

counterexamples due to failed completeness checks. Note that these counterexamples did not require invoking the equivalence oracle.

Given the large alphabet sizes, TTT runs out of memory on all our benchmarks. This is not surprising since the number of queries required by TTT just to construct the *correct* model for a DFA with $128 = 2^7$ states is at least $|\Sigma||Q| \log |Q| = 2^{16} * 2^7 * 7 \approx 2^{26}$. We point out that a corresponding lower bound of $\Omega(|Q| \log |Q| |\Sigma|)$ exists for the number of queries any DFA algorithm may perform and therefore, the size of the alphabet provides a fundamental limitation for any such algorithm.

Analysis. First, we observe that the performance of the algorithm is not always monotone in the number of states or transitions of the s-FA. For example, RE.10 requires more than 10x more membership and equivalence queries than RE.7 despite the fact that both the number of states and transitions of RE.10 are smaller. In this case, RE.10 has fewer transitions, but they contain predicates that are harder to learn—e.g., large character classes. Second, the completeness check and the corresponding counterexamples are not only useful to ensure that the generated guards form a partition but also to restore predicates after new states are discovered. Recall that, once we discover (split) a new state, a number of learning instances is discarded. Usually, the newly created learning instances will simply output \perp as the initial hypothesis. At this point, completeness counterexamples are used to update the newly created hypothesis accordingly and thus save the *MAT** from having to rerun a large number of equivalence queries. Finally, we point out that the equality algebra learner made no special assumptions on the structure of the predicates such as recognizing character classes which are used in regular expressions and others. We expect that providing such heuristics can greatly improve the performance *MAT** in these benchmarks.

6.2 BDD Algebra Learning

In this experiment, we use *MAT** to learn s-FAs over a BDD algebra. We run *MAT** on 1,500 automata obtained by transforming Linear Temporal Logic over finite traces into s-FAs [9]. The formulas have 4 atomic propositions and the height in each BDD used by the s-FAs is four. To learn the underlying BDDs we use *MAT** with the learning algorithm for algebras over finite domains (see Sect. 5) since ordered BDDs can be seen as s-FAs over $\mathcal{D} = \{0, 1\}$.

Figure 4 shows the number of membership (top left) and equivalence (top right) queries performed by *MAT** for s-FAs with different number of states. For this s-FAs, *MAT** is highly efficient with respect to both the number of membership and equivalence queries, scaling linearly with the number of states. Moreover, we note that the size of the set of transitions $|\Delta|$ does not drastically affect the overall performance of the algorithm. This is in agreement with the results presented in the previous section, where we argued that the difficulty of the underlying predicates and not their number is the primary factor affecting performance.

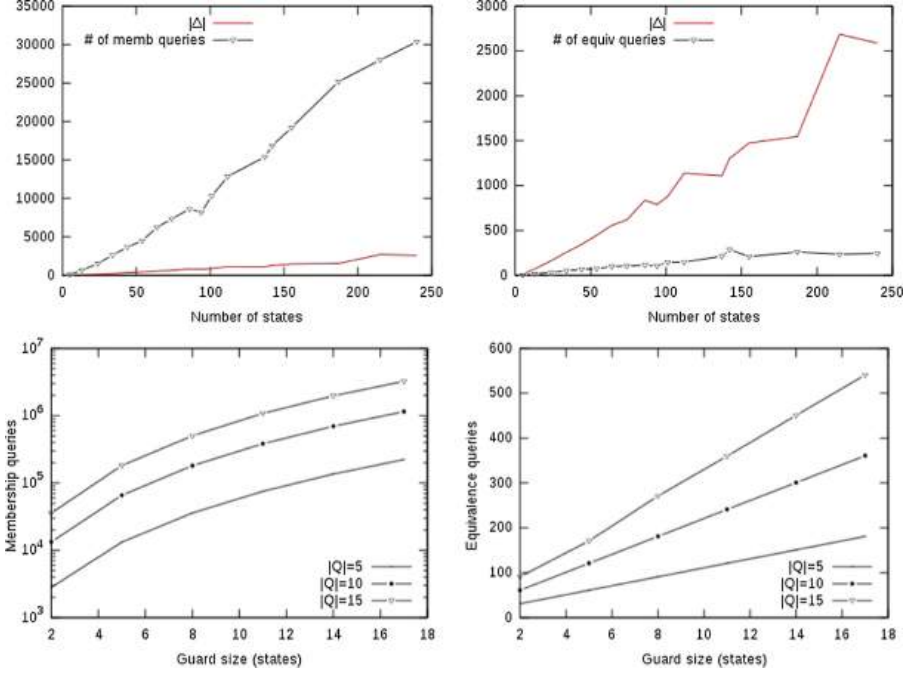


Fig. 4. (Top) Evaluation of MAT^* on s-FAs over a BDD algebra. (Bottom) Evaluation of MAT^* on s-FAs over an s-FA algebra. For an s-FA $\mathcal{M}_{m,n}$, the x -axis denotes the values of n . Different lines correspond to different values of m .

6.3 s-FA Algebra Learning

In this experiment, we use MAT^* to learn 18 s-FAs over s-FAs, which accept strings of strings. We evaluate the scalability of our algorithms when the difficulty of learning the underlying predicates increases. The possible internal s-FAs, which we will use as predicates, operate over the equality algebra and are denoted as I_k (where $2 \leq k \leq 17$). Each s-FA I_k accepts exactly one word $a \cdots a$ of length k and has $k + 1$ states and $2k + 1$ transitions. The external s-FAs are denoted as $\mathcal{M}_{m,n}$ (where $m \in \{5, 10, 15\}$ and $2 \leq n \leq 17$). Each s-FA $\mathcal{M}_{m,n}$ accepts exactly one word $s \cdots s$ of length m where each s is accepted by I_n .

Analysis. For simplicity, let's assume that we have the s-FA $\mathcal{M}_{n,n}$. Consider a membership query performed by one of the underlying algebra learning instances. Answering the membership query requires sifting a sequence in the classification tree of height at most n which requires $O(n)$ membership queries. Therefore, the number of membership queries required to learn each individual predicate is increased by a factor of $O(n)$. Moreover, for each equivalence query performed by an algebra learning instance, the s-FA learning algorithm has to pinpoint the counterexample to the specific algebra learning instance, a process which requires $\log m$ membership queries, where m is the length of the counterexample.

Therefore, we conclude that each underlying guard with n states will require a number of membership queries which is of the order of $O(n^3)$ at the worst and $O(n^2 \log n)$ queries at the best (since the CT has height $\Omega(\log n)$), ignoring the queries required for counterexample processing.

Figure 4 shows the number of membership (bottom left) and equivalence (bottom right) queries, which verify the theoretical analysis presented in the previous paragraph. Indeed, we see that in terms of membership queries, we have a very sharp increase in the number of membership queries which is in fact about quadratic in the number of states in the underlying guards. On the other hand, equivalence queries are not affected so drastically, and only increase linearly.

7 Related Work

Learning Finite Automata. The L^* algorithm proposed by Dana Angluin [3] was the first to introduce the notion of minimally adequate teacher—i.e., learning using membership and equivalence queries—and was also the first for learning finite automata in polynomial time. Following Angluin’s result, L^* has been studied extensively [16, 17], it has been extended to many other models—e.g., to nondeterministic automata [12] alternating automata [4]—and has found many applications in program analysis [2, 5–7, 24] and program synthesis [25]. Since finite automata only operate over finite alphabets, all the automata that can be learned using variants of L^* , can also be learned using MAT^* .

Learning Symbolic Automata. The problem of scaling L^* to large alphabets was initially studied outside the setting of s-FAs using alphabet abstractions [14, 15]. The first algorithm for symbolic automata over ordered alphabets was proposed in [20] but the algorithm assumes that the counterexamples provided to the learning algorithm are of minimal length. Argyros et al. [6] proposed the first algorithm for learning symbolic automata in the standard MAT model and also described the algorithm to distinguish counterexamples leading to new states from counterexamples due to invalid predicates which we adapt in MAT^* . Drews and D’Antoni [11] proposed a symbolic extension to the L^* algorithm, gave a general definition of learnability and demonstrated more learnable algebras such as union and product algebras. The algorithms in [6, 11, 19] are all extensions of L^* and assume the existence of an underlying learning algorithm capable of learning partitions of the domain from counterexamples. MAT^* does not require that the predicate learning algorithms are able to learn partitions, thus allowing to easily plug existing learning algorithms for Boolean algebras. Moreover, MAT^* allows the underlying algebra learning algorithms to perform both equivalence and membership queries, a capability not present in any previous work, thus expanding the class of s-FAs which can be efficiently learned.

Learning Other Models. Argyros et al. [6] and Botincan et al. [7] presented algorithms for learning restricted families of symbolic transducers—i.e., symbolic automata with outputs. Other algorithms can learn nominal [21] and register

automata [8]. In these models, the alphabet is infinite but not structured (i.e., it does not form a Boolean algebra) and characters at different positions can be compared using binary relations.

Acknowledgements. The authors would like to thank the anonymous reviewers for their valuable comments. Loris D’Antoni was supported by National Science Foundation Grants CCF-1637516, CCF-1704117 and a Google Research Award. George Argyros was supported by the Office of Naval Research (ONR) through contract N00014-12-1-0166.

References

1. lorisdanto/symbolicautomata: Library for symbolic automata and symbolic visibly pushdown automata. <https://github.com/lorisdanto/symbolicautomata/>. Accessed 29 Jan 2018
2. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. *SIGPLAN Not.* **40**(1), 98–109 (2005)
3. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987)
4. Angluin, D., Eisenstat, S., Fisman, D.: Learning regular languages via alternating automata. In: *Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI 2015*, pp. 3308–3314. AAAI Press (2015)
5. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A.: SFADiff: automated evasion attacks and fingerprinting using black-box differential automata learning. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1690–1701. ACM (2016)
6. Argyros, G., Stais, I., Kiayias, A., Keromytis, A.D.: Back in black: towards formal, black box analysis of sanitizers and filters. In: *IEEE Symposium on Security and Privacy, SP 2016, 22–26 May 2016, San Jose, CA, USA*, pp. 91–109 (2016)
7. Botincan, M., Babic, D.: Sigma*: symbolic learning of input-output specifications. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, 23–25 January 2013, Rome, Italy*, pp. 443–456 (2013)
8. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. *Formal Aspects Comput.* **28**(2), 233–263 (2016)
9. D’Antoni, L., Kincaid, Z., Wang, F.: A symbolic decision procedure for symbolic alternating finite automata. *arXiv preprint arXiv:1610.01722* (2016)
10. D’Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčák, V. (eds.) *CAV 2017*. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
11. Drews, S., D’Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) *TACAS 2017*. LNCS, vol. 10205, pp. 173–189. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_10
12. García, P., de Parga, M.V., Álvarez, G.I., Ruiz, J.: Learning regular languages using nondeterministic finite automata. In: Ibarra, O.H., Ravikumar, B. (eds.) *CIAA 2008*. LNCS, vol. 5148, pp. 92–101. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70844-5_10

13. Habrard, A., Oncina, J.: Learning multiplicity tree automata. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 268–280. Springer, Heidelberg (2006). https://doi.org/10.1007/11872436_22
14. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_19
15. Isberner, M., Howar, F., Steffen, B.: Inferring automata with state-local alphabet abstractions. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 124–138. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_9
16. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
17. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
18. Li, N., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Reggae: automated test generation for programs using complex regular expressions. In: 2009 24th IEEE/ACM International Conference on Automated Software Engineering. ASE 2009, pp. 515–519. IEEE (2009)
19. Maler, O., Mens, I.-E.: A generic algorithm for learning symbolic automata from membership queries. In: Aceto, L., et al. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 146–169. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_8
20. Mens, I., Maler, O.: Learning regular languages over large ordered alphabets. *Log. Methods Comput. Sci.* **11**(3) (2015)
21. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szyrwelski, M.: Learning nominal automata. In: Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2017)
22. Nakamura, A.: An efficient query learning algorithm for ordered binary decision diagrams. *Inf. Comput.* **201**(2), 178–198 (2005)
23. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. *Inf. Comput.* **103**(2), 299–347 (1993)
24. Sivakorn, S., Argyros, G., Pei, K., Keromytis, A.D., Jana, S.: HVLearn: automated black-box analysis of hostname verification in SSL/TLS implementations. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 521–538. IEEE (2017)
25. Yuan, Y., Alur, R., Loo, B.T.: NetEgg: programming network policies by examples. In: Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, pp. 20:1–20:7. ACM, New York (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Runtime Verification, Hybrid and Timed Systems



Reachable Set Over-Approximation for Nonlinear Systems Using Piecewise Barrier Tubes

Hui Kong^{1(✉)}, Ezio Bartocci², and Thomas A. Henzinger¹

¹ IST Austria, Klosterneuburg, Austria
hui.kong@ist.ac.at

² TU Wien, Vienna, Austria

Abstract. We address the problem of analyzing the reachable set of a polynomial nonlinear continuous system by over-approximating the flowpipe of its dynamics. The common approach to tackle this problem is to perform a numerical integration over a given time horizon based on Taylor expansion and interval arithmetic. However, this method results to be very conservative when there is a large difference in speed between trajectories as time progresses. In this paper, we propose to use combinations of barrier functions, which we call piecewise barrier tube (PBT), to over-approximate flowpipe. The basic idea of PBT is that for each segment of a flowpipe, a coarse box which is big enough to contain the segment is constructed using sampled simulation and then in the box we compute by linear programming a set of barrier functions (called barrier tube or BT for short) which work together to form a tube surrounding the flowpipe. The benefit of using PBT is that (1) BT is independent of time and hence can avoid being stretched and deformed by time; and (2) a small number of BTs can form a tight over-approximation for the flowpipe, which means that the computation required to decide whether the BTs intersect the unsafe set can be reduced significantly. We implemented a prototype called PBTS in C++. Experiments on some benchmark systems show that our approach is effective.

1 Introduction

Hybrid systems [17] are widely used to model dynamical systems which exhibit both discrete and continuous behaviors. The reachability analysis of hybrid systems has been a challenging problem over the last few decades. The hard core of this problem lies in dealing with the continuous behavior of systems that are described by ordinary differential equations (ODEs). Although there are currently several quite efficient and scalable approaches for reachability analysis of linear systems [8–10, 14, 16, 19, 20, 26, 34], nonlinear ODEs are much harder

This research was supported by the Austrian Science Fund (FWF) under grants S11402-N23, S11405-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

© The Author(s) 2018

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 449–467, 2018.

https://doi.org/10.1007/978-3-319-96145-3_24

to handle and the current approaches can be characterized into the following groups.

Invariant Generation [18, 21, 22, 27, 28, 36, 37, 39]. An invariant I for a system S is a set such that any trajectory of S originating from I never escapes from I . Therefore, finding an invariant I such that the initial set $I_0 \subseteq I$ and the unsafe set $U \cap I = \emptyset$ indicates the safety of the system. In this way, there is no need to compute the flowpipe. The main problem with invariant generation is that it is hard to define a set of high quality constraints which can be solved efficiently.

Abstraction and Hybridization [2, 11, 24, 31, 35]. The basic idea of the abstraction-based approach is first constructing a linear model which over-approximates the original nonlinear dynamics and then applying techniques for linear systems to the abstraction model. However, how to construct an abstraction with the fewest discrete states and sufficiently high accuracy is still a challenging issue.

Satisfiability Modulo Theory (SMT) Over Reals [6, 7, 23]. This approach encodes the reachability problem for nonlinear systems as first-order logic formulas over the real numbers. These formulas can be solved using for example δ -complete decision procedures that overcome the theoretical limits in nonlinear theories over the reals, by choosing a desired precision δ . An SMT implementing such procedures can return either *unsat* if the reachability problem is unsatisfiable or δ -sat if the problem is satisfiable given the chosen precision. The δ -sat verdict does not guarantee that the dynamics of the system will reach a particular region. It may happen that by increasing the precision the problem would result *unsat*. In general the limit of this approach is that it does not provide as a result a complete and comprehensive description of the reachability set.

Bounded Time Flowpipe Computation [1, 3–5, 25, 32]. The common technique to compute a bounded flowpipe is based on interval method or Taylor model. Interval-based approach is quite efficient even for high dimensional systems [29], but it suffers the wrapping effect of intervals and can quickly accumulate over-approximation errors. In contrast, the Taylor-model-based approach is more precise in that it uses a vector of polynomials plus a vector of small intervals to symbolically represent the flowpipe. However, for the purpose of safety verification or reachability analysis, the Taylor model has to be further over-approximated by intervals, which may bring back the wrapping effect. In particular, the wrapping effect can explode easily when the flowpipe segment over a time interval is stretched drastically due to a large difference in speed between individual trajectories. This case is demonstrated by the following example.

Example 1 (Running example). Consider the 2D system [30] described by $\dot{x} = y$ and $\dot{y} = x^2$. Let the initial set X_0 be a line segment $x \in [1.0, 1.0]$ and $y \in [-1.05, -0.95]$, Fig. 1a shows the simulation result on three points in X_0 over time interval $[0, 6.6]$. The reachable set at $t = 6.6$ s is a smooth curve connecting the end points of the three trajectories. As can be seen, the trajectory originating from the top is left far behind the one originating from the bottom, which means that the tiny initial line segment is being stretched into a huge curve very quickly,



Fig. 1. (a) Simulation for Example 1 showing flowpipe segment being extremely stretched and deformed, (b) Interval over-approximation of the Taylor model computed by *Flow** [3].

while the width of the flowpipe is actually converging to 0. As a result, the interval over-approximation of this huge curve can be extremely conservative even if its Taylor model representation is precise, and reducing the time step size is not helpful. To prove this point, we computed with *Flow** [3] a Taylor model series for the time horizon of 6.6 s which consists of 13200 Taylor models. Figure 1b shows the interval approximation of the Taylor model series, which apparently starts exploding.

In this paper, we propose to use piecewise barrier tubes (PBTs) to over-approximate flowpipes of polynomial nonlinear systems, which can avoid the issue caused by the excessive stretching of a flowpipe segment. The idea of PBT is inspired from barrier certificate [22, 33]. A barrier certificate $B(\mathbf{x})$ is a real-valued function such that (1) $B(\mathbf{x}) \geq 0$ for all \mathbf{x} in the initial set X_0 ; (2) $B(\mathbf{x}) < 0$ for all \mathbf{x} in the unsafe set X_U ; (3) no trajectory can escape from $\{\mathbf{x} \in \mathbb{R}^n \mid B(\mathbf{x}) \geq 0\}$ through the boundary $\{\mathbf{x} \in \mathbb{R}^n \mid B(\mathbf{x}) = 0\}$. A sufficient condition for this constraint is that the Lie derivative of $B(\mathbf{x})$ w.r.t the dynamics $\dot{\mathbf{x}} = \mathbf{f}$ is positive all over the invariant region, i.e., $\mathcal{L}_{\mathbf{f}}B(\mathbf{x}) > 0$, which means that all the trajectories must move in the increasing direction of the level sets of $B(\mathbf{x})$.

Barrier certificates can be used to verify safety properties without computing the flowpipe explicitly. The essential idea is to use the zero level set of $B(\mathbf{x})$ as a barrier to separate the flowpipe from the unsafe set. Moreover, if the unsafe set is very close to the boundary of the flowpipe, the barrier has to fit the shape of the flowpipe to make sure that all components of the constraint are satisfied. However, the zero level set of a polynomial of fixed degree may not have the power to mimic the shape of the flowpipe, which means that there may exist no solution for the above constraints even if the system is safe. This problem might be addressed using piecewise barrier certificate, i.e., cutting the flowpipe into small pieces so that every piece is straight enough to have a barrier certificate of simple form. Unfortunately, this is infeasible because we know nothing about the flowpipe locally. Therefore, we have to find another way to proceed.

Instead of computing a single barrier certificate, we propose to compute barrier tubes to piecewise over-approximate the flowpipe. Concretely, in the begin-

ning, we first construct a containing box, called **enclosure**, for the initial set using interval approach [29] and simulation, then, using linear programming, we compute a group of barrier functions which work together to form a tight tube (called barrier tube) around the flowpipe. Similarly, taking the intersection of the barrier tube and the boundary of the box as the new initial set, we repeat the previous operations to obtain successive barrier tubes step by step. The key point here is how to compute a group of tightly enclosing barriers around the flowpipe without a constraint on the unsafe set inside the box. Our basic idea is to construct a group of auxiliary state sets U around the flowpipe and then, for each $U_i \in U$, we compute a barrier certificate between U_i and the flowpipe. If a barrier certificate is found, we expand U_i towards the flowpipe iteratively until no more barrier certificate can be found; otherwise, we shrink U_i away from the flowpipe until a barrier certificate is found. Since the auxiliary sets are distributed around the flowpipe, so is the barrier tube. The benefit of such piecewise barrier tubes is that they are time independent, and hence can avoid the issue of stretched flowpipe segments caused by speed differences between trajectories. Moreover, usually a small number of BTs can form a tight over-approximation of the flowpipe, which means that less computation is needed to decide the intersection of PBT and the unsafe set.

The main contributions of this paper are as follows:

1. We transform the constraint-solving problem for barrier certificates into a linear programming problem using Handelman representation [15];
2. We introduce PBT to over-approximate the flowpipe of nonlinear systems, thus dealing with flowpipes independent of time and hence avoiding the error explosion caused by stretched flowpipe segments;
3. We implement a prototype in C++ to compute PTB automatically and we show the effectiveness of our approach by providing a comparison with the state-of-the-art tools for reachability analysis of polynomial nonlinear systems such as *CORA* [1] and *Flow** [3].

The paper is organized as follows. Section 2 is devoted to the preliminaries. Section 3 shows how to compute barrier certificates using Handelman representation, while in Sect. 4 we present a method to compute Piecewise Barrier Tubes. Section 5 provides our experimental results and we conclude in Sect. 6.

2 Preliminaries

In this section, we recall some concepts used throughout the paper. We first clarify some notation conventions. If not specified otherwise, we use boldface lower case letters to denote vectors, we use \mathbb{R} for the real number field and \mathbb{N} for the set of natural numbers, and we consider multivariate polynomials in $\mathbb{R}[\mathbf{x}]$, where the components of \mathbf{x} act as indeterminates. In addition, for all the polynomials $B(\mathbf{u}, \mathbf{x})$, we denote by \mathbf{u} the vector composed of all the u_i and denote by \mathbf{x} the vector composed of all the remaining variables x_i that occur in

the polynomial. We use $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{> 0}$ to denote the domain of nonnegative real number and positive real number respectively.

Let $P \subseteq \mathbb{R}^n$ be a convex and compact polyhedron with non-empty interior, bounded by linear polynomials $p_1, \dots, p_m \in \mathbb{R}[\mathbf{x}]$. Without lose of generality, we may assume $P = \{\mathbf{x} \in \mathbb{R}^n \mid p_i(\mathbf{x}) \geq 0, i = 1, \dots, m\}$.

Next, we present the notation of the Lie derivative, which is widely used in the discipline of differential geometry. Let $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be a continuous vector field such that $\dot{x}_i = f_i(\mathbf{x})$ where \dot{x}_i is the time derivative of $x_i(t)$.

Definition 1 (Lie derivative). For a given polynomial $p \in \mathbb{R}[\mathbf{x}]$ over $\mathbf{x} = (x_1, \dots, x_n)$ and a continuous system $\dot{\mathbf{x}} = \mathbf{f}$, where $\mathbf{f} = (f_1, \dots, f_n)$, the **Lie derivative** of $p \in \mathbb{R}[\mathbf{x}]$ along \mathbf{f} of order k is defined as follows.

$$\mathcal{L}_{\mathbf{f}}^k p \stackrel{\text{def}}{=} \begin{cases} p, & k = 0 \\ \sum_{i=1}^n \frac{\partial \mathcal{L}_{\mathbf{f}}^{k-1} p}{\partial x_i} \cdot f_i, & k \geq 1 \end{cases}$$

Essentially, the k -th order Lie derivative of p is the k -th derivative of p w.r.t. time, i.e., reflects the change of p over time. We write $\mathcal{L}_{\mathbf{f}} p$ for $\mathcal{L}_{\mathbf{f}}^1 p$.

In this paper, we focus on semialgebraic nonlinear systems, which are defined as follows.

Definition 2 (Semialgebraic system). A **semialgebraic system** is a triple $M \stackrel{\text{def}}{=} \langle X, \mathbf{f}, X_0, I \rangle$, where

1. $X \subseteq \mathbb{R}^n$ is the state space of the system M ,
2. $\mathbf{f} \in \mathbb{R}[\mathbf{x}]^n$ is locally Lipschitz continuous vector function,
3. $X_0 \subseteq X$ is the initial set, which is semialgebraic [40],
4. I is the invariant of the system.

