other applications, however, obtaining one model is not enough, but we rather need *all models* of an open QBF formula. Representatives of this class are the synthesis of finite-state systems from temporal logic specifications, which is frequently reduced to game solving, and building a system monitor for identifying that a system

* This work was partly supported by the DFG as part of the AVACS Transregional Collaborative Research Center (SFB/TR 14).

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 370–384, 2012.

reached a potentially bad state, i.e., a state from which some error state can be reached within a short amount of time. As in these applications, there can easily be millions or even billions of models for an open QBF formula, it is furthermore not sufficient to just enumerate the models, but we rather need a *compact representation* of them. Resolution-based *variable elimination* techniques are known not to scale well for many variables to be eliminated, so we need some alternative approach for obtaining such a compact representation of all models of a given SAT or QBF instance, which we call the *ALLSAT* and *ALLQBF* problems for the scope of this paper.

For the ALLSAT problem, some solutions that go beyond simple model enumeration and successive variable elimination are known. A typical ingredient of such an approach is the enumeration of solution cubes [2,3], which leads to a DNF representation of the set of models, possibly combined with some on-the-fly or a-posteriori post-processing to obtain a CNF representation [2,4] of the model set. For ALLQBF, search-based solving approaches that go beyond simple model enumeration [5] and variable elimination by resolution are unknown so far. Thus, for applications in which the ALLQBF problem has to be solved for instances that have many models, but for which there are also many quantified variables, no feasible solution exists yet.

In this paper, we present an approach to extend a state-of-the-art search-based QBF solver to an ALLQBF solver by employing *(active) computational learning* [6], which is the process of deriving a model of some data by asking questions to some *teacher oracle*. Computational learning should not be confused with *clause learning*, a technique to increase the performance of SAT and QBF solvers.

Our approach learns a CNF (conjunctive normal form), DNF (disjunctive normal form) or CDNF (conjunction of DNFs) representation of the set of all models of an open quantified Boolean formula, i.e., a QBF instance in which some variables are left free. The algorithms for all of these result types can equally be applied for ALLSAT solving, but the main target of our approach is ALLQBF solving. Benchmarks from synthesis of finite-state systems and monitor generation show the effectiveness of the new approach.

We start by stating the required preliminaries in Section 2 and describe our approach to learn DNF, CNF or CDNF representations of the set of models of an open QBF problem in Section 3. Section 4 discusses how a modern QBF solver can be adapted to its use as oracle in a learning process. In Section 5, we discuss *synthesis* and *monitor generation* as two of the application of ALLQBF solving, from which we take the Benchmarks for our experimental evaluation of a prototype implementation of our learning approach in Section 6. We conclude with a summary.

2 Preliminaries

In this paper, we consider open quantified Boolean formulas (QBF) in *prenex-cnf-form*, i.e., for a finite set of *free variables* V, we define the set of QBF instances $\mathcal{Q}(V)$ over V as all formulas of the type:

$$\psi = Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n . \phi \tag{1}$$

where $n \in \mathbb{N}$, $Q_i \in \{\forall, \exists\}$ for all $1 \leq i \leq n$, and ϕ is a Boolean formula in conjunctive normal form over the set of variables $\{x_1, \ldots, x_n\} \cup V$. A Boolean formula is said to be in *conjunctive normal form* if it is a conjunction of a set of *clauses* $\phi = \bigwedge_v C_v$. A clause C_v is in turn a set of literals that is treated disjunctively, i.e., $C_v = \bigvee_w l_w$. A *literal* is a variable or its negation. In (1), $Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n$ is the *prefix* and ϕ is the *matrix*. The *level of a variable* x_i is defined to be one plus the number of expressions $Q_j x_j . Q_{j+1} x_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$ (plus 1 if $Q_1 = \forall$). For the sake of simplicity, we will use the term *outermost* (respectively *innermost*) quantifier level to indicate variables having the highest (respectively lowest) level. The level of a literal is the level of its variable. A literal l is *universal* if $l = v_i$ or $l = \neg v_i$ for some $1 \leq i \leq n$ with $Q_i = \forall$. All other literals are *existential*. In (1), a literal l is

- *unit* if l is existential, and, for some $m \geq 0$,
  - a clause $(l \vee l_1 \vee \ldots \vee l_m)$ is a clause in ϕ, and
  - each literal l_i $(1 \leq i \leq m)$ is universal and has a level lower than l.
- *monotone* or *pure* if
  - either l is existential, $\neg l$ does not occur in any clause in ϕ, and l occurs in some clauses in ϕ;
  - or l is universal, l does not occur in any clause in ϕ, and $\neg l$ occurs in some clauses in ϕ.
- *don't care* if l is existential, and neither l nor $\neg l$ occur in any clause in ϕ.

Any element of $\mathcal{Q}(V)$ can be seen as a function that maps some variable valuation $f \in (V \rightarrow \mathbb{B})$ to either $\mathbf{true}$ or $\mathbf{false}$. If $Q_1 = \forall$, we say that ψ is outermost universally quantified, and if $Q_1 = \exists$, we say that ψ is outermost existentially quantified. We call $\mathcal{Q}(\emptyset)$ the set of *closed QBF instances*. For any set V, $\mathcal{Q}(V)$ is the set of *open QBF instances* over V.

Given some set of variables V and some open QBF formula $\psi \in \mathcal{Q}(V)$, we say that some variable valuation $f \in (V \rightarrow \mathbb{B})$ is a model of ψ if $\psi(f) = \mathbf{true}$. Likewise, f is a *co-model* of ψ if $\psi(f) = \mathbf{false}$. A *partial variable valuation* is a function $f' : V \rightarrow \{\mathbf{false}, \mathbf{true}, \perp\}$, and a variable valuation f is a *completion* of f' if for every $v \in V$, $f'(v) = f(v)$ if $f'(v) \neq \perp$. We say that a partial variable valuation is a *partial model* (*partial co-model*) of some open QBF formula ψ if every completion of the partial valuation is a model (co-model) of ψ.

We call a conjunction of literals a *term* and a disjunction of terms a Boolean formula in *disjunctive normal form*. For a partial (or complete) variable valuation $f' \in (V \rightarrow \{\mathbf{false}, \mathbf{true}, \perp\})$, we define the term induced by f' as follows:

$$\mathrm{term}(f') = \Big(\bigwedge_{v \in V, f'(v) = \mathbf{true}} v \Big) \wedge \Big(\bigwedge_{v \in V, f'(v) = \mathbf{false}} \neg v \Big)$$

For some Boolean formula in disjunctive normal form $t_1 \vee t_2 \vee \ldots t_m$, we define $\mathrm{terms}(t_1 \vee t_2 \vee \ldots \vee t_m) = \{t_1, t_2, \ldots, t_m\}$. Likewise, for some term $t = l_1 \wedge l_2 \wedge \ldots \wedge l_m$, we define $\mathrm{lits}(t) = \{l_1, l_2, \ldots, l_m\}$. For a Boolean formula ψ over some set of variables V, some term $t = l_1 \wedge \ldots \wedge l_m$ is an implicant of ψ if $\neg t \vee \psi \equiv \mathbf{true}$. A term is called a *prime implicant* if we cannot remove any of its literals without losing the property that it is an implicant. A Boolean function $F : (V \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$ is called monotone if for every $f, f' : V \rightarrow \mathbb{B}$ with $F(f) = \mathbf{true}$ and for all $v \in V$, $f(v) = \mathbf{true}$ implies $f'(v) = \mathbf{true}$, we have $F(f') = \mathbf{true}$. We denote the *exclusive or* Boolean operator by $\oplus$ and the function composition operator by $\circ$.

3 ALLQBF Solving by Computational Learning

In this section, we show how to use computational learning techniques to obtain compact representations of the sets of models of open QBF formulas that are specified in the commonly used prenex-cnf-form.

Given an open QBF formula ψ over the set of variables V, we consider three representations for a set of models of ψ here: a disjunctive normal form (DNF) Boolean formula, a conjunctive normal form (CNF) Boolean formula, and a conjunction of DNFs (called CDNF), all over V. We start this section by first declaring the requirements to the QBF solver that we use as an oracle in the following learning algorithms, and then explain how to compute a DNF, CNF or CDNF representation of the set of models using the QBF solver as oracle in Sections 3.2, 3.3, and 3.4, respectively.

3.1 Requirements to the QBF Solver Used

For the henceforth algorithms, we use a QBF solver for open QBF formulas in prenex-cnf-form as an oracle in the learning process, and apply it for checking the satisfiability and non-universality of open QBF formulas. In case of a positive answer to one of these checks, the solver must be able to return a (partial) model/co-model of the open QBF formula, respectively, i.e., a valuation for the unquantified variables.

In the experimental evaluation of the following learning schemes, we used an open-QBF version of QuBE 7.2 [7]. The modifications that were required are more complex than one might think, as QuBE 7.2 uses a very advanced preprocessor that can remove and rename variables. This preprocessor allows it to achieve high levels of performance, but it can make the production of models and co-models nontrivial. So, the modifications for satisfying the requirements above, while still using the preprocessor, are described in Section 4.

3.2 Learning DNFs

Let $\psi = Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n . \phi$ be an open QBF formula over the free variables V such that ϕ is in conjunctive normal form. For learning a DNF representation of the set of models of ψ, we apply a variant of the classical algorithms from learning theory for obtaining monotone DNFs [8] or k-term DNFs [9] from a function to learn. Our variant, depicted in Algorithm 1, makes use of the fact that when learning from open QBF formulas, we can take advantage of the possibility to check if a term is an implicant of ψ.

In the algorithm, terms are repeatedly added to the *candidate* DNF representation ψ' until ψ' represents precisely the set of models of ψ. In line 2 of the algorithm, it is checked using a QBF solver as an oracle whether there exists some variable valuation to the variables in V that is a model of ψ, but not yet of ψ'. Whenever this is the case, we know that ψ' is not yet complete and search for a prime implicant of ψ' that also implies the newly found variable valuation.

Note that since the negation of a DNF formula is a CNF formula, $(\phi \wedge \neg\psi')$ is in CNF, and thus the QBF instance ρ computed in line 2 is in prenex-cnf-form. The algorithm uses the partial model of ρ as a starting point for finding the next prime implicant. Of course, the model can also be complete, but does not need to be. In particular, if the QBF solver finds a satisfying assignment to V while still preprocessing, the model will typically be incomplete.

In the remaining lines of the algorithm, the implicant f' (represented in form of a partial variable valuation) obtained in line 3 is reduced as much as possible in order to obtain a prime implicant. In line 7, we take profit from the fact that we use a QBF solver as oracle: by universally quantifying over the variables that are not set in f' and one additional variable v, we can check whether $f'(v)$ can be set to $\perp$ without changing the fact that f' is an implicant. The algorithm guarantees that at the end of the computation, the DNF representation of the set of models of ψ is *irreducible*, i.e., there are no superfluous literals or terms in the obtained DNF ψ'.

Algorithm 1. DNF learning using a QBF solver as oracle

```
Data: An open QBF instance $\psi = Q_1 x_1.Q_2 x_2.\dots Q_n x_n.\phi$ over the set of variables $V$
Result: The set of its models represented as DNF $\psi'$
begin
1   $\psi'$ := false ;
2   while $\rho := Q_1 x_1.Q_2 x_2.\dots Q_n x_n.(\phi \wedge \neg\psi')$ is satisfiable do
3       $f'$ := partial model of $\rho$;
4       for $v \in V$ do
5           if $f'(v) \neq \perp$ then
6               $f'' := f' \setminus (v \mapsto f'(v)) \cup (v \mapsto \perp)$;
7               if $\forall\{v' \in V \mid f''(v') = \perp\}.Q_1 x_1.Q_2 x_2.\dots Q_n x_n.(\phi \wedge \mathrm{term}(f'')) \not\equiv$ false
                then
8                   $f' := f''$;
9       $L := \{\neg v \mid v \in V, f'(v) = \textbf{false}\} \cup \{v \mid v \in V, f'(v) = \textbf{true}\}$;
10      $\psi' := \psi' \vee \bigwedge_{l \in L} l$;
end
```

3.3 Learning CNFs

Learning CNFs instead of DNFs as described above can be seen as the dual case. Since for search-based QBF solving, the matrix of a Boolean formula has to be in CNF, we would however have to re-encode this matrix to complement the original formula. Thus, we apply a slightly different method, aiming to skip the re-encoding step into prenex-normal-form of the formula.

Algorithm 2 describes the modified procedure. In this algorithm, the function Enc^- is used to map a Boolean formula in CNF form into a CNF formula that encodes its negation (using a Tseitin encoding [10]), and EncV^- describes the necessary variables. Let $\psi = \bigwedge_{1 \leq i \leq m} \bigvee_{1 \leq j \leq k_i} l_{ij}$ be a CNF formula with $m \in \mathbb{N}$, $k_1, \dots, k_m \in \mathbb{N}$, and $l_{ij} \in V \cup \{\neg v \mid v \in V\}$ for $1 \leq i \leq m$ and $1 \leq j \leq k_i$. We define:

$$\mathsf{EncV}^-(\psi) = \{v_i \mid 1 \leq i \leq m\}$$

$$\mathsf{Enc}^-(\psi) = \Big(\bigvee_{1 \leq i \leq m} v_i\Big) \wedge \bigwedge_{1 \leq i \leq m} \bigwedge_{1 \leq j \leq k_i} (\neg v_i \vee \neg l_{ij})$$

Algorithm 2. CNF learning using a QBF solver as oracle

```
Data: An open QBF instance $\psi = Q_1x_1.Q_2x_2.\dots Q_nx_n.\phi$ over the set of variables $V$
Result: The set of its models represented as CNF $\psi'$
begin
1   $\psi' := \mathbf{true}$ ;
2   while $\rho := \exists s, \mathsf{EncV}^-(\psi').Q_1x_1.Q_2x_2.\dots Q_nx_n.((\phi \vee s) \wedge (\mathsf{Enc}^-(\psi') \vee \neg s))$ is
    non-universal do
3       $f'$ := partial co-model of $\rho$;
4       for $v \in V$ do
5           if $f'(v) \neq \bot$ then
6               $f'' := f' \setminus (v \mapsto f'(v)) \cup (v \mapsto \bot)$;
7               if $Q_1x_1.Q_2x_2.\dots Q_nx_n.(\phi \wedge \mathrm{term}(f'')) \equiv$ false then
8                   $f' := f''$;
9       $\psi' := \psi' \wedge \left(\bigvee_{v \in V, f'(v) = \mathbf{false}} v \vee \bigvee_{v \in V, f'(v) = \mathbf{true}} \neg v\right)$;
end
```

The algorithm is based on the idea to iterative find cubes of variable valuations to the free variables that falsify the input formula. Such cubes are then added to the input formula in the next round of the algorithm by encoding these using the Enc^- function in line 2 of the algorithm.

3.4 Learning CDNF

In [11], Bshouty describes a learning algorithm for conjunctions of Boolean formulas in disjunctive normal form based on the *monotone theory*. Given some set of variables V, we call a Boolean function f over V c-monotone for some $c : V \to \mathbb{B}$ if $f' \circ m_c$ is monotone for m_c being the function mapping a variable valuation $x : V \to \mathbb{B}$ to some other valuation $x' : V \to \mathbb{B}$ such that for all $v \in V$: $x'(v) = x(v) \oplus c(v)$.

The main idea of Bshouty's learning algorithm is to represent the function to learn as a conjunction of Boolean formulas in disjunctive normal forms, where each of these formulas is c-monotone for some $c \in (V \to \mathbb{B})$. During the learning process, the algorithm maintains and updates the candidate CDNF formula ψ' learned so far. Whenever there exists a false-positive for this CDNF formula, i.e., there exists a valuation $f : (V \to \mathbb{B})$ for which $\psi'(f) = \mathbf{true}$ but $\psi(f) = \mathbf{false}$, a new DNF is added to ψ' that is kept f-monotone during the learning process. Whenever a false-negative is found for ψ', i.e., there exists a valuation $f : (V \to \mathbb{B})$ for which $\psi'(f) = \mathbf{false}$ but $\psi(f) = \mathbf{true}$, for every DNF ρ that is a conjunct of ψ' and its associated monotonicity base c, ρ is extended by some prime implicant for the c-monotone closure of ψ that is implied by f. For more details on Bshouty's CDNF learning algorithm, the interested reader is referred to [12].

Algorithm 3 shows the overall learning algorithm, adapted to the treatment of open QBF formulas. During its run, the set C contains the CDNF learned so far, split up into its DNF formulas, which are paired together with the respective monotonicity base.

Algorithm 3. CDNF learning using a QBF solver as oracle

```
Data: An open QBF instance $\psi = Q_1x_1.Q_2x_2.\dots Q_nx_n.\phi$ over the set of variables $V$
Result: The set of its models represented as CDNF $\psi'$
begin
    $C := \emptyset$ ;
    while true do
        if $\rho := \exists s, \mathsf{EncV}^-(C).Q_1x_1.Q_2x_2.\dots Q_nx_n.((\phi \vee s) \wedge (\mathsf{Enc}^-(C) \vee \neg s))$ is
        non-universal then
            $f$ = co-model of $\rho$;
            $C := C \cup \{(f, \mathbf{false})\}$;
        if $\rho := \exists \mathsf{EncV}^-(C).Q_1x_1.Q_2x_2.\dots Q_nx_n.\phi \wedge \mathsf{Enc}^-(C)$ is satisfiable then
            $f$ = model of $\rho$;
            $C' = \emptyset$;
            for $(c, \eta) \in C$ do
                if $f \models \eta$ then
                    $C' = C' \cup \{(c, \eta)\}$
                else
                    for $v \in V$ do
                        if $c(v) \neq f(v)$ then
                            $f' = f \setminus (v \mapsto f(v)) \cup (v \mapsto c(v))$;
                            if $Q_1x_1.Q_2x_2.\dots Q_nx_n.\phi \wedge \mathrm{term}(f')$ is satisfiable then
                                $f := f'$;
                    $\eta' = \left(\bigwedge_{v \in V, f(v)=\mathbf{true}, c(v)=\mathbf{false}} v\right) \wedge \left(\bigwedge_{v \in V, f(v)=\mathbf{false}, c(v)=\mathbf{true}} \neg v\right)$;
                    $C' = C' \cup \{(c, \eta \vee \eta')\}$;
            $C := C'$;
        else
            for $(c, \eta) \in C$ do
                for $t \in \mathrm{terms}(\eta)$ do
                    $\eta' := \bigvee_{t' \in \mathrm{terms}(\eta) \setminus \{t\}} t'$;
                    $C' := C \setminus \{(c, \eta)\} \cup \{(c, \eta')\}$;
                    if $\exists V, \mathsf{EncV}^+(C), \mathsf{EncV}^-(C').(\mathsf{Enc}^+(C) \wedge \mathsf{Enc}^-(C')) \equiv \mathbf{false}$ then
                        $C := C' \cup \{(c, \eta')\}$;
                    else
                        for $l \in \mathrm{lits}(t)$ do
                            $\eta' := \bigvee_{t' \in \mathrm{terms}(\eta) \setminus \{t\}} t' \vee \bigwedge_{l' \in \mathrm{lits}(t) \setminus \{l\}} l'$;
                            $C' := C \setminus \{(c, \eta)\} \cup \{(c, \eta')\}$;
                            if $\exists V, \mathsf{EncV}^-(C), \mathsf{EncV}^+(C').(\mathsf{Enc}^-(C) \wedge \mathsf{Enc}^+(C')) \equiv \mathbf{false}$ then
                                $C := C' \cup \{(c, \eta')\}$;
            return $\psi' = \bigwedge_{(c,\eta) \in C} \rho$;
end
```

In this algorithm, the function Enc^- is used to map a set C onto a CNF that encodes the negation of the CDNF formula represented by $C = \{(c_1, \rho_1), \ldots, (c_m, \rho_m)\}$, and EncV^- describes the necessary variables. We define:

$$\mathsf{EncV}^-(C) = \{v_i \mid 1 \le i \le m\}$$
$$\mathsf{Enc}^-(C) = \Big(\bigvee_{1\le i\le m} v_i\Big) \wedge \bigwedge_{1\le i\le m} (\neg v_i \vee \neg \rho_i)$$

Likewise, the function Enc^+ is used to map a set C onto a CNF that encodes it in non-negated form using the Tseitin encoding [10] and EncV^+ represents the necessary variables. The number of variables needed here is higher than for EncV^-, as one variable is introduced for every term in the DNFs of C.

In line 3 of the algorithm, it is checked whether C has a false-positive. A false-positive can be found by obtaining a satisfying assignment to the formula $\bigwedge_{(c,\rho)\in C} \rho$ $\wedge \neg\psi$. However, when ψ is in prenex-normal form, negating ψ requires a re-encoding to get $\neg\psi$ back into prenex-cnf. In Algorithm 3, this problem is circumvented by searching for a witness for the non-universality of $\bigvee_{(c,\rho)\in C} \neg\rho \vee \psi$ instead (which is the negation of the former formula).

Whenever a false-negative is found, all DNFs in C for which the false-negative is not a model are updated to change this fact. At the end of the algorithm (starting with line 22), redundant terms in the DNFs and redundant literals are identified using standard SAT solving (lines 26 and 32), and then removed.

4 QBF Solver Modification

All operations on open QBF formulas can easily be translated to the closed QBF case as follows: checking for satisfiability of an open QBF formula ψ over V amounts to testing if $\exists V.\psi \equiv \mathbf{true}$, and checking the non-universality of an open QBF formula ψ amounts to testing if $\forall V.\psi \equiv \mathbf{false}$. In the former case, the solver must be able to output a partial model of ψ. This is supported by many modern QBF solvers when the outermost quantification level is existentially quantified, which is the case for $\exists V.\psi \equiv \mathbf{true}$. For the non-universal (unsatisfiable) case, where the outermost quantification level is universal ($\forall V.\psi \equiv \mathbf{false}$), our QBF solver oracle needs to output a trace, or path, that leads to an unsatisfiable branch of the search space. This is referred to here as a co-model.

In this work we use the QBF solver QuBE 7.2 [7]. QuBE is a state-of-the-art DPLL search-based solver designed to take closed formulas as input, where $V = \emptyset$. This obstacle is circumvented by making free variables quantified as described above, and pushing them to the outermost quantifier level. Furthermore, QuBE incorporates many modern techniques: with respect to this paper, it includes for instance pure/don't care literal detection [13], conflict and solution analysis with solution cube minimization and learning [14,15,16], and an advanced preprocessor [17] that allows it to achieve unmatched performance when compared to the pure search-based algorithm. However, many of these advanced techniques have to be modified in order to produce correct partial models and co-models of the input formula.

Both the preprocessor and the solver were modified in order to keep information on the outermost quantified variables in a slightly more advanced way than a plain use of *don't touch literals* techniques (also called *frozen literals*) as previously done, e.g. in [18]. Indeed, either existential or universal variables are selectively "not touched" according to the outermost binding quantifier.

4.1 Preprocessing Phase

The preprocessor must behave in a slightly different way depending on the quantifier at the highest level of the prefix of the input formula. In case this is an existential quantifier, no variable elimination nor variable renaming techniques —such as equivalence reasoning, variable elimination by Q-resolution, and the subsumption through resolution (self-subsumption) that are not model preserving transformations which can be applied to existential variables— are performed on don't touch variables. Rather, unit and pure literals can still be given a value, simply adding it to the model. In the second case, where the input formula is bound by a universal quantifier, no preprocessing techniques have to be deactivated. Basically, the only rules of inference normally applied to universal variables are pure literal detection and universal reduction, also known as clause minimization [19]. Universal pure literals are immediately pushed into the current model: Note, by the definition of universal pure literal in Section 2, that this valuation to the variable will be sound in order to falsify the clause in case the formula is unsatisfiable. The universal reduction rule states that in a clause, whose literals have been simplified according to their evaluations, the literals quantified at the innermost prefix level among all the others can be removed if universal. This operation is performed every time a clause is added to the matrix — for instance, when a resolvent computed in a Q-resolution step substitutes its two antecedents, and every time a clause is simplified (e.g. when in a clause the existential literal l is deleted because $\neg l$ is unit). Whenever a universal reduction rule is applied, we track the universal literals being deleted: if all the literals in the clause are eliminated, resulting in the empty clause that proves the unsatisfiability of the whole formula, those literals are pushed into the model with their sign flipped.

4.2 Search Phase

During the search, in order to extract the model we have to record the value given to the outermost quantified variables as soon as a conflict occurs or a solution is found. This is done selectively for conflicts if the outermost quantifier is existential, or for solutions when the outermost quantifier is universal. When the solving procedure completes the exploration of the search space, the assignment values saved previously can be eligible to be included into the model. Indeed, because of the on-the-fly universal reduction performed by the solver during both exploration and backtrack phases on the clauses (respectively, its dual existential reduction being performed on the solution cubes), it may be the case that some valuations must be changed, or even further variables must be pushed into the model as well. This can happen when their values have already been taken from the assignment stack and put into the model or no valuation is currently given. In these situations, the valuation must be either flipped in case it is currently satisfying the clause/cube, or forced in case it has no valuation yet. Consider the QBF $\varphi = \forall y_1 y_2 \exists x_3 \phi$ and the empty clause $y_1 \vee y_2 \vee x_3 \in \phi$. Assume that y_2 is a decision literal at level d, and $\neg x_3$ is assigned because of unit propagation at the same decision level d. No value was given to $|y_1|$. As soon as the conflict occurred, the assignment to the outermost quantified universal variables was cached as y_2. Since the empty clause has led the conflict analysis to the root of the search tree, witnessing the unsatisfiability of the whole QBF, the model has to be modified as follows: $\neg y_1$ is added, and the value for $|y_2|$ is flipped into $\neg y_2$.

5 Applications of ALLQBF Solving

In this section, we sketch two applications of ALLQBF solving. The aim of this section is twofold: first of all, our experimental evaluation in the next section is based on benchmarks from these two applications. Second, we want to show the interested reader *why* the transition from plain QBF solving to ALLQBF solving is such an interesting one.

5.1 Synthesis of Reactive Systems

In formal verification, one analyses a system for correctness with respect to a specification after it has been designed. The idea behind *synthesis* is that we can actually construct a system directly from the specification, and save the manual work of actually designing it. After choosing a formal specification language, and describing which inputs and outputs the system under design has, synthesis is essentially a push-button technique.