The local Lipschitz continuity guarantees the existence and uniqueness of the differential equation $\dot{\mathbf{x}} = \mathbf{f}$ locally. A trajectory of a semialgebraic system is defined as follows.

Definition 3 (Trajectory). Given a semialgebraic system M , a **trajectory** originating from a point $\mathbf{x}_0 \in X_0$ to time $T > 0$ is a continuous and differentiable function $\zeta(\mathbf{x}_0, t) : [0, T] \rightarrow \mathbb{R}^n$ such that (1) $\zeta(\mathbf{x}_0, 0) = \mathbf{x}_0$, and (2) $\forall \tau \in [0, T] : \frac{d\zeta}{dt} \big|_{t=\tau} = \mathbf{f}(\zeta(\mathbf{x}_0, \tau))$. T is assumed to be within the maximal interval of existence of the solution from \mathbf{x}_0 .

For ease of readability, we also use $\zeta(t)$ for $\zeta(\mathbf{x}_0, t)$. In addition, we use $\text{Flow}_{\mathbf{f}}(X_0)$ to denote the flowpipe of initial set X_0 , i.e.,

$$\text{Flow}_{\mathbf{f}}(X_0) \stackrel{\text{def}}{=} \{\zeta(\mathbf{x}_0, t) \mid \mathbf{x}_0 \in X_0, t \in \mathbb{R}_{\geq}, \dot{\zeta} = \mathbf{f}(\zeta)\} \quad (1)$$

Definition 4 (Safety). Given an unsafe set $X_U \subseteq X$, a semialgebraic system $M = \langle X, \mathbf{f}, X_0, I \rangle$ is said to be **safe** if no trajectory $\zeta(\mathbf{x}_0, t)$ of M satisfies that $\exists \tau \in \mathbb{R}_{\geq 0} : \mathbf{x}(\tau) \in X_U$, where $\mathbf{x}_0 \in X_0$.

3 Computing Barrier Certificates

Given a semialgebraic system M , a barrier certificate is a real-valued function $B(\mathbf{x})$ such that (1) $B(\mathbf{x}) \geq 0$ for all \mathbf{x} in the initial set; (2) $B(\mathbf{x}) < 0$ for all \mathbf{x} in the unsafe set; (3) no trajectory can escape from the region of $B(\mathbf{x}) \geq 0$. Then, the hyper-surface $\{\mathbf{x} \in \mathbb{R}^n \mid B(\mathbf{x}) = 0\}$ forms a barrier separating the flowpipe from the unsafe set. To compute such a barrier certificate, the most common approach is template based constraint solving, i.e., firstly figure out a sufficient condition for the above condition and then, set up a template polynomial $B(\mathbf{u}, \mathbf{x})$ of fixed degree, and finally solve the constraint on \mathbf{u} derived from the sufficient condition on $B(\mathbf{u}, \mathbf{x})$. There are a couple of sufficient conditions available for this purpose [13, 22, 27]. In order to have an efficient constraint solving method, we adopt the following condition [33].

Theorem 1. *Given a semialgebraic system M , let X_0 and U be the initial set and the unsafe set respectively, the system is guaranteed to be safe if there exists a real-valued function $B(\mathbf{x})$ such that*

$$\forall \mathbf{x} \in X_0 : B(\mathbf{x}) > 0 \quad (2)$$

$$\forall \mathbf{x} \in I : \mathcal{L}_f B > 0 \quad (3)$$

$$\forall \mathbf{x} \in X_U : B(\mathbf{x}) < 0 \quad (4)$$

In Theorem 1, the condition (3) means that all the trajectories of the system always point in the increasing direction of the level sets of $B(\mathbf{x})$ in the region I . Therefore, no trajectory starting from the initial set would cross the zero level set. The benefit of this condition is that it can be solved more efficiently than other existing conditions [13, 22] although it is relatively conservative. The most widely used approach is to transform the constraint-solving problem into a sum-of-squares (*SOS*) programming problem [33], which can be solved in polynomial time. However, a serious problem with *SOS* programming based approach is that automatic generation of polynomial templates is very hard to perform. We now show an example to demonstrate the reason. For simplicity, we assume that the initial set, the unsafe set and the invariant are defined by the polynomial inequalities $X_0(\mathbf{x}) \geq 0$, $X_U(\mathbf{x}) \geq 0$ and $I(\mathbf{x}) \geq 0$ respectively, then the *SOS* relaxation of Theorem 1 is that the following polynomials are all *SOS*

$$B(\mathbf{x}) - \mu_1(\mathbf{x})X_0(\mathbf{x}) + \epsilon_1 \quad (5)$$

$$\mathcal{L}_f B - \mu_2(\mathbf{x})I(\mathbf{x}) + \epsilon_2 \quad (6)$$

$$-B(\mathbf{x}) - \mu_3(\mathbf{x})X_U(\mathbf{x}) + \epsilon_3 \quad (7)$$

where $\mu_i(\mathbf{x}), i = 1, \dots, 3$ are *SOS* polynomials as well and $\epsilon_i > 0, i = 1, \dots, 3$. Suppose the degrees of $X_0(\mathbf{x})$, $I(\mathbf{x})$ and $X_U(\mathbf{x})$ are all odd numbers. Then, the degree of the template for $B(\mathbf{x})$ must be an odd number too. The reason is that, if $\deg(B)$ is an even number, in order for the first and third polynomials to be *SOS* polynomials, $\deg(B)$ must be greater than both $\deg(\mu_3 X_U)$ and $\deg(\mu_1 X_0)$, which are odd numbers. However, since the first and third condition contain $B(\mathbf{x})$

and $-B(\mathbf{x})$ respectively, their leading monomials must have the opposite sign, which means that they cannot be *SOS* polynomial simultaneously. Moreover, the degrees of the templates for the auxiliary polynomials $\mu_1(\mathbf{x}), \mu_3(\mathbf{x})$ must also be chosen properly so that $\deg(\mu_1 X_0) = \deg(\mu_3 X_U) = \deg(B)$, because only in this way the leading monomials (which has an odd degree) of (5) and (7) have the chance to be resolved so that the resultant polynomial can be a *SOS*. Similarly, in order to make the second polynomial a *SOS* as well, one has to choose an appropriate degree for $\mu_2(\mathbf{x})$ according to the degree of $\mathcal{L}_f B$ and $I(\mathbf{x})$. As a result, the tangled constraints on the relevant template polynomials reduce the power of *SOS* programming significantly.

Due to the above reason, inspired by the work [38], we use Handelman representation to relax Theorem 1. We assume that the initial set X_0 , the unsafe set X_U and the invariant I are all convex and compact polyhedra, i.e., $X_0 = \{\mathbf{x} \in \mathbb{R}^n \mid p_1(\mathbf{x}) \geq 0, \dots, p_{m_1}(\mathbf{x}) \geq 0\}$, $I = \{\mathbf{x} \in \mathbb{R}^n \mid q_1(\mathbf{x}) \geq 0, \dots, q_{m_2}(\mathbf{x}) \geq 0\}$ and $X_U = \{\mathbf{x} \in \mathbb{R}^n \mid r_1(\mathbf{x}) \geq 0, \dots, r_{m_3}(\mathbf{x}) \geq 0\}$, where $p_i(\mathbf{x}), q_j(\mathbf{x}), r_k(\mathbf{x})$ are linear polynomials. Then, we have the following theorem.

Theorem 2. *Given a semialgebraic system M , let X_0 , X_U and I be defined as above, the system is guaranteed to be safe if there exists a real-valued polynomial function $B(\mathbf{x})$ such that*

$$B(\mathbf{x}) \equiv \sum_{|\alpha| \leq M_1} \lambda_\alpha p_1^{\alpha_1} \cdots p_{m_1}^{\alpha_{m_1}} + \epsilon_1 \quad (8)$$

$$\mathcal{L}_f B \equiv \sum_{|\beta| \leq M_2} \lambda_\beta q_1^{\beta_1} \cdots q_{m_2}^{\beta_{m_2}} + \epsilon_2 \quad (9)$$

$$-B(\mathbf{x}) \equiv \sum_{|\gamma| \leq M_3} \lambda_\gamma r_1^{\gamma_1} \cdots r_{m_3}^{\gamma_{m_3}} + \epsilon_3 \quad (10)$$

where $\lambda_\alpha, \lambda_\beta, \lambda_\gamma \in \mathbb{R}_{\geq 0}$, $\epsilon_i \in \mathbb{R}_{> 0}$ and $M_i \in \mathbb{N}, i = 1, \dots, 3$.

Theorem 2 provides us with an alternative to *SOS* programming to find barrier certificate $B(\mathbf{x})$ by transforming it into a linear programming problem. The basic idea is that we first set up a template $B(\mathbf{u}, \mathbf{x})$ of fixed degree as well as the appropriate $M_i, i = 1, \dots, 3$ that make the both sides of the three identities (8)–(10) have the same degree. Since (8)–(10) are identities, the coefficients of the corresponding monomials on both sides must be identical as well. Thus, we derive a system S of linear equations and inequalities over $\mathbf{u}, \lambda_\alpha, \lambda_\beta, \lambda_\gamma$. Now, finding a barrier certificate is just to find a feasible solution for S , which can be solved by linear programming. Compared to *SOS* programming based approach, this approach is more flexible in choosing the polynomial template as well as other parameters. We consider now a linear system to show how it works.

Example 2. Given a 2D system defined by $\dot{x} = 2x + 3y, \dot{y} = -4x + 2y$, let $X_0 = \{(x, y) \in \mathbb{R}^2 \mid p_1 = x + 100 \geq 0, p_2 = -90 - x \geq 0, p_3 = y + 45 \geq 0, p_4 = -40 - y \geq 0\}$, $I = \{(x, y) \in \mathbb{R}^2 \mid q_1 = x + 110 \geq 0, q_2 = -80 - x \geq 0, q_3 = y + 45 \geq 0, q_4 = -20 - y \geq 0\}$ and $X_U = \{(x, y) \in \mathbb{R}^2 \mid r_1 = x + 98 \geq 0, r_2 =$

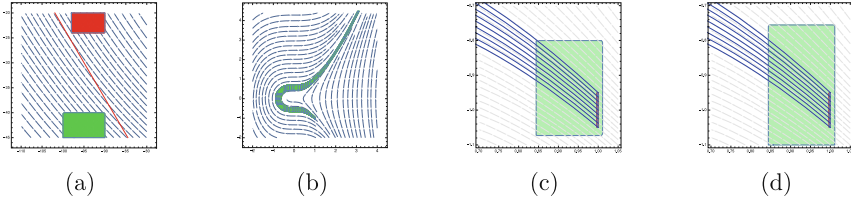


Fig. 2. (a) Linear barrier certificate (straight red line) for Example 2. Rectangle in green: initial set, rectangle in red: unsafe set. (b) PBT for the running Example 5, consisting of 45 BTs. (c) Enclosure (before bloating) for flowpipe of Example 3 (green shadow region). (d) Enclosure (after bloating) for flowpipe of Example 3. (Color figure online)

$-90-x \geq 0, r_3 = y+24 \geq 0, r_4 = -20-y \geq 0\}$. Assume $B(\mathbf{u}, \mathbf{x}) = u_1 + u_2x + u_3y$, $M_i = \epsilon_i = 1$ for $i = 1, \dots, 3$, then we obtain the following polynomial identities according to Theorem 2

$$\begin{aligned} u_1 + u_2x + u_3y - \sum_{i=1}^4 \lambda_{1i}p_i - \epsilon_1 &\equiv 0 \\ u_2(2x + 3y) + u_3(-4x + 2y) - \sum_{j=1}^4 \lambda_{2j}q_j - \epsilon_2 &\equiv 0 \\ -(u_1 + u_2x + u_3y) - \sum_{k=1}^4 \lambda_{3k}r_k - \epsilon_3 &\equiv 0 \end{aligned}$$

where $\lambda_{ij} \geq 0$ for $i = 1, \dots, 3, j = 1, \dots, 4$. By collecting the coefficients of x, y in the above polynomials, we obtain a system S of linear polynomial equations and inequalities over u_i, λ_{jk} . By solving S using linear programming, we obtain a feasible solution and Fig. 2a shows the computed linear barrier certificate. Note that, for the aforementioned reason, it is impossible to find a linear barrier certificate using *SOS* programming for this example.

4 Piecewise Barrier Tubes

In this section, we introduce how to construct PBTs for nonlinear polynomial systems. The basic idea of constructing PBT is that, for each segment of the flowpipe, an enclosure box is first constructed and then, a BT is constructed to form a tighter over-approximation for the flowpipe segment inside the box.

4.1 Constructing an Enclosure Box

Given an initial set, the first task is to construct an enclosure box for the initial set and the following segment of the flowpipe. As pointed out in Sect. 1, one

principle to construct an enclosure box is to simplify the shape of the flowpipe segment, or in other words, to approximately bound the twisting of trajectories by some θ in the box, where the *twisting* of a trajectory is defined as follows.

Definition 5 (Twisting of a trajectory). Let M be a continuous system and $\zeta(t)$ be a trajectory of M . Then, $\zeta(t)$ is said to have a twisting of θ on the time interval $I = [T_1, T_2]$, written as $\xi_I(\zeta)$, if it satisfies that $\xi_I(\zeta) = \theta$, where $\xi_I(\zeta) \stackrel{\text{def}}{=} \sup_{t_1, t_2 \in I} \arccos \left(\frac{\langle \dot{\zeta}(t_1), \dot{\zeta}(t_2) \rangle}{\|\dot{\zeta}(t_1)\| \|\dot{\zeta}(t_2)\|} \right)$.

The basic idea to construct an enclosure box is depicted in Algorithm 1.

Algorithm 1. Algorithm to construct an enclosure box

input : M : dynamics of the system; n : dimension of system; X_0 : initial set
 θ_1 : twisting of simulation; d : maximum distance of simulation;
output: E : an enclosure box containing X_0 ; P : plane where flowpipe exits ;
 G : range of intersection of $Flow_f(X_0)$ with plane P by simulation

- 1 sample a set S_0 of points from X_0 ;
- 2 select a point $\mathbf{x}_0 \in S_0$;
- 3 find a time step size ΔT_0 by (θ, d) -bounded simulation for \mathbf{x}_0 ;
- 4 $\Delta T \leftarrow \Delta T_0$;
- 5 **while** $\Delta T > \epsilon$ **do**
- 6 $[found, E] \leftarrow$ find an enclosure box by interval arithmetic using ΔT ;
- 7 **if** $found$ **then**
- 8 do a simulation for all $\mathbf{x}_i \in S_0$, select the plane P which intersects with
 the most of simulations; generate G ;
- 9 bloat E s.t $Flow_f(X_0)$ gets out of E only through the facet in P ;
- 10 **break**;
- 11 **else**
- 12 $\Delta T \leftarrow 1/2 * \Delta T$;

Remark 1. In Algorithm 1, we use interval arithmetic [29] and simulation to construct an enclosure box E for a given initial set and its following flowpipe segment. Meanwhile, we obtain a coarse range of the intersection of the flowpipe and the boundary of the enclosure, which helps to accelerate the construction of barrier tube. To be simple, the enclosure is constructed in a way such that the flowpipe gets out of the box through a single facet. Given an initial set X_0 , we first sample a set S_0 of points from X_0 for simulation. Then, we select a point \mathbf{x}_0 from S_0 and do (θ, d) -simulation on \mathbf{x}_0 to obtain a time step ΔT . A (θ, d) -simulation is a simulation that stops either when the twisting of the simulation reaches θ or when the distance between \mathbf{x}_0 and the end point reaches d . On the one hand, by using a small θ , we aim to achieve a straight flowpipe segment. On the other hand, by specifying a maximal distance d , we make sure that the

simulation can stop for a long and straight flowpipe. At each iteration of the *while* loop in line 5, we first try to construct an enclosure box by interval arithmetic over ΔT . If such an enclosure box is created, we then perform a simulation (see line 8) for all the points in S_0 to find out the plane P of facet which intersects with the most of the simulations. The idea behind line 9 is that in order to better over-approximate the intersection of the flowpipe with the boundary of the box using intervals, we push the other planes outwards to make P the only plane where the flowpipe get out of the box. Certainly, simply by simulation we cannot guarantee that the flowpipe does not intersect the other facets. Therefore, we have the following theorem for the decision.

Theorem 3. *Given a semialgebraic system M and an initial set X_0 , a box E is an enclosure of X_0 and F_i is a facet of E . Then, $(\text{Flow}_f(X_0) \cap E) \cap F_i = \emptyset$ if there exists a barrier certificate $B_i(\mathbf{x})$ for X_0 and F_i inside E .*

Remark 2. According to the definition of barrier certificate, the proof of Theorem 3 is straightforward, which is ignored here. Therefore, to make sure that the flowpipe does not intersect the facet F_i , we only need to find a barrier certificate, which can be done using the approach presented in Sect. 3. Moreover, if no barrier certificate can be found, we further bloat the facet. Next, we still use the running Example 1 to demonstrate the process of constructing an enclosure.

Example 3 (running example). Consider the system in Example 1 and the initial set $x = 1.0, -1.05 \leq y \leq -0.95$, let the bounding twisting of simulation be $\theta = \pi/18$, then the time step size we computed for interval evaluation is $\Delta T = 0.2947$. The corresponding enclosure computed by interval arithmetic is shown in Fig. 2c. Furthermore, by simulation, we know that the flowpipe can reach both left facet and top facet. Therefore, we have two options to bloat the facet: bloat the left facet to make the flowpipe intersects the top facet only or bloat the top facet to make the flowpipe intersects left facet only. In this example, we choose the latter option and the bloated enclosure is shown in Fig. 2d. In this way, we can over-approximate the intersection of the flowpipe and the facet by intervals if we can obtain its boundary on every side. This can be achieved by finding barrier tube.

4.2 Compute a Barrier Tube Inside a Box

An important fact about the flowpipe of continuous system is that it tends to be straight if it is short enough, given that the initial set is straight as well (otherwise, we can split it). Suppose there is a small box E around a straight flowpipe, it will be easy to compute a barrier certificate for a given initial set and unsafe set inside E . A barrier tube for the flowpipe in E is a group of barrier certificates which form a tube around a flowpipe inside E . Formally,

Definition 6 (Barrier Tube). *Given a semialgebraic system M , a box E and an initial set $X_0 \subseteq E$, a barrier tube is a set of real-valued functions $BT = \{B_i(\mathbf{x}), i = 1, \dots, m\}$ such that for all $B_i(\mathbf{x}) \in BT$: (1) $\forall \mathbf{x} \in X_0 : B_i(\mathbf{x}) > 0$ and, (2) $\forall \mathbf{x} \in E : \mathcal{L}_f B_i > 0$.*

According to Definition 6, a barrier tube BT is defined by a set of real-valued functions and every function inequality $B_i(\mathbf{x}) > 0$ is an invariant of M in E and so do their conjunction. The property of a barrier tube BT is formally described in the following theorem.

Theorem 4. *Given a semialgebraic system M , a box E and an initial set $X_0 \subseteq E$, let $BT = \{B_i(\mathbf{x}) : i = 1, \dots, m\}$ be a barrier tube of M and $\Omega = \{\mathbf{x} \in \mathbb{R}^n \mid \bigwedge B_i(\mathbf{x}) > 0, B_i \in BT\}$, then $Flow_f(X_0) \cap E \subseteq \Omega \cap E$.*

Remark 3. Theorem 4 states that an arbitrary barrier tube is able to form an over-approximation for the reach pipe in the box E . Compared to a single barrier certificate, multiple barrier certificates could over-approximate the flowpipe more precisely. However, since there is no constraint on unsafe sets in Definition 6, a barrier tube satisfying the definition could be very conservative. In order to obtain an accurate approximation for the flowpipe, we choose to create additional auxiliary constraints.

Auxiliary Unsafe Set (AUS). To obtain an accurate barrier tube, there are two main questions to be answered: (1) How many barrier certificates are needed? and (2) How do we control their positions to make the tube well-shaped to better over-approximate the flowpipe? The answer for the first question is quite simple: the more, the better. This will be explained later on. For the second question, the answer is to construct a group of properly distributed auxiliary state sets (AUSs). Each set of the AUSs is used as an unsafe set U_i for the system and then we compute a barrier certificate B_i for U_i according to Theorem 2. Since the zero level set of B_i serves as a barrier between the flowpipe and U_i , the space where a barrier could appear is fully determined by the position of U_i . Roughly speaking, when U_i is far away from the flowpipe, the space for a barrier to exist is wide as well. Correspondingly, the barrier certificate found would usually locate far away from the flowpipe as well. Certainly, as U_i gets closer to the flowpipe, the space for barrier certificates also contracts towards the flowpipe accordingly. Therefore, by expanding U_i towards the flowpipe, we can get more precise over-approximations for the flowpipe.

Why Multiple AUS? Although the accuracy of the barrier certificate over-approximation can be improved by expanding the AUS towards the flowpipe, the capability of a single barrier certificate is very limited because it can erect a barrier which only matches a single profile of the flow pipe. However, if we have a set U of AUSs which are distributed evenly around the flowpipe and there is a barrier certificate B_i for each $U_i \in U$, these barrier certificates would be able to over-approximate the flowpipe from a number of profiles. Therefore, increasing the number of AUSs can increase the quality of the over-approximation as well. Furthermore, if all these auxiliary sets are connected, all the barriers would form a tube surrounding the flowpipe. Therefore, if we can create a series of boxes piecewise covering the flowpipe and then construct a barrier tube for every piece of the flowpipe, we obtain an over-approximation for the flowpipe by PBT.

Based on the above idea, we provide Algorithm 2 to compute barrier tube.

Algorithm 2. Algorithm to compute barrier tube

input : M : dynamics of the system; X_0 : Initial set;
 E : interval enclosure of initial set;
 G : interval approx. of $(\partial E \cap \text{Flow}_f(X_0))$ by simulation;
 P : plane where flowpipe exits from box;
 D : candidate degree list for template polynomial;
 ϵ : difference in size between AUS (auxiliary unsafe set)

output: BT: barrier tube; X'_0 : interval over-approximation of $(\text{BT} \cap E)$

```

1 foreach  $G_{ij}$ : an facet of  $G$  do
2    $\text{found} \leftarrow \text{false}$ ;
3   foreach  $d \in D$  do
4      $\text{AUS} \leftarrow \text{CreateAUS}(G, P, G_{ij})$ ;
5     while true do
6        $[\text{found}, B_{ij}] \leftarrow \text{ComputeBarrierCert}(X_0, E, \text{AUS}, d)$ ;
7       if  $\text{found}$  then  $\text{AUS}' \leftarrow \text{Expand}(\text{AUS})$ ;
8       else  $\text{AUS}' \leftarrow \text{Contract}(\text{AUS})$ ;
9       if  $\text{Diff}(\text{AUS}', \text{AUS}) \leq \epsilon$  then break;
10      else  $\text{AUS}' \leftarrow \text{AUS}$ ;
11   if  $\text{found}$  then  $\text{BT} \leftarrow \text{Push}(\text{BT}, B_{ij})$ ; break;
12   else return FAIL;
13 return SUCCEED;

```

Remark 4. In Algorithm 2, for an n -dimensional flowpipe segment, we aim to build a barrier tube composed of $2(n-1)$ barrier certificates, which means we need to construct $2(n-1)$ AUSs. According to Algorithm 1, we know that the plane P is the only exit of the flowpipe from the enclosure E and G is roughly the region where they intersect. Let F^G be the facet of E that contains G , then for every facet F_{ij}^G of F^G , we can take an $(n-1)$ -dimensional rectangle between F_{ij}^G and G_{ij} as an AUS, where G_{ij} is the facet of G adjacent to F_{ij}^G . Therefore, enumerating all the facets of G in line 1 would produce $2(n-1)$ positions for AUS. The loop in line 3 is attempting to find a polynomial barrier certificate of different degrees in D . In the while loop 5, we iteratively compute the best barrier certificate by adjusting the width of AUS through binary search until the difference in width between two successive AUSs is less than the specified threshold ϵ .

Example 4 (Running example). Consider the initial set and the enclosure computed in Example 3, we use Algorithm 2 to compute a barrier tube. The initial set is $X_0 = [1.0, 1.0] \times [-1.05, -0.95]$ and the enclosure of X_0 is $E = [0.84, 1.01] \times [-1.1, -0.75]$, $G = [0.84, 0.84] \times [-0.91, -0.80]$, the plane P is $x = 0.84$, $D = \{2\}$ and $\epsilon = 0.001$. The barrier tube consists of two barrier certificates. As shown in Fig. 3, each of the barrier certificates is derived from an AUS (red line segment) which is located respectively on the bottom-left and top-left boundary of E .

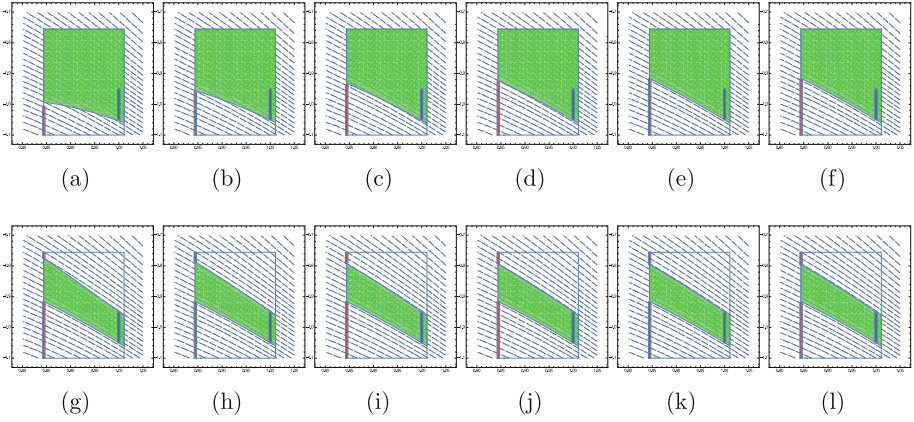


Fig. 3. Computing process of BT for Example 4. Blue line segment: initial set, red line segment: AUS. Figure a–l show how intermediate barrier certificates changed with the width of the AUSs and Fig. l shows the final BT (shadow region in green). (Color figure online)

4.3 Compute Piecewise Barrier Tube

During the computation of a barrier tube by Algorithm 2, we create a series of AUSs around the flowpipe, which build up a rectangular enclosure for the intersection of the flowpipe and the facet of the enclosure box. As a result, such a rectangular enclosure can be taken as an initial set for the following flowpipe segment and then Algorithm 2 can be applied repeatedly to compute a PBT. The basic procedure to compute PBT is presented in Algorithm 3.

Remark 5. In Algorithm 3, initially a box that contains the initial set X_0 is constructed using Algorithm 1. The loop in line 2 consists of 3 major parts: (1) In lines 3–6, a barrier tube BT is firstly computed using Algorithm 2. The **while** loop keeps shrinking the box until a barrier tube is found; (2) In line 8, the initial set X_0 is updated for the next box; (3) In line 9, a new box is constructed to contain X_0 and the process is repeated.

Example 5 (Running example). Let us consider again the running example. We set the length of PBT to 45 and the PBT we obtained is shown in Fig. 2b. Compared to the interval over-approximation of the Taylor model obtained using $Flow^*$, the computed PBT consists of a significantly reduced number of segments and is more precise for the absence of stretching.

Safety Verification Based on PBT. The idea of safety verification based on PBT is straightforward. Given an unsafe set X_U , for each intermediate initial set X_0 and the corresponding enclosure box E , we first check whether $X_U \cap E = \emptyset$. If not empty, we would further find a barrier certificate between X_U and the flowpipe of X_0 inside E . If empty or barrier found, we continue to compute

Algorithm 3. Algorithm to compute PBT

input : M : dynamics of the system; X_0 : Initial set;
 N : length of piecewise barrier tube
output: PBT: piecewise barrier tube

```

1  $E \leftarrow$  construct an initial box containing  $X_0$ ;
2 for  $i \leftarrow 1$  to  $N$  do
3    $[Found, BT] \leftarrow \text{findBarrierTube}(E, X_0)$  ;
4   while not Found do
5      $E \leftarrow \text{Shrink}(E)$  ;
6      $[Found, BT] \leftarrow \text{findBarrierTube}(E, X_0)$  ;
7   if Found then
8      $X_0 \leftarrow \text{OverApprox}(BT \cap \text{Facet}(E))$  ;
9      $E \leftarrow$  construct the next box containing  $X_0$ ;

```

Table 1. Model definitions

| Model | Dynamics | Initial set X_0 | Time horizon (TH) |
|---------------------------|---------------------------------------------------------------------|------------------------------------------------------------------|-------------------|
| Controller 2D | $\dot{x} = xy + y^3 + 2$ $\dot{y} = x^2 + 2x - 3y$ | $x \in [29.9, 30.1]$ $y \in [-38, -36]$ | 0.0125 |
| Van der Pol Oscillator | $\dot{x} = y$ $\dot{y} = y - x - x^2y$ | $x \in [1, 1.5]$ $y \in [2.0, 2.45]$ | 6.74 |
| Lotka-Volterra | $\dot{x} = x(1.5 - y)$ $\dot{y} = -y(3 - x)$ | $x \in [4.5, 5.2]$ $y \in [1.8, 2.2]$ | 3.2 |
| Controller 3D | $\dot{x} = 10(y - x)$ $\dot{y} = x^3$ $\dot{z} = xy - 2.667z$ | $x \in [1.79, 1.81]$ $y \in [1.0, 1.1]$ $y \in [0.5, 0.6]$ | 0.51 |

longer PBT. The refinement of PBT computation can be achieved by using smaller E and higher d for template polynomial.

5 Implementation and Experiments

We have implemented the proposed approach as a C++ prototype called Piecewise Barrier Tube Solver (*PBTS*), choosing *Gurobi* [12] as our internal linear programming solver. We have also performed some experiments on a benchmark of four nonlinear polynomial dynamical systems (described in Table 1) to compare the efficiency and the effectiveness of our approach w.r.t. other tools. Our experiments were performed on a desktop computer with a 3.6 GHz *Intel Core i7-7700* 8 Core CPU and 32 GB memory. The results are presented in Table 2.

Remark 6. There are a number of outstanding tools for flowpipe computation [1, 3–5]. Since our approach is to perform flowpipe computation for polynomial

Table 2. Tool Comparison on Nonlinear Systems. #var: number of variables; T: computing time; NFS: number of flowpipe segments; DEG: candidate degrees for template polynomial (only for *PBTS*); TH: time horizon for flowpipe (only for *Flow** and *CORA*). FAIL: failed to terminate under 30 min.

| Model | #var | PBTS | | | TH | Flow* | | CORA | |
|----------------|------|-------|-----|-----|--------|-------|------|--------|-------|
| | | T | NFS | DEG | | T | NFS | T | NFS |
| Controller 2D | 2 | 5.62 | 46 | 2 | 0.0125 | 22.17 | 6250 | FAIL | - |
| Van der Pol | 2 | 13.38 | 110 | 2,3 | 6.74 | 15.28 | 337 | 212.51 | 12523 |
| Lotka-Volterra | 2 | 6.65 | 30 | 3,4 | 3.2 | 10.59 | 3200 | 35.84 | 2903 |
| Controller 3D | 3 | 83.65 | 15 | 4 | 0.51 | 11.61 | 5100 | 65.18 | 6767 |

nonlinear systems, we pick two of the most relevant state-of-the-art tools for comparison: *CORA* [1] and *Flow** [3]. Note that a big difference between our approach and the other two approaches is that *PBTS* is time-independent, which means that we cannot compare *PBTS* with *CORA* or *Flow** over the exactly same time horizon. To be fair enough, for *Flow** and *CORA*, we have used the same time horizon for the flowpipe computation, while we have computed a slightly longer flowpipe using *PBTS*. To guide the reader, we have also used different plotting colors to visualize the difference between the flowpipes obtained from the three different tools.

Evaluation. As pointed out in Sect. 1, a common problem with the bounded-time integration based approaches is that the flowpipe segment of a dynamics system can be extremely stretched with time so that the interval over-approximation of the flowpipe segment is very conservative and usually the solver has to stop prematurely due to the error explosion. This fact can be found easily from the figures Fig. 4, 5, 6 and 7. In particular, for *Controller 2D*, *Flow** can give quite nice result in the beginning but started producing an exploding flowpipe very quickly (Note that *Flow** offers options to produce better plotting which however is expensive and was not used for safety verification. *CORA* even failed to give a result after over 30 min of running). This phenomenon reappeared with both *Flow** and *CORA* for *Controller 3D*. Notice that most of the time horizons used in the experiment are basically the time limits that *Flow** and *CORA* can reach, i.e., a slightly larger value for the time horizon would cause the solvers to fail. In comparison, our tool has no such problem and can survive a much longer flowpipe before exploding or even without exploding as shown in Fig. 4a.

Another important factor of the approaches is the efficiency. As is shown in Table 2, our approach is more efficient on the first three examples but slower on the last example than the other two tools. The reason for this phenomenon is that the degree d of the template polynomial used in the last example is higher than the others and increasing d led to an increase in the number of decision variables in the linear constraint. This suggests that using smaller d on shorter flowpipe segment would be better. In addition, we can also see in Table 2 that the number of the flowpipe segments produced by *PBTS* is much fewer than that



Fig. 4. Flowpipe for Controller 2D.

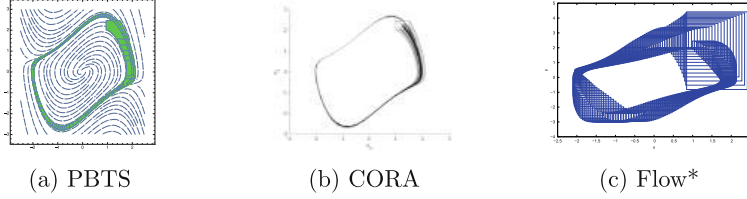


Fig. 5. Flowpipe for Van der Pol Oscillator.

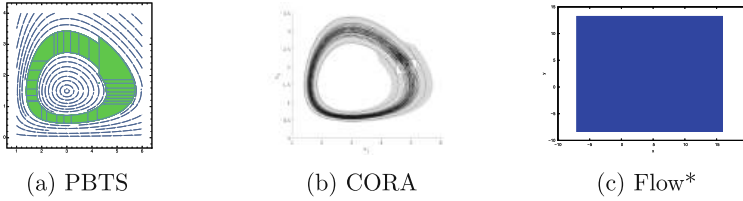


Fig. 6. Flowpipe for Lotka-Volterra.

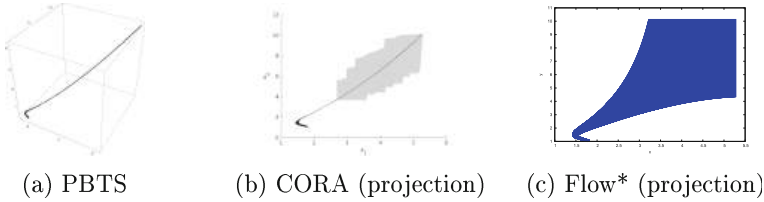


Fig. 7. Flowpipe (projection) for Controller 3D.

produced by *Flow** and *CORA*. In this respect, *PBTS* would be more efficient on safety verification.

6 Conclusion

We have presented *PBTS*, a novel approach to over-approximate flowpipes of nonlinear systems with polynomial dynamics. The benefit of using BTs is that they are time-independent and hence cannot be stretched or deformed by time.

Moreover, this approach only results in a small number of BTs which are sufficient to form a tight over-approximation for the flowpipe, hence the safety verification with PBT can be very efficient.

References

1. Althoff, M., Grebenyuk, D.: Implementation of interval arithmetic in CORA 2016. In: Proceedings of ARCH@CPSWeek 2016: The 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems, EPiC Series in Computing, vol. 43, pp. 91–105. EasyChair (2017)
2. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. *Acta Inform.* **43**(7), 451–476 (2007)
3. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
4. 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). https://doi.org/10.1007/978-3-642-03845-7_9
5. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
6. Fränzle, M., Herde, C.: HySAT: an efficient proof engine for bounded model checking of hybrid systems. *Form. Methods Syst. Des.* **30**(3), 179–198 (2007)
7. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT* **1**(3–4), 209–236 (2007)
8. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
9. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
10. Girard, A., Le Guernic, C.: Efficient reachability analysis for linear systems using support functions. In: Proceedings of IFAC World Congress, vol. 41, no. 2, pp. 8966–8971 (2008)
11. Grosu, R., et al.: From cardiac cells to genetic regulatory networks. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 396–411. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_31
12. Gu, Z., Rothberg, E., Bixby, R.: Gurobi optimizer reference manual (2017). <http://www.gurobi.com/documentation/7.5/refman/refman.html>
13. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_18
14. Gurung, A., Ray, R., Bartocci, E., Bogomolov, S., Grosu, R.: Parallel reachability analysis of hybrid systems in xspeed. *Int. J. Softw. Tools Technol. Transf.* (2018)

15. Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. *Pac. J. Math.* **132**(1), 35–62 (1988)
16. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) *TACAS 2014*. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
17. Henzinger, T.A.: The theory of hybrid automata. In: *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 278–292 (1996)
18. Huang, Z., Fan, C., Mereacre, A., Mitra, S., Kwiatkowska, M.: Invariant verification of nonlinear hybrid automata networks of cardiac cells. In: Biere, A., Bloem, R. (eds.) *CAV 2014*. LNCS, vol. 8559, pp. 373–390. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_25
19. Jiang, Y., Yang, Y., Liu, H., Kong, H., Gu, M., Sun, J., Sha, L.: From state-flow simulation to verified implementation: a verification approach and a real-time train controller design. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–11. IEEE (2016)
20. Jiang, Y., Zhang, H., Li, Z., Deng, Y., Song, X., Ming, G., Sun, J.: Design and optimization of multiclocked embedded systems using formal techniques. *IEEE Trans. Ind. Electron.* **62**(2), 1270–1278 (2015)
21. Kong, H., Bogomolov, S., Schilling, C., Jiang, Y., Henzinger, T.A.: Safety verification of nonlinear hybrid systems based on invariant clusters. In: *Proceedings of HSCC 2017: The 20th International Conference on Hybrid Systems: Computation and Control*, pp. 163–172. ACM (2017)
22. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 242–257. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_17
23. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ -reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
24. Krilavicius, T.: Hybrid techniques for hybrid systems. Ph.D. thesis, University of Twente, Enschede, Netherlands (2006)
25. Lal, R., Prabhakar, P.: Bounded error flowpipe computation of parameterized linear systems. In: *Proceedings of EMSOFT 2015: The International Conference on Embedded Software*, pp. 237–246. IEEE (2015)
26. 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). https://doi.org/10.1007/978-3-642-02658-4_40
27. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: *Proceedings of EMSOFT 2011: The 11th International Conference on Embedded Software*, pp. 97–106. ACM (2011)
28. Matringe, N., Moura, A.V., Rebiha, R.: Generating invariants for non-linear hybrid systems by linear algebraic methods. In: Cousot, R., Martel, M. (eds.) *SAS 2010*. LNCS, vol. 6337, pp. 373–389. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_23
29. Nedialkov, N.S.: Interval tools for ODEs and DAEs. In: *Proceedings of SCAN 2006: The 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, p. 4. IEEE (2006)
30. Neher, M., Jackson, K.R., Nedialkov, N.S.: On Taylor model based integration of ODEs. *SIAM J. Numer. Anal.* **45**(1), 236–262 (2007)

31. Prabhakar, P., Soto, M.G.: Hybridization for stability analysis of switched linear systems. In: Proceedings of HSCC 2016: The 19th International Conference on Hybrid Systems: Computation and Control, pp. 71–80. ACM (2016)
32. Prabhakar, P., Viswanathan, M.: A dynamic algorithm for approximate flow computations. In: Proceedings of HSCC 2011: The 14th International Conference on Hybrid Systems: Computation and Control, pp. 133–142. ACM (2011)
33. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
34. Ray, R., et al.: XSpeed: accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26287-1_1
35. Roohi, N., Prabhakar, P., Viswanathan, M.: Hybridization based CEGAR for hybrid automata with affine dynamics. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 752–769. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_48
36. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: Proceedings of HSCC 2010: The 13th ACM International Conference on Hybrid Systems: Computation and Control, pp. 221–230. ACM (2010)
37. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_36
38. Sankaranarayanan, S., Chen, X., et al.: Lyapunov function synthesis using handelman representations. In: IFAC Proceedings Volumes, vol. 46, no. 23, pp. 576–581 (2013)
39. Sogokon, A., Ghorbal, K., Jackson, P.B., Platzer, A.: A method for invariant generation for polynomial continuous systems. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 268–288. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_13
40. Stengle, G.: A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Math. Ann.* **207**(2), 87–97 (1974)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Space-Time Interpolants

Goran Frehse¹, Mirco Giacobbe^{2(✉)}, and Thomas A. Henzinger²

¹ Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France

² IST Austria, Klosterneuburg, Austria

mgiacobbe@ist.ac.at

Abstract. Reachability analysis is difficult for hybrid automata with affine differential equations, because the reach set needs to be approximated. Promising abstraction techniques usually employ interval methods or template polyhedra. Interval methods account for dense time and guarantee soundness, and there are interval-based tools that overapproximate affine flowpipes. But interval methods impose bounded and rigid shapes, which make refinement expensive and fixpoint detection difficult. Template polyhedra, on the other hand, can be adapted flexibly and can be unbounded, but sound template refinement for unbounded reachability analysis has been implemented only for systems with piecewise constant dynamics. We capitalize on the advantages of both techniques, combining interval arithmetic and template polyhedra, using the former to abstract time and the latter to abstract space. During a CEGAR loop, whenever a spurious error trajectory is found, we compute additional space constraints and split time intervals, and use these *space-time interpolants* to eliminate the counterexample. Space-time interpolation offers a lazy, flexible framework for increasing precision while guaranteeing soundness, both for error avoidance and fixpoint detection. To the best of our knowledge, this is the first abstraction refinement scheme for the reachability analysis over *unbounded* and *dense* time of affine hybrid systems, which is both *sound* and *automatic*. We demonstrate the effectiveness of our algorithm with several benchmark examples, which cannot be handled by other tools.

1 Introduction

Formal verification techniques can be used to either provide rigorous guarantees about the behaviors of a critical system, or detect instances of violating behavior if such behaviors are possible. Formal verification has become widely used in the design of software and digital hardware, but has yet to show a similar success for physical and cyber-physical systems. One of the reasons for this is a scarcity of suitable algorithmic verification tools, such as model checkers, which are formally sound, precise, and scale reasonably well. In this paper, we propose a novel verification algorithm that meets these criteria for systems with piecewise affine dynamics. The performance of the approach is illustrated experimentally on a number of benchmarks. Since systems with affine dynamics have been studied before, we first describe why the available methods and tools do not handle this

class of systems sufficiently well, and then describe our approach and its core contributions.

Previous Approaches. The algorithmic verification of systems with continuous or discrete-continuous (hybrid) dynamics is a hard problem both in theory and practice. For piecewise constant dynamics (PCD), the continuous successor states (a.k.a. flow pipe) can be computed exactly, and the complexity is exponential in the number of variables [17, 19]. While in principle, any dynamics can be approximated arbitrarily well by PCD systems using an approach called hybridization [20], this requires partitioning of the state space, which often leads to prohibitive computational costs. For piecewise affine dynamics (PWA), one-step successors can be computed approximately using complex set representations. However, all published approaches suffer either from a possibly exponential increase in the complexity of the set representation, or from a possibly exponential increase in the approximation error as the considered time interval increases; this will be argued in detail in Sect. 4.

In addition to these theoretical obstacles, we note the following practical obstacles for the available tools and their performance in experiments. The only available model checkers that are (i) *sound* (i.e., they compute provable dense-time overapproximations), (ii) *unbounded* (i.e., they overapproximate the flow-pipe for an infinite time horizon), and (iii) *arbitrarily precise* (i.e., they support precision refinement) are, with one exception, limited to PCD systems, namely, HyTech [18], PHAVer [13], and Lyse [7]. The tool Ariadne [6] can deal with affine dynamics and is sound, unbounded, and precise. However, Ariadne discretizes the reachable state space with a rectangular grid. This invariably leads to an exponential complexity in terms of the number of variables. Other tools that are applicable to PWA systems do not meet our criteria in that they are either not formally sound (e.g., CORA [2], SpaceEx [15]), not arbitrarily precise because of templates or particular data structures (e.g., SpaceEx, Flow* [8], CORA), or limited to bounded model checking (e.g., dReach [24], Flow*). All the above tools exhibit fatal limitations in scalability or precision on standard PWA benchmarks; they typically work only on well-chosen examples. Note that while these tools do not meet the criteria we advance in this paper, they of course have strengths in other areas handling nonlinear and nondeterministic dynamics.

Our Approach. We view iterative abstraction refinement as critical for soundness and precision management, and fixpoint detection as critical for evaluating unbounded properties. We implement, for the first time, a CEGAR (counterexample-guided abstraction refinement) scheme in combination with a fixpoint detection criterion for PWA systems. Our abstraction refinement scheme manages complexity and precision trade-offs in a flexible way by decoupling time from space: the dense timeline is partitioned into a sequence of intervals that are refined individually and lazily, by splitting intervals, to achieve the necessary precision and detect fixpoints; state sets are overapproximated using template polyhedra that are also refined individually and lazily, by adding normal directions to templates; and both refinement processes are interleaved for optimal results, while maintaining soundness with each step. A similar approach was

recently proposed for the limited class of PCA systems [7]; this paper can be seen as an extension of the approach to the class of piecewise affine dynamics.

With each iteration of the CEGAR loop, a spurious counterexample is removed by computing a proof of infeasibility in terms of a sequence of linear constraints in space and interval constraints in time, which we call a sequence of *space-time interpolants*. We use linear programming to construct a suitable sequence of space-time intervals and check for fixpoints. If a fixpoint check fails, we increase the time horizon by adding new intervals. The separation of time from space gives us the flexibility to explore different refinement strategies. Fine-tuning the iteration of space refinement (adding template directions), time refinement (splitting intervals), and fixpoint checking (adding intervals), we find that it is generally best to prefer fewer time intervals over fewer space constraints. Based on performance evaluation, we even expand individual intervals time when this is possible without sacrificing the necessary precision for removing a counterexample.

2 Motivating Example

The ordinary differential equation over the variables x and y

$$\begin{aligned}\dot{x} &= 0.1x - y + 1.8 \\ \dot{y} &= x + 0.1y - 2.2\end{aligned}\tag{1}$$

moves counterclockwise around the point $(2, 2)$ in an outward spiral. We center a box B (of side 0.92) on the same point and place a diagonal segment S close to the bottom right corner of B , without touching it (between $(2, 1)$ and $(3.5, 2)$; see Fig. 1). Then, we consider the problem of proving that every trajectory starting from any point in S never hits B . This is a time-unbounded reachability problem for a hybrid automaton with piecewise affine dynamics and two control modes. The first mode has the dynamics above (Eq. 1) and S as initial region. It has a transition to a second mode, which in its turn has B as invariant. The second mode is a bad mode, which all trajectories indeed avoid.

We tackle the reachability problem by abstraction refinement. In particular, we aim at automatically constructing an enclosure for the flowpipe—i.e., for the set of trajectories from S —which (i) avoids the bad state B and (ii) covers the continuous timeline up to infinity. Figure 1 shows three abstractions that result from different strategies for refining an initial space partition (i.e., template) and time partition (i.e., sequence of time intervals). All three refinement schemes start by enclosing S with an initial template polyhedron P , and then transforming P into a sequence of abstract flowpipe sections $\text{inflow}^{[\underline{t}, \bar{t}]}(P)$, one for each interval $[\underline{t}, \bar{t}]$ of an initial partitioning of the unbounded timeline. The computation of new flowpipe sections stops when a fixpoint is reached,—i.e., we reach a time threshold t^* whose flowpipe section closes a cycle with $\text{inflow}^{t^*}(P) \subseteq P$, sufficient condition for any further flowpipe section to be contained within the union of previously computed sections.