On the technical level, synthesis is typically reduced to *solving a game with an ω-regular winning condition*. A play in this game represents a *trace* of the system to be synthesized, and plays that are winning for a designated *system* player in turn represent traces that are allowed by the specification. Winning plays are of infinite length, meaning that the system they represent has no predefined point of going out of service. Games frequently have huge state spaces, but are representable in a symbolic way. Determining the positions in the game from which the system player wins is typically done by evaluating a *fixed point* expression. For example, in case of a safety specification, the synthesis problem reduces to safety game solving, and the winning positions are the ones that are not in the *attractor* of the *bad positions*, i.e., the ones from which the system player can enforce never to visit any of the bad states. Using a quantified Boolean formula, we can represent the problem if for a position in game there exists an output such that for every input, a non-bad state is reached. By performing ALLQBF solving on this formula, we obtain a small representation of *all* of these positions if the game transition function has a small encoding as CNF formula. Then, we can plug in the negation of this positions set as the new set of bad states, and obtain a formula whose models represent the positions from which the system player does not lose in two steps. Iterating this idea until we reach a fixed point finally gets us the set of states that are winning for the system player.

Currently, synthesis tools use BDDs [20] or anti-chains [21] for symbolic reasoning, which are both techniques with well-known scalability limits. Plain QBF solving has been proposed earlier for *bounded* safety game solving [22], but was found to have a bad performance there. For the initial experimental evaluation in this paper, we used a modified version of the UNBEAST synthesis tool [20] that lazily tracks the operations performed on BDDs and generates QBF instances to represent pre-fixed points in the game solving fixed point computation when synthesizing a load balancer. As the computations performed by UNBEAST are optimized towards BDD usage, we refrain from declaring a “winner“ in this setting.

5.2 Monitor Synthesis

Consider a safety-critical sequential circuit. Adding a *runtime monitor* for such a circuit, that observes the transitions in the system and produces a warning signal if the system is about to enter a bad state, allows warning other parts of the system that the output of this circuit should not be trusted any more. This way, malicious bit flips in the hardware, as well as assumptions about the system environment that actually do not hold in practice, can often be found even after the system is deployed.

ALLQBF solving can help us in constructing such a monitor. For some value of k, we encode the problem “for a given state in the system, along some trace of length k starting from there, we do not visit a bad state” into an open QBF formula, and leave the starting state variables open. A CDNF, DNF or CNF representation for all of these states can then be interpreted as a circuit that checks if the system is still in a state that is not potentially bad, and outputs **false** if this is not the case.

For our experimental evaluation in the next section, we used the single-property circuits from the hardware model checking competition HWMCC’11 [23] and translated the monitor synthesis problem for $k = 2$ into an open QBF formula using a standard Tseitin encoding.

6 Experimental Results

We evaluate a prototype implementation of the learning techniques presented in Section 3 using a version of QuBE 7.2 that has been modified as described in Section 4. Before learning, we simplify the open QBF instance by applying a restricted version of QuBE’s preprocessor that does not alter its set of models.

The aim of this experimental evaluation is to show that the proposed approaches already scale to systems of practical size. Due to the fact that ALLQBF solving is a relatively new topic, comparing against other solvers is difficult. For example, the QBF solver QUANTOR [24] is based on removing quantifiers by resolution and expansion, and in principle it is possible to obtain a CNF representation of the set of models of an open QBF formula. However, it comes with no possibility to ensure that the open (or outermost existential) variables remain intact and thus cannot be used.Techniques for removing only existential variables cannot handle the universal ones dealt with here.

All computation times given in the following are obtained on a Sun XFire computer with 2.6 Ghz AMD Opteron processors running an x64-version of Linux. The memory usage was never observed to exceed 2GB.

6.1 Synthesis of Reactive Systems

Out of the 3411 game solving/synthesis benchmarks, the maximally allowed computation time of 3600 seconds was enough for CDNF learning to work in 3307 cases, CNF learning to finish in 3358 cases, and DNF learning to succeed in 3297 case. Figure 1 shows the numbers of instances learned over time, while Figure 2 compares the number of literals in the learned model set representation. Table 1 shows the properties of some example instances used in this comparison.

It can be seen that in many cases, the result sizes of the techniques coincide - then, DNF learning is often the fastest method. However, for complicated benchmarks, when granting more time for solving an instance, the CNF variant overtakes the DNF variant in terms of instances solved. The experiments show that the CDNF learning method is a reasonable compromise between the two.

Table 1. Properties of some example problem instances in the game solving benchmarks. For every instance, "# V.", "# F." and "# Clauses" denote the numbers of variables, free variables and clauses in the instance, respectively. The column "P.S.-time" contains the time for plain QBF solving the instance. For CDNF, DNF and CNF learning, the sizes (s.) of the resulting formulas and the time (t.) to obtain these are reported. All times are given in seconds.

Instance	**#**	**#**	**#**	**P.S.**	**#**	**CDNF**		**CNF**		**DNF**	
	V.	**F.**	**Cl.**	**-time**	**Models**	**s.**	**t.**	**s.**	**t.**	**s.**	**t.**
`load_16.xml_SAT_4_10`	1185	18	1349	0.04	90112	10	3.62	9	2.12	12	1.83
`load_18.xml_SAT_4_10`	1935	28	2179	0.06	9.22747e+07	10	7.02	9	3.90	12	3.05
`load_31.xml_SAT_4_10`	11299	108	13090	0.16	1.08939e+29	31	192	19	286	42	45.6
`load_33.xml_SAT_5_7`	3396	49	3588	0.08	1.5668e+13	14	21.4	21	21.2	31	6.90
`load_35.xml_SAT_5_7`	5401	75	5746	0.11	2.62866e+20	16	50.7	23	60.3	39	16.0
`load_57.xml_UNSAT_5_6`	2924	37	4330	0.05	unknown	timeout		timeout		timeout	
`load_72.xml_UNSAT_4_3`	1181	37	1470	0.01	3.70482e+09	timeout		109	645	timeout	
`load_74.xml_SAT_2_8`	10402	73	10115	0.14	2.95148e+20	6	49.6	6	69.3	6	8.09
`load_75.xml_SAT_2_9`	14390	88	14425	0.21	9.67141e+24	6	95.2	6	130	6	13.4

6.2 Monitor Synthesis

Out of the 465 monitor synthesis benchmarks, in 286 cases, the learning process did not finish for any mode within a time limit of 15 minutes. In 65 additional cases, the set of models was empty or contained all possible variable valuations (and is therefore uninteresting for the purpose of monitoring). Table 2 shows some representative remaining cases. Compared to the game solving benchmarks, it can be seen that the performance is worse, which is due to the fact that the monitor synthesis benchmarks are harder to solve: they have more variables, more clauses, and are derived from challenging hardware model checking problems.

For monitor synthesis, CDNF learning is not as competitive as in the game solving case, and DNF learning is more advisable to use here than CNF learning. Figure 3 shows the performance.

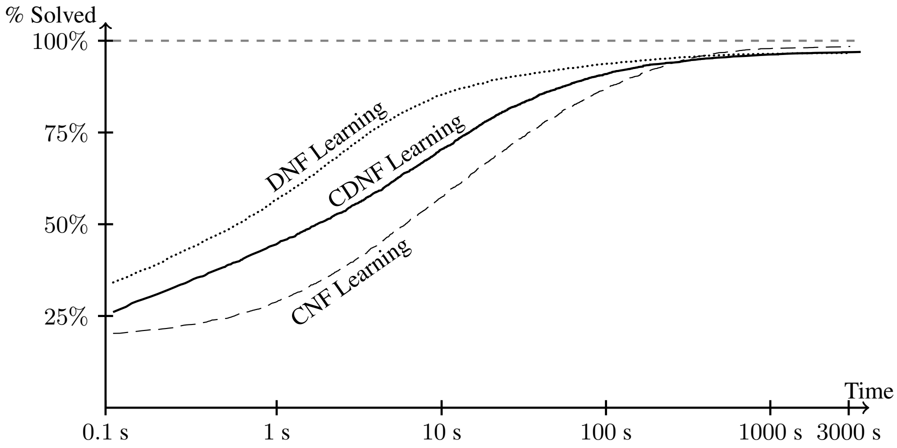
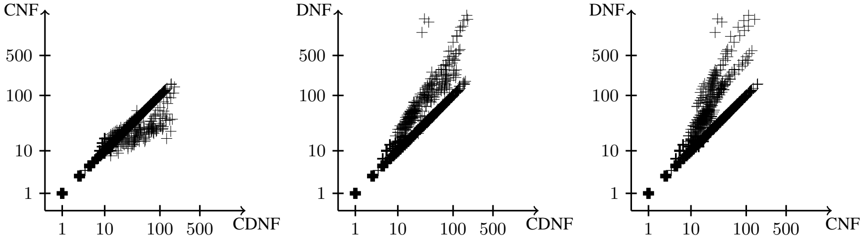


Fig. 1. Graph showing how many of the reactive synthesis benchmarks could be solved (i.e., the set of their models is learned) within certain time bounds



(a) CDNF size vs. CNF size **(b)** CDNF size vs. DNF size **(c)** CNF size vs. DNF size

Fig. 2. Formula size comparisons of the learning results for CDNF, CNF, and DNF on the game solving benchmarks

Table 2. Properties of some example problem instances in the game solving benchmarks. The notation is the same as in Table 1.

Instance	#	#	#	P.S.	#	CDNF		CNF		DNF	
	V.	F.	Cl.	-time	Models	s.	t.	s.	t.	s.	t.
bob3	1232	74	3218	0.1	7.41919e+21	timeout		timeout		1211	485
bob9234redmiter	1843	119	4627	0.15	2.10288e+35	timeout		384	798	358	371
irstdme4	2534	124	6088	0.21	1.66153e+37	25	245	46	180	15	48.9
pdtvisgigamax0	2276	16	6566	0.21	64512	timeout		7	185	7	21.7
vis4arbitp1	747	23	2024	0.07	4.18867e+06	timeout		652	389	331	29.3

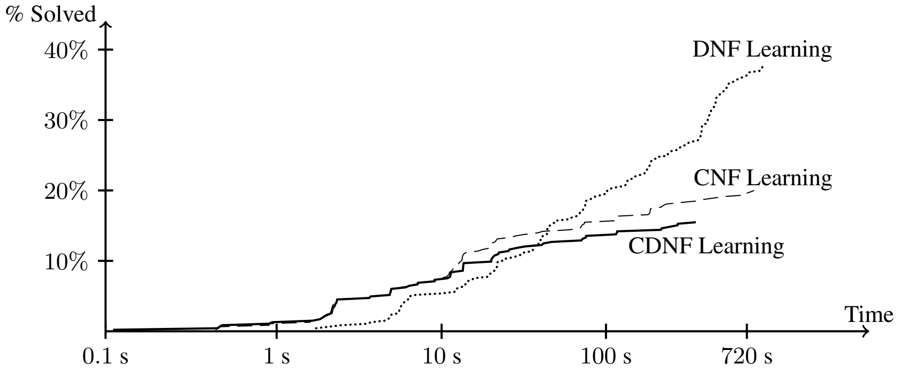


Fig. 3. Graph showing how many of the monitor synthesis benchmarks could be solved (i.e., the set of their models is learned) within certain time bounds

7 Conclusion

We have presented a way to turn a search-based QBF solver into an ALLQBF solver for open quantified Boolean formulas by using computational learning. The resulting set of models of a formula is represented either in DNF, CNF, or CDNF form, and we gave suitable learning algorithms for all of these forms.

The initial evaluation of the approach in this paper shows its potential. We conjecture that a future tighter integration of the solver and the learning algorithm will provide a significant further speed improvement.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Rance Cleaveland, W. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Brauer, J., King, A., Kriener, J.: Existential Quantification as Incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
3. McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
4. Brauer, J., Simon, A.: Inferring Definite Counterexamples through Under-Approximation. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 54–69. Springer, Heidelberg (2012)
5. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. JSAT 5, 133–191 (2008)
6. Valiant, L.G.: A theory of the learnable. Commun. ACM 27, 1134–1142 (1984)
7. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE7.0, System Description. JSAT 7, 83–88 (2010)
8. Angluin, D.: Queries and concept learning. Machine Learning 2, 319–342 (1987)

9. Hellerstein, L., Raghavan, V.: Exact learning of DNF formulas using DNF hypotheses. J. Comput. Syst. Sci. 70, 435–470 (2005)
10. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic, Part 2, 115–125 (1970)
11. Bshouty, N.H.: Exact learning Boolean function via the monotone theory. Inf. Comput. 123, 146–153 (1995)
12. Sloan, R.H., Szörényi, B., Turán, G.: Learning Boolean functions with queries. In: Crama, Y., Hammer, P.L. (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge University Press (2010)
13. Gent, I., Giunchiglia, E., Narizzano, M., Rowley, A., Tacchella, A.: Watched Data Structures for QBF Solvers. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 25–36. Springer, Heidelberg (2004)
14. Gent, I.P., Rowley, A.G.D.: Solution Learning and Solution Directed Backjumping Revisited. In: Technical Report APES-80-2004, APES Research Group (2004)
15. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. Journal of Artificial Intelligence Research (JAIR) 26, 371–416 (2006)
16. Marin, P., Giunchiglia, E., Narizzano, M.: Conflict and solution driven constraint learning in QBF. In: Doctoral Program of Constraint Programming Conference 2010, CP 2010 (2010)
17. Giunchiglia, E., Marin, P., Narizzano, M.: sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 85–98. Springer, Heidelberg (2010)
18. Kupferschmid, S., Lewis, M., Schubert, T., Becker, B.: Incremental preprocessing methods for use in BMC. In: Int'l Workshop on Hardware Verification (2010)
19. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Information and Computation 117, 12–18 (1995)
20. Ehlers, R.: Unbeast: Symbolic Bounded Synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
21. 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)
22. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. STTT 7, 118–128 (2005)
23. Biere, A., Heljanko, K., Wieringa, S., Sörensson, N.: 4th hardware model checking competition (HWCC 2011) (Affiliated with FMCAD 2011), http://fmv.jku.at/hwmcc11/
24. Biere, A.: Resolve and Expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005)

Equivalence of Games with Probabilistic Uncertainty and Partial-Observation Games

Krishnendu Chatterjee[1], Martin Chmelík[1], and Rupak Majumdar[2]

[1] IST Austria (Institute of Science and Technology Austria)
[2] MPI-SWS, Germany

Abstract. We introduce games with probabilistic uncertainty, a model for controller synthesis in which the controller observes the state through imprecise sensors that provide correct information about the current state with a fixed probability. That is, in each step, the sensors return an observed state, and given the observed state, there is a probability distribution (due to the estimation error) over the actual current state. The controller must base its decision on the observed state (rather than the actual current state, which it does not know). On the other hand, we assume that the environment can perfectly observe the current state. We show that controller synthesis for qualitative ω-regular objectives in our model can be reduced in polynomial time to standard partial-observation stochastic games, and vice-versa. As a consequence we establish the precise decidability frontier for the new class of games, and establish optimal complexity results for all the decidable problems.

1 Introduction

In reactive control systems, a controller interacts with its environment through sensors and actuators. The controller observes the state of the environment through a set of sensors, computes a control law that depends on the history of observed sensor readings, and feeds the computed control signal to the environment through actuators. The state of the environment is then updated as a function of the control signal as well as a disturbance signal that models external inputs to the environment. The sense-compute-actuate cycle repeats forever, resulting in an infinite trace of states. The controller synthesis problem asks to design a control law for the controller that ensures that the trace of states belongs to a given specification of "good" traces, no matter how the external disturbance behaves.

Controller synthesis has been studied extensively for deterministic games with ω-regular specifications [5,15,14]. In this setting, the problem is modeled as a game on a graph. The vertices of the graph represent system states, and are divided into "controller states" and "disturbance states." At a controller state, the controller chooses an outgoing edge and moves to a neighboring vertex along this edge. At a disturbance state, the disturbance chooses an outgoing edge and moves along this edge. If the infinite trace so obtained satisfies the specification, the controller wins; otherwise, the disturbance wins. The games are called *perfect*

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 385–399, 2012.

observation, since both players have exact knowledge of the current state and the history of the game. They model the case of perfect sensors and actuators, and where the entire state is observable through sensors.

The model of perfect-observation deterministic games have been extended to systems with *partial observation*, in which the controller can only observe part of the environment's state [16,8], and to *stochastic dynamics* [13,9,11,12], in which the state updates happen according to a probabilistic law. The standard way to model partial-observation and stochastic dynamics [8,3,2] is to extend the preceding graph model by fixing an equivalence relation on the vertices (the "observation function"), and stipulating that the controller only sees the equivalence class of the current vertex, not the particular vertex the state is in. In addition, the transitions of the graph are stochastic: the controller and the disturbance each choose some move, and the next vertex is chosen according to a probability distribution based on the current vertex and the chosen move. Partial observation models the scenario in which sensors observe only part of the state. Stochastic dynamics models imperfect actuators and probabilistic disturbances. So far, models for reactive games do not model sensor imperfections.

In this paper, we introduce a different, albeit natural, model of probabilistic uncertainty in controller synthesis motivated by imperfect sensors. Consider a state given by n bits. We assume the sensors used to measure the state are not perfect, and observing the state through the sensor results in some bits being flipped with some known probability (probabilistic noise). In contrast, we allow the disturbance to precisely observe the state as well as the observation of the controller, corresponding to a worst case assumption on the disturbance. The objective of the controller is to find a strategy that ensures that the system satisfies the specification under this probabilistic uncertainty on the current state. This model is natural in applications where the controller observes the state bits through unreliable sensing channels, where probabilistic noise in the communication channels results in bits being flipped with some known probability (according to the classical communication channel model of Shannon). Thus, the controller observes n bits through the sensor, and this estimate defines a probability distribution over the state space for the current state.

Our model (which we call *games with probabilistic uncertainty*) is inspired by analogous models of state estimation under probabilistic noise in continuous control systems. We believe this model of games with probabilistic uncertainty captures the behavior of many imperfect-sensor-based control systems, and is a formulation of the problem in a discrete setting. Intuitively, the standard model of partial-observation games represents "partial but correct information" where the controller can observe correctly only the first $k < n$ bits of the state (i.e., the observation is partial as the controller observes only a part of the state bits, but the information about the observed state bits is always correct). In contrast, our model of games with probabilistic uncertainty represents "complete but uncertain information" where the controller can observe all the n bits of the state but with uncertainty of observation (i.e., the controller can observe all the bits, but each bit is correct with some probability). Since the type of uncertain information in our model is very different from the standard models of partial-observation games studied in the literature, the relationship between them is not immediate.

Our main contribution is establishing the equivalence of the new class of games and partial-observation games. Our main technical result is a polynomial-time reduction from games with probabilistic uncertainty to standard partial-observation games, and a converse reduction from partially-observable Markov decision processes (POMDPs) to games with probabilistic uncertainty. The technical constructions that establish the equivalence of the two classes of games which represent two different notions of information (partial but correct vs complete but uncertain) are quite intricate. For example, for the new class of games the inductive definition of probability measure is subtle and different from the classical definition of probability measure for probabilistic systems [18,10]. This is because the controller observes a history that can be completely different from the actual history, whereas the environment (or disturbance) observes the actual history. We first inductively define a probability measure of observed history, given the actual history, and use it to define the required probability measure by another inductive construction. We show how our polynomial constructions for reduction capture the subtleties in the probability measure, and by establishing a precise mapping of strategies (which is at the heart of the proof of correctness of the reduction) we obtain the desired equivalence result.

In the positive direction, our reduction allows us to solve controller synthesis problems for games with probabilistic uncertainty against ω-regular specifications, using algorithms of [8,2]. In the negative direction, we get lower bounds on the hardness of problems by using known lower bounds for POMDPs using the hardness results of [1,7]. In particular, with our reductions we establish precisely the decidability frontier of games with probabilistic uncertainty for various classes of parity objectives (a canonical form to express ω-regular specifications); and for all the decidable problems we establish optimal complexity bounds (most of them EXPTIME-complete, and one PTIME-complete; see Table 1). Moreover, our reduction allows the rich body of algorithms (such as symbolic and anti-chain based algorithms [8,2]) for partial-observation games, along with any future algorithmic developments for partial-observation games, to be applicable to solve games with probabilistic uncertainty. In summary, our results provide precise decidability frontier, optimal complexity, and algorithmic solutions for games with probabilistic uncertainty, that is a natural model for control problems with state estimation under probabilistic noise. Detailed proofs, omitted due to lack of space, available in [6].

2 Games with Probabilistic Uncertainty

We now introduce games with probabilistic uncertainty, a class of games with probabilistic imperfect information.

Probability distribution. A *probability distribution* on a finite set A is a function $\kappa : A \to [0,1]$ such that $\sum_{a\in A} \kappa(a) = 1$. We denote by $\mathcal{D}(A)$ the set of probability distributions on A.

Table 1. Decidability and complexity of games with probabilistic uncertainty with parity objectives

	Sure	Almost	Positive
Safety	EXP-complete	EXP-complete	EXP-complete
Reachability	EXP-complete	EXP-complete	PTIME-complete
Büchi	EXP-complete	EXP-complete	Undec.
coBüchi	EXP-complete	Undec.	EXP-complete
Parity	EXP-complete	Undec.	Undec.

Game structures with probabilistic uncertainty. A game structure with probabilistic uncertainty consists of a tuple $\mathcal{G} = (L, \Sigma_I, \Sigma_O, \Delta, \mathsf{un})$, where (a) L is a set of *locations*; (b) Σ_I and Σ_O are two sets of input and output alphabets, respectively; (c) $\Delta : L \times \Sigma_I \times \Sigma_O \to \mathcal{D}(L)$ is a probabilistic transition function that given a location, an input and an output letter gives the probability distribution over the next locations; and (d) $\mathsf{un} : L \to \mathcal{D}(L)$ is the *probabilistic uncertainty function* that given the true current location describes the probability distribution of the observed location. If un is the identity function we obtain perfect-observation games.

Intuitively, a game proceeds as follows. The game starts at some location $\ell \in L$. Player 1 observes a state drawn from the distribution $\mathsf{un}(\ell)$, which represents a potentially faulty observation process. Intuitively, at every step the player can observe the value of all variables that corresponds to the state of the game, but there is a probability that the observed value of some variables is incorrect. Player 2 observes both the "correct" state ℓ as well as the "observed" state of Player 1, corresponding to the worst possible behavior of the adversary. Given the observation of the history of the game so far, Player 1 picks an input alphabet $\sigma^i \in \Sigma_i$. Player 2 then picks an output letter $\sigma^o \in \Sigma_o$: Player 2 observes the history of correct locations, the moves of the players, and also observes the history of observed locations of Player 1. The state of the game is updated to ℓ' with probability $\Delta(\ell, \sigma^i, \sigma^o)(\ell')$. This process is repeated ad infinitum.

Plays. A *play* of $\mathcal{G}$ is a sequence $\rho = \ell_0 \sigma_0^i \sigma_0^o \ell_1 \sigma_1^i \sigma_1^o \dots$ of locations, input letter, and output letter, such that for all $j \geq 0$ we have $\Delta(\ell_j, \sigma_j^i, \sigma_j^o)(\ell_{j+1}) > 0$. The *prefix up to* ℓ_n of the play ρ is denoted by $\rho(n)$, its *length* is $|\rho(n)| = n + 1$ and its *last element* is $\mathsf{Last}(\rho(n)) = \ell_n$. The set of plays in $\mathcal{G}$ is denoted by $\mathsf{Plays}(\mathcal{G})$, and the set of corresponding finite prefixes is denoted $\mathsf{Prefs}(\mathcal{G})$.

Strategies. A strategy for Player 1 observes the finite prefix of a play and then selects an input letter (pure strategies) or a probability distribution over input letters in Σ_i. Formally, a pure strategy for Player 1 is a function $\alpha : \mathsf{Prefs}(\mathcal{G}) \to \Sigma_i$, and a randomized strategy for Player 1 is a function $\alpha : \mathsf{Prefs}(\mathcal{G}) \to \mathcal{D}(\Sigma_i)$. Similarly, pure and randomized strategies for Player 2 are defined as functions $\beta : \mathsf{Prefs}(\mathcal{G}) \times \mathsf{Prefs}(\mathcal{G}) \times \Sigma_i \to \Sigma_o$ and $\beta : \mathsf{Prefs}(\mathcal{G}) \times \mathsf{Prefs}(\mathcal{G}) \times \Sigma_i \to \mathcal{D}(\Sigma_o)$, respectively, where the output letter is chosen based on the original history and observed history. Note that Player 2 sees Player 1's choice of input action at each step.

Outcomes. The *outcome* of two randomized strategies α for Player 1 and β for Player 2 from a location $\ell \in L$ is the set of plays $\rho = \ell_0 \sigma_0^i \sigma_0^o \dots$ such that (1) $\ell = \ell_0$, (2) there exists a sequence $\ell'_0 \ell'_1 \dots$ such that $\mathsf{un}(\ell_j)(\ell'_j) > 0$ for each $j \geq 0$, (3) for each $j \geq 0$, we have $\alpha(\ell'_0 \sigma_0^i \sigma_0^o \dots \ell'_j)(\sigma_j^i) > 0$ and $\beta(\rho(j), \ell'_0 \sigma_0^i \sigma_0^o \ell'_1 \dots \ell'_j, \sigma_j^i)(\sigma_j^o) > 0$, and $\Delta(\ell_j, \sigma_j^i, \sigma_j^o)(\ell_{j+1}) > 0$. The primed sequence $\ell'_0 \ell'_1 \dots$ gives the sequence of observations made by Player 1 using the probabilistic uncertainty function. Note that this sequence may be incorrect with some probability due to probabilistic uncertainty in the observation. We denote this set of plays as $\mathrm{Outcome}(\mathcal{G}, \ell, \alpha, \beta)$. The outcome of two pure strategies is defined analogously, considering pure strategies as degenerate randomized strategies which pick a letter with probability one. The *outcome set* of the pure (resp. randomized) strategy α for Player 1 in $\mathcal{G}$ is the set $\mathsf{Outcome}_1(\mathcal{G}, \ell, \alpha)$ of plays ρ such that there exists a pure (resp. randomized) strategy β for Player 2 with $\rho \in \mathrm{Outcome}(\mathcal{G}, \ell, \alpha, \beta)$. The outcome set $\mathsf{Outcome}_2(\mathcal{G}, \ell, \beta)$ for Player 2 is defined symmetrically.