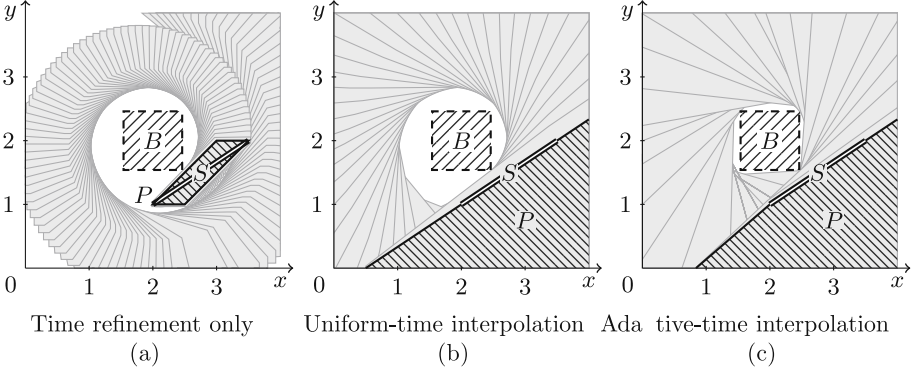


Fig. 1. Comparison of abstraction refinement methods for the ODE in Eq. 1, the segment S as initial region, and the box B as bad region. The polyhedron P is the template polyhedron of S , and the gray polyhedra are the flowpipe sections $\text{inflow}^{[t, \bar{t}]}(P)$.

Refinement scheme (a) sticks to a fixed octagonal template P —i.e., to the normals of a regular octagon—and iteratively halves all time intervals until every flowpipe section avoids the bad set B . This is achieved at interval width $1/64$, but the computation does not terminate because no fixpoint is reached. Refinement scheme (b) splits time similarly but also computes a different, more accurate template for every iteration: first, an interval $[t, \bar{t}]$ is halved until it admits a halfspace interpolant —i.e., a halfspace H that $S \subseteq H$ and $\text{inflow}^{[t, \bar{t}]}(H) \cap B = \emptyset$; then, a maximal set of linearly independent directions is chosen as template from the normals of the obtained halfspaces. Refinement scheme (b) succeeds at interval width $1/16$ to avoid B and reach a fixpoint; the latter at time 6.25 , with $\text{inflow}^{6.25}(P) \subseteq P$. Refinement scheme (c) modifies (b) by optimizing the refinement of the time partition: instead of halving time intervals, the maximal intervals which admit halfspace interpolants are chosen. This scheme produces a nonuniform time partitioning with an average interval width of about $1/8$, discovers five template directions, and finds a fixpoint in fewer steps.

Each iteration of the abstraction refinement loop consists of first abstracting the initial region into a template polyhedron, second solving the differential equation into a sequence of interval matrices, and finally transforming the template polyhedron using each of the interval matrices. We represent each transformation symbolically, by means of its support function. Then, we verify (i) the separation between every support function and the bad region, and (ii) the containment of any support function in the initial template polyhedron. The separation problem amounts to solving one LP, and the inclusion problem amounts to solving an LP in each template direction. If the separation fails, then we independently bisect each time that does not admit halfspace interpolants and expand each that does, until all are proven separated. Together, these halfspace interpolants form an infeasibility proof for the counterexample: a space-time interpolant. We forward the resulting new time intervals and halfspaces to the abstraction

generator, and repeat, using the refined partitioning and the augmented template. If the inclusion fails, then we increase the time horizon by some amount Δ , and repeat. Once we succeed with both separation and inclusion, the system is proved safe.

This example shows the advantage of lazily refining *both* the space partitioning (i.e., the template) by adding directions, and the time partitioning, by splitting intervals.

3 Hybrid Automata with Piecewise Affine Dynamics

A hybrid automaton with piecewise affine dynamics consists of an n -dimensional vector x of real-valued variables and a finite directed multigraph (V, E) , the control graph. We call it the control graph, the vertices $v \in V$ the control modes, and the edges $e \in E$ the control switches. We decorate each mode $v \in V$ with an initial condition $Z_v \subseteq \mathbb{R}^n$, a nonnegative invariant condition $I_v \subseteq \mathbb{R}_{\geq 0}^n$, and a flow condition given by the system of ordinary differential equations

$$\dot{x} = A_v x + b_v. \quad (2)$$

We decorate each switch $e \in E$ with a guard condition $G_e \subseteq \mathbb{R}^n$ and an update condition given the difference equations $x := R_e x + s_e$. All constraints I , G , and Z are conjunctions of rational linear inequalities, A and R are constant matrices, and b and s constant vectors of rational coefficients. In this paper, whenever an indexing of modes and switches is clear from the context, we index the respective constraints and transformations similarly, e.g., we abbreviate A_{v_i} with A_i .

A trajectory is a possibly infinite sequence of states $(v, x) \in V \times \mathbb{R}^n$ repeatedly interleaved first by a switching time $t \in \mathbb{R}_{\geq 0}$ and then by a switch $e \in E$

$$(v_0, x_0)t_0(v_0, y_0)e_0(v_1, x_1)t_1(v_1, y_1)e_1 \dots \quad (3)$$

for which there exists a sequence of solutions $\psi_0, \psi_1, \dots : \mathbb{R} \rightarrow \mathbb{R}^n$ such that $\psi_i(0) = x_i$, $\psi_i(t_i) = y_i$ and they satisfy (i) the invariant conditions $\psi_i(t) \in I_i$ and (ii) the flow conditions $\dot{\psi}_i(t) = A_i \psi_i(t) + b_i$, for all $t \in [0, t_i]$. Moreover, $x_0 \in Z_0$, every switch e_i has source v_i , destination v_{i+1} , and the respective states satisfy (i) the guard condition $y_i \in G_i$ and (ii) the update $x_{i+1} = R_i y_i + s_i$. The maximal set of its trajectories is the semantics of the hybrid automaton, which is safe if none of them contains a special bad mode.

Every hybrid automaton with affine dynamics can be transformed into an equivalent hybrid automaton with linear dynamics, i.e., the special case where $b = 0$ on every mode. We obtain such transformation by adding one extra variable y , rewriting the flow of every mode into $\dot{x} = Ax + by$, and forcing y to be always equal to 1, i.e., invariant $y = 1$ and flow $\dot{y} = 0$ on every mode and update $y' = y$ on every switch. For this reason, in the following sections we discuss w.l.o.g. the reachability analysis of hybrid automata with linear dynamics.

4 Time Abstraction Using Interval Arithmetic

We abstract the reach set of the hybrid automaton with a union of convex polyhedra. In particular, we abstract the states that are reachable in a mode using a finite sequence of images of the initial region over a *time partitioning*, until a completeness threshold is reached. Thereafter, we compute the *template polyhedron* of each of the images that can take a switch. Then, we repeat in the destination mode and we continue until a fixpoint is found.

Precisely, a time partitioning T is a (possibly infinite) set of disjoint closed time intervals whose union is a single (possibly open) interval. For a finite set of directions $D \subseteq \mathbb{R}^n$, the D -polyhedron of a closed convex set X is the tightest polyhedral enclosure whose facets normals are in D . In the following, we associate every mode v to a template D_v and a time partitioning T_v of the time axis $\mathbb{R}_{\geq 0}$, we employ interval arithmetic for abstracting the continuous dynamics (Sect. 4.1), and on top of it we develop a procedure for hybrid dynamics (Sect. 4.2).

4.1 Continuous Dynamics

We consider w.l.o.g. a mode with ODE reduced to the linear form $\dot{x} = A_v x$, invariant I_v , and a given time interval $[\underline{t}, \bar{t}]$. Every linear ODE $\dot{x} = Ax$ has the unique solution

$$\psi(t) = \exp(At)\psi(0). \quad (4)$$

It follows (see also [16]) that the set of states reachable in v after exactly t time units from an initial region X is

$$\text{flow}_v^t(X) \stackrel{\text{def}}{=} \exp(A_v t)X \cap \bigcap_{0 \leq \tau \leq t} \exp(A_v(t - \tau))I_v, \quad (5)$$

Then, the flowpipe section over the time interval $[\underline{t}, \bar{t}]$ is

$$\text{flow}_v^{[\underline{t}, \bar{t}]}(X) \stackrel{\text{def}}{=} \cup \{\text{flow}_v^t(X) \mid t \in [\underline{t}, \bar{t}]\}. \quad (6)$$

We note three straightforward but consequential properties of the reach set: (i) The accuracy of any convex abstraction depends on the size of the time interval: While $\text{flow}_v^t(X)$ is convex for convex X , this is generally not the case for $\text{flow}_v^{[\underline{t}, \bar{t}]}(X)$. (ii) We can prune the time interval whenever we detect that the reach set no longer overlaps with the invariant: If for any $t^* \geq 0$, $\text{flow}_v^{t^*}(X) = \emptyset$, then for all $\bar{t} \geq t^*$, $\text{flow}_v^{\bar{t}}(X) = \emptyset$ and $\text{flow}_v^{[\underline{t}, \bar{t}]}(X) = \text{flow}_v^{[\underline{t}, t^*]}(X)$. (iii) We can prune the time interval whenever we detect containment in the initial states: If $\text{flow}_v^{t^*}(X) \subseteq X$, then $\text{flow}_v^{[t, \infty]}(X) = \text{flow}_v^{[t, t^*]}(X)$.

For given A and t , the matrix $\exp(At)$ can be computed with arbitrary, but only finite, accuracy. We resolve this problem by computing a rational interval matrix $[\underline{M}, \bar{M}]$, which we denote $\text{intexp}(A, \underline{t}, \bar{t})$, such that for all $t \in [\underline{t}, \bar{t}]$ we have element-wise that

$$\exp(At) \in \text{intexp}(A, \underline{t}, \bar{t}). \quad (7)$$

This interval matrix can be derived efficiently with a variety of methods [25], e.g., using a guaranteed ODE solver or using interval arithmetic. The width of the interval matrix can be made arbitrarily small at the price of increasing the number of computations and the size of the representation of the rational numbers. In our approach, we do not rely in a fixed accuracy of the interval matrix. Instead, we require that the accuracy increases as the width of the time interval goes to zero. That way, we don't need to introduce an extra parameter. To ensure progress in our refinement loop, we require that the interval matrix decreases monotonically when we split the time interval. Formally, if $[\underline{t}, \bar{t}] \subseteq [\underline{u}, \bar{u}]$ we require the element-wise inclusion $\text{intexp}(A, \underline{t}, \bar{t}) \subseteq \text{intexp}(A, \underline{u}, \bar{u})$. This can be ensured by intersecting the interval matrices with the original interval matrix after time splitting.

While the mapping with interval matrices is in general not convex [29], we can simplify the problem by assuming that all points of X are in the positive orthant. As long as X is bounded from below, this condition can be satisfied by inducing an appropriate coordinate change. Under the assumption that $X \subseteq \mathbb{R}_{\geq 0}^n$,

$$[\underline{M}, \overline{M}](X) = \{y \in \mathbb{R}^n \mid \underline{M}x \leq y \leq \overline{M}x \text{ and } x \in X\}. \quad (8)$$

Combining the above results, we obtain a convex abstraction of the flowpipe over a time interval as

$$\text{intflow}_v^{[\underline{t}, \bar{t}]}(X) \stackrel{\text{def}}{=} \text{intexp}(A, \underline{t}, \bar{t})X \cap I_v. \quad (9)$$

The abstraction is conservative in the sense that $\text{flow}_v^{[\underline{t}, \bar{t}]}(X) \subseteq \text{intflow}_v^{[\underline{t}, \bar{t}]}(X)$. On the other hand, the longer is the time interval, the coarser is the abstraction. For this reason, we construct an abstraction of the flowpipe in terms of a union of convex approximations over a time partitioning. The abstract flowpipe over the time partitioning T is

$$\text{intflow}_v^T(X) \stackrel{\text{def}}{=} \cup \{\text{intflow}_v^{[\underline{t}, \bar{t}]}(X) \mid [\underline{t}, \bar{t}] \in T\}. \quad (10)$$

Again, this is conservative w.r.t. the concrete flowpipe, i.e., for all time partitionings T it holds that $\text{flow}_v^{\cup T}(X) \subseteq \text{intflow}_v^T(X)$. Moreover, it is conservative w.r.t. any refinement of T , i.e., the time partitioning U refines T if $\cup U = \cup T$ and $\forall [\underline{u}, \bar{u}] \in U: \exists [\underline{t}, \bar{t}] \in T: [\underline{u}, \bar{u}] \subseteq [\underline{t}, \bar{t}]$, then $\text{intflow}_v^U(X) \subseteq \text{intflow}_v^T(X)$.

4.2 Hybrid Dynamics

We embed the flowpipe abstraction routine into a reachability algorithm that accounts for the switching induced by the hybrid automaton. The discrete post operator is the image of a set $Y \subseteq \mathbb{R}^n$ through a switch $e \in E$

$$\text{jump}_e(Y) \stackrel{\text{def}}{=} R_e(Y \cap G_e) \oplus \{s_e\}. \quad (11)$$

We explore the hybrid automaton constructing a set of abstract trajectories, namely sequences abstract states interleaved by time intervals and switches

$$(v_0, X_0)[\underline{t}_0, \bar{t}_0](v_0, Y_0)e_0(v_1, X_1)[\underline{t}_1, \bar{t}_1](v_1, Y_1)e_1 \dots \quad (12)$$

input : Template $\{D_v\}$ and partitioning $\{T_v\}$ indexed by V
output: Optionally an abstract trajectory (counterexample)

```

1 foreach  $v \in V$  with nonempty  $Z_v$  do
2    $\text{push } (v, Z_v)[0, \Delta]$  into the stack  $W$ ;
3   add the  $D_v$ -polyhedron of  $Z_v$  to  $P_v$ ;
4 while  $W$  is not empty do
5    $\text{pop } \dots (v, X)[\underline{t}, \bar{t}]$  from  $W$ ;
6    $P \leftarrow D_v$ -polyhedron of  $X$ ;
7   if  $v$  is bad and  $P \cap I_v$  is nonempty then                                // check counterexample
8      $\text{return } \dots (v, X)$ ;
9   foreach  $t^* \in \{\underline{t} + \delta, \underline{t} + 2\delta, \dots, \bar{t}\}$  do                                // find completeness threshold
10     $\text{if } \text{inflow}_v^{t^*}(P) \subseteq P_v$  then break;
11  if  $t^* = \bar{t}$  and  $\text{inflow}_v^{\bar{t}}(P) \not\subseteq P_v$  then                                // otherwise extend time horizon
12     $\text{push } \dots (v, X)[\underline{t}, \bar{t} + \Delta]$  into  $W$ ;
13  foreach  $[\underline{u}, \bar{u}] \in T_v$  and  $[\underline{u}, \bar{u}] \cap [\underline{t}, t^*] \neq \emptyset$  do            // construct flowpipe
14     $Y \leftarrow \text{inflow}_v^{[\underline{u}, \bar{u}]}(P)$ ;
15    foreach  $e \in E$  with source  $v$  and destination  $v'$  do
16       $X' \leftarrow \text{jump}_e(Y)$ ;
17      if  $X' \subseteq P_{v'}$  then continue;
18       $\text{push } \dots (v, X)[\underline{u}, \bar{u}](v, Y)e(v', X')[0, \Delta]$  into  $W$ ;
19      add the  $D_{v'}$ -polyhedron of  $X'$  to  $P_{v'}$ ;

```

Algorithm 1. Reachability procedure.

where $X_0, Y_0, \dots \subseteq \mathbb{R}^n$ are nonempty sets of states that comply with template $\{D_v\}$ and partitioning $\{T_v\}$ in the following sense. First, $X_0 = Z_0$ and $X_{i+1} = \text{jump}_i(Y_i)$ for all $i \geq 0$. Second, $Y_i = \text{inflow}_i^{[\underline{t}_i, \bar{t}_i]}(P_i)$ for all $i \geq 0$, where P_i is the D_i -polyhedron of X_i and $[\underline{t}_i, \bar{t}_i] \in T_i$. The maximal set of abstract trajectories, the abstract semantics induced by $\{D_v\}$ and $\{T_v\}$, overapproximates the concrete semantics in the sense that every concrete trajectory (see Eq. 3) has an abstract trajectory that subsumes it, i.e., modes and switches match, $x_i \in X_i$, $t_i \in [\underline{t}_i, \bar{t}_i]$, and $y_i \in Y_i$, for all $i \geq 0$.

Computing the abstraction involves several difficulties. First, the trajectories might be not finitary. Indeed, this is unsolvable in theory, because the reachability problem is undecidable [21]. Second, the post operators are hard to compute. In particular, obtaining the sets X and Y in terms of conjunctions of linear inequalities in \mathbb{R}^n requires eliminating quantifiers. In Algorithm 1, we present a procedure (which does not necessarily terminate) for tackling the first problem. In the next section, we show how to tackle the second using support functions.

We employ Algorithm 1 to explore the tree of abstract trajectories. We store in the stack W the leaves to process $\dots (v, X)$, followed by a candidate interval $[\underline{t}, \bar{t}]$. For each leaf, we retrieve P , the template polyhedron of X . If it leads to a bad mode, we return, otherwise we search for a completeness threshold t^* between \underline{t} excluded and \bar{t} , checking for inclusion in the union of visited polyhedra P_v . In case of failure, we extend the time horizon of Δ and push the next candidate to the stack. Then, we partition the time between \underline{t} and t^* , construct the flowpipe, and process switching. Upon each successful switch, we augment $P_{v'}$ with the $D_{v'}$ -polyhedron of the switching region X' , avoiding to store redundant polyhedra. Notably, the latter operation is efficient because all polyhedra

comply with the same template. For the same reason, we obtain efficient inclusion checks, which we implement by first computing the template polyhedron of the left hand side, and then comparing the constant terms of the respective linear inequalities.

In conclusion, this reachability procedure that takes a template $\{D_v\}$ and a partitioning $\{T_v\}$ and constructs a tree of reachable sets of states X and Y . It manipulates them through the post operators and overapproximate them into template polyhedra. In the next section, we discuss how to efficiently represent X and Y , so to efficiently compute their template polyhedra. In Sect. 6 we discuss how to discover appropriate $\{D_v\}$ and $\{T_v\}$, so to eliminate spurious counterexamples.

5 Space Abstraction Using Support Functions

Abstracting away time left us with the task of representing the state space of the hybrid automaton, namely the space of its variable valuations. Such sets consists of polyhedra emerging from operations such as intersections, Minkowski sums, and linear maps with simple or interval matrices. In this section, we discuss how to represent precisely all sets emerging from any of these operations by means of their support functions (Sect. 5.1) and then how to abstract them into template polyhedra (Sect. 5.2). In the next section, we discuss how to refine the abstraction.

5.1 Support Functions

The support function of a closed convex set $X \subseteq \mathbb{R}^n$ in direction $d \in \mathbb{R}^n$ consists of the maximizer scalar product of d over X

$$\rho_X(d) = \sup\{d^\top x \mid x \in X\}, \quad (13)$$

and, indeed, uniquely represents any closed convex set [28]. Classic work on the verification of hybrid automata with affine dynamic have posed a framework for the construction of support functions from basic set operations, but under the assumption of unboundedness and nonemptiness of the represented set, and with approximated intersection [16]. Indeed, if the set is empty then its support function is $-\infty$, while if it is unbounded an d points toward a direction of recession is $+\infty$, making the framework end up into undefined values. Such conditions turn out to be limiting in our context, first because we find desirable to represent unbounded sets so to accelerate the convergence to a fixpoint of the abstraction procedure, but most importantly because when encoding support functions for long abstract trajectories we might be not aware whether its concretization is infeasible. Checking this is a crucial element of a counterexample-guided abstraction refinement routine.

Recent work on the verification of hybrid automata with constant dynamics, i.e., with flows defined by constraints on the derivative only, provides us with

a generalization of the classic support function framework which relaxes away the assumptions of boundedness and nonemptiness and yields precise intersection [7]. The framework encodes combinations of convex sets of states into LP (linear programs) which enjoy strong duality with their support function. Similarly, we encode the support function in direction d of any set X into the LP

$$\begin{aligned} & \text{minimize } c^\top \lambda \\ & \text{subject to } A\lambda = Bd, \end{aligned} \quad (14)$$

over the nonnegative vector of variables λ . The LP is dual to $\rho_X(d)$, which is to say that if the LP is infeasible then X is unbounded in direction d , and if the LP is unbounded then X is the empty set. Moreover, if the LP has bounded solution so does $\rho_X(d)$ and the solutions coincide.

The construction is inductive on operations between sets. For the base case, we recall that from duality of linear programming the support function of a polyhedron given by a system of inequalities $Px \leq q$ is dual to the LP over $\lambda \geq 0$

$$\begin{aligned} & \text{minimize } q^\top \lambda \\ & \text{subject to } P^\top \lambda = d. \end{aligned} \quad (15)$$

Then, inductively, we assume that for the set $X \subseteq \mathbb{R}^n$ we are given an LP with the coefficients A_X , B_X , and c_X , and similarly for the set $Y \subseteq \mathbb{R}^n$. For the support functions of $X \oplus Y$, MX , and $X \cap Y$ we respectively construct the following LP over the nonnegative vectors of variables λ , μ , α , and β :

$$\begin{aligned} & \text{minimize } c_X^\top \lambda + c_Y^\top \mu \\ & \text{subject to } A_X \lambda = B_X d \text{ and } A_Y \mu = B_Y d, \end{aligned} \quad (16)$$

$$\begin{aligned} & \text{minimize } c_X^\top \lambda \\ & \text{subject to } A_X \lambda = B_X M^\top d, \text{ and} \end{aligned} \quad (17)$$

$$\begin{aligned} & \text{minimize } c_X^\top \lambda + c_Y^\top \mu \\ & \text{subject to } A_X \lambda - B_X(\alpha - \beta) = 0 \text{ and} \\ & \quad A_Y \mu + B_Y(\alpha - \beta) = B_Y d. \end{aligned} \quad (18)$$

Such construction follows as a special case of [7], which we extend with the support function of a map through an interval matrix.

The time abstraction of Sect. 4 additionally requires us to represent the map of sets of states through interval matrices. Precisely, we are given convex set of nonnegative values $X \subseteq \mathbb{R}_{\geq 0}^n$, the coefficients for the respective LP, an interval matrix $[\underline{M}, \overline{M}] \subseteq \mathbb{R}^{n \times n}$, and we aim at computing the support function of all values in X mapped by all matrices in $[\underline{M}, \overline{M}]$. To this end, we define the LP

$$\begin{aligned} & \text{minimize } c_X^\top \lambda \\ & \text{subject to } A_X \lambda + B_X(\underline{M}^\top \mu - \overline{M}^\top \nu) = 0 \text{ and} \\ & \quad -\mu + \nu = d, \end{aligned} \quad (19)$$

over the vectors λ , μ , and ν of nonnegative variables. This linear program corresponds to the the dual of the interval matrix map in Eq. 8.

5.2 Computing Template Polyhedra

We represent all space abstractions X and Y in our procedure by their support functions. In particular, whenever set operations are applied, instead of solving the operation by removing quantifiers, we construct an LP. We delay solving it until we need to compute a template polyhedron. In that case, we compute the D -polyhedron of the set X by computing its support function in each of the directions in D , and constructing the intersection of halfspaces $\cap \{d^\top x \leq \rho_X(d) \mid d \in D\}$.

6 Abstraction Refinement Using Space-Time Interpolants

The reachability analysis of hybrid automata by means of the combination of interval arithmetic and support functions presented in Sects. 4 and 5 builds an overapproximation of the system dynamics. It is always sound for safety, but it may produce spurious counterexamples, due to an inherent lack of precision of the time abstraction and the polyhedral approximation. The level of precision is given by two factors, namely the choice of time partitioning and the choice of template directions, excluding the parameters for approximation of the exponential function, which we assume constant (see Sect. 4.1). In the following, we present a procedure to extract infeasibility proofs from spurious counterexamples. We produce them in the form of time partitions and bounding polyhedra, which we call space-time interpolants. Space-time interpolants can then be used to properly refine time partitioning and template directions.

Consider the bounded path $v_0, e_0, v_1, e_1, \dots, v_k, e_k, v_{k+1}$ over the control graph and a sequence of dwell time intervals $[\underline{t}_0, \bar{t}_0], [\underline{t}_1, \bar{t}_1], \dots, [\underline{t}_k, \bar{t}_k]$ emerging from an abstract trajectory. We aim at extracting a sequence X_0, X_1, \dots, X_{k+1} of (possibly nonconvex) polyhedra and a sequence T_0, T_1, \dots, T_k of refinements of the respective dwell times such that $Z_0 \subseteq X_0$, $\text{jump}_0 \circ \text{inflow}_0^{T_0}(X_0) \subseteq X_1$, \dots , $\text{jump}_k \circ \text{inflow}_k^{T_k}(X_k) \subseteq X_{k+1}$, and $X_{k+1} \cap I_{k+1}$ is empty. In other words, we want every X_{i+1} to contain all states that can enter mode v_{i+1} after dwelling on v_i between \underline{t}_i and \bar{t}_i time, and the last to be separated from the invariant of mode v_{k+1} . Containment is to hold inductively, namely X_{i+1} has to contain what is reachable from X_i , and the time refinements T are to be chosen in such a way that containment holds in the abstraction. Then, we call the sequence $X_0, T_0, X_1, T_1, \dots, X_k, T_k, X_{k+1}$ a sequence of space-time interpolants for the path and the dwell times above.

We compute a sequence of space-time interpolants by alternating multiple strategies. First, for the given sequence of dwell times, we attempt to extract a sequence of halfspace interpolants using linear programming (Sect. 6.1). In case of failure, we iteratively partition the dwell times in sets of smaller intervals, separating nonswitching from switching times and until every combination of intervals along the path admits halfspace interpolants (Sect. 6.2). We accumulate all halfspaces to form a sequence of unions of convex polyhedra that, together with the obtained time partitionings, will form a valid sequence of space-time interpolants. Finally, we refine the abstraction using the time partitionings and

the outwards pointing directions of all computed halfspaces, in order to eliminate the spurious counterexample (Sect. 6.3).

6.1 Halfspace Interpolation

Halfspace interpolants are the special case of space-time interpolants where every polyhedron in the sequence is defined by a single linear inequality [1]. Indeed, they are the simplest kind of space-time interpolants, and, for the same reason, the ones that best generalize the reachable states along the path. Unfortunately, not all paths admit halfspace interpolants, but, if one such sequence exists, then it can be extrapolated from the solution of a linear program.

Consider a path v_0, e_0, \dots, v_{k+1} with the respective dwell times $[\underline{t}_0, \bar{t}_0], \dots, [\underline{t}_k, \bar{t}_k]$. A sequence of halfspace interpolants consists of a sequence of sets H_0, \dots, H_{k+1} among either any halfspace, or the empty set, or the universe, such that $Z_0 \subseteq H_0$, $\text{jump}_0 \circ \text{inflow}_0^{[\underline{t}_0, \bar{t}_0]}(H_0) \subseteq H_1, \dots, \text{jump}_k \circ \text{inflow}_k^{[\underline{t}_k, \bar{t}_k]}(H_k) \subseteq H_{k+1}$, and $H_{k+1} \cap I_{k+1}$ is empty. In contrast with general space-time interpolants, every time partition consists of a single time interval and therefore the support function of every post operator $\text{jump} \circ \text{inflow}^{[\underline{t}, \bar{t}]}$ can be encoded into a single LP (see Sect. 5). We exploit the encoding for extracting halfspace interpolants, similarly to a recent interpolation technique for PCD systems [7].

We encode the support function in direction d of the closure of the image of the post operators along the path, i.e., the set $\text{jump}_k \circ \text{inflow}_k^{[\underline{t}_k, \bar{t}_k]} \circ \dots \circ \text{jump}_0 \circ \text{inflow}_0^{[\underline{t}_0, \bar{t}_0]}(Z_0)$, intersected with the invariant I_{k+1} . We obtain the following LP over the free vectors $\alpha_0, \dots, \alpha_{k+1}$ and the nonnegative vectors $\beta, \delta_0, \dots, \delta_k, \gamma_0, \dots, \gamma_{k+1}, \mu_0, \dots, \mu_k$, and ν_0, \dots, ν_k :

$$\begin{aligned}
 & \text{minimize } q_{Z_0}^\top \beta + \sum_{i=0}^k (q_{I_i}^\top \gamma_i + q_{G_i}^\top \delta_i + s_i^\top \alpha_{i+1}) + q_{I_{k+1}}^\top \gamma_{k+1} \\
 & \text{subject to } P_{Z_0}^\top \beta = \alpha_0, \\
 & \quad \underline{M}_i^\top \mu_i - \overline{M}_i^\top \nu_i = -\alpha_i \quad \text{for each } i \in [0..k], \\
 & \quad -\mu_i + \nu_i + P_{I_i}^\top \gamma_i + P_{G_i}^\top \delta_i = R_i^\top \alpha_{i+1} \quad \text{for each } i \in [0..k], \\
 & \quad P_{I_{k+1}}^\top \gamma_{k+1} = -\alpha_{k+1} + d,
 \end{aligned} \tag{20}$$

where every system of inequalities $Px \leq q$ corresponds to the constraints of the respective init, guard, or invariant, every $R_i x + s_i$ is an update equation, and every interval matrix $[\overline{M}_i, \underline{M}_i] = \text{intexp}(A_i, \underline{t}_i, \bar{t}_i)$. In general, one can check whether the closure is contained in a halfspace $a^\top x \leq b$ by setting the direction to its linear term $d = a$ and checking whether the objective function can equal its constant term b . In particular, we check for emptiness, which we pose as checking inclusion in $0x \leq -1$. Therefore, we set $d = 0$ and the objective function to equal -1 . Upon affirmative answer, from the solution $\alpha_0^*, \alpha_1^*, \dots, \nu_k^*$ we obtain a valid sequence of halfspace interpolants whose i -th linear term is given by α_i^* and i -th constant term is given by $q_{Z_0}^\top \beta^* + \sum_{j=0}^{i-1} (q_{I_j}^\top \gamma_j^* + q_{G_j}^\top \delta_j^* + s_j^\top \alpha_{j+1}^*)$.

input : sequence of intervals $[u_0, \bar{u}_0], \dots, [u_j, \bar{u}_j]$
output: set of intervals

```

1  $b \leftarrow u_j$ ;
2 while  $b < \bar{u}_j$  do
3    $a \leftarrow b$ ;
4    $b \leftarrow b + \varepsilon$ ;
5    $c \leftarrow \bar{u}_j$ ;
6   if  $[u_0, \bar{u}_0], \dots, [u_{j-1}, \bar{u}_{j-1}], [a, b]$  does not admit halfspace interpolants then
7     continue;
8   if  $[u_0, \bar{u}_0], \dots, [u_{j-1}, \bar{u}_{j-1}], [a, c]$  admits halfspace interpolants then
9     push  $[a, c]$  to the output;
10    return;
11    while  $c - b > \varepsilon$  do
12      if  $[u_0, \bar{u}_0], \dots, [u_{j-1}, \bar{u}_{j-1}], [a, \varepsilon \lfloor \frac{b+c}{2\varepsilon} \rfloor]$  admits halfspace interpolants then
13         $b \leftarrow \varepsilon \lfloor \frac{b+c}{2\varepsilon} \rfloor$ ;
14      else
15         $c \leftarrow \varepsilon \lfloor \frac{b+c}{2\varepsilon} \rfloor$ ;
16    push  $[a, b]$  to the output;
```

Algorithm 2. Nonswitching time partitioning.

6.2 Time Partitioning

Halfspace interpolation attempts to compute a sequence of enclosures that are convex for a sequence of sets that are not necessarily convex. Specifically, it requires each halfspace to enclose the set of solutions of a linear differential equation, which is nonconvex, by enclosing its convex overapproximation along a whole time interval. As a result, large time intervals produce large overapproximations, on which halfspace interpolation might be impossible. Likewise, shorter intervals produce tighter overapproximations, which are more likely to admit halfspace interpolants. In this section, we exploit such observation to enable interpolation over large time intervals. In particular, we properly partition the time into smaller subintervals and we treat each of them as a halfspace interpolation problem. Later, we combine the results to refine the abstraction.

Time partitioning is a delicate task in the whole abstraction refinement loop. In fact, while template refinement affects linearly the performance of the abstractor, partitioning time intervals that can switch induces branching in the search, possibly leading to an exponential blowup. For this reason, we partition time by narrowing down the switching time, for incremental precision, until no more is left. In particular, we use Algorithm 2 to compute a set N of maximal intervals that admit halfspace interpolants, by enlarging or narrowing them of ε amounts. We embed this procedure in Algorithm 3 which, along the sequence, excludes the time in N , constructing a set of intervals S that overapproximate the switching time. In particular, we construct the set with the widest possible intervals that are disjoint from N . Algorithm 3 succeeds when no more intervals are left, otherwise we half ε and reapply it to the sequences that are left to process.

input : sequence of intervals $[\underline{t}_0, \bar{t}_0], \dots, [\underline{t}_k, \bar{t}_k]$
output: set of sequences of intervals

```

1 push  $[\underline{t}_0, \bar{t}_0]$  to the queue  $Q$ ;
2 while  $Q$  is not empty do
3   pop  $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_j, \bar{u}_j]$  from  $Q$ ;
4    $N \leftarrow$  nonswitching time partitioning of  $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_j, \bar{u}_j]$ ;
5   foreach  $[\underline{a}, \bar{a}] \in N$  do
6     push  $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_{j-1}, \bar{u}_{j-1}], [\underline{a}, \bar{a}]$  to the output;
7   if  $j = k$  then
8     assert  $[\underline{u}_j, \bar{u}_j] \setminus \cup N = \emptyset$ ;
9     continue;
10   $S \leftarrow$  choose set of intervals that cover  $[\underline{u}_j, \bar{u}_j] \setminus \cup N$ ;
11  foreach  $[\underline{b}, \bar{b}] \in S$  do
12    push  $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_{j-1}, \bar{u}_{j-1}], [\underline{b}, \bar{b}], [\underline{t}_{j+1}, \bar{t}_{j+1}]$  to  $Q$ ;

```

Algorithm 3. Dwell time partitioning.

6.3 Abstraction Refinement

The procedures above construct sequences of time intervals $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_j, \bar{u}_j]$ that are included in $[\underline{t}_0, \bar{t}_0], \dots, [\underline{t}_k, \bar{t}_k]$ and that, with the respective halfspace interpolants, this constitutes a proof of infeasibility for the counterexample. Yet, it does not form a sequence of space-time interpolants X_0, T_0, \dots, X_{k+1} . We form each partitioning T_i by splitting $[\underline{t}_i, \bar{t}_i]$ in such a way each element of T_i is either contained in $[\underline{u}_i, \bar{u}_i]$ or disjoint from it, for all intervals $[\underline{u}_i, \bar{u}_i]$. Then, we refine the partitioning of mode v_i similarly. Each polyhedron X_i is a union of convex polyhedra, each of which is the intersection of all halfspaces H_i corresponding to some sequence $[\underline{u}_0, \bar{u}_0], \dots, [\underline{u}_i, \bar{u}_i]$. Nevertheless, to refine the abstraction we do not need to construct X_i , but just to take the outward point directions of all H_i and add them to the template of v_i .

7 Experimental Evaluation

We implemented our method in C++ using GMP and Eigen for multiple precision linear algebra, Arb for interval arithmetic, and PPL for linear programming [5, 23]. In particular, all libraries we are using are meant to provide guaranteed solutions, as well as our implementation. We evaluate it on several instances of a *filtered oscillator* and a *rod reactor*, which are both parametric in the number of variables, and the latter in the number of modes too [15, 35]. We record several statistics from every execution of our tool: the number $\#cex$ of counterexamples found during the CEGAR loop, the number $\#dir$ of linearly independent directions and the average width of the time partitionings extracted from all space-time interpolants. Moreover, we independently measure three times. First, the time spent in finding counterexamples, namely the total time taken by inconclusive abstractions which returned a spurious counterexample. Second, the refinement time, that is the total time consumed by computing space-time interpolants. Finally, the verification time, that is the time spend in the last

abstraction of the CEGAR loop, which terminates with a fixpoint proving the system safe. We compare the outcome and the performance of our tool against Ariadne which, to the best of our knowledge, is the only verification tool available that is numerically sound and time-unbounded [11].

Table 1. Statistics for the benchmark examples (oot when > 1000 s).

| | # vars | # modes | # cex | # dirs | avg. width | cex. time | ref. time | ver. time | tot. time | Ariadne |
|------------------|-----------|------------|----------|-----------|---------------|--------------|--------------|--------------|--------------|---------|
| filtosc_1st_ord | 3 | 4 | 7 | 13 | 0.55 | 0.57 | 0.96 | 0.13 | 1.66 | 27.56 |
| filtosc_2nd_ord | 4 | 4 | 7 | 15 | 0.55 | 0.83 | 1.78 | 0.20 | 2.81 | 150.7 |
| filtosc_3rd_ord | 5 | 4 | 7 | 16 | 0.55 | 1.28 | 4.65 | 0.32 | 6.25 | oot |
| filtosc_4th_ord | 6 | 4 | 7 | 18 | 0.55 | 1.53 | 11.39 | 0.37 | 13.29 | oot |
| filtosc_5th_ord | 7 | 4 | 7 | 19 | 0.55 | 2.61 | 26.60 | 0.70 | 29.37 | - |
| filtosc_6th_ord | 8 | 4 | 7 | 18 | 0.55 | 4.56 | 101.8 | 1.29 | 107.7 | - |
| filtosc_7th_ord | 9 | 4 | 7 | 18 | 0.55 | 4.36 | 109.9 | 1.13 | 114.6 | - |
| filtosc_8th_ord | 10 | 4 | 7 | 17 | 0.55 | 5.92 | 150.9 | 1.54 | 158.4 | - |
| filtosc_9th_ord | 11 | 4 | 7 | 16 | 0.55 | 6.49 | 383.1 | 1.83 | 391.3 | - |
| filtosc_10th_ord | 12 | 4 | 7 | 17 | 0.55 | 12.84 | 428.87 | 3.73 | 445.4 | - |
| filtosc_11th_ord | 13 | 4 | 7 | 17 | 0.55 | 15.10 | 525.2 | 4.38 | 544.6 | - |
| reactor_1_rod | 2 | 4 | 11 | 3 | 0.11 | 5.24 | 10.64 | 1.59 | 17.47 | oot |
| reactor_2_rods | 3 | 5 | 9 | 7 | 0.79 | 5.68 | 5.36 | 2.33 | 13.37 | oot |
| reactor_3_rods | 4 | 6 | 12 | 13 | 1.07 | 14.46 | 13.94 | 13.13 | 41.53 | - |
| reactor_4_rods | 5 | 7 | 15 | 29 | 1.67 | 45.50 | 42.47 | 111.5 | 199.9 | - |
| reactor_5_rods | 6 | 8 | 16 | 31 | 1.81 | 73.77 | 27.36 | 696.46 | 797.5 | - |

The filtered oscillator is hybrid automaton with four modes that smoothens a signal x into a signal z . It has $k + 2$ variables and a system of $k + 2$ affine ODE, where k is the order of the filter. Table 1 shows the results, for a scaling of k up to the 11-th order. The first observation is that the CEGAR loop behaves quite similarly on all scalings: number of counterexamples, number of directions, and time partitionings are almost identical. On the other hand, the computation times show a growth, particularly in the refinement phase which dominates over abstraction and verification. This suggests us that our procedure exploits efficiently the symmetries of the benchmark. In particular, time partitioning seems unaffected. What affects the performance is linear programming, whose size depends on the number of variables of the system.

The rod reactor consists of a heating reactor tank and k rods each of which cools the tank for some amount of time, excluding each other. The hybrid automaton has one variable x for the temperature, k clock variables, one heating mode, one error mode, and k cooling modes. If the temperature reaches a critical threshold and no rod can intervene, it goes into an error. For this benchmark, we start with a simple template, the interval around x , and we discover further directions. Table 1 highlights two fundamental differences with the previous benchmark. First, the average width grows with the model size. This is because the heating mode requires finer time partitioning than the cooling modes. The cooling modes increase with the number of rods, and so does the average width over all time partitions. Second, while with the filtered oscillator the difficulty laid at interpolation, for the rod reactor interpolation is rather easy as well as finding counterexamples. Most of the time is spent in the verification

phase, where all fixpoint checks must be concluded, without being interrupted by a counterexample. This shows the advantage of our lazy approach, which first processes the counterexamples and finally proves the fixpoint.

Our method outperforms Ariadne on all benchmarks. On the other hand, tools like Flow* and SpaceEx can be dramatically faster [9]. For instance, they analyze `filtosc_8th_ord` in resp. 9.1 s and 0.36 s (time horizon of 4 and jump depth of 10). This is hardly surprising, as our method has primarily been designed to comply with soundness and time-unboundedness, and pays the price for that.

8 Related Work

There is a rich literature on CEGAR approaches for hybrid automata, either abstracting to a purely discrete system [3, 10, 27, 33, 34] or to a hybrid automaton with simpler dynamics [22, 30]. Both categories exploit the principle that the verification step is easier to carry out in the abstract domain. The abstraction entails a considerable loss of precision that can only be counteracted by increasing the number of abstract states. This leads to a state explosion that severely limits the applicability of such approaches. In contrast, our approach allows us to increase the precision by adding template directions, which does not increase the number of abstract states. The only case where we incur additional abstract states is when partitioning the time domain. This is a direct consequence of the nonconvexity of flowpipes of affine systems, and therefore seems to be unavoidable when using convex sets in abstractions. In [26], the abstraction consists of removing selected ODE entirely. This reduces the complexity, but does not achieve any fine-tuning between accuracy and complexity. Template reachability has been shown to be very effective in both scaling up reachability tasks to more efficient successor computations [15, 31, 32] and achieving termination even over unbounded time horizons [12]. The drawback of templates is the lack of accuracy, which may lead to an approximation error that accumulates excessively. Efforts to dynamically refine templates have so far not scaled well for affine dynamics [14]. A single-step refinement was proposed in [4], but as was illustrated in [7], the refinement needs to be inductive in order to exclude counterexamples in a CEGAR scheme.

9 Conclusion

We have developed an abstraction refinement scheme that combines the efficiency and scalability of template reachability with just enough precision to exclude all detected paths to the bad states. At each iteration of the refinement loop, only one template direction is added per mode and time-step. This does not increase the number of abstract states. Additional abstract states are only introduced when required by the nonconvexity of flowpipes of affine systems, a problem that we consider unavoidable. In contrast, existing CEGAR approaches for hybrid automata tend to suffer from state explosion, since refining

the abstraction immediately requires additional abstract states. As our experiments confirm, our approach results in templates over very low complexity and terminates with an unbounded proof of safety after a relatively small number of iterations. Further research is required to extend this work to nondeterministic and nonlinear dynamics.

Acknowledgments. We thank Luca Geretti for helping us setting up Ariadne. This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award), by the European Commission under grant 643921 (UnCoVerCPS).

References

1. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 313–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_22
2. Althoff, M.: An introduction to CORA 2015. In: Frehse, G., Althoff, M. (eds.) ARCH14-15. 1st and 2nd International Workshop on Applied verification for Continuous and Hybrid Systems. EPiC Series in Computer Science, vol. 34, pp. 120–151. EasyChair (2015)
3. Alur, R., Dang, T., Ivančić, F.: Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci.* **354**(2), 250–271 (2006)
4. Asarin, E., Dang, T., Maler, O., Testylier, R.: Using redundant constraints for refinement. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 37–51. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_5
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* **72**(1–2), 3–21 (2008)
6. Benvenuti, L., Bresolin, D., Collins, P., Ferrari, A., Geretti, L., Villa, T.: Assume-guarantee verification of nonlinear hybrid systems with Ariadne. *Int. J. Robust Nonlinear Control* **24**(4), 699–724 (2014)
7. Bogomolov, S., Frehse, G., Giacobbe, M., Henzinger, T.A.: Counterexample-guided refinement of template polyhedra. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 589–606. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_34
8. Chen, X., Abraham, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: RTSS 2012, pp. 183–192 (2012)
9. Chen, X., Schupp, S., Makhoul, I.B., Abraham, E., Frehse, G., Kowalewski, S.: A benchmark suite for hybrid systems reachability analysis. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 408–414. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_29
10. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.* **14**(04), 583–604 (2003)
11. Collins, P., Bresolin, D., Geretti, L., Villa, T.: Computing the evolution of hybrid systems using rigorous function calculus. In: Proceedings of the 4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS12), Eindhoven, The Netherlands, pp. 284–290, June 2012

12. Dang, T., Gawlitza, T.M.: Template-based unbounded time verification of affine hybrid automata. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 34–49. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_6
13. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT **10**(3), 263–279 (2008)
14. Frehse, G., Bogomolov, S., Greitschus, M., Strump, T., Podelski, A.: Eliminating spurious transitions in reachability with support functions. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 149–158. ACM (2015)
15. 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). https://doi.org/10.1007/978-3-642-22110-1_30
16. 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). https://doi.org/10.1007/978-3-642-02658-4_40
17. Halbwachs, N., Proy, Y.-E., Raymond, P.: Verification of linear hybrid systems by means of convex approximations. In: Le Charlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58485-4_43
18. Henzinger, T., Ho, P.H., Wong-Toi, H.: HyTech: a model checker for hybrid systems. *Softw. Tools Technol. Transf.* **1**, 110–122 (1997)
19. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) *Verification of Digital and Hybrid Systems*, vol. 170, pp. 265–292. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59615-5_13
20. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. *IEEE Trans. Autom. Control* **43**, 540–554 (1998)
21. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What’s decidable about hybrid automata? In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, 29 May–1 June 1995, Las Vegas, Nevada, USA, pp. 373–382 (1995)
22. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_24
23. Johansson, F.: Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Trans. Comput.* **66**, 1281–1292 (2017)
24. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ -reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
25. Moler, C., Van Loan, C.: Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Rev.* **45**(1), 3–49 (2003)
26. Nellen, J., Ábrahám, E., Wolters, B.: A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In: Bouabana-Tebibel, T., Rubin, S.H. (eds.) *Formalisms for Reuse and Systems Integration*. AISC, vol. 346, pp. 55–78. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16577-6_3
27. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Trans. Embed. Comput. Syst.* (TECS) **6**(1), 8 (2007)

28. Rockafellar, R.T.: *Convex Analysis*. Princeton University Press, Princeton (1970)
29. Rohn, J.: Systems of linear interval equations. *Linear Algebra Appl.* **126**, 39–78 (1989)
30. Roohi, N., Prabhakar, P., Viswanathan, M.: Hybridization based CEGAR for hybrid automata with affine dynamics. In: Chechik, M., Raskin, J.-F. (eds.) *TACAS 2016*. LNCS, vol. 9636, pp. 752–769. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_48
31. 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). https://doi.org/10.1007/978-3-540-78800-3_14
32. 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). https://doi.org/10.1007/978-3-540-30579-8_2
33. Segelken, M.: Abstraction and counterexample-guided construction of ω -automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_46
34. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) *FORMATS/FTRTFT-2004*. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_25
35. Vaandrager, F.: *Hybrid systems*. Images of SMC Research, pp. 305–316 (1996)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Monitoring Weak Consistency

Michael Emmi¹(✉) and Constantin Enea²

¹ SRI International, New York, NY, USA

`michael.emmi@sri.com`

² IRIF, Univ. Paris Diderot and CRNS, Paris, France

`cenea@irif.fr`

Abstract. High-performance implementations of distributed and multicore shared objects often guarantee only the weak consistency of their concurrent operations, foregoing the de-facto yet performance-restrictive consistency criterion of linearizability. While such weak consistency is often vital for achieving performance requirements, practical automation for checking weak-consistency is lacking. In principle, algorithmically checking the consistency of executions according to various weak-consistency criteria is hard: in addition to the enumeration of linearizations of an execution's operations, such criteria generally demand the enumeration of possible visibility relations among the linearized operations; a priori, both enumerations are exponential.

In this work we identify an optimization to weak-consistency checking: rather than enumerating every possible visibility relation, it suffices to consider only the *minimal* visibility relations which adhere to the various constraints of the given criterion, for a significant class of consistency criteria. We demonstrate the soundness of this optimization, and describe an associated minimal-visibility consistency checking algorithm. Empirically, we show that our algorithm significantly outperforms the baseline weak-consistency checking algorithm, which naïvely enumerates all visibilities, and adds only modest overhead to the baseline linearizability checking algorithm, which does not enumerate visibilities.

Keywords: Linearizability · Consistency · Runtime verification

1 Introduction

Programming software applications that can deal with multiple clients at the same time, and possibly, with clients that connect at different sites in a network, relies on optimized concurrent or distributed objects which encapsulate lock-free shared memory access or message passing protocols into high-level abstract data types. Given the potentially-enormous amount of software that relies on

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

© The Author(s) 2018

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 487–506, 2018.

https://doi.org/10.1007/978-3-319-96145-3_26

these objects, it is important to maintain precise specifications and ensure that implementations adhere to their specifications.

One of the standard correctness criteria used in this context is linearizability (or strong consistency) [22], which ensures that the results of concurrently-executed invocations match the results of some serial execution of those same invocations. Ensuring such a criterion in a distributed context (when data is replicated at different sites in a network) is practically infeasible or even impossible [17, 19]. Therefore, various weak consistency criteria have been proposed like eventual consistency [23, 36], “session guarantees” like read-my-writes or monotonic-reads [35], causal consistency [25, 28], etc.