Probability measure. Given strategies α and β, we define the probability measure $\mathrm{Pr}_{\ell_0}^{\alpha,\beta}(\cdot)$. The definition of the probability measure is subtle and non-standard as the prefix that Player 1 observes can be completely different from the original history. For a finite prefix $\rho \in \mathsf{Prefs}(\mathcal{G})$, let $\mathsf{Cone}(\rho)$ denote the set of plays with ρ as prefix. We will define $\mathrm{Pr}_{\ell_0}^{\alpha,\beta}(\cdot)$ for cones, and then by Caratheodory extension theorem [4] there is a unique extension to all measurable sets of paths. To define the probability measure we also need to define a function $\mathsf{ObsSeq}(\rho)$, that given a finite prefix ρ, gives the probability distribution over finite prefixes ρ', such that $\mathsf{ObsSeq}(\rho)(\rho')$ denotes the probability of observing ρ' given the correct prefix is ρ. The base case is as follows: $\mathrm{Pr}_{\ell_0}^{\alpha,\beta}(\mathsf{Cone}(\ell_0)) = 1$; and $\mathsf{ObsSeq}(\ell_0)(\ell') = \mathsf{un}(\ell_0)(\ell')$. The inductive definition of ObsSeq is as follows: for a prefix ρ of length $n+1$ we have $\mathsf{ObsSeq}(\rho \sigma_n^i \sigma_n^o \ell_{n+1})(\rho' \sigma_n^i \sigma_n^o \ell'_{n+1}) = \mathsf{ObsSeq}(\rho)(\rho') \cdot \mathsf{un}(\ell_{n+1})(\ell'_{n+1})$. Given a sequence $\rho = \ell_0 \sigma_0^i \sigma_0^o \ell_1 \sigma_1^i \sigma_1^o \dots \ell_n$, we define $\mathsf{AcMt}(\rho) = \{\widetilde{\rho} = \widetilde{\ell}_0 \widetilde{\sigma}_0^i \widetilde{\sigma}_0^o \ell_1 \widetilde{\sigma}_1^i \widetilde{\sigma}_1^o \dots \widetilde{\ell}_n \mid \forall 1 \leq j \leq n-1.\ \widetilde{\sigma}_j^i = \sigma_j^i \text{ and } \widetilde{\sigma}_j^o = \sigma_j^o\}$ the sequences of same length as ρ such that the sequence of input and output letter matches (i.e., the set of action-matching prefixes). Note that for non action-matching prefixes the observation sequence function always assigns probability zero. The inductive case for the probability measure is as follows: for a prefix ρ of length $n+1$ with last state ℓ_n, we have

$$\begin{aligned}&\mathrm{Pr}_{\ell_0}^{\alpha,\beta}(\mathsf{Cone}(\rho \sigma_n^i \sigma_n^o \ell_{n+1})) = \\ &\mathrm{Pr}_{\ell_0}^{\alpha,\beta}(\mathsf{Cone}(\rho)) \sum_{\rho' \in \mathsf{AcMt}(\rho)} \mathsf{ObsSeq}(\rho)(\rho') \cdot \alpha(\rho')(\sigma_n^i) \cdot \beta(\rho, \rho', \sigma_n^i)(\sigma_n^o) \cdot \Delta(\ell_n, \sigma_n^i, \sigma_n^o)(\ell_{n+1}).\end{aligned}$$

i.e., $\mathsf{ObsSeq}(\rho)(\rho')$ gives the probability to observe ρ', then $\alpha(\rho')(\sigma_n^i)$ denotes the probability to play σ_n^i given the strategy and observed sequence ρ', and since Player 2 observes the correct sequence as well as the observed sequence ρ', the probability to play σ_n^o is given by $\beta(\rho, \rho', \sigma_n^i)(\sigma_n^o)$ (Player 2 observes both ρ and ρ'), and the final term $\Delta(\ell_n, \sigma_n^i, \sigma_n^o)(\ell_{n+1})$ gives the transition probability.

Winning objectives. An *objective* for Player 1 in $\mathcal{G}$ is a set $\phi \subseteq \mathsf{Plays}(\mathcal{G})$ of plays. A play $\rho \in \mathsf{Plays}(\mathcal{G})$ *satisfies* the objective ϕ, denoted $\rho \models \phi$, if $\rho \in \phi$. We consider ω-regular objectives specified as parity objectives (a canonical form to express all ω-regular objectives [17]). For a play $\rho = \ell_0 \sigma_0^i \sigma_0^o \ldots$, we denote by ρ_k the k-th location ℓ_k of the play and denote by $\mathrm{Inf}(\rho)$ the set of locations that occur infinitely often in ρ, that is, $\mathrm{Inf}(\rho) = \{\ell \mid \forall i \exists j : j > i \text{ and } \ell_j = \ell\}$. We consider the following classes of objectives.

1. *Reachability and safety objectives.* Given a set $\mathcal{T} \subseteq L$ of target locations, the *reachability* objective $\mathsf{Reach}(\mathcal{T})$ requires that a location in $\mathcal{T}$ be visited at least once, that is, $\mathsf{Reach}(\mathcal{T}) = \{\rho \mid \exists k \geq 0 \cdot \rho_k \in \mathcal{T}\}$. Dually, the *safety* objective $\mathsf{Safe}(\mathcal{T})$ requires that only states in $\mathcal{T}$ be visited. Formally, $\mathsf{Safe}(\mathcal{T}) = \{\rho \mid \forall k \geq 0 \cdot \rho_k \in \mathcal{T}\}$.
2. *Büchi and coBüchi objectives.* Let $\mathcal{T} \subseteq L$ be a set of target locations. The *Büchi* objective $\mathsf{Buchi}(\mathcal{T})$ requires that a state in $\mathcal{T}$ be visited infinitely often, that is, $\mathsf{Buchi}(\mathcal{T}) = \{\rho \mid \mathrm{Inf}(\rho) \cap \mathcal{T} \neq \emptyset\}$. Dually, the *coBüchi* objective $\mathsf{coBuchi}(\mathcal{T})$ requires that only states in $\mathcal{T}$ be visited infinitely often. Formally, $\mathsf{coBuchi}(\mathcal{T}) = \{\rho \mid \mathrm{Inf}(\rho) \subseteq \mathcal{T}\}$.
3. *Parity objectives.* For $d \in \mathbb{N}$, let $p : L \to \{0, 1, \ldots, d\}$ be a *priority function*, which maps each state to a nonnegative integer priority. The *parity* objective $\mathsf{Parity}(p)$ requires that the minimum priority that occurs infinitely often be even. Formally, $\mathsf{Parity}(p) = \{\rho \mid \min\{p(\ell) \mid \ell \in \mathrm{Inf}(\rho)\} \text{ is even}\}$. The Büchi and coBüchi objectives are the special cases of parity objectives with two priorities, $p : L \to \{0, 1\}$ and $p : L \to \{1, 2\}$, respectively.

Sure, almost-sure and positive winning. An *event* is a measurable set of plays, and given strategies α and β for the two players, the probabilities of events are uniquely defined. For an objective ϕ, assumed to be Borel, we denote by $\mathrm{Pr}_\ell^{\alpha,\beta}(\phi)$ the probability that ϕ is satisfied by the play obtained from the starting location ℓ when the strategies α and β are used. Given a game $\mathcal{G}$, an objective ϕ, and a location ℓ, we consider the following winning modes: (1) a strategy α for Player 1 is *sure winning* for the objective ϕ from $\ell \in L$ if $\mathrm{Outcome}(\mathcal{G}, \ell, \alpha, \beta) \subseteq \phi$ for all strategies β for Player 2; (2) a strategy α for Player 1 is *almost-sure winning* for the objective ϕ from $\ell \in L$ if $\mathrm{Pr}_\ell^{\alpha,\beta}(\phi) = 1$ for all strategies β for Player 2; and (3) a strategy α for Player 1 is *positive winning* for the objective ϕ from $\ell \in L$ if $\mathrm{Pr}_\ell^{\alpha,\beta}(\phi) > 0$ for all strategies β for Player 2.

Qualitative analysis of a game consists of the computation of the sure, almost-sure and positive winning sets. The sure (resp. almost-sure and positive) winning decision problem for an objective consists of a game and a starting location ℓ, and asks whether there is a sure (resp. almost-sure and positive) winning strategy from ℓ.

3 Partial-Observation Stochastic Games

We now recall the usual definition of partial-observation games and their subclasses. We focus on partial-observation turn-based probabilistic games, where at each round one of the players is in charge of choosing the next action and the transition function is probabilistic. We will present a polynomial time reduction of games with probabilistic uncertainty to these games.

Partial-observation games. A *partial-observation stochastic game* (for short partial-observation game or simply a *game*) is a tuple $G = \langle S_1 \cup S_2, A_1, A_2, \delta_1 \cup \delta_2, \mathcal{O}_1, \mathcal{O}_2 \rangle$ with the following components:

1. *(State space).* $S = S_1 \cup S_2$ is a finite set of states, where $S_1 \cap S_2 = \emptyset$ (i.e., S_1 and S_2 are disjoint), states in S_1 are Player 1 states, and states in S_2 are Player 2 states.
2. *(Actions).* A_i $(i = 1, 2)$ is a finite set of actions for Player i.
3. *(Transition function).* For $i \in \{1, 2\}$, the probabilistic transition function for Player i is the function $\delta_i : S_i \times A_i \to \mathcal{D}(S_{3-i})$ that maps a state $s_i \in S_i$ and an action $a_i \in A_i$ to the probability distribution $\delta_i(s_i, a_i)$ over the successor states in S_{3-i} (i.e., games are alternating).
4. *(Observations).* $\mathcal{O}_1 \subseteq 2^S$ is a finite set of observations for Player 1 that partitions the state space S, and similarly $\mathcal{O}_2$ is the observations for Player 2. These partitions uniquely define functions $\mathsf{obs}_i : S \to \mathcal{O}_i$, for $i \in \{1, 2\}$, that map each state to its observation such that $s \in \mathsf{obs}_i(s)$ for all $s \in S$. We will only focus on the special case where Player 2 is perfectly informed (has complete observation) and refer them as *one-sided* games, i.e., $\mathcal{O}_2 = S$, and $\mathsf{obs}_2(s) = s$ for all $s \in S$ (i.e., the partition consists of singleton states).

Special Class: POMDPs. We will consider one special class of one-sided partial-observation games called *partial-observable Markov decision processes* (POMDPs), where the action set for Player 2 is a singleton (i.e., there is effectively only Player 1 and stochastic transitions). Hence we will omit the action set and observation for Player 2 and represent a POMDP as the following tuple $G = \langle S, A, \delta, \mathcal{O} \rangle$, where $\delta : S \times A \to \mathcal{D}(S)$.

Plays. In a game, in each turn, for $i \in \{1, 2\}$, if the current state s is in S_i, then Player i chooses an action $a \in A_i$, and the successor state is chosen by sampling the probability distribution $\delta_i(s, a)$. A *play* in G is an infinite sequence of states and actions $\rho = s_0 a_0 s_1 a_1 \ldots$ such that for all $j \geq 0$, if $s_j \in S_i$, for $i \in \{1, 2\}$, then $a_j \in A_i$ such that $\delta_i(s_j, a_j)(s_{j+1}) > 0$. The definitions of prefix and length are analogous to the definitions in Section 2. For $i \in \{1, 2\}$, we denote by $\mathsf{Prefs}_i(G)$ the set of finite prefixes in G that end in a state in S_i. The *observation sequence* of $\rho = s_0 a_0 s_1 a_1 \ldots$ for Player i $(i = 1, 2)$ is the unique infinite sequence of observations and actions, i.e., $\mathsf{obs}(\rho) = o_0 a_0 o_1 a_1 o_2 \ldots$ such that $s_j \in o_j$ for all $j \geq 0$. The observation sequence for finite sequences (prefix of plays) is defined analogously.

Strategies. A *pure strategy* in G for Player 1 is a function $\alpha : \mathsf{Prefs}_1(G) \to A_1$. A *randomized strategy* in G for Player 1 is a function $\alpha : \mathsf{Prefs}_1(G) \to \mathcal{D}(A_1)$. A (pure or randomized) strategy α for Player 1 is *observation-based* if for all prefixes $\rho, \rho' \in \mathsf{Prefs}_1(G)$, if $\mathsf{obs}(\rho) = \mathsf{obs}(\rho')$, then $\alpha(\rho) = \alpha(\rho')$. We omit analogous definitions of strategies for Player 2. We denote by $\mathcal{A}_G$, $\mathcal{A}_G^O$, $\mathcal{A}_G^P$, $\mathcal{B}_G$, $\mathcal{B}_G^O$, $\mathcal{B}_G^P$ the set of all Player-1 strategies in G, the set of all observation-based Player-1 strategies, the set of all pure Player-1 strategies, the set of all Player-2 strategies in G, the set of all observation-based Player-2 strategies, and the set of all pure Player-2 strategies, respectively. In the setting where Player 1 has partial-observation and Player 2 has complete observation, the set $\mathcal{B}_G$ of all strategies coincides with the set $\mathcal{B}_G^O$ of all observation-based strategies. We will require the players to play observation-based strategies.

Outcomes. The *outcome* of two randomized strategies α (for Player 1) and β (for Player 2) from a state s in G is the set of plays $\rho = s_0 a_0 s_1 a_1 \ldots \in \mathsf{Plays}(G)$, with $s_0 = s$, where for all $j \geq 0$, if $s_j \in S_1$ (resp. $s_j \in S_2$), then $\alpha(\rho(j))(a_j) > 0$ (resp. $\beta(\rho(j))(a_j) > 0$) and $\delta_1(s_j, a_j)(s_{j+1}) > 0$ (resp. $\delta_2(s_j, a_j)(s_{j+1}) > 0$). This set is denoted $\mathrm{Outcome}(G, s, \alpha, \beta)$. The outcome of two pure strategies is defined analogously by viewing pure strategies as randomized strategies that play their chosen action with probability one. The *outcome set* of the pure (resp. randomized) strategy α for Player 1 in G is the set $\mathsf{Outcome}_1(G, s, \alpha)$ of plays ρ such that there exists a pure (resp. randomized) strategy β for Player 2 with $\rho \in \mathrm{Outcome}(G, s, \alpha, \beta)$. The outcome set $\mathsf{Outcome}_2(G, s, \beta)$ for Player 2 is defined symmetrically.

Probability measure. We define the probability measure $\mathrm{Pr}_s^{\alpha,\beta}(\cdot)$ as follows: for a finite prefix ρ, let $\mathsf{Cone}(\rho)$ denote the set of plays with ρ as prefix. Then we have $\mathrm{Pr}_s^{\alpha,\beta}(\mathsf{Cone}(s)) = 1$, and for a prefix of length n ending in a Player 1 state s_n we have $\mathrm{Pr}_s^{\alpha,\beta}(\mathsf{Cone}(\rho a_n s_{n+1})) = \mathrm{Pr}_s^{\alpha,\beta}(\mathsf{Cone}(\rho)) \cdot \alpha(\rho)(a_n) \cdot \delta_1(s_n, a_n)(s_{n+1})$; and the definition when s_n is a Player 2 state is similar. For a set Q of finite prefixes, we write $\mathrm{Pr}_s^{\alpha,\beta}(\mathsf{Cone}(Q))$ for $\mathrm{Pr}_s^{\alpha,\beta}(\bigcup_{\rho \in Q} \mathsf{Cone}(\rho))$.

The winning modes sure, almost-sure, and positive are defined analogously to Section 2, where we restrict the players to play an observation-based strategy. From the results of [8,2,1,3,7] we obtain the following theorem summarizing the results for partial-observation games and POMDPs.

Theorem 1 ([8,2,1,3,7]). *The following assertions hold:*

1. (One-sided games and POMDPs). *The sure, almost-sure, and positive winning problems for safety objectives; the sure and almost-sure winning problems for reachability objectives and Büchi objectives; the sure and positive winning problems for coBüchi objectives; and the sure winning problem for parity objectives are EXPTIME-complete for one-sided partial-observation games (Player 2 perfectly informed) and POMDPs. The positive winning problem for reachability objectives is PTIME-complete both for one-sided partial-observation games and POMDPs.*
2. (Undecidability results). *The positive winning problem for Büchi objectives, the almost-sure winning problem for coBüchi objectives, and the positive and almost-sure winning problems for parity objectives are undecidable for POMDPs.*

4 Reduction: Games with Probabilistic Uncertainty to Partial-Observation Games

We now present a reduction of games with probabilistic uncertainty to one-sided partial-observation games. Let $G = (L, \Sigma_I, \Sigma_O, \Delta, \mathsf{un})$ be a game with probabilistic uncertainty and we construct a one-sided partial-observation game $H = (L \times L \cup L \times L \times \Sigma_I, A_1 = \Sigma_I, A_2 = \Sigma_O, \delta = \delta_1 \cup \delta_2, \mathcal{O}_1, \mathcal{O}_2)$ as follows:

1. The transition function δ_1 is deterministic and for $(\ell_1, \ell_2) \in L \times L$ and $\sigma_I \in \Sigma_I$ we have $\delta((\ell_1, \ell_2), \sigma_I) = (\ell_1, \ell_2, \sigma_I)$.
2. The transition function δ_2 captures both Δ and un and is defined as follows: for $(\ell_1, \ell_2, \sigma_I) \in L \times L \times \Sigma_I$ and $\sigma_O \in \Sigma_O$ we have $\delta((\ell_1, \ell_2, \sigma_I), \sigma_O)(\ell_1', \ell_2') = \Delta(\ell_1, \sigma_I, \sigma_O)(\ell_1') \cdot \mathsf{un}(\ell_1')(\ell_2')$. Intuitively, the first component of the game H keeps track of the real state of the game G, and the second component keeps track of the information available from probabilistic uncertainty. Hence Player 1 is only allowed to observe the second component which is the probability distribution over the observable state given the current state.
3. The observation mapping is as follows: we have $\mathcal{O}_1 = L$; and $\mathsf{obs}_1(\ell_1, \ell_2) = \mathsf{obs}_1(\ell_1, \ell_2, \sigma_I) = \ell_2$, i.e., only the second component is observable. Player 2 has perfect observation.
4. For a parity objective in G given by priority function $p_G : L \to \{0, 1, \ldots, d\}$, we consider the priority function p_H in H as follows: $p_H((\ell, \ell')) = p_H((\ell, \ell', \sigma_I)) = p_G(\ell)$, for all $\ell, \ell' \in L$ and $\sigma_I \in \Sigma_I$.

Correspondence of strategies. We will now establish the correspondence of probabilistic uncertain strategies in G and the observation based strategies in H. We present a few notations. For simplicity of presentation, we will use a slight abuse of notation: given a history (or finite prefix) $\rho_H = s_0 a_0 s_1 a_1 s_2 a_2 \ldots s_{2n}$ in H we will represent the history as $s_0 a_0 a_1 s_2 a_2 a_3 s_3 \ldots s_{2n}$ as the intermediate state is always uniquely defined by the state and the action. Intuitively this is removing the stuttering and does not affect parity objectives.

Mapping of strategies from G to H. Given a history $\rho_H = s_0 a_0 a_1 s_2 a_2 a_3 s_3 \ldots s_{2n}$ in H, such that $s_{2i} = (\ell_{2i}^1, \ell_{2i}^2)$, we consider two histories in G as follows: (i) $g_1(\rho_H) = \ell_0^1 a_0 a_1 \ell_2^1 a_2 a_3 \ldots \ell_{2n}^1$; and (ii) $g_2(\rho_H) = \ell_0^2 a_0 a_1 \ell_2^2 a_2 a_3 \ldots \ell_{2n}^2$. Intuitively, g_1 gives the first component (which is the correct history) and g_2 gives the second component (which is the observed history). We now define the mapping of strategies from G to H: given strategy α_G for Player 1, a strategy β_G for Player 2, in the game G, we define the corresponding strategies in H as follows: for a history ρ_H and an action a_i for Player 1 we have: (i) $\alpha_H(\rho_H) = \alpha_G(g_2(\rho_H))$; (ii) $\beta_H(\rho_H\ a_i) = \beta_G(g_1(\rho_H), g_2(\rho_H), a_i)$. Note that α_H is an observation-based strategy, and β_H is a strategy with complete-observation. We will use $\widehat{g}$ to denote the mapping of strategies, i.e., $\alpha_H = \widehat{g}(\alpha_G)$ and $\beta_H = \widehat{g}(\beta_G)$.

Mapping of strategies from H to G. We now present the mapping in the other direction. Let $\rho_G^1 = \ell_0^1 \sigma_0^i \sigma_0^o \ell_1^1 \sigma_1^i \sigma_1^o \ldots \ell_n^1$, and $\rho_G^2 = \ell_0^2 \sigma_0^i \sigma_0^o \ell_1^2 \sigma_1^i \sigma_1^o \ldots \ell_n^2$ be two prefixes in G. Intuitively, the first represent the correct history and the second the observed history. Then we consider the following set of histories in H: (i) $h_1(\rho_G^1) = \{\rho_H \mid g_1(\rho_H) = \rho_G^1\}$; and (ii) $h_2(\rho_G^2) = \{\rho_H \mid g_2(\rho_H) = \rho_G^2\}$; and (iii) $h_{12}(\rho_G^1, \rho_G^2) = (\ell_0^1, \ell_0^2) \sigma_0^i \sigma_0^o (\ell_1^1, \ell_1^2) \sigma_1^i \sigma_1^o \ldots (\ell_n^1, \ell_n^2)$. We now define the mapping of strategies. Given an observation-based strategy $\alpha_H \in \mathcal{A}_H^O$ for Player 1, and a complete observation-based strategy $\beta_H \in \mathcal{B}_H$, we define the following strategies in G: for a correct history ρ_G^1, observed history ρ_G^2, and input σ^i we have: (i) $\alpha_G(\rho_G^2) = \alpha_H(\rho_H)$ for $\rho_H \in h_2(\rho_G^2)$; and (ii) $\beta_G(\rho_G^1, \rho_G^2, \sigma^i) =$

$\beta_H(h_{12}(\rho_G^1, \rho_G^2), \sigma_i)$. Note that since α_H is observation-based it plays the same for all $\rho_H \in h_2(\rho_G^2)$. We will use $\widehat{h}$ to denote the mapping of strategies, i.e., $\alpha_G = \widehat{h}(\alpha_H)$ and $\beta_G = \widehat{h}(\beta_H)$.

Given a starting state $\ell_0 \in G$, consider the following probability distribution μ in H: $\mu(\ell_0, \ell) = \mathsf{un}(\ell_0)(\ell)$. Given the mapping of strategies, our goal is to establish the equivalences of the probability measure. We introduce some notations required to establish the equivalence. For $j \geq 0$, we denote by (τ_j^1, τ_j^2) the pair of random variables to denote the j-th Player 1 state of the game H, and by θ_j^i and θ_j^o the random variables for the actions following the j-th state. Our first lemma establishes a connection of the probability of observing the second component in H given the first component along with function ObsSeq. We introduce notations to define two events: given two prefixes $\rho_G^1 = \ell_0^1\sigma_0^i\sigma_0^o\ell_1^1\sigma_1^i\sigma_1^o \dots \ell_n^1$, and $\rho_G^2 = \ell_0^2\sigma_0^i\sigma_0^o\ell_1^2\sigma_1^i\sigma_1^o \dots \ell_n^2$ in G, let $\mathcal{E}_{1,2}(\rho_G^1, \rho_G^2)$ denote the event that for all $0 \leq j \leq n$ we have $\tau_j^1 = \ell_j^1, \tau_j^2 = \ell_j^2$ and for all $0 \leq j \leq n-1$ we have $\theta_j^i = \sigma_j^i, \theta_j^o = \sigma_j^o$; and $\mathcal{E}_1(\rho_G^1)$ denote the event that for all $0 \leq j \leq n$ we have $\tau_j^1 = \ell_j^1$ and for all $0 \leq j \leq n-1$ we have $\theta_j^i = \sigma_j^i, \theta_j^o = \sigma_j^o$.

Lemma 1. *Let $\rho_G^1 = \ell_0^1\sigma_0^i\sigma_0^o\ell_1^1\sigma_1^i\sigma_1^o \dots \ell_n^1$, and $\rho_G^2 = \ell_0^2\sigma_0^i\sigma_0^o\ell_1^2\sigma_1^i\sigma_1^o \dots \ell_n^2$ be two prefixes in G. Then for all strategies α_H and β_H, the probability that the second component sequence in H is ρ_G^2, given the first component sequence is ρ_G^1 is $\mathsf{ObsSeq}(\rho_G^1)(\rho_G^2)$, i.e., formally $\Pr_\mu^{\alpha_H,\beta_H}(\mathcal{E}_{1,2}(\rho_G^1, \rho_G^2) \mid \mathcal{E}_1(\rho_G^1)) = \mathsf{ObsSeq}(\rho_G^1)(\rho_G^2)$.*

Lemma 2 using Lemma 1 and an inductive argument establishes the equivalences of the probabilities of the cones.

Lemma 2. *For all finite prefixes ρ_G^1 in G, the following assertions hold: (1) For all strategies α_G and β_G, we have (i) $\Pr_{\ell_0}^{\alpha_G,\beta_G}(\mathsf{Cone}(\rho_G^1)) = \Pr_\mu^{\widehat{g}(\alpha_G),\widehat{g}(\beta_G)}(\mathsf{Cone}(h_1(\rho_G^1)))$. (2) For all strategies α_H and β_H, we have (i) $\Pr_{\ell_0}^{\widehat{h}(\alpha_H),\widehat{h}(\beta_H)}(\mathsf{Cone}(\rho_G^1)) = \Pr_\mu^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1)))$.*

Proof. We will present the result for the first item, and the proof for second item is identical. Let us denote by $\alpha_H = \widehat{g}(\alpha_G)$ and $\beta_H = \widehat{g}(\beta_G)$. We will prove the result by induction on the length of the prefixes. The base case is as follows: let the length of the prefix ρ_G^1 be 1, with $\rho_G^1 = \ell_0$. We observe that $\Pr_{\ell_0}^{\alpha_G,\beta_G}(\mathsf{Cone}(\ell_0)) = 1$, and $\Pr_\mu^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\ell_0))) = 1$, and for all other cones of length 1 the probability is zero. This completes the base case.