An axiomatic framework for formalizing such criteria has been proposed by Burckhardt et al. [9, 11]. Essentially, this extends the linearizability-based specification methodology with a dynamic *visibility* relation among operations, in addition to the standard dynamic *happens-before* and *linearization* relations. Permitting weaker visibility relations models outcomes in which an operation may not observe the effects of concurrent operations that are linearized before it.

In this work, we propose an online monitoring algorithm that checks whether an execution of a concurrent (or distributed) object satisfies a consistency model defined in this axiomatic framework. This algorithm constructs a linearization and visibility relation satisfying the axioms of the consistency model gradually as the execution extends with more operations. It is possible that the linearization and visibility constructed until some point in time are invalidated as more operations get executed, which requires the algorithm to backtrack and search for different candidates. This exponential blow-up is unavoidable since even the problem of checking linearizability is NP-hard in general [18].

The main difficulty in devising such an algorithm is coming up with efficient strategies for enumerating linearizations and visibility relations which minimize the number of candidates needed to be explored and the number of times the algorithm has to backtrack. We build on previous works that propose such strategies for enumerating linearizations [29, 38] in the context of linearizability checking. Roughly, the linearizations are extended iteratively by appending operations which are minimal in the happens-before order (among non-linearized operations). The choice of the minimal operations to append varies from one approach to the other. Our work focuses on combining such strategies with an efficient enumeration of visibility relations which are compatible with a given linearization.

Rather than specializing our results to one single consistency model, we consider a general class of consistency models from Burckhardt et al.’s axiomatic framework [9, 11] in which the visibility relation among operations is constrained to be contained in the linearization relation. That class includes, for instance, time-stamp based models employed in distributed object implementations, in which time stamps serve to resolve conflicts by effectively linearizing concurrent operations. We show that within this class of consistency models, it is *not* necessary to enumerate the set of all possible visibility relations (included in the

linearization) in order to check consistency of an execution. More precisely, we develop an algorithm for enumerating visibility relations that traverses operations in linearization order and chooses for each operation o , a *minimal* set of operations visible to o that conforms to the consistency axioms (up to the linearization prefix that includes o). In general there may exist multiple such minimal sets of operations, and each of them must be explored. When the visibility relation cannot be extended, the algorithm needs to backtrack and choose different minimal visibility sets for previous operations. However, when all the minimal candidates have been explored, the algorithm can soundly report that the execution is not consistent, without resorting to the exploration of non-minimal visibility relations.

Besides demonstrating the soundness of minimal-visibility consistency checking, we also demonstrate its empirical impact by applying our algorithm to concurrent traces of Java concurrent data structures. We find that our algorithm consistently outperforms the baseline naïve approach to enumerating visibilities, which considers also non-minimal visibility relations. Furthermore, we demonstrate that minimal-visibility checking adds only modest overhead (roughly $2\times$) to the baseline linearizability checking algorithm, which does not enumerate visibilities. This suggests that small sets of minimal visibilities typically suffice in practice, and that the additional exponential enumeration of visibilities, atop the exponential enumeration of linearizations, may be avoidable in practice. Our implementation and experiments are open source, and publicly available on GitHub.¹

In summary, this work makes the following contributions:

- we develop a new *minimal-visibility* consistency-checking algorithm for Burckhardt et al.’s axiomatic consistency framework [9, 11];
- we demonstrate the soundness of minimal-visibility consistency checking; and
- we demonstrate an empirical evaluation comparing minimal-visibility consistency checking with the state-of-the-art consistency-checking algorithms.

To the best of our knowledge, our algorithm is the first completely automatic algorithm for checking weak-consistency of arbitrary abstract data type implementations which avoids the naïve enumeration of all possible visibility relations.

The rest of this paper is organized as follows. Section 2 elaborates a formalization of Burckhardt et al.’s axiomatic consistency framework [9, 11], and Sect. 3 develops a formal argument to the soundness of considering only minimal visibility relations. Section 4 describes our overall consistency checking algorithms, and Sect. 5 describes our implementation and empirical evaluation. Section 6 describes related work, and finally Sect. 7 concludes.

2 Weak Consistency

We describe a formal model for concurrent (distributed) object implementations. Clients interact with an object by making *invocations* from a set \mathbb{I} and receiving

¹ <https://github.com/michael-emmi/violat/releases/tag/cav-2018-submission>.

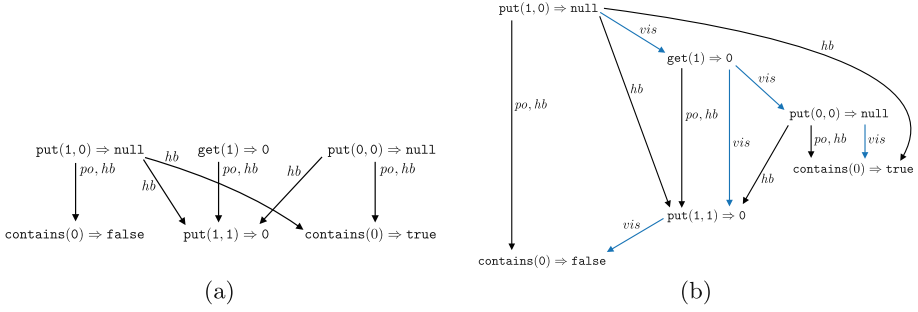


Fig. 1. A history h and an abstract execution containing h .

returns from a set \mathbb{R} (parameters of invocations, if any, are part of the invocation name). An *operation* is an invocation $i \in I$ paired with a return $r \in R$; we denote such an operation by $i \Rightarrow r$. We denote individual operations by o . The invocation, resp., the return, in an operation o is denoted by $\text{inv}(o)$, resp., $\text{ret}(o)$.

The interaction between a client and an object is represented by a *history* $\langle po, hb \rangle$ over a set of operations O which consists of

- a *program (order)* po which is a partial order on O , and
- a *happens-before (order)* hb which is a partial order on O .

The program order is enforced by the client, e.g., by invoking a set of operations within the same thread or process, while the happens-before order represents the order in which the operations finished, i.e., $(o_1, o_2) \in hb$ iff operation o_1 finished before o_2 started. We assume that the program order is included in the happens-before order.

Example 1. Let us consider a key-value map ADT containing operations of the form $\text{put}(\text{key}, \text{value}) \Rightarrow \text{old}$, which insert key-value pairs and return previously-mapped values for the given keys, $\text{remove}(\text{key}) \Rightarrow \text{value}$, which remove key mappings and return previously-mapped values, $\text{contains}(\text{value}) \Rightarrow \text{true/false}$, which test whether values are currently mapped, and $\text{get}(\text{key}) \Rightarrow \text{value}$, which return currently-mapped values for the given keys. Figure 1(a) pictures a history h where edges denote the program order po and happens-before hb . Such a history can be obtained by a client with three threads each making two invocations (the invocations within the same thread are aligned vertically).

The axiomatic specifications of concurrent objects we consider are based on the following abstract representation of executions: an *abstract execution* over operations O is a tuple $\langle po, hb, lin, vis \rangle$ that consists of a history $\langle po, hb \rangle$ over O ,

- a *linearization (order)* lin^2 which is a total order on O , and
- a *visibility (relation)* vis which is an acyclic relation on O .

² The linearization is also called *arbitration* in previous works, e.g., [9].

Intuitively, the visibility relation represents the inter-thread communication, how effects of operations are visible to other threads, while the linearization order models the “conflict resolution policy”, how the effects of concurrent operations are ordered when they become visible to other threads.

We say that an operation o_1 such that $\langle o_1, o_2 \rangle \in \text{vis}$ is *visible* to o_2 , and that o_2 *sees* o_1 . Also, the set of operations visible to o_2 is called the *visibility set* of o_2 . The extensions of *inv* and *ret* to partial orders on O are defined component-wise as usual.

Example 2. Figure 1(b) pictures an abstract execution containing the history in Fig. 1(a). The visibility relation is defined by the edges labeled *vis* together with their transitive closure. The linearization order is defined by the order in which operations are written (from top to bottom).

A consistency criterion for concurrent objects is defined by a set of axioms over the relations in an abstract execution. These axioms relate abstract executions to a sequential semantics of the operations, which is defined by a function $\text{Spec} : \mathbb{I}^* \times \mathbb{I} \rightarrow \mathbb{R}$ that determines the return value of an invocation given the sequence of invocations previously executed on the object³.

Example 3. The sequential semantics of the key-value map ADT considered in Example 1 is defined as expected. For instance, the return value of `put(key, value)` after a sequence of invocations σ is the value `null` if σ contains no invocation `put(key, ...)`, or `old` if `put(key, old)` is the last invocation of the form `put(key, ...)` in σ .

The *domain* $\text{dom}(R)$ of a relation R is the set of elements x such that $\langle x, y \rangle \in R$ for some y ; the *codomain* $\text{codom}(R)$ is the set of elements y such that $\langle x, y \rangle \in R$ for some x . By an abuse of notation, if x is an individual element, $x \in R$ denotes the fact that $x \in \text{dom}(R) \cup \text{codom}(R)$. The (*left*) *composition* $R_1 \circ R_2$ of two binary relations R_1 and R_2 is the set of pairs $\langle x, z \rangle$ such that $\langle x, y \rangle \in R_1$ and $\langle y, z \rangle \in R_2$ for some y . We denote the identity binary relation $\{\langle x, x \rangle : x \in X\}$ on a set X by $[X]$, and we write $[x]$ to denote $\{x\}$.

Return-value consistency [9], a variant of eventual consistency without liveness guarantees, states that the return r of every operation $i \Rightarrow r$ can be obtained from a sequential execution of i that follows the invocations visible to o (in the linearization order). This constraint will be formalized as an axiom called **Ret**. The visibility relation can be chosen arbitrarily. Standard “session guarantees” can be described in the same framework by adding constraints on the visibility relation: for instance, *read my writes*, i.e., operations previously executed in the same thread remain visible, can be stated as $\text{vis} \supseteq \text{po}$ and *monotonic reads*, i.e., the set of visible operations to some thread grows monotonically over time, can

³ Previous works have considered more general, concurrent semantics for operations. We restrict ourselves to sequential semantics in order to simplify the exposition. Our results extend easily to the general case.

$$\begin{array}{ll}
\phi ::= \text{Ret} \mid \text{ord} & \langle po, hb, lin, vis \rangle \models \text{Ret} \text{ iff} \\
ord ::= qrel \supseteq rel & \forall o. ret(o) = Spec(inv(ctxt(lin, vis, o)), inv(o)) \\
qrel ::= lin \mid vis & \langle po, hb, lin, vis \rangle \models ord \text{ iff} \\
rel ::= qrel \mid po \mid hb \mid rel \circ rel & ord[po/po][hb/hb][lin/lin][vis/vis] \text{ is valid}
\end{array}$$

Fig. 2. The grammar of consistency axioms.

Fig. 3. Consistency axiom satisfaction for abstract executions. The satisfaction relation \models is implicitly parameterized by a sequential semantics $Spec$ which we consider fixed.

be stated as $vis \supseteq vis \circ po$. Then, a version of causal consistency [7, 9], called *causal convergence*, is defined by the following set of axioms:

$$vis \supseteq vis \circ vis \quad vis \supseteq po \quad lin \supseteq vis \quad \text{Ret}$$

which state that the visibility relation is transitive, it includes program order, and it is included in the linearization order. Finally, *linearizability* is defined by the set of axioms $lin \supseteq hb$, $vis = lin$, and Ret .

To state our results in a general context that concerns multiple consistency criteria defined in the literature (including the ones mentioned above) and variations there of, we consider a language of *consistency axioms* ϕ defined by the grammar in Fig. 2. A *consistency model* Φ is a set $\{\phi_1, \phi_2, \dots\}$ of consistency axioms.

In the following, we assume that every consistency model is stronger than return-value consistency, and also, that the linearization order is consistent with the visibility and happens-before relations. The assumptions concerning the linearization order correspond to the fact that for instance, concurrent operations are ordered using timestamps that correspond to real-time. Formally, we assume that every consistency model contains the axioms

$$\Phi_0 = \{\text{Ret}, lin \supseteq vis, lin \supseteq hb\}.$$

Figure 3 defines the precise semantics of consistency axioms on abstract executions: the *context* of an operation o according to a linearization lin and visibility vis , denoted $ctxt(lin, vis, o)$ is the restriction $([O_o] \circ lin \circ [O_o])$ of lin to the operations $O_o = \text{dom}(vis \circ [o])$ visible to o . For instance, for the abstract execution in Fig. 1(b), $ctxt(lin, vis, \text{contains}(0) \Rightarrow \text{false})$ is the sequence of operations $\text{put}(1, 0) \Rightarrow \text{null}; \text{get}(1) \Rightarrow 0; \text{put}(1, 1) \Rightarrow 0$.

We extend this semantics to consistency models as $e \models \Phi$ iff $e \models \phi$ for all $\phi \in \Phi$ and to histories as:

$$\langle po, hb \rangle \models \Phi \text{ iff } \exists lin, vis. \langle po, hb, lin, vis \rangle \models \Phi$$

Example 4. The abstract execution in Fig. 1(b) satisfies causal convergence: the visibility relation is transitive, it includes program order, and it is consistent with the linearization order. Moreover, the axiom Ret is also satisfied.

For instance, the invocation `contains(0)` returns exactly `false` when executed after `put(1, 0); get(1); put(1, 1)`. Similarly, it returns `true` when executed after `put(1, 0); get(1); put(0, 0)`.

3 Minimal Visibility Extensions

Checking whether a given history satisfies a consistency model is intractable in general. This essentially follows from the fact that checking linearizability is NP-hard in general [18]. While the main issue in checking linearizability is enumerating the exponentially many linearizations, checking weaker criteria like causal convergence requires also an enumeration of the exponentially many visibility relations (included in a given linearization). We prove in this section that it is enough to enumerate only *minimal* visibility relations (w.r.t. set inclusion), included in a given linearization, in order to conclude whether a given history and linearization satisfy a consistency model.

A *linearized history* $\sigma = \langle po, hb, lin \rangle$ consists of a history and a linearization lin such that $lin \supseteq hb$. The extension of \models to linearized histories is defined as:

$$\langle po, hb, lin \rangle \models \Phi \text{ iff } \exists vis. \langle po, hb, lin, vis \rangle \models \Phi$$

The i -th element of a sequence s is denoted by $s[i]$ and the prefix of s of length i is denoted by s_i . The projection of a linearized history $\sigma = \langle po, hb, lin \rangle$ to a prefix lin_i of lin is denoted by σ_i . Formally, $O_i = \text{dom}(lin_i) \cup \text{codom}(lin_i)$ and $\sigma_i = \langle po \cap (O_i \times O_i), hb \cap (O_i \times O_i), lin_i \rangle$.

For a linearized history $\langle po, hb, lin \rangle$ and a consistency model Φ , a visibility relation vis_i on operations from a prefix lin_i of lin is called Φ -*extensible* when there exists a visibility relation $vis \supseteq vis_i$ such that $\langle po, hb, lin, vis \rangle \models \Phi$. The relation vis is called a Φ -*extension* of vis_i up to lin . By extrapolation, a Φ -extension of vis_i up to lin_j is a visibility relation vis_j such that $\langle \sigma_j, vis_j \rangle \models \Phi$, for any $i < j$. Such an extension is called *minimal* when for every other Φ -extension vis'_j of vis_i up to lin_j , we have that $vis'_j \not\subseteq vis_j$.

Example 5. Consider again the abstract execution in Fig. 1(b). Ignoring the edges labeled by vis , it becomes a linearized history σ . The prefix σ_2 contains just the two operations `put(1, 0) ⇒ null` and `get(1) ⇒ 0`. For causal convergence, the visibility relation $vis_2 = \{\langle \text{put}(1, 0) \Rightarrow \text{null}, \text{get}(1) \Rightarrow 0 \rangle\}$ on operations of σ_2 is extensible, as witnessed by the visibility relation defined for the rest of the operations in this execution. The visibility relation

$$vis_3 = \{\langle \text{put}(1, 0) \Rightarrow \text{null}, \text{get}(1) \Rightarrow 0 \rangle, \langle \text{put}(1, 0) \Rightarrow \text{null}, \text{put}(0, 0) \Rightarrow \text{null} \rangle, \\ \langle \text{get}(1) \Rightarrow 0, \text{put}(0, 0) \Rightarrow \text{null} \rangle\}$$

is an extension of vis_2 up to lin_3 , and contains the operations in σ_2 together with `put(0, 0) ⇒ null`. Note that this extension is *not* minimal. A minimal extension would be exactly equal to vis_2 since, intuitively, `put(0, 0) ⇒ null` is not required to observe operations on keys other than 0.

The next lemma shows that minimizing the visibility sets of operations in a linearization prefix, while preserving the truth of the axioms on that prefix, doesn't exclude visibility choices for future operations (occurring beyond that prefix). In more precise terms, the Φ -extensibility status is not affected by choosing smaller visibility sets for operations in a linearization prefix. For instance, since the visibility vis_3 in Example 5 is extensible (for causal convergence), the smaller visibility relation in which $\text{put}(0,0) \Rightarrow \text{null}$ doesn't see any operation, is also extensible. This result relies on the specific form of the axioms, which ensure that smaller visibility sets impose fewer constraints on the visibility sets of future operations. For instance, the axiom $vis \supseteq vis \circ vis$ enforces that vis contains $\{\langle o, o_2 \rangle : \langle o, o_1 \rangle \in vis\}$ whenever a pair $\langle o_1, o_2 \rangle$ is added to vis . Minimizing the visibility set of o_1 will minimize the set of operations that *must* be seen by o_2 , thus making the choice of the operations visible to o_2 more liberal.

Lemma 1. *For every linearized history σ and consistency model Φ , if*

$$\langle \sigma_i, vis_i \rangle \models \Phi, \quad vis_i \text{ is } \Phi\text{-extensible}, \quad \langle \sigma_i, vis'_i \rangle \models \Phi, \quad \text{and } vis'_i \subseteq vis_i,$$

then vis'_i is Φ -extensible.

Proof (Sketch). We show that the Φ -extension vis of vis_i up to lin can be transformed to a Φ -extension of vis'_i up to lin by simply removing the pairs of operations in $vis_i \setminus vis'_i$. Let vis' be this visibility relation and Φ a consistency model. We prove that $\langle po, hb, lin, vis' \rangle \models \Phi$ by considering the different types of axioms defined in Fig. 2.

Suppose that Φ contains an axiom of the form $vis \supseteq rel$ (according to the notations in Fig. 2). We have that $vis'_i \supseteq (rel[po/po][hb/hb][lin/lin][vis'/vis]) \circ [O_i]$ by the hypothesis (from $\langle \sigma_i, vis'_i \rangle \models \Phi$). Then, $vis'_i \subseteq vis_i$ implies that

$$\begin{aligned} & (rel[po/po][hb/hb][lin/lin][vis/vis]) \circ [O \setminus O_i] \\ & \supseteq (rel[po/po][hb/hb][lin/lin][vis'/vis]) \circ [O \setminus O_i] \end{aligned}$$

which together with $vis' \circ [O \setminus O_i] = vis \circ [O \setminus O_i]$ (the visibility relations vis and vis' are the same for operations which are not included in the prefix lin_i) implies that

$$vis' \circ [O \setminus O_i] \supseteq (rel[po/po][hb/hb][lin/lin][vis'/vis]) \circ [O \setminus O_i].$$

Therefore, $\langle po, hb, lin, vis' \rangle \models vis \supseteq rel$.

The axiom Ret relates the return value of each operation o in σ to the set of operations visible to o . This relation is insensitive to the set of operations seen by an operation before o in the linearization order. Therefore, $\langle po, hb, lin, vis' \rangle \models \text{Ret}$ is an immediate consequence of $\langle \sigma_i, vis'_i \rangle \models \text{Ret}$ and the fact that vis and vis' are the same for operations which are not included in the prefix lin_i .

The axioms of the form $lin \supseteq rel$ (according to the notations in Fig. 2) are straightforward implications of $lin \supseteq hb$ and $lin \supseteq vis$, which are assumed to be included in any consistency model. They hold for any linearized history. \square

The main result of this section shows that a visibility enumeration strategy that considers operations in the linearization order and computes minimal extensions iteratively, possibly backtracking to another choice of minimal extension if necessary, is complete in general (it finds a visibility relation satisfying the consistency axioms Φ iff the input linearized history satisfies Φ). Backtracking is necessary since in general, there may exist multiple minimal extensions and all of them should be explored. For a given linearized history σ and visibility relation vis on operations of σ , $vis_i = vis \circ [O_i]$ denotes the restriction of vis to operations from the prefix lin_i .

Theorem 1. *For every linearized history σ and consistency model Φ , $\sigma \models \Phi$ iff there exists a visibility relation vis such that*

for every i , vis_{i+1} is a minimal Φ -extension of vis_i up to lin_{i+1} .

Proof. (Sketch) Let σ be a linearized history such that $\sigma \models \Phi$. Therefore, there exists a visibility relation vis such that $\langle \sigma, vis \rangle \models \Phi$. We prove by induction that there exists a visibility relation vis' satisfying the claim of the theorem. Assume that there exists a Φ -extensible visibility relation vis^j on operations in lin_j which satisfies the claim of the theorem for every $i < j$ (we take $vis^0 = vis$). Let vis^{j+1} be a minimal visibility relation on operations in lin_{j+1} such that $vis^{j+1} \circ [O_j] = vis^j \circ [O_j]$ and $(\sigma_{j+1}, vis^{j+1}) \models \Phi$ (such a set exists because vis^j is Φ -extensible). By Lemma 1, vis^{j+1} is Φ -extensible. Also, vis^{j+1} satisfies the claim of the theorem for every $i < j + 1$. The reverse direction is trivial. \square

Example 6. In the context of the abstract execution in Fig. 1(b), the visibility relation defined by removing the vis edge ending in $put(0, 0) \Rightarrow null$, and adding the transitive closure, satisfies the requirements in Theorem 1.

4 Efficient Monitoring of Consistency Models

We describe an algorithm for checking whether a given history satisfies a consistency model, which combines linearization enumeration strategies proposed in [29, 38] with the visibility enumeration strategy proposed in Sect. 3.

The algorithm is defined by the procedure **checkConsistency** listed in Fig. 4. This recursive procedure searches for extensions of the input linearization and visibility (initially, **checkConsistency** will be called with $lin = vis = \emptyset$) which witness that the input history h satisfies Φ . It assumes that the inputs lin and vis satisfy the axioms of the consistency model Φ when the input history is projected on the linearized operations (the operations in lin). This projection is denoted by h_{lin} . Formally, the precondition of this procedure is that $\langle h_{lin}, lin, vis \rangle \models \Phi$.

The extensions of lin and vis are built in successive steps. At each step, the linearization is extended according to the procedure **linExtensions** and the visibility according to the procedure **visExtensions**.

The abstract implementation of **linExtensions**, presented in Fig. 4, chooses a set of *non-linearized* operations O which are *minimal* among non-linearized

```

proc checkConsistency( $h, \Phi, \text{lin}, \text{vis}$ ) {
  if (isComplete( $h, \text{lin}$ )) then
    return true;
  forall  $\text{lin}'$  of linExtensions( $h, \text{lin}$ ) do
    forall  $\text{vis}'$  of visExtensions( $h, \text{lin}', \text{vis}$ ) do
      if checkConsistency( $h, \Phi, \text{lin}', \text{vis}'$ ) then
        return true;
  return false;
}

proc linExtensions( $h, \text{lin}$ ) {
  let  $O = \text{minimals}(h, \text{lin})$ ;
  forall  $O'$  of subsets( $O$ )
    forall  $\text{seq}$  of linearizations( $O'$ )
      let  $\text{lin}' = \text{append}(\text{lin}, \text{seq})$ ;
      yield  $\text{lin}'$ ;
}

proc visExtensions( $h, \text{lin}, \text{vis}$ ) {
  forall  $\text{vis}'$  a minimal  $\Phi$ -extension
    of  $\text{vis}$  up to  $\text{lin}$ 
    yield  $\text{vis}'$ ;
}

```

Fig. 4. Checking consistency of a history. The procedures `linExtensions`, resp., `visExtensions` return the set of linearizations, resp., visibilities, produced by the instruction `yield`.

operations w.r.t. happens-before, i.e., returned by `minimals`(h, lin), and appends any linearization of the operations in O to the input linearization lin . Formally, $O \subseteq \{o : o \notin \text{lin} \text{ and } \forall o'. o' \notin \text{lin} \Rightarrow \neg o' \prec o\}$, where \prec denotes the happens-before relation. The fact that the operations in O are minimal among non-linearized operations ensures that the returned linearizations are consistent with the happens-before order.

Two linearization enumeration strategies proposed in the literature can be seen as instances of `linExtensions`. The strategy in [38] corresponds to the case where O contains exactly one minimal operation. For instance, for the history in Fig. 1(a), this strategy will start by picking a minimal element in the happens-before relation, say `put(1, 0) \Rightarrow null`, then, a minimal operation among the rest, say `get(1) \Rightarrow 0`, and so on.

The strategy proposed in [29] is slightly more involved (and according to experimental results, more efficient), but it relies on a presentation of histories h as sequences of call and return actions (an operation spanning the time interval between its call and return action). The happens-before order is extracted as usual: an operation o_1 happens before an operation o_2 if its return occurs before the call of o_2 . This strategy defines O as the first non-linearized operation o that returned in h together with a set of non-linearized operations O' that are concurrent with o (i.e., are not ordered after o in the happens-before order). The operation o is linearized last in the returned extensions. For instance, consider the history h in Fig. 5 represented as a sequence of call/return actions (small boxes at the begin, resp., end, of an interval denote call actions, resp., return actions). The first linearization extension (when $\text{lin} = \emptyset$) includes `put(1, 0) \Rightarrow null` (the first operation to return) after some sequence of operations concurrent with it, for

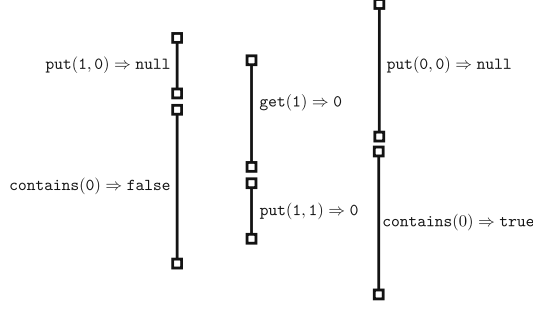


Fig. 5. The history h in Fig. 1 presented as a sequence of call/return actions.

instance the empty sequence. Next, the current linearization $\text{put}(1,0) \Rightarrow \text{null}$ can be extended by adding $\text{put}(0,0) \Rightarrow \text{null}$ (the first operation to return, if we exclude $\text{put}(1,0) \Rightarrow \text{null}$ which is already linearized) and possibly $\text{get}(1) \Rightarrow 0$ before it. Suppose that we choose $\text{put}(1,0) \Rightarrow \text{null}; \text{get}(1) \Rightarrow 0; \text{put}(0,0) \Rightarrow \text{null}$. Then, the extension will include $\text{put}(1,1) \Rightarrow 0$ and possibly $\text{contains}(0) \Rightarrow \text{true}$ or $\text{contains}(0) \Rightarrow \text{false}$, and so on. Compared to the previous strategy, an extension step can add multiple operations.

The extensions of the visibility relation (returned by `visExtensions`) are minimal Φ -extensions of vis up to the input linearization. They can be constructed iteratively by considering the newly linearized operations one by one and each time compute a minimal extension of the visibility. For instance, the linearization construction explained in the previous paragraph can be expanded with a visibility enumeration as follows:

- $\text{lin} = \text{put}(1,0) \Rightarrow \text{null}$: the minimal visibility is $\text{vis}_1 = \emptyset$,
- $\text{lin} = \text{put}(1,0) \Rightarrow \text{null}; \text{get}(1) \Rightarrow 0; \text{put}(0,0) \Rightarrow \text{null}$: the minimal visibility is $\text{vis}_2 = \{\langle \text{put}(1,0) \Rightarrow \text{null}, \text{get}(1) \Rightarrow 0 \rangle\}$, and so on.

The procedure `checkConsistency` backtracks to a different extension when the current one cannot be completed to include all the operations in the input history (checked by the recursive call). The correctness of the algorithm is stated in the following theorem.

Theorem 2. $\text{checkConsistency}(h, \Phi, \emptyset, \emptyset)$ returns true iff $h \models \Phi$.

5 Empirical Results

While our minimal-visibility consistency checking algorithm is applicable to a wide class of distributed and multicore shared object implementations, here we demonstrate its efficacy on histories recorded from executions of Java Development Kit (JDK) Standard Edition concurrent data structures. Recent work demonstrates that JDK concurrent data structures regularly admit

non-atomic behaviors, often by design [14]; these weakly-consistent behaviors span many methods of the `java.util.concurrent` package, including the `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `ConcurrentLinkedQueue`, and the `ConcurrentLinkedDeque`, for instance, including the contains method described in Example 3.

We extracted 4,000 randomly-sampled histories from approximately 8,000 observed over approximately 1,000,000 executions in stress testing 20 randomly-generated client programs of the `ConcurrentSkipListMap` with up to 15 invocations across up to 3 threads. In each program, the given number of threads invokes its share of randomly-generated methods with randomly-generated values. We consider random generation superior to collecting programs *in the wild*, since found client programs can mask inconsistencies by restricting method argument values, or by being agnostic to inconsistent return values. Furthermore, automated generation gives us the ability to evaluate our algorithm on unbiased sample sets, and avoid any technical problems in the collection of programs; it also allows us to test method combinations which might not appear in publicly-available examples.

We subject each client program to 1s of stress testing⁴ to record histories. The return value of each invocation is stored in a different thread-local variable which is read at the end of the execution. Recording the happens-before order between invocations without affecting implementation behavior significantly (e.g., without influencing the memory orderings between shared-memory accesses) is challenging. For instance, we found the use of high-precision timers to be unsuitable, since the response-time of `System.nanoTime` calls is much higher than calls to the implementations under test; invoking such timers between each invocation of implementation methods would prevent implementation methods from overlapping in time, and thus hide any possible inconsistent behaviors. Similarly, the use of atomic operations and volatile variables would impose additional synchronization constraints and prevent many weak-memory reorderings.

Essentially, our solution is to introduce a shared variable per thread storing its program counter – in our context, the program counter stores the number of call and return events thus far executed. A thread’s program counter is read by every other thread before and after each invocation. Figure 6 demonstrates a simplified version⁵ of our encoding for a program with two threads each invoking two methods. The program counter variables `pc0` and `pc1` are not declared volatile, which, in principle, provides stronger guarantees concerning the derived happens-before relation; such declarations would interfere with implementation weak-memory effects. The program counter values read by each thread allows

⁴ For stress testing we leverage OpenJDK’s JCTestress tool: <http://openjdk.java.net/projects/code-tools/jctestress/>.

⁵ In our actual implementation, each program-counter access is encapsulated within a method call in order to avoid compiler reordering between the reads of other threads’ counters and the increment of one’s own. While the Java memory model does not guarantee that such encapsulation will prevent reordering, we found this solution to be adequate on Oracle’s Java SE runtime version 9. Our actual implementation also wraps invocations in try-catch blocks to deal with exceptions.

```

int pc0 = 0, pc1 = 0;
ConcurrentHashMap obj = new ConcurrentHashMap();

void thread0() {
    Object r0, r1;
    int pcs[] [] = new int[4][1];
    int n = 0;

    // first invocation
    pcs[n][0] = pc1; n++; pc0++;
    r0 = obj.elements();
    pcs[n][0] = pc1; n++; pc0++;

    // second invocation
    pcs[n][0] = pc1; n++; pc0++;
    r1 = obj.put(1,0);
    pcs[n][0] = pc1; n++; pc0++;

    // store the values of r0, r1, pcs
    ...
}

void thread1() {
    Object r0, r1;
    int pcs[] [] = new int[4][1];
    int n = 0;

    // first invocation
    pcs[n][0] = pc0; n++; pc1++;
    r0 = obj.remove(1);
    pcs[n][0] = pc0; n++; pc1++;

    // second invocation
    pcs[n][0] = pc0; n++; pc1++;
    r1 = obj.put(0,1);
    pcs[n][0] = pc0; n++; pc1++;

    // store the values of r0, r1, pcs
    ...
}

```

Fig. 6. Our encoding for recording ConcurrentHashMap histories. Each thread’s program counter is read before and after other threads’ invocations, and incremented subsequent to each such read. The two-dimensional `pcs[n][m]` array stores n program counter values for m neighboring threads.

us to extract a happens-before order between invocations which is *sound* in the sense that the actual happens-before may order more operations, but not fewer – assuming that shared-memory accesses satisfy at least the total-store order (TSO) semantics in which writes are guaranteed to be performed according to program order. For instance, when `pcs[0][0] > 2` in the second thread (`thread1`), the first invocation in the other thread (`thread0`) happens-before the first invocation in this thread. Otherwise, if `pcs[0][0] < 2`, then the two invocations are overlapping in time. The latter may not be true in the real happens-before due to the delay in incrementing and reading the program counter variables. Although some loss of precision is possible, we are unaware of other methods for tracking happens-before which avoid significant interference with the implementation under test.

Based on the encoding described above, we generate histories as sequences of call and return actions which serve as input to our consistency checking algorithms. For simplicity, we have considered just two consistency models, linearizability and a weak consistency model defined by $\{\text{Ret}, \text{lin} \supseteq \text{vis}, \text{lin} \supseteq \text{hb}, \text{vis} \supseteq \text{hb}\}$ – see Sect. 2. We consider linearizability in order to measure the overhead of checking weak consistency due to visibility enumeration; the second model is simply the easiest weak-consistency model to support with our implementation; the choice among possible weak-consistency models appears fairly arbitrary, since the enumeration of visibility relations is common to all.

We consider several measurements, the results of which are listed in Figs. 7 and 8; all times are measured in milliseconds on logarithmic scale on a 2.7 GHz Intel Core i5 MacBook Pro with Oracle’s Java SE runtime version 9; and

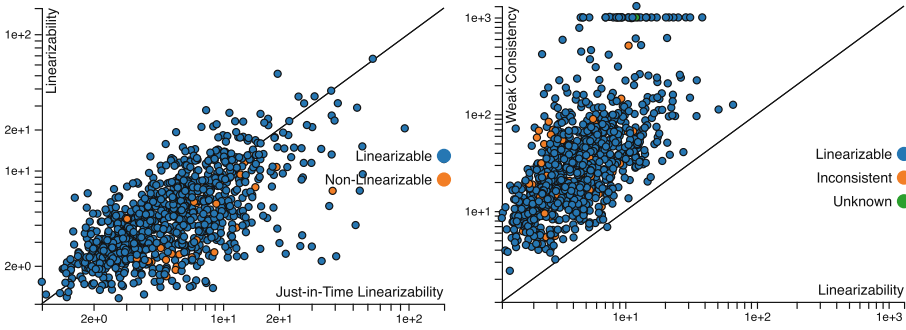


Fig. 7. Empirical comparison of (left) standard linearizability checking versus just-in-time linearizability checking on concurrent traces of Java data structures; and (right) weak-consistency checking versus standard linearizability checking. Each point reflects the time in milliseconds for checking a given trace.

timeouts are set to 1000 ms. We note that while accurate and *recording* of operation timings within an execution without interference is challenging, timing the *validation* of each recorded history, which we report here, is accomplished accurately, without interference, by computing the clock difference just before and after validation.

Our first measurements establish the baseline linearizability and weak-consistency checking algorithms. On the left side of Fig. 7 we consider the time required to check linearizability for each history by our own implementations of Wing and Gong’s standard enumerative approach [38], along with Lowe’s “just-in-time linearizability” algorithm [29] – see Sect. 4. We resolve the non-determinism in these algorithms (e.g., in choosing which pending operation to attempt linearizing first) arbitrarily (e.g., first called), finding no clear winner: each algorithm performs better on some histories. Since these subtleties are outside the scope of our work, we avoid further investigation and choose Wing and Gong’s algorithm as our baseline linearizability-checking algorithm.

Our second measurement exposes the overhead of enumerating visibility relations for checking weak consistency. On the right side of Fig. 7 we consider the time required to check weak consistency of a given history versus the time required to check its linearizability.⁶ We observe an overhead of approximately $10\times$ due to visibility enumeration and validation. Our naïve implementation enumerates candidate visibilities in size-decreasing order since we expect visibility-loss to be the exception rather than the rule; for instance, atomic operations observe all linearized-before operations. We omit the analogous comparison between weak-consistency checking and just-in-time linearizability checking to avoid redundancy, since the just-in-time optimization is a seemingly-insignificant factor in our experiments: the results are nearly identical.

⁶ Due to a benign error in the decoding of results of stress testing, we observe one single point on which the two algorithms conflict – labeled by “Unknown.”.

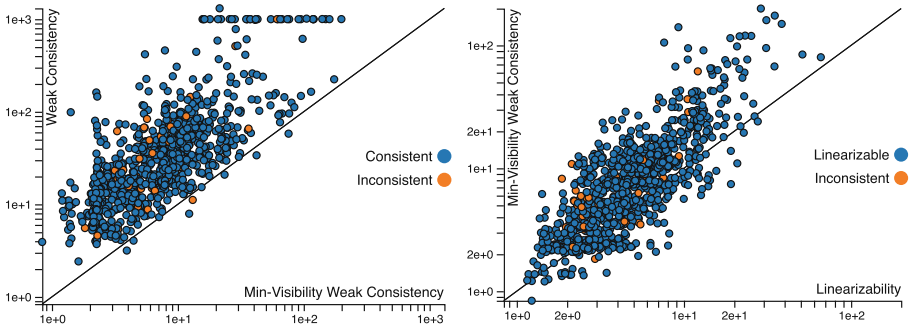


Fig. 8. Empirical comparison of (left) standard weak-consistency checking versus minimal-visibility weak-consistency checking on concurrent traces of Java data structures; and (right) the latter versus standard linearizability checking. Each point reflects the time in milliseconds for checking a given trace.

Our third measurement demonstrates the impact of our minimal-visibility consistency checking optimization. On the left side of Fig. 8 we consider the time required to check weak consistency without and with our optimization. The difference is dramatic, with our optimized algorithm consistently outperforming, sometimes up to multiple orders of magnitude: the leftmost 1000 ms timeout of the naïve algorithm is matched by a roughly 18 ms positive identification. Finally, our fourth measurement, on the right side of Fig. 8, demonstrates that the overhead of our minimal-visibility checking algorithm over linearizability checking is quite modest: we observe roughly a $2\times$ overhead, compared with the observed $10\times$ overhead without optimization.

While our experiments clearly demonstrate the efficacy of our minimal-visibility consistency checking algorithm, we will continue to evaluate this optimization across a wide range of concurrent objects, consistency models, and client programs, e.g., including many more concurrent threads. While we do expect the performance of linearizability- and weak-consistency checking to vary with thread count, we expect the performance gains of minimal-visibility consistency checking to continue to hold.

6 Related Work

Herlihy and Wing [22] described linearizability, which is the standard consistency criterion for shared-memory concurrent objects. Motivated by replication-based distributed systems, Burckhardt et al. [9, 11] describe a more general axiomatic framework for specifying weaker consistencies like eventual consistency [36] and causal consistency [2]. Our weak consistency checking algorithm applies to consistency models described in this framework.

While several static techniques have been developed to prove linearizability [1, 4, 6, 12, 13, 21, 22, 24, 26, 27, 30–34, 37, 39], few have addressed dynamic techniques such as testing and runtime verification. The works in [29, 38] describe

monitors for checking linearizability that construct linearizations of a given history incrementally, in an online fashion. Line-Up [10] performs systematic concurrency testing via schedule enumeration, and offline linearizability checking via linearization enumeration. Our weak consistency checking algorithm combines these approaches with an efficient enumeration of visibility relations. The works in [15, 16] propose a symbolic enumeration of linearizations based on a SAT solver. Although more efficient in practice, this approach applies only to certain ADTs. In this work, we propose a generic approach that assumes no constraints on the sequential semantics of the concurrent objects.

Bouajjani et al. [7] consider the problem of verifying causal consistency. They propose an algorithm for checking whether a given execution satisfies causal consistency, but only for the key-value map ADT with simple `put` and `get` operations. Our work proposes a generic algorithm that can deal with various weak consistency criteria and ADTs.

From the complexity standpoint, Gibbons and Korach [18] showed that monitoring even the single-value register type for linearizability is NP-hard. Alur et al. [3] showed that checking linearizability of all executions of a given implementation is in EXPSpace when the number of concurrent operations is bounded, and then Hamza [20] established EXPSpace-completeness. Bouajjani et al. [5] showed that the problem becomes undecidable once the number of concurrent operations is unbounded. Also, Bouajjani et al. [7, 8] investigate various ADTs for which the problems of checking eventual and causal consistency are decidable.

7 Conclusion

We have developed the first completely-automatic algorithm for checking weak consistency of arbitrary concurrent object implementations which avoids the naïve enumeration of all possible visibility relations. While methodologies for constructing reliable yet weakly-consistent implementations are relatively immature, we believe that such implementations will continue to be important for the development of distributed and multicore software systems. Likewise, automation for testing and verifying such implementations is, and will increasingly be, important. Besides improving state-of-the-art verification algorithms, our results represent an important step for future research which may find other ways to exploit the soundness of considering only minimal visibilities, on which our optimized algorithm relies.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_23
2. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. *Distrib. Comput.* **9**(1), 37–49 (1995). <https://doi.org/10.1007/BF01784241>

3. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. *Inf. Comput.* **160**(1–2), 167–188 (2000). <https://doi.org/10.1006/inco.1999.2847>
4. Amit, D., Rinetzk, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_49
5. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) *ESOP 2013*. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_17
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, 15–17 January 2015, Mumbai, India, pp. 651–662. ACM (2015). <https://doi.org/10.1145/2676726.2677002>
7. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, 18–20 January 2017, Paris, France, pp. 626–638. ACM (2017). <http://dl.acm.org/citation.cfm?id=3009888>
8. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Jagannathan, S., Sewell, P. (eds.) *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*, 20–21 January 2014, San Diego, CA, USA, pp. 285–296. ACM (2014). <https://doi.org/10.1145/2535838.2535877>
9. Burckhardt, S.: Principles of eventual consistency. *Found. Trends Program. Lang.* **1**(1–2), 1–150 (2014). <https://doi.org/10.1561/25000000011>
10. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, 5–10 June 2010, Toronto, Ontario, Canada, pp. 330–340. ACM (2010). <https://doi.org/10.1145/1806596.1806634>
11. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*, 20–21 January 2014, San Diego, CA, USA, pp. 271–284. ACM (2014). <https://doi.org/10.1145/2535838.2535848>
12. Dodds, M., Haas, A., Kirsch, C.M.: A scalable, correct time-stamped stack. In: Rajamani, S.K., Walker, D. (eds.) *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, 15–17 January 2015, Mumbai, India, pp. 233–246. ACM (2015). <https://doi.org/10.1145/2676726.2676963>
13. Drăgoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 174–190. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_11
14. Emmi, M., Enea, C.: Exposing non-atomic methods of concurrent objects. *CoRR abs/1706.09305* (2017). <http://arxiv.org/abs/1706.09305>

15. Emmi, M., Enea, C.: Sound, complete, and tractable linearizability monitoring for concurrent collections. *PACMPL* **2**(POPL), 25:1–25:27 (2018). <https://doi.org/10.1145/3158113>
16. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 15–17 June 2015, Portland, OR, USA, pp. 260–269. ACM (2015). <https://doi.org/10.1145/2737924.2737983>
17. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. *J. ACM* **32**(2), 374–382 (1985). <https://doi.org/10.1145/3149.214121>
18. Gibbons, P.B., Korach, E.: Testing shared memories. *SIAM J. Comput.* **26**(4), 1208–1244 (1997). <https://doi.org/10.1137/S0097539794279614>
19. Gilbert, S., Lynch, N.A.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002). <https://doi.org/10.1145/564585.564601>
20. Hamza, J.: On the complexity of linearizability. In: Bouajjani, A., Fauconnier, H. (eds.) *NETYS 2015. LNCS*, vol. 9466, pp. 308–321. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26850-7_21
21. Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: D’Argenio, P.R., Melgratti, H. (eds.) *CONCUR 2013. LNCS*, vol. 8052, pp. 242–256. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_18
22. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990). <https://doi.org/10.1145/78969.78972>
23. Kawell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I.: Replicated document management in a group communication system. In: *Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work*, p. 395. CSCW 1988. ACM, New York (1988). <https://doi.org/10.1145/62266.1024798>
24. Khyzha, A., Gotsman, A., Parkinson, M.: A generic logic for proving linearizability. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) *FM 2016. LNCS*, vol. 9995, pp. 426–443. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_26
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978). <https://doi.org/10.1145/359545.359563>
26. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: Boehm, H., Flanagan, C. (eds.) *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*, 16–19 June 2013, Seattle, WA, USA, pp. 459–470. ACM (2013). <https://doi.org/10.1145/2462156.2462189>
27. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009. LNCS*, vol. 5850, pp. 321–337. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_21
28. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Wobber, T., Druschel, P. (eds.) *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011*, 23–26 October 2011, Cascais, Portugal, pp. 401–416. ACM (2011). <https://doi.org/10.1145/2043556.2043593>
29. Lowe, G.: Testing for linearizability. *Concurr. Comput.: Pract. Exp.* **29**(4) (2017). <https://doi.org/10.1002/cpe.3928>

30. O'Hearn, P.W., Rinetzký, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: Richa, A.W., Guerraoui, R. (eds.) *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010*, 25–28 July 2010, Zurich, Switzerland, pp. 85–94. ACM (2010). <https://doi.org/10.1145/1835698.1835722>
31. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) *CAV 2012*. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_21
32. Sergey, I., Nanovski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Grove, D., Blackburn, S. (eds.) *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 15–17 June 2015, Portland, OR, USA, pp. 77–87. ACM (2015). <https://doi.org/10.1145/2737924.2737964>
33. Sergey, I., Nanovski, A., Banerjee, A.: Specifying and verifying concurrent algorithms with histories and subjectivity. In: Vitek, J. (ed.) *ESOP 2015*. LNCS, vol. 9032, pp. 333–358. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_14
34. Shacham, O., Bronson, N.G., Aiken, A., Sagiv, M., Vechev, M.T., Yahav, E.: Testing atomicity of composed concurrent operations. In: Lopes, C.V., Fisher, K. (eds.) *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, part of SPLASH 2011, 22–27 October 2011, Portland, OR, USA, pp. 51–64. ACM (2011). <https://doi.org/10.1145/2048066.2048073>
35. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems, PDIS 1994*, pp. 140–150. IEEE Computer Society Press, Los Alamitos (1994). <http://dl.acm.org/citation.cfm?id=381992.383631>
36. Terry, D.B., Theimer, M., Petersen, K., Demers, A.J., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: Jones, M.B. (ed.) *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995*, 3–6 December 1995, Copper Mountain Resort, Colorado, USA, pp. 172–183. ACM (1995). <https://doi.org/10.1145/224056.224070>
37. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) *CAV 2010*. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40
38. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.* **17**(1–2), 164–182 (1993). <https://doi.org/10.1006/jpdc.1993.1015>
39. Zhang, S.J.: Scalable automatic linearizability checking. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, 21–28 May 2011, Waikiki, Honolulu, HI, USA, pp. 1185–1187. ACM (2011). <https://doi.org/10.1145/1985793.1986037>





Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Monitoring CTMCs by Multi-clock Timed Automata

Yijun Feng¹, Joost-Pieter Katoen² , Haokun Li¹ , Bican Xia¹ ,
and Naijun Zhan^{3,4} 

¹ LMAM and School of Mathematical Sciences, Peking University, Beijing, China
ker@protonmail.ch, xbc@math.pku.edu.cn

² RWTH Aachen University, Aachen, Germany
katoen@cs.rwth-aachen.de

³ State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
znj@ios.ac.cn

⁴ University of Chinese Academy of Sciences, Beijing, China

Abstract. This paper presents a numerical algorithm to verify continuous-time Markov chains (CTMCs) against multi-clock deterministic timed automata (DTA). These DTA allow for specifying properties that cannot be expressed in CSL, the logic for CTMCs used by state-of-the-art probabilistic model checkers. The core problem is to compute the probability of timed runs by the CTMC \mathcal{C} that are accepted by the DTA \mathcal{A} . These likelihoods equal reachability probabilities in an embedded piecewise deterministic Markov process (EPDP) obtained as product of \mathcal{C} and \mathcal{A} 's region automaton. This paper provides a numerical algorithm to efficiently solve the PDEs describing these reachability probabilities. The key insight is to solve an ordinary differential equation (ODE) that exploits the specific characteristics of the product EPDP. We provide the numerical precision of our algorithm and present experimental results with a prototypical implementation.