We now consider the inductive case: by inductive hypothesis we assume that $\Pr_{\ell_0}^{\alpha_G,\beta_G}(\mathsf{Cone}(\rho_G^1)) = \Pr_\mu^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1)))$; and show that

$$\Pr_{\ell_0}^{\alpha_G,\beta_G}(\mathsf{Cone}(\rho_G^1 a_n b_n \ell_{n+1})) = \Pr_\mu^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1 a_n b_n \ell_{n+1}))).$$

Let ℓ_n be the last state of ρ_G^1. We first consider the left-hand side (LHS), and for brevity in presentation let us denote $\mathsf{Term} = \alpha_G(\rho')(a_n) \cdot \beta_G(\rho_G^1\rho' a_n)(b_n) \cdot \mathit{\Delta}(\ell_n, a_n, b_n)(\ell_{n+1})$

$$\begin{aligned}
&\mathrm{Pr}\,{}_{\ell_0}^{\alpha_G,\beta_G}\;(\mathsf{Cone}(\rho_G^1 a_n b_n \ell_{n+1}))\\
&\quad= \mathrm{Pr}_{\ell_0}^{\alpha_G,\beta_G}(\mathsf{Cone}(\rho_G^1))\cdot\left(\sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathsf{ObsSeq}(\rho_G^1)(\rho')\cdot\mathsf{Term}\right)\\
&\quad= \mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1)))\cdot\left(\sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathsf{ObsSeq}(\rho_G^1)(\rho')\cdot\mathsf{Term}\right)\\
&\quad= \sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_{12}(\rho_G^1,\rho')))\cdot\mathsf{Term}
\end{aligned}$$

Above the first equality is by definition, the second equality by inductive hypothesis, and the last equality is obtained as follows: by Lemma 1 we have $\mathsf{ObsSeq}(\rho_G^1)(\rho') = \mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathcal{E}_{1,2}(\rho_G^1,\rho') \mid \mathcal{E}_1(\rho_G^1))$, and hence

$$\begin{aligned}
&\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1)))\cdot\sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathsf{ObsSeq}(\rho_G^1)(\rho')\\
&\quad= \sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1)))\cdot\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathcal{E}_{1,2}(\rho_G^1,\rho')\mid\mathcal{E}_1(\rho_G^1))\\
&\quad= \sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_{12}(\rho_G^1,\rho'))).
\end{aligned}$$

We now consider the right-hand side (RHS) $\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_1(\rho_G^1 a_n b_n \ell_{n+1})))$ and the RHS can be expanded as: (below for brevity we write $\widehat{\rho} = h_{12}(\rho_G^1,\rho')$)

$$\sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\sum_{\ell'_{n+1}}\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(\widehat{\rho}))\cdot\alpha_H(\widehat{\rho})(a_n)\cdot\beta_H(\widehat{\rho}a_n)(b_n)\cdot\delta((\ell_n,\ell'_n,a_n),b_n)(\ell_{n+1},\ell'_{n+1})$$

Since we have

$$\alpha_H(h_{12}(\rho_G^1,\rho'))(a_n) = \alpha_G(\rho')(a_n);\;\text{ and }\;\beta_H(h_{12}(\rho_G^1,\rho')a_n)(b_n) = \beta_G(\rho_G^1 a_n)(b_n),$$

the above expression for RHS is equivalently described as:

$$\sum_{\rho'\in\mathsf{AcMt}(\rho_G^1)}\sum_{\ell'_{n+1}}\mathrm{Pr}_{\mu}^{\alpha_H,\beta_H}(\mathsf{Cone}(h_{12}(\rho_G^1,\rho')))\cdot\mathsf{Term}\cdot\mathsf{un}(\ell_{n+1})(\ell'_{n+1})$$

Since $\sum_{\ell'_{n+1}}\mathsf{un}(\ell_{n+1})(\ell'_{n+1}) = 1$, it follows that LHS is equal to the RHS. This completes the proof and the desired result follows. ∎

It follows that there is a sure, almost-sure, positive winning strategy in G for $\mathsf{Parity}(p_G)$ iff there is a corresponding one in H for $\mathsf{Parity}(p_H)$ and hence from Theorem 1 we obtain the following result.

Theorem 2. *The sure, almost-sure, and positive winning problems for safety objectives; the sure and almost-sure winning problems for reachability objectives and Büchi objectives; the sure and positive winning problems for coBüchi objectives; and the sure winning problem for parity objectives can be solved in EXPTIME for games with probabilistic uncertainty. The positive winning problem for reachability objectives can be solved in PTIME.*

5 Reduction: POMDPs to Games with Probabilistic Uncertainty

In this section we present a reduction in the reverse direction and show that POMDPs with parity objectives can be reduced to games with probabilistic uncertainty and parity objectives. We first present the reduction and then show the correctness of the reduction by mapping prefixes, strategies, and establishing the equivalence of the probability measure.

Reduction: POMDPs to games with probabilistic uncertainty. Let $H = (S, A, \delta, \mathcal{O})$ be a POMDP with a parity objective ϕ, we construct the game of probabilistic uncertainty $G = (L, \Sigma_I, \Sigma_O, \Delta, \mathsf{un})$ as follows: (i) $L = S$; (ii) $\Sigma_I = A$; (iii) $\Sigma_O = \{\bot\}$; (iv) for $\ell \in L$ and $a \in \Sigma_I$ let $\Delta(\ell, a, \bot)(\ell') = \delta(\ell, a)(\ell')$, i.e., the transition function is same as the transition function of the POMDP. In other words, the state space is the same, the action choices of the POMDP corresponds to the input action choice, and the output action set is singleton, and the transition function mimics the transition function of the POMDP. Below we use the probabilistic uncertainty to capture the partial-observation of the POMDP. The uncertainty function is as follows: $\mathsf{un}(\ell)(\ell') = 0$ if $\mathsf{obs}(\ell) \neq \mathsf{obs}(\ell')$ and $\frac{1}{|\mathsf{obs}(\ell)|}$ if $\mathsf{obs}(\ell) = \mathsf{obs}(\ell')$. Finally, the parity objective is the same as the original parity objective.

Mapping of prefixes. Given a prefix (or a finite history) $\rho_H = s_0 a_0 s_1 a_1 s_2 \dots s_n$ in H we construct a prefix in G as $\rho_G = s_0 a_0 \bot s_1 a_1 \bot s_2 \dots s_n$ by simply inserting the $\bot$ actions. This construction defines a bijection $h : \mathsf{Prefs}_H \to \mathsf{Prefs}_G$ between prefixes. We can naturally extend the mapping to sets of prefixes. Let $\Psi \subseteq \mathsf{Prefs}_H$, then $h'(\Psi) = \{h(\rho) \mid \rho \in \Psi\}$.

Lemma 3. *For prefixes ρ, ρ' in G we have* $\mathsf{ObsSeq}(\rho)(\rho') = \frac{1}{\prod_{i=1}^{n} |o_i|}$ *if* $\mathsf{obs}(h^{-1}(\rho)) = \mathsf{obs}(h^{-1}(\rho')) = o_1 a_1 o_2 \dots a_{n-1} o_n$*; and 0 otherwise.*

Mapping of strategies. We first present the mapping of strategies from H to G and then from G to H. Note that in the game G, there is no choice for Player 2, and hence we remove the Player 2 strategies in the descriptions below.

Mapping strategies from H to G. Let α_H be an observation-based Player-1 strategy in H and $\rho_G = s_0 a_0 \bot s_1 a_1 \bot s_2 \dots s_n$ be a prefix in G. We define a Player-1 strategy α_G in G as follows: $\alpha_G(\rho_G) = \alpha_H(h^{-1}(\rho_G))$.

Mapping strategies from G to H. Let α_G be a Player-1 strategy in G and $\rho_H = s_0 a_0 s_1 a_1 s_2 \dots s_n$ be a prefix in H with $o = o_0 a_0 o_1 a_1 o_2 \dots o_n$ as its observation sequence. Note that as Player 2 has only one strategy (always playing $\bot$) we omit it from discussion. Note that every $\rho \in \mathsf{AcMt}(h(\rho_H))$ can have different actions with different probabilities enabled. We define a Player 1 strategy α_H in H as follows: for an action $a \in A$ we have $\alpha_H(\rho_H)(a) = \sum_{\rho' \in \mathsf{AcMt}(h(\rho_H))} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_G(\rho')(a)$. Using Lemma 3 we have that ObsSeq only depends on the observation sequence, and thus we obtain Lemma 4.

Lemma 4. *The strategy α_H obtained from strategy α_G is an observation-based strategy for Player 1 in H.*

Correspondence of probabilities. In the following two lemmas we establish the correspondence of the probabilities for the mappings.

Lemma 5. *Consider the mapping of strategies from H to G. For all prefixes ρ_H in H we have* $\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H)) = \Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H)))$.

Proof. The proof is based on induction on the length of the prefix ρ_H. We denote the last state of ρ_H by ℓ_n.

Base case. For prefixes of length 1 where $\rho_H = \ell_0$ we get $\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\ell_0)) = 1$ and $\Pr_{l_0}^{\alpha_G}(\mathsf{Cone}(h(\ell_0))) = 1$. For all other prefixes both sides are equal to 0. Hence the base case follows.

Inductive step. By inductive hypothesis we assume the result for prefixes ρ_H of length n (i.e., we assume that $\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H)) = \Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H))))$ and will show that

$$\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H a_n \ell_{n+1})) = \Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H a_n \ell_{n+1}))).$$

First we expand the left hand side (LHS) and by definition we get that:

$$\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H a_n \ell_{n+1})) = \Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H)) \cdot \alpha_H(\rho_H)(a_n) \cdot \delta(\ell_n, a_n)(\ell_{n+1}).$$

We now expand the right hand side (RHS) and get that:

$$\Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H a_n \ell_{n+1}))) =$$

$$\Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H))) \cdot \sum_{\rho' \in \mathsf{AcMt}(h(\rho_H))} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_G(\rho')(a_n) \cdot \varDelta(\ell_n, a_n, \perp)(\ell_{n+1})$$

Using inductive hypothesis, the definition of the game, and the mapping of strategies we get on RHS:

$$\Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H a_n \ell_{n+1}))) =$$

$$\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H)) \cdot \sum_{\rho' \in \mathsf{AcMt}(h(\rho_H))} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_H(h^{-1}(\rho'))(a_n) \cdot \delta(\ell_n, a_n)(\ell_{n+1})$$

For all ρ' that does not match the observation sequence of $h(\rho_H)$, we have $\mathsf{ObsSeq}(h(\rho_H))(\rho') = 0$ (by Lemma 3), and as α_H is observation based for all $\rho' \in \mathsf{AcMt}(\rho_H)$ that matches the observation sequence of $h(\rho_H)$, the strategy α_H plays the same. Let us denote by $\rho' \approx h(\rho_H)$ that ρ' matches the observation sequence of $h(\rho_H)$. Then we have

$$\begin{aligned}
&\sum_{\rho' \in \mathsf{AcMt}(h(\rho_H))} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_H(h^{-1}(\rho'))(a_n) \\
&\quad = \sum_{\rho' \in \mathsf{AcMt}(h(\rho_H)), \rho' \approx h(\rho_H)} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_H(h^{-1}(\rho'))(a_n) \\
&\quad = \sum_{\rho' \in \mathsf{AcMt}(h(\rho_H)), \rho' \approx h(\rho_H)} \mathsf{ObsSeq}(h(\rho_H))(\rho') \cdot \alpha_H(\rho_H)(a_n) \\
&\quad = \alpha_H(\rho_H)(a_n);
\end{aligned}$$

where the first equality follows as for all sequences ρ' that does not match the observation sequence of $h(\rho_H)$ we have $\mathsf{ObsSeq}(h(\rho_H))(\rho') = 0$; the second equality follows as for all $\rho' \approx h(\rho_H)$ we have $\alpha_H(h^{-1}(\rho'))(a_n) = \alpha_H(\rho_H)(a_n)$ (as α_H is observation based); and the last equality follows because as ObsSeq is a probability distribution we have $\sum_{\rho' \in \mathsf{AcMt}(h(\rho_H)), \rho' \approx h(\rho_H)} \mathsf{ObsSeq}(h(\rho_H))(\rho') = 1$. Hence we have

$$\Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(h(\rho_H a_n \ell_{n+1}))) = \Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(\rho_H)) \cdot \alpha_H(\rho_H)(a_n) \cdot \delta(\ell_n, a_n)(\ell_{n+1})$$

Thus we have that LHS and RHS coincide and this completes the proof. ∎

Lemma 6. *Let us consider the mapping of strategies from G to H. For all prefixes ρ_G in G we have $\Pr_{\mu}^{\alpha_H}(\mathsf{Cone}(h^{-1}(\rho_G))) = \Pr_{\ell_0}^{\alpha_G}(\mathsf{Cone}(\rho_G))$.*

The proof of Lemma 6 is similar to the proof of Lemma 5. The previous two lemmas establish the equivalence of the probability measure and completes the reduction of POMDPs to games with probabilistic uncertainty. Hence the lower bounds for POMDPs also gives us the lower bound for games with probabilistic uncertainty. Hence Theorem 2, along with the reduction from POMDPs and Theorem 1 gives us the following result for games with probabilistic uncertainty (the results are also summarized in Table 1).

Theorem 3. *The following assertions hold:*

1. (Complexity results). *The sure, almost-sure, and positive winning problems for safety objectives; the sure and almost-sure winning problems for reachability and Büchi objectives; the sure and positive winning problems for coBüchi objectives; and the sure winning problem for parity objectives are all EXPTIME-complete for games with probabilistic uncertainty. The positive winning problem for reachability objectives is PTIME-complete.*
2. (Undecidability results). *The positive winning problem for Büchi objectives, the almost-sure winning problem for coBüchi objectives, and the positive and almost-sure winning problems for parity objectives are undecidable for games with probabilistic uncertainty.*

6 Conclusion

We considered games with probabilistic uncertainty, a natural model for control under sensing uncertainties. We present a reduction of such games to classical partial-observation games and a reduction of POMDPs to such games. As a consequence we establish the precise decidability frontier and optimal complexities for games with probabilistic uncertainty (Table 1).

Acknowledgements. The research was supported by Austrian Science Fund (FWF) Grant No P 23499-N23 on Modern Graph Algorithmic Techniques in Formal Verification, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award.

References

1. 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)
2. Bertrand, N., Genest, B., Gimbert, H.: Qualitative determinacy and decidability of stochastic games with signals. In: LICS, pp. 319–328. IEEE Computer Society (2009)
3. Berwanger, D., Doyen, L.: On the power of imperfect information. In: FSTTCS, Dagstuhl Seminar Proceedings 08004. IBFI (2008)
4. Billingsley, P.: Probability and Measure. Wiley-Interscience (1995)
5. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the AMS 138, 295–311 (1969)
6. Chatterjee, K., Chmelik, M., Majumdar, R.: Equivalence of games with probabilistic uncertainty and partial-observation games. CoRR, abs/1202.4140 (2012)
7. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative Analysis of Partially-Observable Markov Decision Processes. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010)
8. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games of incomplete information. LMCS 3 (2007)
9. Condon, A.: The complexity of stochastic games. I. & C. 96(2), 203–224 (1992)
10. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
11. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. TCS 386(3), 188–217 (2007)
12. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. In: STOC 2001, pp. 675–683. ACM Press (2001)
13. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer (1997)
14. Kupferman, O., Vardi, M.Y.: μ-Calculus Synthesis. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 497–507. Springer, Heidelberg (2000)
15. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. Conference Series in Mathematics, vol. 13. American Mathematical Society (1969)
16. Reif, J.H.: Universal games of incomplete information. In: STOC, pp. 288–308. ACM Press (1979)
17. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages. Beyond Words, vol. 3, ch. 7, pp. 389–455. Springer (1997)
18. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state systems. In: FOCS 1985, pp. 327–338. IEEE Computer Society Press (1985)

A Probabilistic Kleene Theorem*

Benedikt Bollig[1], Paul Gastin[1], Benjamin Monmege[1], and Marc Zeitoun[2]

[1] LSV, ENS Cachan, CNRS & Inria, France
firstname.lastname@lsv.ens-cachan.fr
[2] LaBRI, Univ. Bordeaux & CNRS, France
mz@labri.fr

Abstract. We provide a Kleene Theorem for (Rabin) probabilistic automata over finite words. Probabilistic automata generalize deterministic finite automata and assign to a word an acceptance probability. We provide probabilistic expressions with probabilistic choice, guarded choice, concatenation, and a star operator. We prove that probabilistic expressions and probabilistic automata are expressively equivalent. Our result actually extends to two-way probabilistic automata with pebbles and corresponding expressions.

1 Introduction

Kleene's Theorem states the equivalence of rational and recognizable languages in the free monoids. Naturally, this fundamental result has been generalized to various settings and in particular to quantitative extensions of classical languages, called formal power series [24,23,4].

The present paper aims at a probabilistic counterpart of Kleene's Theorem. There are actually a variety of models for probabilistic systems, comprising Segala systems, generative systems, stratified systems, Markov chains, etc. (see [29,25] for overviews). Those models may involve non-determinism and *generate* some behavior according to probability distributions over states. Alternatively, they may make a probabilistic decision depending on the input letter, like (reactive) probabilistic automata [20]. The latter go back to Rabin [21] and are an object of ongoing research considering decision problems such as emptiness, language equivalence [24,28,9,10,18], and the value 1 problem [17].

Our starting point of view is that expressions and automata shall represent quantitative properties of words. In particular, rather than at bisimulation equivalence, we are looking at language equivalence in terms of formal power series (i.e., mappings from strings to elements from the real-valued interval $[0, 1]$). This actually has an immediate impact on the choice of both the automaton model and the syntax of expressions that are supposed to characterize it. On the automata side, a probabilistic decision should depend on the *given* input. Therefore, we consider probabilistic automata. On the specification side, we would like to adopt concepts from rational expressions. In this paper, we actually provide a

* Supported by ANR 2010 BLAN 0202 01 FREC, LIA INFORMEL.

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 400–415, 2012.

simple fragment of classical weighted rational expressions over the nonnegative real numbers, including a star operator and concatenation. The star operator has to be handled with care, though. It comes with a subtle restriction to make sure that an expression associates with every word a probability. In this way, we obtain a class of probabilistic expressions that have the same expressive power as probabilistic automata. Translations forth and back are effective so that decidability results for automata directly carry over to expressions.

Actually, we prove a more general result. Expressions are extended in such a way that they capture two-way probabilistic automata [14,22] and automata with pebbles (similar to two-way word automata and tree-walking automata). Our expressions can then be considered as a probabilistic generalization of XPath [19,27]. Note that (non-probabilistic) two-way automata are in fact an appropriate machine model for compiling XPath queries [5]. The concept presented in this paper may therefore constitute a first step towards probabilistic database query languages: an expression is considered as a query, and an equivalent automaton can be used as a tool for evaluating queries efficiently (see [16] for recent developments on weighted query evaluation).

Related Work. It has to be noted that there have been numerous approaches to characterizing probabilistic systems in terms of algebraic expressions and process calculi [29,7,12,11]. A unifying framework is due to Silva et al. [26], who consider probabilistic systems in a general coalgebraic setting. This allows them to derive algebraic expressions and a corresponding Kleene Theorem, as well as full axiomatizations for many of those (and even for weighted automata over arbitrary semirings). Their and above-mentioned works are mainly aiming at axiomatization of probabilistic-system behaviors in terms of bisimulation equivalence, so their focus is on *system* models including non-determinism and generative probability distributions. In this paper, we consider probabilistic automata, which are a more appropriate machine model for our purpose, i.e., for *evaluating* queries. Moreover, while the syntax of process-algebraic expressions is tailored to modeling probabilistic systems and uses action prefixing, fixed points, and process variables, we provide expressions with concatenation and a proper Kleene star. Thus, our expressions are closer to language-theoretic operations and more convenient to use in query languages. So far, there have been only few attempts to define quantitative query languages. In [15], Flesca et al. introduce a weighted XPath. Their approach, however, does not extend to probabilistic automata. Note that the fact that we also consider two-way devices distinguishes our work from all above-mentioned references. To the best of our knowledge, we present the first Kleene-Schützenberger correspondence for probabilistic two-way automata.

Outline. In Section 2, after some motivating example, we recall the definition of (reactive) probabilistic automata, introduce our probabilistic regular expressions, and present a corresponding Kleene Theorem. A part of the proof of this theorem is postponed to Section 4, where a more general result is shown for two-way probabilistic automata and corresponding generalized expressions (which are introduced in Section 3): automata and expressions are expressively equivalent and can be transformed effectively into each other.

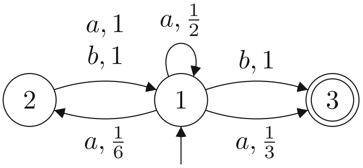


Fig. 1. A probabilistic automaton equivalent to $\left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (\frac{1}{3}a+b)$

2 Probabilistic Automata and Expressions

Preliminaries. We fix a finite alphabet A. We restrict our attention to nonempty words over A, i.e., sequences $w = a_0 \cdots a_{n-1} \in A^+$ with $n \geq 1$ and $a_i \in A$ for every i. The *length* n of w is denoted $|w|$, and $\text{pos}(w) = \{0, \ldots, |w|\}$ is the set of *positions* of w. The last position of index n is added to facilitate the definition of runs in two-way automata. To be able to deal with infinite sums over the non-negative real numbers, we extend $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}^{\infty} = \mathbb{R}_{\geq 0} \cup \{\infty\}$. In other words, we consider the *continuous semiring* $(\mathbb{R}_{\geq 0}^{\infty}, +, \cdot, 0, 1)$ where ∞ is assigned to any infinite sum that does not converge.

2.1 Probabilistic Automata

We consider classical probabilistic finite automata (PFA) [21,20]. A PFA over alphabet A is a tuple $\mathcal{A} = (Q, \iota, Acc, \mathbb{P})$ where Q is the finite set of states, $\iota \in Q$ is the initial state, and $Acc \subseteq Q$ is the set of final states. Moreover, $\mathbb{P} : Q \times A \times Q \to [0, 1]$ is a function that assigns a probability to each transition. PFA are *reactive* automata, whose probabilistic choice depends on the current input letter. Thus, we require that $\sum_{q' \in Q} \mathbb{P}(q, a, q') \leq 1$ for all $(q, a) \in Q \times A$. We will moreover assume that a final state has no outgoing transitions with positive probability: $\mathbb{P}(q, a, q') > 0$ implies $q \notin Acc$. An example PFA is depicted in Fig. 1 where 1 is the initial state and 3 is the only final state. Transitions with probability 0 are omitted.

An *accepting run* of $\mathcal{A}$ over $w = a_0 \cdots a_{n-1} \in A^+$ is a sequence of transitions $\rho = \delta_1 \cdots \delta_n$ such that $\delta_i = (q_{i-1}, a_{i-1}, q_i)$ with $q_0 = \iota$ and $q_n \in Acc$. Given a run ρ, we set $\mathbb{P}(\rho) = \prod_{i=1}^{n} \mathbb{P}(\delta_i)$. The semantics of the PFA $\mathcal{A}$ is the mapping (series) $[\![\mathcal{A}]\!] : A^+ \to [0, 1]$ given by $[\![\mathcal{A}]\!](w) = \sum_{\rho} \mathbb{P}(\rho)$, where the sum ranges over all accepting runs ρ over w. For instance, given the automaton of Fig. 1, we have $[\![\mathcal{A}]\!](aab) = \frac{1}{4} + \frac{1}{6} = \frac{5}{12}$.

2.2 Probabilistic Expressions

While PFAs are a machine model, we are aiming at denotational probabilistic regular expressions with the same expressiveness as PFAs. We start with the definition of classical weighted expressions (WEs) given by the syntax

$$E ::= s \mid a \mid E + E \mid E \cdot E \mid E^*$$

with $s \in \mathbb{R}_{\geq 0}^{\infty}$ and $a \in A$. In the following, we often simply write EF instead of $E \cdot F$. Also, we let $E^0 \overset{\text{def}}{=} 1$ and $E^{m+1} = EE^m$ for $m \geq 0$. The semantics of a WE E is a mapping $[\![E]\!] : A^+ \to \mathbb{R}_{\geq 0}^{\infty}$ which is defined inductively by

$$[\![s]\!](w) = \begin{cases} s & \text{if } w = \varepsilon \\ 0 & \text{otherwise} \end{cases} \qquad [\![a]\!](w) = \begin{cases} 1 & \text{if } w = a \\ 0 & \text{otherwise} \end{cases}$$

$$[\![E_1 + E_2]\!](w) = [\![E_1]\!](w) + [\![E_2]\!](w) \quad [\![E^*]\!](w) = \textstyle\sum_{m \in \mathbb{N}} [\![E^m]\!](w)$$

$$[\![E_1 \cdot E_2]\!](w) = \sum_{w=uv} [\![E_1]\!](u) \cdot [\![E_2]\!](v) \,.$$

In the following, we consider expressions modulo the following trivial identities:

$$0 + E \equiv E + 0 \equiv E \qquad E \cdot 0 \equiv 0 \cdot E \equiv 0 \qquad E \cdot 1 \equiv 1 \cdot E \equiv E \qquad 0^* \equiv 1$$

We introduce below probabilistic regular expressions (PREs) as a fragment of WEs. We have to restrict WEs since otherwise values greater than 1 could be obtained. For instance, the WE $(a + ab)(ba + a)$ should not be a PRE since it evaluates to 2 on the word aba. The restriction will be both on sum and star. Since we aim at PREs which are equivalent to PFAs, let us examine first which type of WEs are obtained from PFAs. A transition $\delta = (q, a, q')$ with probability $\mathbb{P}(\delta) = s$ could be denoted by the expression sa. Applying classical algorithms to build a regular expression from finite-state automata, we then obtain, for the automaton in Fig. 1, the expression $\left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (\frac{1}{3}a+b)$. Now, the expression $\left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (a+b)$, obtained by changing the subexpression $\frac{1}{3}a$ into a, should be disallowed, because it corresponds to an automaton violating condition $\sum_{q' \in Q} \mathbb{P}(q, a, q') \leq 1$. On the other hand, $\left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (\frac{1}{3}a+\frac{1}{2}b)$ would be acceptable: we obtain a corresponding PFA from the automaton depicted in Fig. 1 by setting $\mathbb{P}(1, b, 3) = \frac{1}{2}$.

Definition 1. Probabilistic regular expressions (PREs) *is the fragment of* WEs *built inductively as follows:*

(Atoms) $s \in [0,1]$ *and* $a \in A$ *are* PREs.

$(+_a)$ *If* $(E_a)_{a \in A}$ *are* PREs, *then* $\sum_{a \in A} a \cdot E_a$ *is a* PRE.

$(+_s)$ *If* E *and* F *are* PREs *and* $s \in [0,1]$, *then* $s \cdot E + (1-s) \cdot F$ *is a* PRE.

$(\cdot)$ *If* E *and* F *are* PREs, *then* $E \cdot F$ *is a* PRE.

$(*)$ *If* $E + F$ *is a* PRE, *then* $E^* \cdot F$ *is a* PRE.

(ACD) *Every* WE *that is obtained from a* PRE *by applying commutativity of* $+$, *associativity of* $+$ *or* $\cdot$, *or distributivity of* $\cdot$ *over* $+$ *is a* PRE.

There are two *guarded* sums. The first one $(+_a)$ is deterministic and guarded by the next letter to be read. The second one $(+_s)$ is probabilistic. Also, the star operation contains an implicit choice which is either to iterate again the expression or to exit the loop. This choice also has to be guarded which is the reason for the precondition $E + F \in$ PRE in the rule $(*)$. The guard could be deterministic as in $(ab)^*b$ or probabilistic as in $(\frac{1}{3}(aa+bb))^* \frac{2}{3}(a+b)$. Finally, with the above restrictions, we lose the classical ACD identities, hence we enforce these properties explicitely with the ACD-rules which allow to rewrite a PRE in order to apply the star rule as needed.

Since PREs form a fragment of WEs, the semantics is inherited. From Theorem 1 below, PREs are equivalent to PFAs. We deduce that the semantics of $E \in \mathsf{PRE}$ takes values in $[0,1]$, so that one can interpret $[\![E]\!](w)$ as a probability.

Example 1. A simple PRE is $(\frac{1}{3}a)^* \cdot \frac{2}{3}b$, which assigns to a word $a^m b$ the probability $(\frac{1}{3})^m \cdot \frac{2}{3}$, and 0 to words not in a^*b. Moreover, $E = \left[\frac{1}{6}a(a+b) + \frac{1}{2}a\right]^* \cdot (\frac{1}{3}a + b)$ is indeed a PRE for the automaton from Fig. 1. To show that E is a PRE, we use some semantical equivalences such as $sa \equiv as$ or $\frac{5}{6} \equiv \frac{1}{2} + \frac{1}{3}$. The expression $a(\frac{1}{6}(a+b) + \frac{5}{6}) + b$ uses two deterministic sums (first and third) and a probabilistic sum. Using the above semantical equivalences and ACD-rules, we deduce that $\frac{1}{6}a(a+b) + \frac{1}{2}a + \frac{1}{3}a + b$ is a PRE and it remains to apply the star rule to get E.

In order to construct a PRE which is equivalent to a PFA, we need to be able to concatenate a PRE after an arbitrary term of another PRE. This is possible thanks to the following result.

Proposition 1. *If $E + F$ and G are* PREs*, then $E + F \cdot G$ is also a* PRE*.*

For a PFA, we can always find an equivalent PRE, and vice versa. This first theorem, which is non-trivial even in the one-way setting, is generalized in the next sections allowing two-way moves and pebbles.

Theorem 1. PFAs *and* PREs *are effectively equivalent.*

Proof. We only prove the translation from automata to expressions. The other direction will be proved in a more general setting in Section 3.

Let $\mathcal{A} = (Q, \iota, Acc, \mathbb{P})$ be a PFA. For each $q \in Q \setminus Acc$ we construct a PRE $E_q = \sum_{q' \in Acc} E_{q,q'}$ where $[\![E_{q,q'}]\!](w)$ computes the sum of the probabilities of *nonempty* runs over w starting from state q, ending in state q'. Hence, we will obtain the PRE E_ι, which computes exactly the behavior of $\mathcal{A}$.

We follow usual procedures to translate automata into expressions. For $q' \in Q$ and $X \subseteq Q \setminus Acc$, we define $f_{q'}^X = 0$ if $q' \in X$ and 1 otherwise. For $q \in Q \setminus Acc$ and $X \subseteq Q \setminus Acc$, we construct by induction on X a PRE $E_q^X = \sum_{q' \in Q} E_{q,q'}^X f_{q'}^X$ where $E_{q,q'}^X$ is a PRE such that $[\![E_{q,q'}^X]\!](w)$ is the sum of the probabilities of *nonempty* runs over w starting from state q, ending in state q' and using only *intermediary* states in X. Hence, we have $E_q = E_q^{Q \setminus X}$ and $E_{q,q'} = E_{q,q'}^{Q \setminus X}$.

The base of the induction is when $X = \emptyset$. For each state $q \in Q \setminus Acc$ and letter $a \in A$, by definition of PFAs we have $\sum_{q' \in Q} \mathbb{P}(q,a,q') \leq 1$. Hence, using rules $(+_a)$ and $(+_s)$ we obtain the PRE

$$E_q^\emptyset = \sum\nolimits_{a \in A} a \cdot \sum\nolimits_{q' \in Q} \mathbb{P}(q,a,q') = \sum\nolimits_{q' \in Q} E_{q,q'}^\emptyset f_{q'}^\emptyset$$

where the last equality is obtained using ACD-rules and $f_{q'}^\emptyset = 1$.

For the induction step, let $X \cup \{r\} \subseteq Q \setminus Acc$ with $r \notin X$. By induction, we assume that PREs E_q^X have been constructed for all $q \in Q \setminus Acc$, and we construct $E_q^{X\cup\{r\}}$. We have $E_r^X = \sum_{q'\in Q} E_{r,q'}^X f_{q'}^X \in \mathsf{PRE}$ and $f_r^X = 1$ since $r \notin X$. Using rule $(*)$, we get $G_r^X = \left(E_{r,r}^X\right)^* \cdot \left(\sum_{q'\in Q\setminus\{r\}} E_{r,q'}^X f_{q'}^X\right) \in \mathsf{PRE}$. Now, $E_q^X = \sum_{q'\in Q} E_{q,q'}^X f_{q'}^X \in \mathsf{PRE}$ and $f_r^X = 1$. Using Proposition 1, we can plug G_r^X after $E_{q,r}^X$ and we obtain the PRE

$$\begin{aligned} E_q^{X\cup\{r\}} &= E_{q,r}^X \cdot G_r^X + \sum\nolimits_{q'\in Q\setminus\{r\}} E_{q,q'}^X f_{q'}^X \\ &= \sum\nolimits_{q'\in Q\setminus\{r\}} \left(E_{q,q'}^X + E_{q,r}^X \left(E_{r,r}^X\right)^* E_{r,q'}^X\right) f_{q'}^X \qquad \text{(ACD-rules)} \\ &= \sum\nolimits_{q'\in Q} E_{q,q'}^{X\cup\{r\}} f_{q'}^{X\cup\{r\}} \end{aligned}$$

using $f_r^{X\cup\{r\}} = 0$ and $f_{q'}^{X\cup\{r\}} = f_{q'}^X$ if $q' \in Q \setminus \{r\}$. □

With Theorem 1, decidability of the equivalence problem for PFAs carries over to PREs (provided the probabilities in an expression are rational numbers), whereas their threshold problem is undecidable.

Corollary 1.

1. *The equivalence problem for* PREs *is decidable: given* PREs *E and F, does $[\![E]\!] = [\![F]\!]$ hold?*
2. *The threshold problem for* PREs *is undecidable: given an alphabet A, a* PRE *E over A and $0 < s < 1$, is there a word $w \in A^+$ such that $[\![E]\!](w) \geq s$?*

Note that PFAs cannot recognize all series recognized by usual Rabin automata, i.e., PFAs without the blocking assumption over accepting states. For example, the map $g\colon A^+ \to [0,1]$, defined by $g(a^n) = 1$ if $n > 0$ and $g(w) = 0$ for all other words w, is not recognizable by a PFA (note that a^*a is not a PRE). However, g is recognized by a Rabin automaton with a single state. To deal with this issue, we can add a fresh symbol $\lhd$ at the end of a word. For a function $f\colon A^+ \to [0,1]$, we define $f_\lhd\colon (A \cup \{\lhd\})^+ \to [0,1]$ by $f_\lhd(w\lhd) = f(w)$ if $w \in A^+$, and 0 otherwise. For example, the series $g_\lhd$ is defined by $a(a^*\lhd)$, which is a PRE since $a + \lhd \in \mathsf{PRE}$. More generally, we can prove the following:

Proposition 2. *Let $f\colon A^+ \to [0,1]$. The function $f_\lhd$ is recognizable by a* PFA *(or equivalently by a* PRE*) iff f is recognizable by a Rabin automaton.*

3 Adding Two-Way Navigation and Pebbles

In this section, we extend probabilistic automata and expressions such that they allow us to navigate in a given word and place pebbles that can be recovered later. Before we extend PFAs and PREs accordingly, let us give a motivating example.

Example 2. Using pebbles in probabilistic expressions or automata is a natural and powerful way to deal with nesting in LTL formulas. Indeed, temporal logics implicitly use a free variable to denote the position where a formula has to be evaluated. We will mark this position with a pebble, say x, in expressions $E_\varphi(x)$ or automata $\mathcal{A}_\varphi(x)$ associated with LTL formulas φ.

Consider an LTL formula $\mathsf{F}\varphi$, for *Finally* φ. Given a word w and a position i in w, we are interested in the probability $\mathbb{P}(\mathsf{F}\varphi, w, i)$ that φ holds in w at position i. For instance, for $\varphi = \frac{1}{3}a$, we should obtain $\mathbb{P}(\mathsf{F}\varphi, abba, 0) = \frac{1}{3} + \frac{2}{3}(0 + \frac{2}{3}(0 + \frac{2}{3}(\frac{1}{3} + 0)))$: either φ is satisfied immediately with probability $\frac{1}{3}$, or it is not (probability $\frac{2}{3}$) so that (product) it has to be satisfied later. More generally, we have

$$\begin{aligned}\mathbb{P}(\mathsf{F}\varphi, w, i) &= \mathbb{P}(\varphi, w, i) + \mathbb{P}(\neg\varphi, w, i) \times \mathbb{P}(\mathsf{F}\varphi, w, i+1)\\ &= \sum_{k \geq i} \mathbb{P}(\varphi, w, k) \times \prod_{i \leq j < k} \mathbb{P}(\neg\varphi, w, j)\,.\end{aligned}$$

For every LTL formula φ, we are aiming at an equivalent expression $E_\varphi(x)$ which evaluates to $\mathbb{P}(\varphi, w, i)$ over word w when pebble x marks position i. For this, we use a new construct, $y!E_\varphi(y)$, which marks the current position with pebble y, and computes φ on the whole word (from beginning to end) with this position marked (this is a non-progressing construct). Let us illustrate this inductive construction for LTL formulas. For *Finally* φ, we set

$$E_{\mathsf{F}\varphi}(x) = \rhd?{\rightarrow^*}x?\big(\big(y!E_{\neg\varphi}(y)\big){\rightarrow}\big)^*\big(y!E_\varphi(y)\big){\rightarrow^*}\lhd?\,.$$

The expression starts at the beginning of the word ($\rhd?$), and moves to the right ($\rightarrow^*$) until it discovers the marked position ($x?$). Then, for each $n \geq 0$, it iterates n times the computation of $\neg\varphi$ with the current position marked by y ($y!E_{\neg\varphi}(y)$), moving to the right ($\rightarrow$) between two computations. Finally, it computes φ with $y!E_\varphi(y)$ before moving to the last position of the word ($\rightarrow^*\lhd?$).

Similarly, for *Globally* φ ($\mathsf{G}\varphi$), we have $\mathbb{P}(\mathsf{G}\varphi, w, i) = \prod_{j \geq i} \mathbb{P}(\varphi, w, j)$, leading to the simpler expression

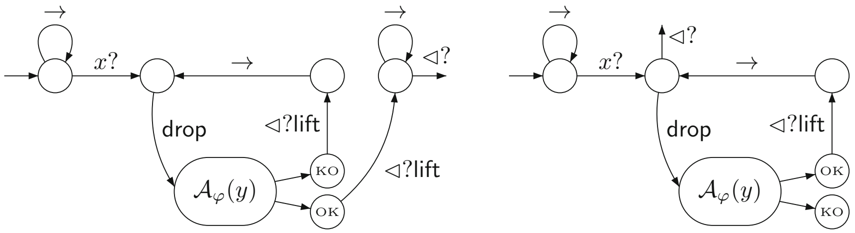
$$E_{\mathsf{G}\varphi}(x) = \rhd?{\rightarrow^*}x?\big(\big(y!E_\varphi(y)\big){\rightarrow}\big)^*\lhd?\,.$$

The last test ($\lhd?$) is useful to enforce the preceding star operation to capture the whole suffix of the word from the position marked by x.

Finally, based on the equivalence $\varphi \,\mathsf{U}\, \psi \equiv (\neg\psi \wedge \varphi) \,\mathsf{U}\, \psi$, the expression for the *Until* modality is

$$E_{\varphi\mathsf{U}\psi}(x) = \rhd?{\rightarrow^*}x?\big(\big(y!(E_{\neg\psi}(y){\leftarrow^*}E_\varphi(y))\big){\rightarrow}\big)^*\big(y!E_\psi(y)\big){\rightarrow^*}\lhd?\,.$$

In terms of automata, let us assume that, for every formula φ, there is an automaton $\mathcal{A}_\varphi$ with two designated terminal states, OK and KO, such that runs ending in OK (and at the end of the word) compute expression E_φ and those ending in KO compute expression $E_{\neg\varphi}$. These automata are 2-way, and can drop and lift pebbles on word positions. Dropping a pebble resets the control at the beginning. Lifting a pebble can be performed anywhere and resets the control to the position of the last dropped pebble (which then gets removed). The figure below depicts automata for the modalities *Finally* and *Globally*.



3.1 Probabilistic Pebble Automata

Our extended automata navigate (just like extended expressions defined below) inside a word in both directions, denoted → and ←. Motivated by Example 2, we moreover equip automata with $p \in \mathbb{N}$ pebbles, $1, \ldots, p$. Naturally, two-way automata can check if the current position carries some particular letter or pebble, or if it is a border (i.e., the first or last) position of the input word. Thus, they make their probabilistic choice depending on the type of the current position, which is a set containing the letter and pebbles that can be found at the current position. Formally, a *p-type* is a set $t \subseteq A \cup \{\rhd, \lhd\} \cup \{1, \ldots, p\}$ such that (i) $\lhd \in t$ implies $t = \{\lhd\}$, and (ii) $\lhd \notin t$ implies $|t \cap A| = 1$. In other words, t indicates the current letter from A, unless the control is beyond the word ($\lhd \in t$). Moreover, it reveals if the current position is the first ($\rhd \in t$), or the last one ($\lhd \in t$), or neither of them ($t \cap \{\rhd, \lhd\} = \emptyset$). Let $\mathcal{T}_p$ be the set of p-types, and let $\mathcal{T} = \bigcup_{p \in \mathbb{N}} \mathcal{T}_p$ denote the set of *types.* The current state and type of a configuration will give rise to a probability function, which triggers a move of the automaton, taken from the set $\mathcal{M} = \{\rightarrow, \leftarrow, \mathsf{drop}, \mathsf{lift}, \mathsf{stay}\}$.

Let us formally define pebble probabilistic automata, which generalize PFAs.

Definition 2. *A* pebble probabilistic automaton (PPA) *over A is a tuple $\mathcal{A} = (p, Q, \iota, Acc, \mathbb{P})$ where $p \in \mathbb{N}$ is the number of pebbles, Q is a finite set of states, $\iota \in Q$ is the initial state, and $Acc \subseteq Q \setminus \{\iota\}$ is the set of final states. Moreover, $\mathbb{P}\colon Q \times \mathcal{T}_p \times \mathcal{M} \times Q \to [0,1]$ is a transition probability function such that:*

- *For all $\delta = (q, t, d, q')$, if $\mathbb{P}(\delta) > 0$ then $q \notin Acc$ and $\rhd \in t$ implies $d \neq \leftarrow$ and $\lhd \in t$ implies $d \notin \{\rightarrow, \mathsf{drop}\}$.*
- *For all $q \in Q \setminus Acc$ and all types $t \in \mathcal{T}_p$, we have* $\sum_{(d,q') \in \mathcal{M} \times Q} \mathbb{P}(q, t, d, q') \leq 1.$

Next, we define the behavior of PPAs. Fix a word $w = a_0 \cdots a_{n-1}$ and a PPA $\mathcal{A}$ with p pebbles. A *configuration* of $\mathcal{A}$ over w is a triple $\kappa = (q, i, \pi)$ with $q \in Q$, $i \in \mathrm{pos}(w) = \{0, \ldots, n\}$, and $\pi \in \{0, \ldots, n-1\}^{\leq p}$, where $X^{\leq p}$ is the set of words over X of length at most p. Intuitively, q is the current state, i is the current position, and π represents a stack recording the $k = |\pi| \leq p$ positions of the currently dropped pebbles: $\pi = i_1 \cdots i_k$ means that pebble $\ell \in \{1, \ldots, k\}$ is currently at position i_ℓ and pebbles $k+1, \ldots, p$ are currently not dropped. Pebbles are dropped and lifted using a stack policy: if $\pi = i_1 \cdots i_k$, only pebble $k+1$ can be dropped (provided $k + 1 \leq p$), and only pebble k can be lifted (provided $k \geq 1$).

Let $\kappa = (q, i, \pi)$ be a configuration with $\pi = i_1 \cdots i_k$. We call κ *initial* if $q = \iota$, $i = 0$, and $\pi = \varepsilon$. It is said to be *final* if $q \in Acc$, $i = n$, and $\pi = \varepsilon$. Moreover, the type of κ is denoted by $type(\kappa)$ and defined as $\{a_i \mid i < n\} \cup \{\triangleright \mid i = 0\} \cup \{\triangleleft \mid i = n\} \cup \big\{\ell \in \{1, \ldots, k\} \mid i_\ell = i\big\}$.

Given $\delta = (q, t, d, q') \in Q \times \mathcal{T}_p \times \mathcal{M} \times Q$ as well as configurations $\kappa = (q, i, \pi)$ and $\kappa' = (q', i', \pi')$, we write $\kappa \xrightarrow{\delta} \kappa'$ if $t = type(\kappa)$ and the following hold:

1. if $d = \rightarrow$ then $i' = i + 1$ and $\pi' = \pi$
2. if $d = \leftarrow$ then $i' = i - 1$ and $\pi' = \pi$
3. if $d = \mathsf{stay}$ then $i' = i$ and $\pi' = \pi$
4. if $d = \mathsf{drop}$ then $i' = 0$ and $\pi' = \pi i$ and $i < n$
5. if $d = \mathsf{lift}$ then $\pi' i' = \pi$.

In other words, drop saves the current position on the stack of pebbles ($\pi' = \pi i$ in 4) and resets the head to the first position ($i' = 0$ in 4). Moreover, $\pi' i' = \pi$ in 5 means that lift pops the last dropped pebble and resets the control to the position where this pebble was dropped.

A *run* of $\mathcal{A}$ over w is a finite sequence $\rho = (\kappa_0, \delta_1, \kappa_1, \ldots, \delta_h, \kappa_h)$ ($h \geq 0$) of configurations and transitions such that, for all $0 < m \leq h$, we have $\mathbb{P}(\delta_m) > 0$ and $\kappa_{m-1} \xrightarrow{\delta_m} \kappa_m$. We say that ρ is *accepting* if κ_0 is an initial configuration and κ_h is final. The probability of this run is the product of the probabilities of the transitions all along the run: $\mathbb{P}(\rho) = \prod_{m=1}^{h} \mathbb{P}(\delta_m)$. Now, let $[\![\mathcal{A}]\!](w)$ be the sum $\sum_\rho \mathbb{P}(\rho)$, where ρ ranges over the accepting runs of $\mathcal{A}$ over w. Note that the number of accepting runs over a given word may be infinite so that, a priori, $[\![\mathcal{A}]\!]$ is a mapping $A^+ \to \mathbb{R}_{\geq 0}^{\infty}$. However, one can show the following:

Proposition 3. *For every* PPA $\mathcal{A}$ *and every* $w \in A^+$*, we have* $[\![\mathcal{A}]\!](w) \in [0, 1]$.

Proof. Let us define, for every $w \in A^+$, a probability space $(\Omega^w, \mathfrak{E}^w, \mathbb{P}^w)$. The set Ω^w of outcomes is the set of maximal runs of $\mathcal{A}$ over w, starting in the initial configuration (for technical reasons, we have to include infinite ones). Moreover, the set $\mathfrak{E}^w$ of events is the smallest σ-algebra containing, for all finite runs ρ, the *cylinder set* $Cyl(\rho) \overset{\text{def}}{=} \{\rho' \mid \rho'$ is a maximal run with prefix $\rho\}$. With this, there is a unique probability measure $\mathbb{P}^w$ for Ω^w and $\mathfrak{E}^w$, which is given by $\mathbb{P}^w(Cyl(\rho)) = 1 \cdot \mathbb{P}(\delta_1) \cdot \ldots \cdot \mathbb{P}(\delta_h)$ for all finite runs $\rho = (\kappa_0, \delta_1, \kappa_1, \ldots, \delta_h, \kappa_h)$. As the set $AR(w)$ of accepting runs over w is countable, it is measurable in the probability space, and its probability is $\mathbb{P}^w(AR(w)) = \sum_{\rho \in AR(w)} \mathbb{P}^w(Cyl(\rho))$. Note that the latter equals $[\![\mathcal{A}]\!](w)$ and that $Cyl(\rho) = \{\rho\}$ for all $\rho \in AR(w)$. □

The proof above establishes a strong connection between PPAs and Markov chains. This connection also provides an algorithm for evaluating a PPA wrt. a given word, which reduces to computing the probability of reaching a final configuration in the *synchronized* Markov chain (see, for example, [3]).

Example 3. Consider the PPA $\mathcal{A}$ (2-way, without pebbles) depicted on the left of Fig. 2 (where $0 < s < 1$) over alphabet $A = \{a\}$. The synchronized Markov chain of $\mathcal{A}$ wrt. a word of length n is depicted on the right of the same figure. This Markov chain represents a random walk over a straight line of bounded length.

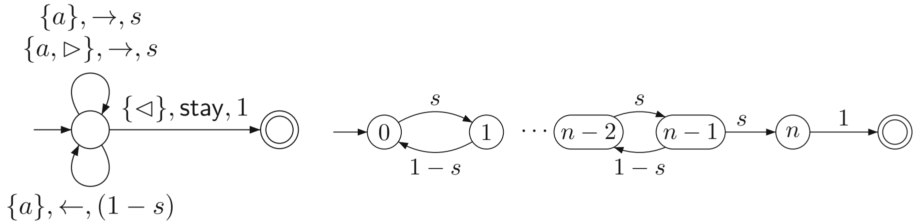

$\{a\}, \rightarrow, s$
$\{a, \triangleright\}, \rightarrow, s$
$\{\triangleleft\}$, stay, 1
$\{a\}, \leftarrow, (1-s)$
0
1
s
$1-s$
$\cdots$
$n-2$
$n-1$
s
$1-s$
s
n
1

Fig. 2. Markov chain obtained by synchronizing $\mathcal{A}$ with a word of length n

3.2 Probabilistic Pebble Expressions

Now, we define probabilistic expressions that capture the expressive power of PPAs. Just like PPAs, expressions are equipped with some pebbles from an infinite supply $\mathcal{P} = \{1, 2, \ldots\}$ (though every expression will only employ a finite number thereof). Also, expressions should be able to move left or right and to check if pebbles are dropped on the current position. Hence, we adopt an XPath-like syntax: we explicitly distinguish progression in the word (with $\leftarrow$ or $\rightarrow$) from tests $\varphi \in$ Tests checking the current type. We define *test* formulas inductively by

$$\varphi ::= a? \mid x? \mid \triangleright? \mid \triangleleft? \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

where $a \in A$ and $x \in \mathcal{P}$. In particular, formulas $\triangleright?$ and $\triangleleft?$ allow an expression to test whether it is at the first or last position of the word. Moreover, $x?$ is true if pebble x is on the current position. Formulas $\varphi \in$ Tests are interpreted over a type $t \in \mathcal{T}$ in a straightforward manner: for $\theta \in A \cup \{\triangleright, \triangleleft\} \cup \mathcal{P}$, we write $t \models \theta?$ if $\theta \in t$. The semantics of boolean connectives is defined as expected.

We start with *unrestricted* weighted pebble expressions (WPEs) defined by

$$E ::= s \mid \varphi \mid \leftarrow \mid \rightarrow \mid E + E \mid E \cdot E \mid E^* \mid x!E$$

with $s \in \mathbb{R}^{\infty}_{\geq 0}$, $\varphi \in$ Tests and $x \in \mathcal{P}$. The interpretation of s, sum, concatenation, and star does not change wrt. WEs. The new atomic expressions $\leftarrow$ and $\rightarrow$ perform a step to the left or to the right. The expressions φ and $x!E$, on the other hand, perform *non-progressing* computations (similar to s). The atomic WE a is obtained by checking first if the current letter is a and then moving right, hence the abbreviation $a \overset{\text{def}}{=} a?{\rightarrow}$. The new construct $x!E$ is read "compute E with x assigned to the current position". The computation of E "rescans" the whole word, from position 0 to position $|w|$.

The set of *free* pebbles of an expression $E \in$ WPE is defined in the obvious way; in particular, we set $\mathit{Free}(x?) = \{x\}$ and $\mathit{Free}(x!E) = \mathit{Free}(E) \setminus \{x\}$.

The semantics $[\![E]\!]$ of an expression E assigns a value from $\mathbb{R}^{\infty}_{\geq 0}$ to each tuple (w, σ, i, j) where $w = a_0 a_1 \cdots \in A^+$, $i, j \in \mathrm{pos}(w)$ and $\sigma : \mathit{Free}(E) \to \{0, \ldots, |w| - 1\}$. It computes the weight of going from i to j with the pebble assignment σ. Formally, the semantics is given in Table 1. Hereby, we let $\mathit{type}(i) \in \mathcal{T}$ denote the set $\{a_i \mid i \neq |w|\} \cup \{\triangleright \mid i = 0\} \cup \{\triangleleft \mid i = |w|\} \cup \sigma^{-1}(i)$. Moreover, $\sigma[x \mapsto i]$ stands for the assignment coinciding with σ except on x, which is

Table 1. Semantics of WPEs

$$[\![s]\!](w,\sigma,i,j) = \begin{cases} s & \text{if } j=i \\ 0 & \text{otherwise} \end{cases} \qquad [\![\varphi]\!](w,\sigma,i,j) = \begin{cases} 1 & \text{if } j=i \text{ and } \mathit{type}(i) \models \varphi \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\rightarrow]\!](w,\sigma,i,j) = \begin{cases} 1 & \text{if } j=i+1 \\ 0 & \text{otherwise} \end{cases} \qquad [\![\leftarrow]\!](w,\sigma,i,j) = \begin{cases} 1 & \text{if } j=i-1 \\ 0 & \text{otherwise} \end{cases}$$

$$[\![E_1+E_2]\!] = [\![E_1]\!] + [\![E_2]\!] \qquad [\![E^*]\!](w,\sigma,i,j) = \textstyle\sum_{m\in\mathbb{N}} [\![E^m]\!](w,\sigma,i,j)$$

$$[\![E_1 \cdot E_2]\!](w,\sigma,i,j) = \sum_{k\in\mathrm{pos}(w)} [\![E_1]\!](w,\sigma,i,k) \times [\![E_2]\!](w,\sigma,k,j)$$

$$[\![x!E]\!](w,\sigma,i,j) = \begin{cases} [\![E]\!](w,\sigma[x\mapsto i],0,|w|) & \text{if } j=i<|w| \\ 0 & \text{otherwise} \end{cases}$$

mapped to i. If $\mathit{Free}(E)=\emptyset$, then we let $[\![E]\!](w) = [\![E]\!](w,0,|w|)$ (omitting σ, as it is irrelevant).

One can easily check that concatenation, also called Cauchy product, is associative, i.e., $[\![(E_1\cdot E_2)\cdot E_3]\!] = [\![E_1\cdot(E_2\cdot E_3)]\!]$. Hence, one can define E^m by induction: $E^0=1$ and $E^{m+1}=E\cdot E^m$. Moreover, $+$ is associative and commutative, concatenation distributes over $+$, and $x!$ distributes over $+$.

As for expressions without pebbles, we need to restrict the sum and star operations to get the *probabilistic* fragment. Doing so, we lose commutativity, associativity and distributivity hence we enforce these properties explicitly. The proof of Proposition 1 cannot be extended to cope with $x!E$, hence we strengthen the rule for concatenation.

Definition 3. Probabilistic pebble expressions (PPEs) *is the fragment of* WPEs *built inductively as follows:*

(Atoms) $s\in[0,1]$, $\varphi\in$ Tests, $\leftarrow$ *and* $\rightarrow$ *are* PPEs.
$(+_\varphi)$ *If E and F are* PPEs *and* $\varphi\in$ Tests, *then* $\varphi\cdot E+(\neg\varphi)\cdot F$ *is a* PPE.
$(+_s)$ *If E and F are* PPEs *and* $s\in[0,1]$, *then* $s\cdot E+(1-s)\cdot F$ *is a* PPE.
$(\cdot)$ *If $E+F$ is a* PPE *and G is a* PPE, *then* $E+F\cdot G$ *is a* PPE.
$(*)$ *If $E+F$ is a* PPE, *then* $E^*\cdot F$ *is a* PPE.
$(x!)$ *If E is a* PPE, *then* $x!E$ *is a* PPE.
(ACD) PPEs *are closed under the following associativity, commutativity and distributivity rules (*ACD*-rules):*

$$\begin{array}{llcl}
\mathbf{A}_+ & E+(F+G)\in\mathsf{PPE} & \longleftrightarrow & (E+F)+G\in\mathsf{PPE} \\
\mathbf{C}_+ & E+F\in\mathsf{PPE} & \longleftrightarrow & F+E\in\mathsf{PPE} \\
\mathbf{A}_\cdot & E\cdot(F\cdot G)\in\mathsf{PPE} & \longleftrightarrow & (E\cdot F)\cdot G\in\mathsf{PPE} \\
\mathbf{D}_\cdot & E\cdot(F+G)\in\mathsf{PPE} & \longleftrightarrow & E\cdot F+E\cdot G\in\mathsf{PPE} \\
\mathbf{D}_\cdot & (E+F)\cdot G\in\mathsf{PPE} & \longleftrightarrow & E\cdot G+F\cdot G\in\mathsf{PPE} \\
\mathbf{D}_{x!} & x!(E+F)\in\mathsf{PPE} & \longleftrightarrow & x!E+x!F\in\mathsf{PPE}
\end{array}$$

The semantics of PPEs is inherited from WPEs and we will show later that when $E\in$ PPE then $[\![E]\!](w,\sigma,i,j)$ actually always belongs to $[0,1]$.

Example 4. We continue Example 3. An equivalent PPE to denote the random walk is given by

$$E = (\neg\lhd?(s\rightarrow + (1-s)\neg\rhd?\leftarrow))^* \lhd?\,.$$

Moreover, let $F = \neg\lhd?(s\rightarrow + (1-s)\neg\rhd?\leftarrow)$ so that $E = F^* \lhd?$. Notice that E is indeed an expression in PPE because $F + \lhd? \in \mathsf{PPE}$. Let w be a word of length $m \geq 2$. We can easily see that, for all $i, j \in \mathrm{pos}(w)$ and all $n \geq |j-i|$, the expression F^n computes a *positive* value on (w, i, j). Therefore, the expression F^* computes an infinite sum on (w, i, j). In the present case $(0 < s < 1)$, the series $\sum_{n\geq 0}\llbracket F^n\rrbracket(w,i,j)$ converges: with $\alpha = \frac{1-s}{s}$, one can show that $\llbracket E\rrbracket(w, 0, |w|) = 1/(1+\alpha+\ldots+\alpha^{|w|})$, and $\llbracket F^*\rrbracket(w,i,j) \in [0,1]$.

We define now the multiset $\mathsf{Terms}(E)$ of terms of an expression $E \in \mathsf{WPE}$. This will be crucial for the translation from expressions to automata in the next section. Intuitively, if we suppose that summation is pushed up as much as possible by means of ACD-rules, then the multiset of terms consists of all expressions that occur in this big outermost sum. Formally, the definition is by induction over $E \in \mathsf{WPE}$. When E is an atom, we let $\mathsf{Terms}(E) = \{\!\{E\}\!\}$ be the singleton multiset containing only the atom itself. Moreover,

$$\begin{aligned}
\mathsf{Terms}(E+F) &= \mathsf{Terms}(E) \uplus \mathsf{Terms}(F)\\
\mathsf{Terms}(E\cdot F) &= \{\!\{E'\cdot F' \mid E' \in \mathsf{Terms}(E), F' \in \mathsf{Terms}(F)\}\!\}\\
\mathsf{Terms}(E^*) &= \{\!\{E^*\}\!\}\\
\mathsf{Terms}(x!E) &= \{\!\{x!E' \mid E' \in \mathsf{Terms}(E)\}\!\}\,.
\end{aligned}$$

Note that, if an expression F can be obtained from an expression E through ACD-rules, then we have $\mathsf{Terms}(E) = \mathsf{Terms}(F)$. The converse also holds as can be seen from the following proposition which can be easily proved by structural induction on the expression.

Proposition 4. *Let $E \in \mathsf{WPE}$ with $\mathsf{Terms}(E) = \{\!\{E_i \mid i \in I\}\!\}$. Using ACD-rules, we can rewrite E into $\sum_{i\in I} E_i$. In particular, we have $\llbracket E\rrbracket = \sum_{i\in I}\llbracket E_i\rrbracket$. Hence, we identify E and $\sum_{i\in I} E_i$.*

4 The Probabilistic Kleene Theorem

We prove in this section that PPAs are effectively equivalent to PPEs. We start with the construction of PPAs from PPEs. The problematic cases are concatenation and iteration due to the precondition $E + F \in \mathsf{PPE}$. To deal with these cases, we construct from PPE E a PPA $\mathcal{A}$ which simultaneously recognizes all *terms* of E.

Theorem 2. *From any expression $E \in \mathsf{PPE}$ we can effectively construct an equivalent PPA $\mathcal{A} = (p, Q, \iota, Acc, \mathbb{P})$. More precisely, if $\mathsf{Terms}(E) = \{\!\{E_i \mid i \in I\}\!\}$, the set of accepting states of $\mathcal{A}$ is $Acc = \{f_i \mid i \in I\}$ and for all $i \in I$ the expression E_i is equivalent to the PPA $\mathcal{A}[f_i] = (p, Q, \iota, \{f_i\}, \mathbb{P})$.*

Proof. The construction is by structural induction on the expression $E \in \mathsf{PPE}$ which may have free pebbles. As usual, the assignment of free pebbles will be encoded in the alphabet of the word read by the automaton $\mathcal{A} \in \mathsf{PPA}$, hence (w, σ) will be interpreted as a word over $A \times 2^{\mathit{Free}(E)}$ in the automata. Then, equivalence $\mathcal{A} \approx E$ means that, for all words $w \in A^+$, assignments $\sigma\colon \mathit{Free}(E) \to \mathrm{pos}(w) \setminus \{|w|\}$, and positions $i, j \in \mathrm{pos}(w)$, the value $[\![E]\!](w, \sigma, i, j)$ is the sum of the probabilities of the runs of $\mathcal{A}$ over (w, σ) starting in the initial state in position i, ending in an accepting state in position j.

The cases $s \in [0,1]$, $\varphi \in \mathsf{Tests}$, $\to$, and $\leftarrow$ are clear. For each atom, the resulting automaton has only two states (it uses stay transitions for s and φ).

Now let $E, E' \in \mathsf{PPE}$ be such that $\mathsf{Terms}(E) = \{\!\{E_i \mid i \in I\}\!\}$ and $\mathsf{Terms}(E') = \{\!\{E'_j \mid j \in J\}\!\}$. By induction hypothesis, we have constructed two suitable PPAs $\mathcal{A} = (p, Q, \iota, \mathit{Acc}, \mathbb{P})$ and $\mathcal{A}' = (p', Q', \iota', \mathit{Acc}', \mathbb{P}')$ with $\mathit{Acc} = \{f_i \mid i \in I\}$ and $\mathit{Acc}' = \{f'_j \mid j \in J\}$. Without loss of generality, we assume that $p = p'$, $Q \cap Q' = \emptyset$ and that a final state may only be reached when no pebble is dropped (if necessary, this may be enforced by keeping the number of dropped pebbles in the states).

Consider $E'' = \varphi{\cdot}E + \neg\varphi{\cdot}E'$. We have $\mathsf{Terms}(E'') = \{\!\{\varphi{\cdot}E_i \mid i \in I\}\!\} \uplus \{\!\{\neg\varphi{\cdot}E'_j \mid j \in J\}\!\}$. We construct a PPA $\mathcal{A}'' = (p, Q \uplus Q' \uplus \{\iota''\}, \iota'', \mathit{Acc} \uplus \mathit{Acc}', \mathbb{P}'')$. From the new initial state ι'', we add stay transitions with probability 1 going to ι for all types t satisfying φ, and going to ι' for all other types.

The construction for $E'' = s \cdot E + (1-s) \cdot E'$ is similar. We have $\mathsf{Terms}(E'') = \{\!\{s \cdot E_i \mid i \in I\}\!\} \uplus \{\!\{(1-s) \cdot E'_j \mid j \in J\}\!\}$. We add stay transitions with probability s from ι'' to ι and stay transitions with probability $1-s$ from ι to ι'.

For the concatenation, we assume that $E = F + G$ and $E'' = F + G \cdot E'$. We have $I = K \uplus L$ with $\mathsf{Terms}(F) = \{\!\{E_i \mid i \in K\}\!\}$ and $\mathsf{Terms}(G) = \{\!\{E_i \mid i \in L\}\!\}$. Hence, we have $\mathsf{Terms}(E'') = \{\!\{E_i \mid i \in K\}\!\} \uplus \{\!\{E_i \cdot E'_j \mid (i,j) \in L \times J\}\!\}$. The automaton $\mathcal{A}''$ for E'' consists of one copy of $\mathcal{A}$ and a copy $\mathcal{A}'_i$ of $\mathcal{A}'$ for every $i \in L$. First, $\mathcal{A}''$ simulates $\mathcal{A}$ until it reaches some final state f_i of $\mathcal{A}$. Then, if $i \in L$, a stay transition leads with probability 1 into the initial state of $\mathcal{A}'_i$. The final states of $\mathcal{A}''$ consist of $\{f_i \mid i \in K\}$ and a copy of Acc' for each $i \in L$.

For the Kleene star we assume that $E = F + G$ and $E'' = F^* \cdot G$. We have $I = K \uplus L$ with $\mathsf{Terms}(F) = \{\!\{E_i \mid i \in K\}\!\}$ and $\mathsf{Terms}(G) = \{\!\{E_i \mid i \in L\}\!\}$. Hence, we have $\mathsf{Terms}(E'') = \{\!\{F^* \cdot E_i \mid i \in L\}\!\}$. We construct the PPA $\mathcal{A}'' = (p, Q, \iota, \mathit{Acc}'', \mathbb{P}'')$ with $\mathit{Acc}'' = \{f_i \mid i \in L\}$ by adding stay transitions with probability 1 from all states f_i for $i \in K$ back to the initial state ι.

Consider $E'' = x!E$. We have $\mathsf{Terms}(E'') = \{\!\{x!E_i \mid i \in I\}\!\}$. We construct the PPA $\mathcal{A}'' = (p+1, Q \uplus \{\iota''\} \uplus \mathit{Acc}'', \iota'', \mathit{Acc}'', \mathbb{P}'')$ with Acc'' a copy of Acc. To this aim, we shift the numbers of pebbles of $\mathcal{A}$ to $\{2, \ldots, p+1\}$ keeping pebble 1 to be the fresh one, used to mark the position of variable x. We start by dropping pebble 1 with transitions of the form $(\iota'', t, \mathsf{drop}, \iota)$, each with probability 1. At the end of a computation of $\mathcal{A}$, pebble 1 is lifted, again with probability 1, using transitions $(q, \{\lhd\}, \mathsf{lift}, q'')$ where $q'' \in \mathit{Acc}''$ is the copy of $q \in \mathit{Acc}$.

Finally, if E'' is obtainted from E via ACD-rules, we have $\mathsf{Terms}(E'') = \mathsf{Terms}(E)$ so we can keep the same automaton: $\mathcal{A}'' = \mathcal{A}$. □

Note that, even if we start from a PRE, the proof above does not yield a PFA since it adds *stay* transitions. Hence, this proof has to be adapted to translate PREs into PFAs. By Proposition 3, the semantics of automata only assumes values in $[0, 1]$. This carries over to PPE.

Corollary 2. *For every* PPE E *and every* $w \in A^+$*, we have* $[\![E]\!](w) \in [0, 1]$.

We turn now to the construction of expressions (PPEs) which are equivalent to automata (PPAs). We follow usual procedures for the translation from automata to expressions, ensuring throughout the proof that we produce expressions in PPE. To this aim, we strongly rely on ACD-rules.

Theorem 3. *Let* $\mathcal{A} = (p, Q, \iota, Acc, \mathbb{P})$ *be a* PPA *with* p *pebbles. We can effectively construct a* PPE $E_\iota = \sum_{q \in Acc} E_{\iota,q}$ *such that* $[\![E_{\iota,q}]\!](w, 0, |w|)$ *is the sum of the runs from the initial configuration* $(\iota, 0, \varepsilon)$ *to the final configuration* $(q, |w|, \varepsilon)$.

5 Conclusion

In this paper, we presented a probabilistic Kleene Theorem, first for classical probabilistic automata and then for extended automata with two-way navigation and pebbles. This constitutes a first step towards probabilistic XPath, so we aim at extending our work to finite trees and probabilistic tree automata. We also raise the question of whether our technique can be used to obtain ω-expressions for probabilistic Büchi automata, which have attracted a lot of attention [2,1,8]. Just like classical finite automata, weighted automata over semirings enjoy characterizations in terms of monadic second-order logic [13,6]. Continuing this line of research, a recent paper establishes a logical characterization of probabilistic automata [30]. It would be interesting to study whether alternative characterizations exist that use, for example, a transitive-closure operator [6].

References

1. 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)
2. Baier, C., Größer, M.: Recognizing ω-regular languages with probabilistic automata. In: Proc. of LICS 2005, pp. 137–146. IEEE Computer Society (2005)
3. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
4. Berstel, J., Reutenauer, C.: Noncommutative rational series with applications, Cambridge. Encyclopedia of Mathematics & Its Applications, vol. 137, Cambridge (2011)
5. Bojańczyk, M.: Tree-Walking Automata. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 1–2. Springer, Heidelberg (2008)
6. Bollig, B., Gastin, P., Monmege, B., Zeitoun, M.: Pebble Weighted Automata and Transitive Closure Logics. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 587–598. Springer, Heidelberg (2010)

7. Buchholz, P., Kemper, P.: Quantifying the Dynamic Behavior of Process Algebras. In: de Luca, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 184–199. Springer, Heidelberg (2001)
8. Chadha, R., Sistla, A.P., Viswanathan, M.: Power of randomization in automata on infinite strings. Logical Methods in Computer Science 7(3:22) (2011)
9. Cortes, C., Mohri, M., Rastogi, A.: L_p distance and equivalence of probabilistic automata. Int. J. Found. Comput. Sci. 18(4), 761–779 (2007)
10. Cortes, C., Mohri, M., Rastogi, A., Riley, M.: On the computation of the relative entropy of probabilistic automata. Int. J. Found. Comput. Sci. 19(1), 219–242 (2008)
11. Deng, Y., Palamidessi, C.: Axiomatizations for probabilistic finite-state behaviors. Theor. Comput. Sci. 373(1-2), 92–114 (2007)
12. Deng, Y., Palamidessi, C., Pang, J.: Compositional Reasoning for Probabilistic Finite-State Behaviors. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity. LNCS, vol. 3838, pp. 309–337. Springer, Heidelberg (2005)
13. Droste, M., Gastin, P.: Weighted automata and weighted logics. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. EATCS Monographs in Theoretical Computer Science, ch. 5, pp. 175–211. Springer (2009)
14. Dwork, C., Stockmeyer, L.: On the power of 2-way probabilistic finite state automata. In: Proc. of FoCS 1989, pp. 480–485. IEEE Computer Society (1989)
15. Flesca, S., Furfaro, F., Greco, S.: Weighted path queries on semistructured databases. Inform. and Comput. 204(5), 679–696 (2006)
16. Gastin, P., Monmege, B.: Adding Pebbles to Weighted Automata. In: Moreira, N., Reis, R. (eds.) CIAA 2012. LNCS, vol. 7381, pp. 28–51. Springer, Heidelberg (2012)
17. Gimbert, H., Oualhadj, Y.: Probabilistic Automata on Finite Words: Decidable and Undecidable Problems. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 527–538. Springer, Heidelberg (2010)
18. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: On the Complexity of the Equivalence Problem for Probabilistic Automata. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 467–481. Springer, Heidelberg (2012)
19. Marx, M.: Conditional XPath. ACM Transactions on Database Systems 30(4), 929–959 (2005)
20. Paz, A.: Introduction to probabilistic automata (Computer science and applied mathematics). Academic Press (1971)
21. Rabin, M.O.: Probabilistic automata. Inform. and Control 6, 230–245 (1963)
22. Ravikumar, B.: On some variations of two-way probabilistic finite automata models. Theor. Comput. Sci. 376, 127–136 (2007)
23. Sakarovitch, J.: Rational and recognizable power series. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. EATCS Monographs in Theoretical Computer Science, ch. 4, pp. 103–172. Springer (2009)
24. Schützenberger, M.P.: On the definition of a family of automata. Inform. and Control 4, 245–270 (1961)
25. Segala, R.: Probability and Nondeterminism in Operational Models of Concurrency. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 64–78. Springer, Heidelberg (2006)
26. Silva, A., Bonchi, F., Bonsangue, M.M., Rutten, J.: Quantitative Kleene coalgebras. Inf. Comput. 209(5), 822–849 (2011)
27. ten Cate, B., Segoufin, L.: XPath, transitive closure logic, and nested tree walking automata. In: Proc. of PODS 2008, pp. 251–260. ACM (2008)

28. Tzeng, W.-G.: A polynomial-time algorithm for the equivalence of probabilistic automata. SIAM J. Comput. 21(2), 216–227 (1992)
29. van Glabbeek, R.J., Smolka, S.A., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Inform. and Comput. 121(1), 59–80 (1995)
30. Weidner, T.: Probabilistic Automata and Probabilistic Logic. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 813–824. Springer, Heidelberg (2012)

Higher-Order Approximations for Verification of Stochastic Hybrid Systems

Sadegh Esmaeil Zadeh Soudjani and Alessandro Abate

Delft Center for Systems & Control, TU Delft,
Mekelweg 2, 2628 CD, Delft, The Netherlands
{S.EsmaeilZadehSoudjani,A.Abate}@tudelft.nl

Abstract. This work investigates the approximate verification of probabilistic specifications expressed as any non-nested PCTL formula over Markov processes on general state spaces. The contribution puts forward new algorithms, based on higher-order function approximation, for the efficient computation of approximate solutions with explicit bounds on the error. Approximation error related to higher-order approximations can be substantially lower than those for piece-wise constant (zeroth-order) approximations known in the literature and, unlike the latter, can display convergence in time to a finite value. Furthermore, higher-order approximation procedures, which depend on the partitioning of the state space, can lead to lower partition cardinality than the related piece-wise constant ones. The work is first presented for Markov processes over Euclidean spaces and thereafter extended to hybrid spaces characterizing models known as Stochastic Hybrid Systems.

Keywords: General State-Space Markov Processes, Stochastic Hybrid Systems, PCTL Verification, Bounded-Until and Reach-Avoid, Interpolation Theory.

1 Introduction and background

This work addresses the investigation of complex properties over Markov processes evolving in discrete time over continuous (uncountable) state spaces [10,14]. We are in particular interested in Markov models with state spaces displaying a *hybrid* structure, namely characterized by a finite collection of bounded continuous domains (typically taken to be subsets of Euclidean spaces). These models are known in the literature as Stochastic Hybrid Systems (SHS) [6,7].

With regards to the probabilistic properties under investigation in this work, we focus on formulae expressed via a modal logic known as PCTL [4]. PCTL encodes probabilistic specifications that can be equivalently expressed via value functions [15] and computed by recursive application of known operators or by solving integral equations, as typical in dynamic programming problems over continuous spaces [5]. This work zooms in on autonomous models (namely, on models admitting no controller, nor scheduler, nor non-determinism), on non-nested PCTL specifications, and mostly on finite-horizon properties that admit

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 416–434, 2012.

a finite recursive expression. Extensions to non-autonomous models have been explored in [3], whereas to infinite-horizon specifications in [17].

With focus on a particular PCTL specification expressing probabilistic invariance, the work in [1] has put forward a formal connection between the study of probabilistic invariance over SHS and the computation of a related property over a discretized version of the model, namely a Markov chain (MC) – the latter property can be computed with a probabilistic model checker, such as PRISM [11] or MRMC [12]. The work in [2] has extended the approach to automata-based properties. Both contributions are formal in that they allow an exact computation of a bound on the formula-dependent approximation error. Recent contributions in [8,9] have investigated the development of enhanced computational approaches with tightened bounds on the error, to translate a SHS into a MC with the end goal of model checking PCTL formulae. In approximating SHS as MC, the surveyed results [1,2,8,9] have leveraged piece-wise constant interpolations of the kernels characterizing the SHS models under study, which has direct consequences on the derived error bounds. In contrast, this work provides approximation methods via higher-order interpolations of the value functions that are aimed at requiring less computational effort. More precisely, drawing on the expression of non-nested PCTL formulae as value functions [1,15], this work builds on the premises in [1,2,8,9] and puts forward higher-order approximation methods, obtained via interpolation procedures, in order to express the value functions under study as compactly as possible. The claim is that using higher-order interpolations (versus piece-wise constant ones) can be beneficial in terms of obtaining tighter bounds on the approximation error. Furthermore, since the approximation procedures depend on the partitioning of the state space, higher-order schemes display an interesting tradeoff between more parsimonious representations versus more complex local computation – this work explores the computational compromise between partition size and local interpolation. In assessing the computability of the results, an underlying tenet is that the total number of integrations required in the interpolation is a proxy for total computational time. An additional advantage of the present study over previous work is that in some cases the approximation error converges in time, which allows the applicability of the method to the approximate solution of infinite-horizon PCTL properties.

The article is structured as follows: Section 2 introduces a general state-space Markov process and zooms in on a specific PCTL formula – finite-time *bounded-until* – equivalently expressing it, via value functions, as a bounded-horizon *reach-avoid* problem. Section 3 considers higher-order approximation schemes over the value functions of interest, and quantifies explicitly the introduced approximation error over the formula (or problem). Section 4 tailors the results to a well studied case in the literature, and specializes the proposed approach to explicit schemes for low-dimensional models and known interpolation bases. Section 5 extends the results to SHS models. Finally, Section 6 develops a few numerical case studies to test and benchmark the proposed schemes. Due to length limitations, the statements are presented without proofs.

2 PCTL *bounded-until* Formula as a *reach-avoid* Problem

Consider a discrete-time, homogeneous Markov process $X = (X_n)_{n\in\mathbb{N}}$, taking values on a general (namely, uncountable) state space $\mathscr{S}$, with $\mathscr{B}(\mathscr{S})$ representing the associated σ-algebra. The evolution of the Markov process is fully characterized by a transition kernel $T(dy|x)$ as:

$$T(\mathcal{A}|x) = P_x\{X_{n+1} \in \mathcal{A}|X_n = x\}, \quad \forall \mathcal{A} \in \mathscr{B}(\mathscr{S}), \quad n \geq 0.$$

In this work we suppose that the transition kernel $T(dy|x)$ of the Markov process admits a density function $t(y|x)$, such that $T(dy|x) = t(y|x)dy$. We consider a bounded-until PCTL formula over a finite time horizon $[0, N]$ and express it as a reach-avoid problem over that time horizon. Given two Borel measurable bounded sets $\mathcal{A} \in \mathscr{B}(\mathscr{S})$ and $\mathcal{B} \subset \mathcal{A}$, we are interested in computing the probability that executions of the Markov process reach the target set $\mathcal{B}$, while never leaving the safe set $\mathcal{A}$ (that is, while avoiding $\mathcal{A}^c$) during the time horizon $[0, N]$, namely [16]:

$$P_x(\mathcal{A},\mathcal{B}) \doteq P\{\exists k \in [0, N], X_k \in \mathcal{B} \wedge \forall l \in [0, k-1], X_l \in \mathcal{A}|X_0 = x\}. \qquad (1)$$

(Notice that the expression above holds also for $k = 0$ since $\mathcal{B} \subset \mathcal{A}$, and can easily be extended to the case where $\mathcal{B} \cap \mathcal{A} = \emptyset$.) Given a probability level $\epsilon \in [0, 1]$ and the inequality operator $\sim \in \{>, \geq, <, \leq\}$, the quantity in (1) can be employed to perform a satisfiability check over the corresponding bounded-until PCTL formula, namely:

$$P_x(\mathcal{A},\mathcal{B}) \sim \epsilon \quad \Leftrightarrow \quad x \models \mathbb{P}_{\sim\epsilon}\{\mathcal{A}\,\mathcal{U}^{\leq N}\mathcal{B}\}.$$

Next, we show that the quantity in (1), characterizing the satisfiability set of the bounded-until PCTL formula, can be equivalently expressed by introducing time-dependent value functions $W_k : \mathscr{S} \to [0, 1], k \in [0, N]$, which lead to the alternative expression $P_x(\mathcal{A},\mathcal{B}) = W_N(x)$. The value functions W_k are obtained recursively according to the following Bellman scheme, which characterizes the reach-avoid problem in (1) [16]:

$$W_{k+1}(x) = \mathbb{1}_{\mathcal{B}}(x) + \mathbb{1}_{\mathcal{A}\setminus\mathcal{B}}(x) \int_{\mathscr{S}} W_k(y)T(dy|x), \quad k \in [0, N-1], \qquad (2)$$

initialized as $W_0(x) = \mathbb{1}_{\mathcal{B}}(x), \forall x \in \mathscr{S}$, and where $\mathbb{1}_{\mathcal{C}}$ denotes the indicator function of set $\mathcal{C} \subseteq \mathscr{S}$. The Belman recursion in (2) indicates that the value functions W_k are always equal to one within the target set $\mathcal{B}$, while their supports are contained in the set $\mathcal{A}$ (namely, they are equal to zero over the complement of $\mathcal{A}$). We are thus only interested in computing the value functions over the set $\mathcal{A}\setminus\mathcal{B}$, which allows simplifying the recursion in (2) as follows, for $k \in [0, N-1]$:

$$W_{k+1}(x) = T(\mathcal{B}|x) + \int_{\mathcal{A}\setminus\mathcal{B}} W_k(y)T(dy|x), \quad W_0(x) = 0, \quad \forall x \in \mathcal{A}\setminus\mathcal{B}. \qquad (3)$$

Let us denote with $\mathbb{B}(\mathcal{A}\backslash\mathcal{B})$ the space of bounded and measurable functions $f : \mathcal{A}\backslash\mathcal{B} \rightarrow \mathbb{R}$, and let us assign to this space the infinity norm $\|f\|_\infty = \sup\{|f(x)|, x \in \mathcal{A}\backslash\mathcal{B}\}$, $\forall f \in \mathbb{B}(\mathcal{A}\backslash\mathcal{B})$. The affine operator $\mathcal{R}_{\mathcal{A},\mathcal{B}}$, defined over $\mathbb{B}(\mathcal{A}\backslash\mathcal{B})$ by

$$\mathcal{R}_{\mathcal{A},\mathcal{B}}f(x) = T(\mathcal{B}|x) + \int_{\mathcal{A}\backslash\mathcal{B}} f(y)T(dy|x), \quad \forall f \in \mathbb{B}(\mathcal{A}\backslash\mathcal{B}), \quad \forall x \in \mathcal{A}\backslash\mathcal{B}, \tag{4}$$

characterizes the solution of the recursion in (3) as $W_k(x) = \mathcal{R}^k_{\mathcal{A},\mathcal{B}}(W_0)(x)$, for any $k = 1, 2, ..., N$.

3 Approximation Schemes and Error Quantification

The solution of the recursion in (3) cannot be characterized analytically in general. The goal of this section is to propose numerical schemes for approximating the value functions $W_k, k = 0, 1, \ldots, N$, with an explicit quantification of the approximation error. While previous work proposed approximations of the value functions W_k by piece-wise constant functions [1,2,8,9], in this contribution we are interested in considering approximations via higher-order interpolations.

3.1 Quantification of the Error of a Projection over a Function Space

Consider a function space $\varPhi = span\{\phi_1(x), \phi_2(x), \cdots, \phi_n(x)\}$ as a subset of $\mathbb{B}(\mathcal{A}\backslash\mathcal{B})$, and a projection operator $\varPi_{\mathcal{A}\backslash\mathcal{B}} : \mathbb{B}(\mathcal{A}\backslash\mathcal{B}) \rightarrow \varPhi$ that satisfies the inequality

$$\|\varPi_{\mathcal{A}\backslash\mathcal{B}}(f) - f\|_\infty \leq \mathcal{E}(f) \tag{5}$$

under some regularity conditions on f (beyond $f \in \mathbb{B}(\mathcal{A}\backslash\mathcal{B})$, see assumptions in Theorem 1), and where the bound $\mathcal{E}$ depends on the properties of the function f. With focus on a linear projection operator, the next result provides a useful tool for approximating the solution of the reach-avoid problem.

Theorem 1. *Assume that a linear operator $\varPi_{\mathcal{A}\backslash\mathcal{B}}$ satisfies the inequality*

$$\left\|\varPi_{\mathcal{A}\backslash\mathcal{B}}(t(y|\cdot)) - t(y|\cdot)\right\|_\infty \leq \epsilon, \quad \forall y \in \mathcal{A}, \tag{6}$$

and that there exists a finite constant $\mathcal{M}$, such that

$$\int_{\mathcal{A}\backslash\mathcal{B}} \left|\varPi_{\mathcal{A}\backslash\mathcal{B}}(t(y|x))\right| dy \leq \mathcal{M}, \quad \forall x \in \mathcal{A}\backslash\mathcal{B}. \tag{7}$$

Define the value functions $\bar{W}_k$ as approximations of the value functions W_k (cfr. (4)), by

$$\bar{W}_k = (\varPi_{\mathcal{A}\backslash\mathcal{B}}\mathcal{R}_{\mathcal{A},\mathcal{B}})^k(W_0), \quad k = 0, 1, \ldots, N. \tag{8}$$

Then it holds that

$$\|W_k - \bar{W}_k\|_\infty \leq E_k, \quad k = 1, 2, ..., N, \tag{9}$$

where the error E_k satisfies the difference equation

$$E_{k+1} = \mathcal{M}E_k + \mathcal{L}(\mathcal{A})\epsilon,$$

initialized by $E_0 = 0$, and where $\mathcal{L}(\mathcal{A})$ denotes the Lebesgue measure of the set $\mathcal{A}$.

Corollary 1. *Under the assumptions raised in (6)-(7), the error E_k can be alternatively expressed explicitly as*

$$E_k = \epsilon\mathcal{L}(\mathcal{A})\frac{1-\mathcal{M}^k}{1-\mathcal{M}},\ \textit{for } \mathcal{M} \neq 1, \quad \textit{and} \quad E_k = \epsilon\mathcal{L}(\mathcal{A})k,\ \textit{for } \mathcal{M} = 1.$$

One possible general choice for the constant $\mathcal{M}$ is $\mathcal{M} = 1 + \epsilon\mathcal{L}(\mathcal{A}\backslash\mathcal{B})$.

Notice that the above error converges if $\mathcal{M} < 1$ as k goes to infinity, which makes the result applicable to the approximate computation of the infinite-horizon reach-avoid property (unbounded-until operator) with a finite approximation error.

3.2 Construction of the Projection Operator

In the ensuing sections we focus, for the sake of simplicity, on a state space that is Euclidean, namely $\mathscr{S} = \mathbb{R}^d$, where d is its finite dimension. In Section 5 we extend the upcoming results to be valid over general models known as Stochastic Hybrid Systems.

We discuss a general form for the interpolation operator. Let $\phi_j : \mathcal{D} \subset \mathbb{R}^d \to \mathbb{R}, j = 1, \cdots, n$, be independent functions defined over a generic set $\mathcal{D}$. The interpolation operator $\mathit{\Pi}_{\mathcal{D}}$ is defined as a projection map into the function space $\mathit{\Phi} = span\{\phi_1(x), \phi_2(x), \cdots, \phi_n(x)\}$, which projects any function $f : \mathcal{D} \to \mathbb{R}$ to a unique function $\mathit{\Pi}_{\mathcal{D}}(f) = \sum_{j=1}^{n} \alpha_j\phi_j$, using a finite set of data $\{(x_j, f(x_j))|x_j \in \mathcal{D}, j = 1, \cdots, n\}$ and such that $\mathit{\Pi}_{\mathcal{D}}(f)(x_j) = f(x_j)$. The operator $\mathit{\Pi}_{\mathcal{D}}$ is guaranteed to verify the inequality in (5), namely $\|\mathit{\Pi}_{\mathcal{D}}(f) - f\|_\infty \leq \mathcal{E}_{\mathcal{D}}(f)$, under some regularity assumptions on its argument function f (cfr. Corollary 2).

With focus on the problem described in Section 2, let us select a partition $\{\mathcal{D}_i\}_{i=1}^m$ for the set $\mathcal{A}\backslash\mathcal{B}$, with finite cardinality m. Using a basis $\{\phi_{ij}\}_{j=1}^n$, let us introduce the interpolation operators $\mathit{\Pi}_{\mathcal{D}_i}$ for the projection over each partition set $\mathcal{D}_i$, which is done as described above by replacing the domain $\mathcal{D}$ with $\mathcal{D}_i$. Finally, let us introduce the (global) linear operator $\mathit{\Pi}_{\mathcal{A}\backslash\mathcal{B}}$ on a function $f : \mathcal{A}\backslash\mathcal{B} \to \mathbb{R}$ by

$$\mathit{\Pi}_{\mathcal{A}\backslash\mathcal{B}}(f) = \sum_{i=1}^{m} \mathbb{1}_{\mathcal{D}_i}\mathit{\Pi}_{\mathcal{D}_i}(f|_{\mathcal{D}_i}), \tag{10}$$

where $f|_{\mathcal{D}_i}$ represents the restriction of the function f over the partition set $\mathcal{D}_i$. The following result holds:

Theorem 2. *The operator in (10) satisfies the inequality in (5) with constant $\mathcal{E}(f) = \max_{i=1,\ldots,m} \mathcal{E}_{\mathcal{D}_i}(f|_{\mathcal{D}_i})$, and where $\|\mathit{\Pi}_{\mathcal{D}_i}(f|_{\mathcal{D}_i}) - f|_{\mathcal{D}_i}\|_\infty \leq \mathcal{E}_{\mathcal{D}_i}(f|_{\mathcal{D}_i})$.*

Corollary 2. *The result in Theorem 1 can be tailored to the operator in* (10) *and applied to the density $t = f$, under the assumptions* (6)-(7) *on t and using the following two quantities:*

$$\epsilon = \max_i \epsilon_i, \text{ where } \|\mathit{\Pi}_{\mathcal{D}_i}(t(y|\cdot)|_{\mathcal{D}_i}) - t(y|\cdot)|_{\mathcal{D}_i}\|_\infty \leq \epsilon_i, \text{ for all } y \in \mathcal{A};$$

$$\mathcal{M} = \max_i \mathcal{M}_i, \text{ where } \int_{\mathcal{A}\setminus\mathcal{B}} |\mathit{\Pi}_{\mathcal{D}_i}(t(y|x))|\, dy \leq \mathcal{M}_i, \text{ for all } x \in \mathcal{D}_i.$$

Here ϵ_i represents the interpolation error on the density function over the partition set $\mathcal{D}_i$.

3.3 Approximation Algorithm

An advantage of the interpolation operator in (10) is that $\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}(f)$ is fully characterized by the interpolation coefficients α_{ij}, such that

$$\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}(f) = \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij}\phi_{ij}\mathbb{1}_{\mathcal{D}_i}.$$

The set of interpolation coefficients α_{ij} are computable by matrix multiplication based on the data $\{f(x_{ij})\}_{i,j=1}^{m,n}$, where the matrix depends on the interpolation points x_{ij} and on the basis functions ϕ_{ij} and can be computed off-line (see step 5 in Algorithm 1).

Let us now focus on the recursion in (8), $\bar{W}_{k+1} = \mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}\mathcal{R}_{\mathcal{A},\mathcal{B}}(\bar{W}_k)$, given the initialization $\bar{W}_0 = 0$, for the approximate computation of the value functions. This recursion indicates that the approximate value functions $\bar{W}_k, k = 1, \ldots, N$, belong to the image of the operator $\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}$. Let us express these value functions by

$$\bar{W}_k = \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij}^k\phi_{ij}\mathbb{1}_{\mathcal{D}_i},$$

where α_{ij}^k denote the interpolation coefficients referring to $\bar{W}_k$ (at step k). This suggests that we need to store and update the coefficients α_{ij}^k for each iteration in (8). Writing the recursion in the form $\bar{W}_{k+1} = \mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}\left(\mathcal{R}_{\mathcal{A},\mathcal{B}}(\bar{W}_k)\right)$ indicates that it is sufficient to evaluate the function $\mathcal{R}_{\mathcal{A},\mathcal{B}}(\bar{W}_k)$ over the interpolation points in order to compute the coefficients α_{ij}^{k+1}. In the following, the pair i, s indicate the indices of the related partition sets, namely $\mathcal{D}_i, \mathcal{D}_s$, whereas the pair of indices j, t show the ordering positions within partition sets. For an arbitrary interpolation point x_{st} we have:

$$\begin{aligned}\mathcal{R}_{\mathcal{A},\mathcal{B}}(\bar{W}_k)(x_{st}) &= T(\mathcal{B}|x_{st}) + \int_{\mathcal{A}\setminus\mathcal{B}} \bar{W}_k(y)t(y|x_{st})dy\\ &= T(\mathcal{B}|x_{st}) + \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij}^k \int_{\mathcal{D}_i} \phi_{ij}(y)t(y|x_{st})dy.\end{aligned}$$

Introducing the following quantities

$$Q(s,t) = \int_{\mathcal{B}} t(y|x_{st})dy, \quad P_{ij}(s,t) = \int_{\mathcal{D}_i} \phi_{ij}(y)t(y|x_{st})dy,$$

we have that

$$\bar{W}_{k+1}(s,t) = \mathcal{R}_{\mathcal{A},\mathcal{B}}(\bar{W}_k)(x_{st}) = Q(s,t) + \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij}^k P_{ij}(s,t).$$

Algorithm 1 provides a general procedure for the discrete computation of the interpolation coefficients and of the approximate value functions.

Algorithm 1. Approximate computation of the value functions $\bar{W}_k$

```
Require: Density function $t(y|x)$, safe set $\mathcal{A}\backslash\mathcal{B}$
 1: Select a finite $m$-dimensional partition of the set $\mathcal{A}\backslash\mathcal{B} = \cup_{i=1}^m \mathcal{D}_i$ ($\mathcal{D}_i$ are non-overlapping)
 2: For each $\mathcal{D}_i$, select interpolation basis functions $\phi_{ij}$ and points $x_{ij} \in \mathcal{D}_i$, where $j = 1, \dots, n$
 3: Compute $P_{ij}(s,t) = \int_{\mathcal{D}_i} \phi_{ij}(y)t(y|x_{st})dy$, where $1 \le i, s \le m$ and $1 \le j, t \le n$
 4: Compute matrix $Q$ with entries $Q(s,t) = \int_{\mathcal{B}} t(y|x_{st})dy$
 5: Compute matrix representation of operators $\Pi_{\mathcal{D}_i}$
 6: Set $k = 0$ and $\bar{W}_0(i,j) = 0$ for all $i, j$
 7: if $k < N$ then
 8:    Compute interpolation coefficients $\alpha_{ij}^k$ given $\bar{W}_k(i,j)$, using matrices in step 5
 9:    Compute values $\bar{W}_{k+1}(s,t)$ based on $\bar{W}_{k+1}(s,t) = Q(s,t) + \sum_i \sum_j \alpha_{ij}^k P_{ij}(s,t)$
10:    $k = k + 1$
11: end if
Ensure: Approximate value functions $\bar{W}_k, k = 0, 1, \dots, N$
```

Next, we provide a condition on the selection of the basis functions and of the interpolation points, leading to a simplification of Algorithm 1.

Theorem 3 ([13]). *Assume that there exists a choice of interpolation points x_{ij} and of basis functions ϕ_{ij} such that*

$$\det \begin{bmatrix} \phi_{i1}(x_{i1}) & \cdots & \phi_{in}(x_{i1}) \\ \vdots & \ddots & \vdots \\ \phi_{i1}(x_{in}) & \cdots & \phi_{in}(x_{in}) \end{bmatrix} \neq 0, \quad \forall i \in \{1, 2, \cdots, m\}.$$

Then, there additionally exists an equivalent basis made up of functions ψ_{ij} such that

$$span\{\psi_{i1}, \psi_{i2}, \cdots, \psi_{in}\} = span\{\phi_{i1}, \phi_{i2}, \cdots, \phi_{in}\}$$

for all i, and which is related to the interpolation coefficients $\alpha_{ij}^k = \bar{W}_k(i,j)$.

Theorem 3 ensures that by utilizing the basis functions ψ_{ij} step 5 in Algorithm 1 can be skipped, and that the main update (steps 8 and 9) can be simplified as follows:

$$\bar{W}_{k+1}(s,t) = Q(s,t) + \sum_{i=1}^{m}\sum_{j=1}^{n} \bar{W}_k(i,j)P_{ij}(s,t), \quad \bar{W}_0(i,j) = 0.$$

A sufficient condition for the satisfaction of the assumption in Theorem 3 is the selection of a basis $\{\phi_{i1}, \cdots, \phi_{in}\}$ as a Chebyshev (or Haar) system [13], for all i. In this case, the choice of the distinct interpolation points x_{ij} can be made freely, for each partition set $\mathcal{D}_i$ (instances of this selection will be given below).

In Algorithm 1, the interpolation points x_{ij} are in general pair-wise distinct. Extending the domain of interpolation $\mathcal{D}_i$ to its closure $\bar{\mathcal{D}}_i$, it is legitimate to use boundary points as interpolation points, which can lead to a reduction of the number of integrations required in Algorithm 1. However, special care should be taken, since the interpolation operator should produce a continuous output over the boundaries of the neighboring partition sets. In the ensuing sections, we will exploit this feature upon selecting equally spaced points.

4 Special Forms of the Projection Operator

In this section we leverage known interpolation theorems for the construction of the projection operator $\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}$. These theorems are presented over a general domain $\mathcal{D}$ and are then used to derive specific error bounds for the problem of interest presented in Section 2.[1]

4.1 Piece-Wise Constant Approximations

We focus on the approximation of a function by a piece-wise constant one, which has inspired the previous work in [1,2,8,9]. The procedure is detailed in Algorithm 2, while the associated error is quantified in Theorem 4.

Consider a continuous, partially differentiable scalar field $f : \mathcal{D} \subset \mathbb{R}^d \to \mathbb{R}$ such that $\|\frac{\partial f}{\partial x}\| \leq M_0, \forall x \in \mathcal{D}$. Then $|f(x) - f(x')| \leq M_0\|x - x'\|, \forall x, x' \in \mathcal{D}$.

Theorem 4. *Suppose the density function $t(\cdot|x)$ is Lipschitz continuous with constant $\mathcal{M}_0$:*

$$|t(y|x) - t(y|x')| \leq \mathcal{M}_0\|x - x'\|, \quad \forall x, x' \in \mathcal{A}\setminus\mathcal{B}.$$

Then the approximation error of Algorithm 2 is upper bounded by the quantity $N\mathcal{L}(\mathcal{A})\mathcal{M}_0\delta$, where $\delta = \max_i \delta_i$ is the partition size of $\cup_{i=1}^{m}\mathcal{D}_i = A\setminus\mathcal{B}$, with $\delta_i = \sup\{\|x - x'\| : x, x' \in \mathcal{D}_i\}$.

[1] In the rest of the article, we employ normal typeset for bounds derived from general interpolation theorems, whereas calligraphic letters are used for theorems developed specifically for this article.

Algorithm 2. Piece-wise constant computation of the value functions $\bar{W}_k$

```
Require: Density function $t(y|x)$, safe set $\mathcal{A}\backslash\mathcal{B}$
1: Select a finite $m$-dimensional partition of the set $\mathcal{A}\backslash\mathcal{B} = \cup_{i=1}^m \mathcal{D}_i$ ($\mathcal{D}_i$ are non-overlapping)
2: For each $\mathcal{D}_i$, select one representative point $x_i \in \mathcal{D}_i$
3: Compute matrix $P$ with entries $P(i,j) = \int_{\mathcal{D}_i} t(y|x_j)dy$, where $1 \leq i,j \leq m$
4: Compute vector $Q$ with entries $Q(j) = \int_{\mathcal{B}} t(y|x_j)dy$
5: Set $k = 0$ and $\bar{W}_0(i) = 0$ for all $i$
6: if $k < N$ then
7:    Compute the vector $\bar{W}_{k+1}$ based on $\bar{W}_{k+1} = Q + \bar{W}_k P$
8:    $k = k + 1$
9: end if
Ensure: Approximate value functions $\bar{W}_k, k = 0, 1, \ldots, N$
```

Notice that in some cases [17] it is possible to find a constant $\mathcal{M} = \max_{x\in\mathcal{A}\backslash\mathcal{B}} \int_{\mathcal{A}\backslash\mathcal{B}} t(y|x)dy$ that is less than one, which leads to an error (cfr. Corollary 1) that converges as time horizon N grows.

Let us compare Algorithms 1 and 2 in terms of their computational complexity. Algorithm 1 requires $mn(mn+1)$ integrations in the marginalization steps (3 and 4), whereas $m(m+1)$ integrations are required in Algorithm 2. Furthermore, steps 5 and 8 in Algorithm 1 can be skipped only if a Chebyshev (Haar) system can be selected, whereas these steps are not needed at all in Algorithm 2. As a bottom line, higher interpolation orders increase the computational complexity of the approximation procedure, however this can as well lead to a lower global approximation error. Since the global approximation error depends on the local partitioning sets (their diameter, size, and the local continuity of the density function), for a given error higher interpolation procedures may require partitions with lower cardinality.

4.2 Higher-Order Approximations for One-Dimensional Systems

In this section we study higher-order interpolations over the real axis, where the partition sets $\mathcal{D}_i$ are real intervals. We use this simple setting to quantify the error related to the approximate solution of the reach-avoid problem. In order to assess the effect of the choice of the interpolation points on the approximation error and on the computational complexity of the method, we compare two different sets of interpolation points: equally spaced points and Chebyshev nodes.

Theorem 5 ([13]). *Let f be a real $(n+1)$-times continuously differentiable function on the bounded (one-dimensional) interval $\mathcal{D} = [\alpha, \beta]$. For the interpolation polynomial $\Pi_{\mathcal{D}}(f) \in span\{1, x, x^2, ..., x^n\}$, with $(n+1)$ pair-wise distinct points $\{x_0, x_1, ..., x_n\} \subset \mathcal{D}$, and condition $\Pi_{\mathcal{D}}(f)(x_j) = f(x_j), j = 0, \ldots, n$, there exist a $\xi \in \mathcal{D}$ such that*

$$f(x) - \Pi_{\mathcal{D}}(f)(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j), \quad \forall x \in \mathcal{D}.$$

Equally spaced interpolation points. The following result can be adapted from [13].

Theorem 6. *Consider equally spaced interpolation points $x_0, x_1, ..., x_n$:*

$$x_j = \alpha + j\frac{\beta - \alpha}{n}, \quad j = 0, 1, 2, ..., n.$$

The interpolation error is upper bounded, $\forall x \in \mathcal{D}$, by

$$|f(x) - \mathit{\Pi}_{\mathcal{D}}(f)(x)| \leq \frac{M_n}{4(n+1)} \left(\frac{\beta - \alpha}{n}\right)^{n+1},$$

where $M_n = \max_{x \in \mathcal{D}} |f^{n+1}(x)|$.

Application to the reach-avoid problem. Consider a one dimensional reach-avoid problem with a partitioning of $\mathcal{A} \backslash \mathcal{B} = \cup_{i=1}^{m} \mathcal{D}_i$ which is such that $\mathcal{D}_i = [\alpha_i, \beta_i]$. Define the interpolation operator $\mathit{\Pi}_{\mathcal{D}_i}(t|_{\mathcal{D}_i})$ over the basis $\mathit{\Phi} = span\{1, x, x^2, ..., x^n\}$ using equally spaced interpolation points $x_{ij} \in \mathcal{D}_i, j = 0, \ldots, n$. Then we can easily derive the following constants:

$$\mathcal{M}_n = \max_{x,y \in \mathcal{A} \backslash \mathcal{B}} \left| \frac{\partial^{n+1} t(y|x)}{\partial x^{n+1}} \right|, \qquad \epsilon = \frac{\mathcal{M}_n}{4(n+1)} \left(\frac{\delta}{n}\right)^{n+1},$$

and $\delta_i = \beta_i - \alpha_i, \delta = \max_i \delta_i, i = 1, 2, ..., m$. Changing the basis of interpolation gives us the opportunity to obtain another value for $\mathcal{M}$ to be used in the error computation. Let us select the interpolation basis functions to be Lagrange polynomials:

$$L_{ij}(x) = \prod_{s=1, s \neq j}^{n+1} \frac{x - x_{is}}{x_{ij} - x_{is}}.$$

This leads to a projection with a special form, namely $\mathit{\Pi}_{\mathcal{D}_i}(t(y|x)|_{\mathcal{D}_i}) = \sum_{j=1}^{n+1} \alpha_{ij} x^{j-1} = \sum_{j=1}^{n+1} t(y|x_{ij}) L_{ij}(x)$. Computing the constants $\kappa_i = \max_{x \in \mathcal{D}_i} \sum_{j=1}^{n+1} |L_{ij}(x)|$ yields the following choice of $\mathcal{M}$:

$$\int_{\mathcal{A} \backslash \mathcal{B}} |\mathit{\Pi}_{\mathcal{D}_i}(t(y|x)|_{\mathcal{D}_i})|\, dy \leq \kappa_i \int_{\mathcal{A} \backslash \mathcal{B}} t(y|x_{ij}) dy \leq \kappa_i, \text{ and } \mathcal{M} = \max_i \kappa_i.$$

Having the values of ϵ and $\mathcal{M}$ we are ready to implement Algorithm 1 for equally spaced points and polynomial basis functions of degree at most n, with the pre-specified error of Theorem 1.

Chebyshev nodes. The following statement can be adapted from [13].

Theorem 7. *Let f be a real $(n+1)$-times continuously differentiable function on the bounded interval $\mathcal{D} = [\alpha, \beta]$. For the interpolation polynomial $\Pi_{\mathcal{D}}(f) \in$ span$\{1, x, x^2, ..., x^n\}$ with Chebyshev nodes*

$$x_j = \frac{\alpha+\beta}{2} + \frac{\beta-\alpha}{2}\cos\left(\frac{2j+1}{2(n+1)\pi}\right), \quad j = 0, 1, 2, ..., n,$$

and values $\Pi_{\mathcal{D}}(f)(x_j) = f(x_j)$, we have

$$|f(x) - \Pi_{\mathcal{D}}(f)(x)| \leq \frac{M_n}{2^n(n+1)!}\left(\frac{\beta-\alpha}{2}\right)^{n+1}, \quad \forall x \in \mathcal{D},$$

where $M_n = \max_{x\in\mathcal{D}} |f^{n+1}(x)|$.

Application to the reach-avoid problem. We can implement Algorithm 1 for Chebyshev nodes and Chebyshev polynomials of degree n, given a pre-specified error in Theorem 1, and with the following value of ϵ:

$$\epsilon = \frac{\mathcal{M}_n}{2^n(n+1)!}\left(\frac{\delta}{2}\right)^{n+1},$$

where the quantity $\mathcal{M}_n$ is that defined for equally spaced points. The only difference between the selection of equally spaced points and of Chebyshev nodes is the value of ϵ. The ratio of ϵ for these two cases (denoted respectively ϵ_1 and ϵ_2) is presented in Table 1 as a function of n (interpolation order). The advantage gained by using Chebyshev nodes is distinctive over larger values of the interpolation order.

Table 1. Ratio between equally spaced pints (ϵ_1) vs. Chebyschev nodes (ϵ_2), expressed with double digit precision, for different orders of interpolation order (n).

n	1	2	3	4	5	6	7	8	9	10	11	12
$\frac{\epsilon_2}{\epsilon_1}$	0.50	0.50	0.42	0.33	0.25	0.19	0.14	0.10	0.07	0.05	0.04	0.03

It is worth mentioning that, unlike the piece-wise constant case [1,2,8,9], with higher-order approximation approaches the global error is a nonlinear function of the partition size δ, namely it depends on a power of the partition size contingent on the order of the selected interpolation operator.

4.3 Bilinear Interpolation for Two-Dimensional Systems

We directly tailor the results above to a general two-dimensional system.

Theorem 8. *Consider a partially differentiable function $f(x_1, x_2)$, defined (for simplicity) over the unit square $\mathcal{D} = [0, 1]^2$. For the interpolation operator*

$$\begin{aligned}\Pi_{\mathcal{D}}(f)(x_1, x_2) =& a_1 + a_2x_1 + a_3x_2 + a_4x_1x_2\\ =& x_1(1-x_2)f(1,0) + x_1x_2f(1,1) + (1-x_1)(1-x_2)f(0,0) + (1-x_1)x_2f(0,1),\end{aligned}$$

the error is upper bounded by

$$\|f - \Pi_{\mathcal{D}}(f)\|_\infty \le \frac{1}{8}\left[M_{x_1^2} + M_{x_2^2} + 2M_{x_1^2 x_2} + 2M_{x_2^2 x_1}\right],$$

where $\left|\frac{\partial^2 f}{\partial x_i^2}\right| \le M_{x_i^2}, \left|\frac{\partial^3 f}{\partial x_i^2 x_{3-i}}\right| \le M_{x_i^2 x_{3-i}}, i = 1, 2, \quad \forall (x_1, x_2) \in \mathcal{D}.$

Application to the reach-avoid problem. With focus on a two-dimensional reach-avoid problem, consider a uniform partition (using squared partition sets) of size δ for the set $\mathcal{A}\backslash\mathcal{B}$. We employ a bilinear interpolation within each partition set $\mathcal{D}_i = [\alpha_{i1}, \alpha_{i2}] \times [\beta_{i1}, \beta_{i2}]$ with basis $\{\phi_1(x) = 1, \phi_2(x) = x_1, \phi_3(x) = x_2, \phi_4(x) = x_1 x_2\}$, or with Lagrange polynomials

$$\psi_{i1}(x) = \frac{(\alpha_{i2} - x_1)(\beta_{i2} - x_2)}{(\alpha_{i2} - \alpha_{i1})(\beta_{i2} - \beta_{i1})}, \qquad \psi_{i2}(x) = \frac{(\alpha_{i2} - x_1)(x_2 - \beta_{i1})}{(\alpha_{i2} - \alpha_{i1})(\beta_{i2} - \beta_{i1})},$$
$$\psi_{i3}(x) = \frac{(x_1 - \alpha_{i1})(\beta_{i2} - x_2)}{(\alpha_{i2} - \alpha_{i1})(\beta_{i2} - \beta_{i1})}, \qquad \psi_{i4}(x) = \frac{(x_1 - \alpha_{i1})(x_2 - \beta_{i1})}{(\alpha_{i2} - \alpha_{i1})(\beta_{i2} - \beta_{i1})},$$

and compute the associated error, given the following value for ϵ:

$$\epsilon = \frac{\delta^2}{16}\left[\mathcal{M}_{x_1^2} + \mathcal{M}_{x_2^2} + \delta\sqrt{2}\mathcal{M}_{x_1^2 x_2} + \delta\sqrt{2}\mathcal{M}_{x_2^2 x_1}\right],$$

where $\left|\frac{\partial^2 t}{\partial x_i^2}(y|x)\right| \le \mathcal{M}_{x_i^2}$, $\left|\frac{\partial^3 t}{\partial x_i^2 x_{3-i}}(y|x)\right| \le \mathcal{M}_{x_i^2 x_{3-i}}, i = 1, 2, \forall x, y \in A\backslash\mathcal{B}$. Note that the basis function ψ_{ij} is non-negative on the partition set $\mathcal{D}_i$ and that $\sum_{j=1}^4 \psi_{ij}(x) = 1$, which leads to a constant $\mathcal{M} = \max_{x\in A\backslash\mathcal{B}} \int_{A\backslash\mathcal{B}} t(y|x)dy \le 1$.

4.4 Trilinear Interpolation for Three-Dimensional Systems

We now apply the results above to a general three-dimensional system.

Theorem 9. *Consider a partially differentiable function $f(x_1, x_2, x_3)$, defined (for simplicity) over the unit cube $\mathcal{D} = [0, 1]^3$. For the interpolation operator*

$$\begin{aligned}\Pi_{\mathcal{D}}(f)(x_1, x_2, x_3) =& a_1 + a_2 x_1 + a_3 x_2 + a_4 x_3 + a_5 x_1 x_2 + a_6 x_1 x_3 + a_7 x_2 x_3 + a_8 x_1 x_2 x_3\\ =&(1 - x_1)(1 - x_2)(1 - x_3) f(0, 0, 0) + x_1 x_2 x_3 f(1, 1, 1)\\ &+ x_1(1 - x_2)(1 - x_3) f(1, 0, 0) + (1 - x_1) x_2 x_3 f(0, 1, 1)\\ &+ (1 - x_1) x_2 (1 - x_3) f(0, 1, 0) + x_1(1 - x_2) x_3 f(1, 0, 1)\\ &+ (1 - x_1)(1 - x_2) x_3 f(0, 0, 1) + x_1 x_2 (1 - x_3) f(1, 1, 0),\end{aligned}$$

the error is upper bounded by the expression

$$\begin{aligned}\|f - \Pi_{\mathcal{D}}(f)\|_\infty \le \frac{1}{8}[&M_{x_1^2} + M_{x_2^2} + M_{x_3^2} + 2M_{x_1^2 x_2} + 2M_{x_2^2 x_1} + 2M_{x_1^2 x_3}\\ &+ 2M_{x_3^2 x_1} + 2M_{x_2^2 x_3} + 2M_{x_3^2 x_2} + 6M_{x_1 x_2 x_3}],\end{aligned}$$

where $\left|\frac{\partial^2 f}{\partial x_i^2}\right| \le M_{x_i^2}, \left|\frac{\partial^3 f}{\partial x_i^2 x_j}\right| \le M_{x_i^2 x_j}, \left|\frac{\partial^3 f}{\partial x_1^2 x_2 x_3}\right| \le M_{x_1 x_2 x_3}, \forall x = (x_1, x_2, x_3) \in \mathcal{D}$.

Application to the reach-avoid problem. With focus on a three-dimensional reach-avoid problem, consider a uniform partition (using cubic sets) of size δ for the set $\mathcal{A}\backslash\mathcal{B}$. We employ a trilinear interpolation within each partition set and compute the associated error, given the following value for ϵ:

$$\begin{aligned}\epsilon = &\frac{\delta^2}{24}\left[\mathcal{M}_{x_1^2}+\mathcal{M}_{x_2^2}+\mathcal{M}_{x_3^2}\right]\\ &+\frac{\delta^3}{12\sqrt{3}}\left[\mathcal{M}_{x_1^2x_2}+\mathcal{M}_{x_2^2x_1}+\mathcal{M}_{x_2^2x_3}+\mathcal{M}_{x_3^2x_2}+\mathcal{M}_{x_1^2x_3}+\mathcal{M}_{x_3^2x_1}+3\mathcal{M}_{x_1x_2x_3}\right],\end{aligned}$$

where, $\forall x=(x_1,x_2,x_3), y=(y_1,y_2,y_3)\in D$, $\left|\frac{\partial^2 t}{\partial x_i^2}(y|x)\right|\le M_{x_i^2}$, $\left|\frac{\partial^3 t}{\partial x_i^2 x_j}(y|x)\right|\le M_{x_i^2x_j}$, and $\left|\frac{\partial^3 t}{\partial x_1^2 x_2 x_3}(y|x)\right|\le M_{x_1x_2x_3}$. Similar to the bilinear interpolation case, the function ψ_{ij} is non-negative on the partition set $\mathcal{D}_i$ and $\sum_{j=1}^{8}\psi_{ij}(x)=1$, which leads to a constant $\mathcal{M}=\max_{x\in\mathcal{A}\backslash\mathcal{B}}\int_{\mathcal{A}\backslash\mathcal{B}}t(y|x)dy\le 1$.

5 Extensions to Stochastic Models with Hybrid State Spaces

Stochastic Hybrid Systems are Markov processes defined over a hybrid state space $\mathscr{S}$ made up of a finite, disjoint union of continuous domains, namely $\mathscr{S}=\cup_{q\in\mathcal{Q}}\{q\}\times\mathbb{R}^{n(q)}$, where $\mathcal{Q}=\{q_1,q_2,\cdots,q_{\mathfrak{m}}\}$, and the function $n:\mathcal{Q}\to\mathbb{N}$ assigns to each discrete location $q\in\mathcal{Q}$ a (finite) dimension for the associated continuous domain $\mathbb{R}^{n(q)}$. The conditional stochastic kernel $T:\mathscr{B}(\mathscr{S})\times\mathscr{S}\to[0,1]$ on $\mathscr{S}$ is fully characterized by three kernels T_q,T_x,T_r, dealing respectively with the discrete evolution over locations, the continuous evolution in the domain of a given location, and the continuous reset between domains of different locations:

$$T(\{q'\}\times A_{q'}|(q,x))=T_q(q'|(q,x))\times\begin{cases}T_x(A_{q'}|(q,x)), & q'=q,\\ T_r(A_{q'}|(q,x),q'), & q'\neq q.\end{cases}$$

Consider a safe set $\mathcal{A}=\cup_{q\in\mathcal{Q}}\{q\}\times\mathcal{A}_q$ and a target set $\mathcal{B}=\cup_{q\in\mathcal{Q}}\{q\}\times\mathcal{B}_q$, where $\mathcal{B}_q\subset\mathcal{A}_q$. Since the conditional kernels T_x,T_r admit density functions t_x,t_r, we can define the operator $\mathcal{R}_{\mathcal{A},\mathcal{B}}$ acting on $f\in\mathbb{B}(\mathcal{A}\backslash\mathcal{B})$ as

$$\begin{aligned}\mathcal{R}_{\mathcal{A},\mathcal{B}}f(q,x)=&\,T(\mathcal{B}|(q,x))+T_q(q|(q,x))\int_{\mathcal{A}_q\backslash\mathcal{B}_q}f(q,y)t_x(y|(q,x))dy\\ &+\sum_{\bar{q}\neq q}T_q(\bar{q}|(q,x))\int_{\mathcal{A}_{\bar{q}}\backslash\mathcal{B}_{\bar{q}}}f(\bar{q},y)t_r(y|(q,x),\bar{q})dy,\quad\forall q\in\mathcal{Q},\forall x\in\mathcal{A}_q\backslash\mathcal{B}_q.\end{aligned}$$

Given a partition $\mathcal{A}_q\backslash\mathcal{B}_q=\cup_i\mathcal{D}_{q,i}$ and a basis of interpolation functions $\{\psi_{q,ij}(x)\}$, we can construct the projection operator $\varPi_{\mathcal{A}\backslash\mathcal{B}}$ on $\mathbb{B}(\mathcal{A}\backslash\mathcal{B})$ by separately interpolating over the continuous domains associated to each discrete location. The following holds:

Theorem 10. *Suppose the conditional kernels of the SHS model satisfy the following inequalities*

$$\begin{aligned}
&\|\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}(T_q(q|(q,\cdot))t_x(y|(q,\cdot))) - T_q(q|(q,\cdot))t_x(y|(q,\cdot))\|_\infty \leq \mathcal{E}_x, \quad \forall q \in \mathcal{Q}, \forall y \in \mathcal{A}_q,\\
&\|\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}(T_q(\bar{q}|(q,\cdot))t_r(y|(q,\cdot),\bar{q})) - T_q(\bar{q}|(q,\cdot))t_r(y|(q,\cdot),\bar{q})\|_\infty \leq \mathcal{E}_r, \quad \forall q,\bar{q} \in \mathcal{Q}, \bar{q} \neq q, \forall y \in \mathcal{A}_{\bar{q}},
\end{aligned}$$

then the following error bound can be established:

$$\begin{aligned}
&\|\mathcal{R}^k_{\mathcal{A},\mathcal{B}}(W_0) - (\mathit{\Pi}_{\mathcal{A}\setminus\mathcal{B}}\mathcal{R}_{\mathcal{A},\mathcal{B}})^k(W_0)\|_\infty \leq E_k, \quad W_0 = 0,\\
&E_{k+1} = \lambda(\mathcal{E}_x + (\mathfrak{m}-1)\mathcal{E}_r) + \kappa E_k, \quad E_0 = 0,
\end{aligned}$$

where $\lambda = \max_q \mathcal{L}(A_q)$, $\kappa = \max\left\{\sum_j |\psi_{q,ij}(x)| \,\middle|\, x \in A_{q,i}, \forall i, q\right\}$, *and* $\mathfrak{m}$ *is the cardinality of the set of discrete locations.*

6 Case Studies

The probabilistic safety (or invariance) problem over a finite time horizon can be defined as follows:

$$P_x(\mathcal{A}) \doteq P\{\forall k \in [0,N], X_k \in \mathcal{A}|X_0 = x\}. \tag{11}$$

Safety is the dual of reachability, which in turn is a special case of the reach-avoid problem. In order to compute the solution of the safety problem over the safe set $\mathcal{A}$, we can compute that of the reach-avoid problem with a safe set $\mathscr{S}$ and a target set $\mathcal{A}^c = \mathscr{S}\setminus\mathcal{A}$. In this instance, the operator $\mathcal{R}_{\mathscr{S},\mathcal{A}^c}$ is used to compute the associated value functions W_k, which leads to the solution of the safety problem as $1 - W_N$. The errors associated to this procedure can be computed exactly as done for the reach-avoid problem. We develop a few case studies to investigate the probabilistic safety problem.

6.1 A One-Dimensional Case Study

Consider a probabilistic safety problem over the safe set $\mathcal{A} = [0,2]$ and the time horizon $N = 10$, over a model characterized by the kernel $T(dy|x) = g(x + c - y)dy$, where $c = 1.3035$, and the function g is defined as:

$$g(t) = \begin{cases} 3.57485\dfrac{1}{t^2}\exp\left(-t - \dfrac{1}{t}\right), & t > 0,\\ 0, & t \leq 0. \end{cases}$$

Selecting an approximation error $E_N = 0.01$, we compute the required number of partition sets to abide by such figure. Using piece-wise constant approximations based on a global Lipschitz constant (cfr. Sec. 4.1) yields a value $\mathcal{M}_0 = 6.90$ and the error function $E_N = N\mathcal{L}(\mathcal{A})\mathcal{M}_0\delta$. This leads to a required number of partition sets $m = 27616$ and a total number of integrations $m(m+1) = 7.6 \times 10^8$ (the number of integrations is here conceived as a proxy for computational complexity).

Now consider algorithms and error bounds developed for higher-order approximations. The constants $\mathcal{M}_n$ are: $\mathcal{M}_1 = 88.93, \mathcal{M}_2 = 2063.65, \mathcal{M}_3 = 79064.41, \mathcal{M}_4 = 5428040$, whereas $\mathcal{M}$ is computed based on the following optimization problem:

$$\mathcal{M} = \max_{x\in\mathcal{A}} \int_{\mathcal{A}} t(y|x)dy = \max_{x\in\mathcal{A}} \int_0^2 g(x+c-y)dy = \max_{x\in\mathcal{A}} \int_{x+c-2}^{x+c} g(u)du,$$

which leads to $x_{opt} = 0.82$ and $\mathcal{M} = 0.96$.

Table 2 compares the number of partition sets and the number of integrations required to reach an approximation error $E_N = 0.01$, using equally spaced points and Chebyshev nodes. Notice that the two methods coincide for $n = 0$. The formulas for the number of integrations are an adaptation of the corresponding ones developed to assess Algorithm 1 (this case deals with invariance, rather than the more general reach-avoid for Algorithm 1). Similar outcomes, performed for an experiment with error $E_N = 0.001$, are also reported. These results show that Chebyshev nodes require in general a lower number of partition sets and therefore fewer integrations. The values are comparable since the ratio ϵ_2/ϵ_1 is smaller for larger values of n, as per Table 1. Notice further that equally spaced points give the opportunity to select common boundary points over adjacent partition sets as interpolation points, which can lead to a reduction on the associated number of integrations. However, interestingly the complexity is in general not monotonically decreasing with the order.

Table 2. Number of partition sets and integrations for equally spaced points (indexed by 1) and for Chebyshev nodes (indexed by 2), given two error bounds $E_N = 0.01, 0.001$.

uniform partitioning	total # of partitions		# of integrations	
$E_N = 0.01$	m_1	m_2	$m_1(n+1)(m_1 n+1)$	$m_2^2(n+1)^2$
piecewise constant, $n = 0$	23357	23357	$5.5 \cdot 10^8$	$5.5 \cdot 10^8$
piecewise linear, $n = 1$	275	194	$1.5 \cdot 10^5$	94864
piecewise quadratic, $n = 2$	67	53	27135	25281
third-order, $n = 3$	36	29	15696	13456
fourth-order, $n = 4$	28	22	15820	12100
uniform partitioning	total # of partitions		# of integrations	
$E_N = 0.001$	m_1	m_2	$m_1(n+1)(m_1 n+1)$	$m_2^2(n+1)^2$
piecewise constant, $n = 0$	233563	233563	$5.5 \cdot 10^{10}$	$5.5 \cdot 10^{10}$
piecewise linear, $n = 1$	868	614	1508584	1507984
piecewise quadratic, $n = 2$	143	114	123123	116964
third-order, $n = 3$	64	52	49408	43264
fourth-order, $n = 4$	43	35	37195	30625

6.2 A Two-Dimensional Case Study

Consider a d-dimensional linear, stochastic difference equation over $\mathbb{R}^d$

$$x(k+1) = Ax(k) + w(k), \quad k \in \mathbb{N},$$

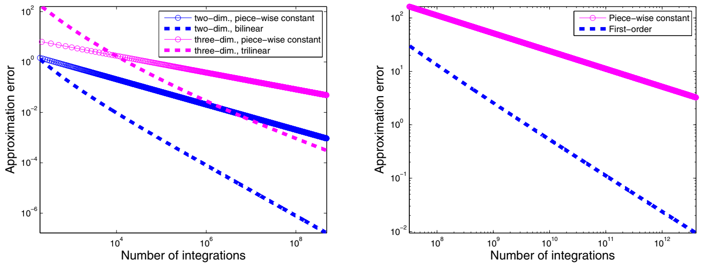
where $w(k), k \geq 0$, is the process noise, taken to be Normal i.i.d. random variables with zero mean and covariance matrix Σ: $w(k) \sim \mathcal{N}(0, \Sigma)$. Given any point $x \in \mathbb{R}^d$ at any time, the distribution at the next time can be characterized by a transition probability kernel $T(\cdot|x) \sim \mathcal{N}(\cdot; Ax, \Sigma)$. For a detailed description of the model and of its parameters the reader may refer to [8]. Let us consider the probabilistic invariance problem over a safe set $\mathcal{A} = [-1, 1]^d$, namely a hypercube pointed at the origin, and a time horizon $[0, N]$. Select a two dimensional state space $d = 2$ and a covariance matrix $\Sigma = 0.5\mathbb{I}_2$. The following constants are needed to compute the error: $\mathcal{M} = 0.71, \quad \mathcal{M}_{x_1^2} = 2.23, \quad \mathcal{M}_{x_2^2} = 0.72, \quad \mathcal{M}_{x_1^2 x_2} = 3.80, \quad \mathcal{M}_{x_1 x_2^2} = 2.17$. Table 3 compares the complexity of piece-wise constant and bilinear approximations, for different values of the global error E_N. Similarly, Figure 1a (on the left) compares the two approximations over the probabilistic safety problem (blue lines). The vertical axis represents the global approximation error, whereas the horizontal axis indicates the corresponding number of integrations, pointing to the computational complexity of each method. For a given computational complexity, bilinear interpolations approximate the solution with less error and their performance is dimensionally better in compared to the piece-wise constant approximations. Similarly, for a given error threshold, less computations are required when using bilinear interpolations.

Table 3. Piece-wise constant versus bilinear approximations

	piece-wise constant		bilinear	
error	# of partitions per dimension	# of integrations	# of partitions per dimension	# of integrations
E_N	m_1	m_1^2	m_2	$4(m_2+1)^2$
0.1	206	$4.2 \cdot 10^4$	18	1444
0.01	2053	$4.2 \cdot 10^6$	49	10^4
0.001	20525	$4.2 \cdot 10^8$	145	$8.5 \cdot 10^4$
0.0001	205241	$4.2 \cdot 10^{10}$	448	$8.1 \cdot 10^5$

6.3 A Three-Dimensional Case Study

Consider the above system with three dimensional state space $d = 3$ and covariance matrix $\Sigma = 0.5\mathbb{I}_3$. The following constants are needed to compute the error: $\mathcal{M} = 0.60, \mathcal{M}_{x_1^2} = 2.66, \mathcal{M}_{x_2^2} = 0.33, \mathcal{M}_{x_3^2} = 1.50, \mathcal{M}_{x_1^2 x_2} = 3.47, \mathcal{M}_{x_2^2 x_1} = 1.28, \mathcal{M}_{x_2^2 x_3} = 0.95, \mathcal{M}_{x_3^2 x_2} = 1.92, \mathcal{M}_{x_1^2 x_3} = 8.37, \mathcal{M}_{x_3^2 x_1} = 6.27, \mathcal{M}_{x_1 x_2 x_3} = 2.56$. Table 4 compares piece-wise constant and trilinear approximations, for different values of the global error E_N. Similarly, Figure 1a (on the left) compares the two approximations over the solution of the safety problem (magenta lines). Recall that there is a tradeoff between local computations and global error for higher-order interpolations. Thus, if we consider a large global error, piece-wise approximations may be computationally favorable (left of the crossing in the magenta curves). However, for small error bounds the performance of trilinear interpolations is much better in comparison with that of piece-wise constant approximations.



(a) 2- and 3-dimensional models, Secs. 6.2, 6.3

(b) Hybrid model, Sec. 6.4

Fig. 1. Error comparison between piece-wise constant versus higher-order approximations, as a function of their computational complexity, for three case studies.

Table 4. Piece-wise constant versus trilinear approximations

	piece-wise constant		trilinear	
error	# of partitions per dimension	# of integrations	# of partitions per dimension	# of integrations
E_N	m_1	m_1^3	m_2	$8(m_2+1)^3$
0.1	383	$5.6 \cdot 10^7$	30	$2.4 \cdot 10^5$
0.01	3825	$5.6 \cdot 10^{10}$	78	$3.9 \cdot 10^6$
0.001	38250	$5.6 \cdot 10^{13}$	220	$8.6 \cdot 10^7$
0.0001	382498	$5.6 \cdot 10^{16}$	681	$2.5 \cdot 10^9$

6.4 Case Study for a Hybrid Model

Consider the hybrid model of a chemical reaction network, with continuous dynamics described by stochastic difference equations, where time is discrete with sampling interval Δ (see [9] for a complete derivation of the model and for its parameters):

$$\begin{cases} x_1(k+1) = k_r \Delta q(k) + (1-\gamma_r\Delta)x_1(k) + \sqrt{k_r\Delta q(k) + \gamma_r \Delta x_2(k)}w_1(k) \\ x_2(k+1) = k_p \Delta x_1(k) + (1-\gamma_p\Delta)x_2(k) + \sqrt{k_p\Delta x_1(k) + \gamma_p \Delta x_2(k)}w_2(k). \end{cases}$$

The model has two locations $\mathcal{Q} = \{q_1, q_2\}$ indicating a gene in active or inactive mode. The variables x_1, x_2 are concentrations of m-RNA and of a protein, respectively. The signals $w_i(k), i = 1, 2, k \in \mathbb{N} \cup \{0\}$, are independent standard Normal random variables. The transition kernels can be directly derived from the above dynamics [9]. The safe set $\mathcal{A}$ is selected to cover an interval of 10% variation around the steady state of the model. We study the probabilistic safety of $\mathcal{A}$ over a 10-step interval.

Figure 1b compares the approximation errors of piece-wise constant and first-order approximations. The total number of integrations differ roughly only by a factor of two. Furthermore, considering for instance 1000 bins per dimension, the piece-wise constant (zeroth-order) approximation has a global error equal to 32.64, whereas the first-order approximation leads to an error equal to 0.62, with only twice as many integrations involved in the procedure.

7 Conclusions

This contribution has put forward new algorithms, based on higher-order function approximation, for the efficient computation of approximate solutions of probabilistic specifications expressed as PCTL formulae over Markov processes on general state spaces (and in particular over Stochastic Hybrid Systems).

The authors plan to extend the technique to nested PCTL formulae, to further investigate its convergence properties, and to integrate the presented procedures within the algorithms worked out in [8,9], with the goal of developing a flexible software tool for abstraction and verification of Stochastic Hybrid Systems.

References

1. Abate, A., Katoen, J.-P., Lygeros, J., Prandini, M.: Approximate model checking of stochastic hybrid systems. European Journal of Control (6), 624–641 (2010)
2. Abate, A., Katoen, J.-P., Mereacre, A.: Quantitative automata model checking of autonomous stochastic hybrid systems. In: ACM Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, Chicago, IL, pp. 83–92 (April 2011)
3. Abate, A., Prandini, M., Lygeros, J., Sastry, S.: Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems. Automatica 44(11), 2724–2734 (2008)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
5. Bertsekas, D.P., Shreve, S.E.: Stochastic Optimal Control: The Discrete-Time Case. Athena Scientific (1996)
6. Blom, H.A.P., Lygeros, J. (eds.): Stochastic Hybrid Systems: Theory and Safety Critical Applications. LNCIS, vol. 337. Springer, Heidelberg (2006)
7. Cassandras, C.G., Lygeros, J. (eds.): Stochastic Hybrid Systems. Control Engineering, vol. 24. CRC Press, Boca Raton (2006)
8. Esmaeil Zadeh Soudjani, S., Abate, A.: Adaptive gridding for abstraction and verification of stochastic hybrid systems. In: Proceedings of the 8th International Conference on Quantitative Evaluation of Systems, Aachen, DE, pp. 59–69 (September 2011)
9. Esmaeil Zadeh Soudjani, S., Abate, A.: Probabilistic invariance of mixed deterministic-stochastic dynamical systems. In: ACM Proceedings of the 15th International Conference on Hybrid Systems: Computation and Control, Beijing, PRC, pp. 207–216 (April 2012)
10. Hernández-Lerma, O., Lasserre, J.B.: Discrete-time Markov control processes. Applications of Mathematics, vol. 30. Springer, New York (1996)
11. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)

12. Katoen, J.-P., Khattri, M., Zapreev, I.S.: A Markov reward model checker. In: IEEE Proceedings of the International Conference on Quantitative Evaluation of Systems, Los Alamos, CA, USA, pp. 243–244 (2005)
13. Mastroianni, G., Milovanovic, G.V.: Interpolation Processes: Basic Theory and Applications. Springer (2008)
14. Meyn, S.P., Tweedie, R.L.: Markov chains and stochastic stability. Springer (1993)
15. Ramponi, F., Chatterjee, D., Summers, S., Lygeros, J.: On the connections between PCTL and dynamic programming. In: ACM Proceedings of the 13th International Conference on Hybrid Systems: Computation and Control, pp. 253–262 (April 2010)
16. Summers, S., Lygeros, J.: Verification of discrete time stochastic hybrid systems: A stochastic reach-avoid decision problem. Automatica 46(12), 1951–1961 (2010)
17. Tkachev, I., Abate, A.: Regularization of Bellman equations for infinite-horizon probabilistic properties. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, Beijing, PRC, pp. 227–236 (April 2012)

Author Index