1 Introduction

Continuous-time Markov chains (CTMCs) [17] are ubiquitous. They are used to model safety-critical systems like communicating networks and power management systems, are key to performance and dependability analysis, and naturally describe chemical reaction networks. The algorithmic verification of CTMCs has received quite some attention. Aziz *et al.* [3] proved that verifying CTMCs against CSL (Continuous Stochastic Logic) is decidable. CSL is a probabilistic and timed branching-time logic that allows for expressing properties like “is the probability of a given chemical reaction within 50 time units at least 10^{-3} ?”. Baier *et al.* [5] gave efficient numerical algorithms for CSL model checking that nowadays provide the basis of CTMC model checking in PRISM [23], MRMC [22] and Storm [15], as well as GreatSPN [2]. Extensions of CSL to cascaded timed-until operators [27], conditional probabilities [19], and (simple) timed regular expressions [4] have been considered.

This paper considers the verification of CTMCs against *linear-time* real-time properties. These include relevant properties in the design of a gas burner [28], like “the probability that the duration of leaking is more than one twentieth over an interval with a length more than 20s is less than 10^{-6} ”. Such real-time properties can be conveniently expressed by deterministic timed automata (DTA) [1]. The core problem in the verification of CTMC \mathcal{C} against DTA \mathcal{A} is to compute the probability of \mathcal{C} ’s timed runs that are accepted by \mathcal{A} , i.e. $\Pr(\mathcal{C} \models \mathcal{A})$. Chen *et al.* [10, 11] showed that this quantity equals the reachability probability in a piecewise deterministic Markov process (PDP) [14]. This PDP is obtained by taking the product of CTMC \mathcal{C} and the region automaton of \mathcal{A} . Computing reachability probabilities in PDPs is a challenge.

Practical implementations of verifying CTMCs against DTA specifications are rare. Barbot *et al.* [7] showed that for *single-clock* DTA, the PDP is in fact a Markov regenerative process. (This observation is also at the heart of model-checking CSL^{TA} [16].) This implies that for single-clock DTA, off-the-shelf CSL model-checking algorithms can be employed resulting in an efficient procedure [7]. Mikeev *et al.* [24] generalised these ideas to infinite-state CTMCs obtained from stoichiometric equations, whereas Chen *et al.* [12] showed the theory to generalize verifying single-clock DTA to continuous-time Markov decision processes.

Multi-clock DTA are however much harder to handle. The characterisation of PDP reachability probabilities as the unique solution of a set of partial differential equations (PDEs) [10, 11] does not give insight into an efficient computational procedure. With the notable exception of [25], verifying PDPs has not been considered. Fu [18] provided an algorithm to approximate the probabilities using finite difference methods and gave an error bound. This method hampers scalability and therefore was never implemented. The same holds for model-checking using other linear-time real-time formalisms such as MTL and timed automata [9], linear duration invariants [8], and probabilistic duration calculus [13]. All these multi-clock approaches suffer from scalability issues due to the low efficiency of solving PDEs and/or integral equations on which they heavily depend.

This paper presents a numerical technique to approximate the reachability probability in the product PDP. The DTA \mathcal{A} is approximated by DTA $\mathcal{A}[t_f]$ which extends \mathcal{A} with an additional clock that is never reset and that needs to be at most t_f when accepting. By increasing the time-bound t_f , DTA $\mathcal{A}[t_f]$ approximates \mathcal{A} arbitrarily closely. We show that the set of PDPs characterizing the reachability probability in the embedded PDP of \mathcal{C} and $\mathcal{A}[t_f]$ can be reduced to solving an ordinary differential equation (ODE). The specific characteristics of the product EPDP, in particular the fact that all clocks run at the same pace, are key to obtain these ODEs. Our numerical algorithm to solve the ODEs is based on computing the approximations in a backward manner using t_f and the sum of all clocks. The complexity of the resulting procedure is linear in the EPDP size, and exponential in $\lceil \frac{t_f}{\delta} \rceil$ where δ is the discretization step size. We show the approximations converges to the real solution of the ODEs at a linear

speed of δ . Using a prototypical tool implementation we present some results on a number of case studies such as robot navigation with varying number of clocks in their specification. The experimental results show promising results for checking CTMCs against multi-clock DTA.

Organization of the Paper. Section 2 introduces basic notions including CTMCs, DTA, and PDPs. Section 3 presents the product of a CTMC and the region graph of a DTA and shows this is an embedded PDP. Section 4 derives the PDE (fixing some flaw in [10]), the reduction to the set of ODEs and presents the numerical algorithm to solve these ODEs. Section 5 presents the experimental results and Sect. 6 concludes.

2 Preliminaries

In this section, we introduce some basic notions which will be used later.

A probability space is denoted by a triple $(\Omega, \mathcal{F}, Pr)$, where Ω is a set of samples, \mathcal{F} is a σ -algebra over Ω , and $Pr : \mathcal{F} \rightarrow [0, 1]$ is a probability measure on \mathcal{F} with $Pr(\Omega) = 1$. Let $\mathbb{P}_r(\Omega)$ denote the set of all probability measures over Ω . For a random variable X on the probability space, its expectation is denoted by $\mathbb{E}(X)$.

2.1 Continuous-Time Markov Chain (CTMC)

Definition 1 (CTMC). A CTMC is a tuple $\mathcal{C} = (S, \mathbf{P}, \alpha, AP, L, E)$, where

- S is a finite set of states;
- $\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability function, which is identified with the matrix $\mathbf{P} \in [0, 1]^{|S| \times |S|}$ such that $\sum_{t \in S} \mathbf{P}(s, t) = 1$, for all $s \in S$;
- $\alpha \in \mathbb{P}_r(S)$ is the initial distribution;
- AP is a finite set of atomic propositions;
- $L : S \rightarrow 2^{AP}$ is a labeling function; and
- $E : S \rightarrow \mathbb{R}_{>0}$ is the exit rate function.

We denote by $s \xrightarrow{t} s'$ a transition from state s to state s' after residing in state s for t time units. The probability of the occurrence of this transition within t time units is $\mathbf{P}(s, s') \int_0^t E(s) \exp^{-E(s)x} dx$, where $\int_0^t E(s) \exp^{-E(s)x} dx$ stands for the probability to leave state s in t time units, and $\mathbf{P}(s, s')$ for the probability to select the transition to s' from all transitions outgoing from s . A state s is called *absorbing* if $\mathbf{P}(s, s) = 1$. Given a CTMC \mathcal{C} , removing the exit rate function E results in a discrete-time Markov chain (DMTC), which is called *embedded* DTMC of \mathcal{C} . A CTMC \mathcal{C} is called *irreducible* if there exists a unique stationary distribution α , such that $\alpha(s) > 0$ for all $s \in S$, and *weakly irreducible* if $\alpha(s)$ may be zero for some $s \in S$.

Definition 2 (CTMC Path). Let \mathcal{C} be a CTMC, a path ρ of \mathcal{C} starting from s_0 with length n is a sequence $\rho = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} s_n \in S \times (\mathbb{R}_{>0} \times S)^n$. The

set of paths in \mathcal{C} with length n is denoted by $\text{Path}_n^{\mathcal{C}}$; the set of all finite paths of \mathcal{C} is $\text{Path}_{\text{fin}}^{\mathcal{C}} = \cup_n \text{Path}_n^{\mathcal{C}}$ and the set of infinite paths of \mathcal{C} is $\text{Path}_{\text{inf}}^{\mathcal{C}} = (S \times \mathbb{R}_{>0})^\omega$. We use $\text{Path}^{\mathcal{C}} = \text{Path}_{\text{fin}}^{\mathcal{C}} \cup \text{Path}_{\text{inf}}^{\mathcal{C}}$ to denote all paths in \mathcal{C} . As a convention, ε stands for the empty path.

Note that we assume the time to exit a state is strictly greater than 0. For an infinite path ρ , we use $\text{Pref}(\rho)$ to denote the set of its finite prefixes. For a (finite or infinite) path ρ with prefix $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots$, the trace of the path is the sequence of states $\text{trace}(\rho) = s_0 s_1 \dots$. Let $\rho(n) = s_n$ be the n -th state in the path and $\rho[n] = t_n$ be the corresponding exit time for s_n . For a finite path $\rho = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} s_n$, we use $T(\rho) = \sum_{i=0}^{n-1} t_i$ to denote the total time spent on this path if $n \geq 1$, otherwise $T(\rho) = 0$. For a time $t \leq T(\rho)$, $\rho(0 \dots t)$ denotes the prefix of ρ within t time units, i.e., $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{m-1}} s_m$ if there exists some $m \leq n$ with $\sum_{i=0}^{m-1} \rho[m] \leq t \wedge \sum_{i=0}^m \rho[m] > t$, otherwise ε .

A basic cylinder set $C(s_0, I_0, \dots, I_{n-1}, s_n)$ consists of all paths $\rho \in \text{Path}^{\mathcal{C}}$ such that $\rho(i) = s_i$ for $0 \leq i \leq n$, and $\rho[i] \in I_i$ for $0 \leq i < n$. Then the σ -algebra $\mathcal{F}_{s_0}(\mathcal{C})$ associated with CTMC \mathcal{C} and initial state s_0 is the smallest σ -algebra that contains all cylinder sets $C(s_0, I_0, \dots, I_{n-1}, s_n)$ with $\alpha(s_0) > 0$, and $\mathbf{P}(s_i, s_{i+1}) > 0$, for $1 \leq i \leq n$, and I_0, \dots, I_{n-1} are non-empty intervals in $\mathbb{R}_{\geq 0}$. There is a unique probability measure $\text{Pr}^{\mathcal{C}}$ on the σ -algebra $\mathcal{F}_{s_0}(\mathcal{C})$, by which the probability for a cylinder set is given by

$$\text{Pr}^{\mathcal{C}}(C(s_0, I_0, \dots, I_n, s_n)) = \alpha(s_0) \cdot \prod_{i=1}^n \int_{I_i} E(s_{i-1}) \exp^{-E(s_{i-1})x} dx \cdot \mathbf{P}(s_{i-1}, s_i)$$

Example 1. An example of CTMC is shown in Fig. 1, with $AP = \{a, b, c\}$ and initial state s_0 . The exit rate r_i , $i = 0, 1, 2, 3$ and transition probability are shown in the figure.

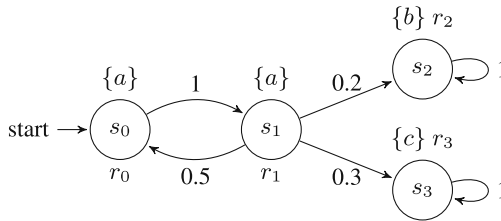


Fig. 1. An example of CTMC

2.2 Deterministic Timed Automaton (DTA)

A timed automaton is a finite state graph equipped with a finite set of non-negative real-valued clock variables, or clocks for short. Clocks can only be

reset to zero, or proceed with rate 1 as time progresses independently. Let $\mathcal{X} = \{x_1, \dots, x_n\}$ be a set of clocks. $\eta(x) : \mathcal{X} \rightarrow \mathbb{R}_{\geq 0}$ is a \mathcal{X} -valuation which records the amount of time since its last reset. Let $\text{Val}(\mathcal{A})$ be the set of all clock valuations of \mathcal{A} . For a subset $X \subseteq \mathcal{X}$, the reset of X , denoted as $\eta[X := 0]$, is the valuation η' such that $\eta'(x) = 0, \forall x \in X$, and $\eta'(x) = \eta(x)$, otherwise. For $d \in \mathbb{R}_{>0}$, $(\eta + d)(x) = \eta(x) + d$ for any clock $x \in \mathcal{X}$.

A clock constraint over \mathcal{X} is a formula with the following form

$$g := x < c \mid x \leq c \mid x > c \mid x \geq c \mid x - y \geq c \mid g \wedge g,$$

where x, y are clocks, $c \in \mathbb{N}$. Let $\text{Con}(\mathcal{X})$ denote the set of clock constraints over \mathcal{X} . A valuation η satisfies a guard g , denoted as $\eta \models g$, iff $\eta(x) \bowtie c$ when g is $x \bowtie c$, where $\bowtie \in \{<, \leq, >, \geq\}$; and $\eta \models g_1$ and $\eta \models g_2$ iff $g = g_1 \wedge g_2$.

Definition 3 (DTA). A DTA is a tuple $\mathcal{A} = (\Sigma, \mathcal{X}, Q, q_0, Q_F, \hookrightarrow)$, where

- Σ is a finite set of actions;
- \mathcal{X} is a finite set of clocks;
- Q is a finite set of locations;
- $q_0 \in Q$ is the initial location;
- $Q_F \subseteq Q$ is the set of accepting locations;
- $\hookrightarrow \in (Q \setminus Q_F) \times \Sigma \times \text{Con}(\mathcal{X}) \times 2^{\mathcal{X}} \times Q$ is the transition relation, satisfying if $q \xrightarrow{a, g, X} q'$ and $q \xrightarrow{a, g', X'} q''$ with $q' \neq q''$ then $g \cap g' = \emptyset$.

Each transition relation, or edge, $q \hookrightarrow q'$ in \mathcal{A} is endowed with (a, g, X) , where $a \in \Sigma$ is an action, $g \in \text{Con}(\mathcal{X})$ is the guard of the transition, and $X \subseteq \mathcal{X}$ is a set of clocks, which should be reset to 0 after the transition. An intuitive interpretation of the transition is that \mathcal{A} can move from q to q' by taking action a and resetting all clocks in X to be 0 only if g is satisfied. There are no outgoing transitions from any accepting location in Q_F .

A finite timed path of \mathcal{A} is of the form $\theta = q_0 \xrightarrow{a_0, t_0} q_1 \xrightarrow{a_1, t_1} \dots \xrightarrow{a_{n-1}, t_{n-1}} q_n$, where $t_i \geq 0$, for $i = 0, \dots, n-1$. Moreover, there exists a sequence of transitions $q_j \xrightarrow{a_j, g_j, X_j} q_{j+1}$, for $0 \leq j \leq n-1$, such that $\eta_0 = \mathbf{0}$, $\eta_j + t_j \models g_j$ and $\eta_{j+1} = \eta_j[X_j := 0]$, where η_k denotes the clock valuation when entering q_k . θ is said to be *accepted by \mathcal{A}* if there exists a state $q_i \in Q_F$ for some $0 \leq i \leq n$. As normal, it is assumed all DTA are non-Zeno [6], that is any circular transition sequence takes nonzero dwelling time.

A region is a set of valuations, usually represented by a set of clock constraints. Let $\text{Reg}(\mathcal{X})$ be the set of regions over \mathcal{X} . Given $\Theta, \Theta' \in \text{Reg}(\mathcal{X})$, Θ' is called a *successor* of Θ if for all $\eta \models \Theta$, there exists $t > 0$ such that $\eta + t \models \Theta'$ and $\forall t' < t, \eta + t' \models \Theta \vee \Theta'$. A region Θ satisfies a guard g , denoted as $\Theta \models g$, iff $\forall \eta \models \Theta$ implies $\eta \models g$. The reset operation on a region Θ is defined as $\Theta[X := 0] = \{\eta[X := 0] \mid \eta \models \Theta\}$. Then the region graph, viewed as a quotient transition system related to clock equivalence [6] can be defined as follows:

Definition 4 (Region Graph). The region graph for DTA $\mathcal{A} = (\Sigma, \mathcal{X}, Q, q_0, Q_F, \hookrightarrow)$ is a tuple $\mathcal{G}(\mathcal{A}) = (\Sigma, \mathcal{X}, \bar{Q}, \bar{q}_0, \bar{Q}_F, \mapsto)$, where

- $\overline{Q} = Q \times \text{Reg}(\mathcal{X})$ is the set of states;
- $\overline{q_0} = (q_0, \mathbf{0}) \in \overline{Q}$ is the initial state;
- $\overline{Q_F} \subseteq Q_F \times \text{Reg}(\mathcal{X})$ is the set of final states;
- $\mapsto \subseteq \overline{Q} \times ((\Sigma \times 2^{\mathcal{X}}) \cup \{\lambda\}) \times \overline{Q}$ is the transition relation satisfying
 - $(q, \Theta) \xrightarrow{\lambda} (q, \Theta')$ if Θ' is a successor of Θ ;
 - $(q, \Theta) \xrightarrow{a, X} (q', \Theta'')$ if there exists $g \in \text{Con}(\mathcal{X})$ and transition $q \xrightarrow{a, g, X} q'$ such that $\Theta \models g$ and $\Theta'' = \Theta[X := 0]$.

Example 2 (Adapted from [10]). Figure 2 presents an example of DTA and Fig. 3 gives its region graph, in which double circle and double rectangle stand for final states, respectively.

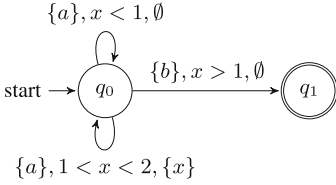


Fig. 2. A DTA \mathcal{A}

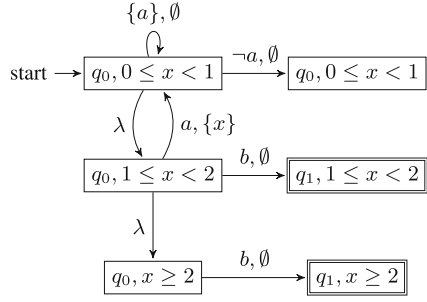


Fig. 3. The region graph of \mathcal{A}

2.3 Piecewise-Deterministic Markov Process (PDP)

Piecewise-deterministic Markov Processes (PDPs for short) [14] cover a wide range of stochastic models in which the randomness appears as discrete events at fixed or random times, whose evolution is deterministically governed by an ODE system between these times. A PDP consists of a mixture of deterministic motion and random jumps between a finite set of locations. During staying in a location, a PDP evolves deterministically following a flow function, which is a solution to an ODE system. A PDP can jump between locations either randomly, in which case the residence time of a location is governed by an exponential distribution, or when the location invariant is violated. The successor state of the jump follows a probability measure depending on the current state. A PDP is right-continuous and has the strong Markov property [14].

Definition 5 (PDP [14]). A PDP is a tuple $\mathcal{Q} = (Z, \mathcal{X}, \text{Inv}, \phi, \Lambda, \mu)$ with

- Z is a finite set of locations;
- \mathcal{X} is a finite set of variables;
- $\text{Inv} : Z \rightarrow 2^{\mathbb{R}^{|\mathcal{X}|}}$ is an invariant function;

- $\phi : Z \times \mathbb{R}^{|\mathcal{X}|} \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{|\mathcal{X}|}$, is a flow function, which is a solution of a system of ODEs with Lipschitz continuous vector fields;
- $\Lambda : \mathbb{S} \rightarrow \mathbb{R}_{>0}$ is an exit rate function;
- $\bar{\mathbb{S}} \rightarrow \mathbb{P}_r(\mathbb{S})$, is the transition probability function, where $\mathbb{S} = \{\xi := (z, \eta) \mid z \in Z, \eta \models \text{Inv}(z)\}$ is the state space for \mathcal{Q} , $\bar{\mathbb{S}}$ is the closure of \mathbb{S} , $\mathbb{S}^o = \{(z, \eta) \mid z \in Z, \eta \models \text{Inv}(z)^o\}$ is the interior of \mathbb{S} , in which $\text{Inv}(z)^o$ stands for the interior of $\text{Inv}(z)$, and $\partial\mathbb{S} = \bigcup_{z \in Z} \{z\} \times \partial\text{Inv}(z)$ is the boundary of \mathbb{S} , in which $\partial\text{Inv}(z) = \text{Inv}(z) \setminus \text{Inv}^o$ and $\text{Inv}(z)$ is the closure of $\text{Inv}(z)$.

For any $\xi = (z, \eta) \in \mathbb{S}$, there is an $\delta(\xi) > 0$ such that $\Lambda(z, \phi(z, \eta, t))$ is integrable on $[0, \delta(\xi))$. $\mu(\xi)(A)$ is measurable for any $A \in \mathcal{F}(\mathbb{S})$, where $\mathcal{F}(\mathbb{S})$ is the smallest σ -algebra generated by $\{\bigcup_{z \in Z} z \times A_z \mid A_z \in \mathcal{F}(\text{Inv}(z))\}$ and $\mu(\xi)(\{\xi\}) = 0$.

There are two ways to take transitions between locations in PDP \mathcal{Q} . A PDP \mathcal{Q} is allowed to stay in a current location z only if $\text{Inv}(z)$ is satisfied. During its residence, the valuation η evolves time-dependently according to the flow function. Let $\xi \oplus t = (z, \phi(z, \eta, t))$ be the successor state of $\xi = (z, \eta)$ after residing t time units in z . Thus, \mathcal{Q} is piecewise-deterministic since its behavior is determined by the flow function ϕ in each location. In a state $\xi = (z, \eta)$ with $\eta \models \text{Inv}(z)^o$, the PDP \mathcal{Q} can either evolve to a state $\xi' = \xi \oplus t$ by delaying t time units, or take a Markovian jump to $\xi'' = (z'', \eta'') \in \mathbb{S}$ with probability $\mu(\xi)(\{\xi''\})$. When $\eta \models \partial\text{Inv}(z)$, \mathcal{Q} is forced to take a boundary jump to $\xi'' = (z'', \eta'') \in \mathbb{S}$ with probability $\mu(\xi)(\{\xi''\})$.

3 Reduction to the Reachability Probability of EPDP

As proved in [10], model-checking of a given CTMC \mathcal{C} against a linear real-time property expressed by a DTA \mathcal{A} , i.e., determining $\Pr(\mathcal{C} \models \mathcal{A})$, can be reduced to computing the reachability probability of the product of \mathcal{C} and $\mathcal{G}(\mathcal{A})$. This can be further reduced to computing the reachability probability of the embedded PDP (EPDP) of the product. But how to efficiently compute the reachability probability of the EPDP still remains challenging, as existing approaches [7, 10, 16] can only handle DTA with one clock. We will attack this challenge in this paper. For self-containedness, we reformulate the reduction reported in [10] in this section.

A path $\rho = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots$ of CTMC \mathcal{C} is accepted by DTA \mathcal{A} if $\hat{\rho} = q_0 \xrightarrow{L(s_0), t_0} q_1 \xrightarrow{L(s_1), t_1} \dots \xrightarrow{L(s_{n-1}), t_{n-1}} q_n$ induced by some ρ 's prefix is an accepting path of \mathcal{A} . Then $\Pr(\mathcal{C} \models \mathcal{A}) = \Pr\{\rho \in \text{Path}^{\mathcal{C}} \mid \rho \text{ is accepted by } \mathcal{A}\}$.

Definition 6 (Product Region Graph [7]). *The product of CTMC $\mathcal{C} = (S, \mathbf{P}, \alpha, AP, L, E)$ and the region graph of DTA $\mathcal{G}(\mathcal{A}) = (\Sigma, \mathcal{X}, \bar{Q}, \bar{q}_0, \bar{Q}_F, \mapsto)$, denoted by $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, is a tuple $(\mathcal{X}, V, \alpha', V_F, \rightarrow, \Lambda)$, where*

- $V = S \times \bar{Q}$ is the state space;
- $\alpha'(s, \bar{q}_0) = \alpha(s)$ is the initial distribution;
- $V_F = S \times \bar{Q}_F$ is the set of accepting states;
- $\rightarrow \subseteq V \times (([0, 1] \times 2^{\mathcal{X}}) \cup \{\lambda\}) \times V$ is the smallest relation satisfying

- $(s, \bar{q}) \xrightarrow{\lambda} (s, \bar{q}')$ (called delay transition), if $\bar{q} \xrightarrow{\lambda} \bar{q}'$;
 - $(s, \bar{q}) \xrightarrow{p, X} (s'', \bar{q}'')$ (called Markovian transition), if $\mathbf{P}(s, s'') = p, p > 0$ and $\bar{q} \xrightarrow{L(s), X} \bar{q}''$;
- $\Lambda : V \rightarrow \mathbb{R}_{>0}$ is the exit rate function, where

$$\Lambda(s, \bar{q}) = \begin{cases} E(s) & \text{if there exists a Markovian transition from } (s, \bar{q}) \\ 0 & \text{otherwise} \end{cases}$$

Remark 1. Note that the definition of region graph here is slightly different from the usual one in the sense that Markovian transitions starting from a boundary do not contribute to the reachability probability. Therefore we can merge the boundary into its unique delay successor.

Example 3 (Adapted from [10]). Figure 4 shows the product region graph of CTMC \mathcal{C} in Example 1 and DTA \mathcal{A} in Example 2. The graph can be split into three subgraphs in a column-wise manner, where all transitions within a subgraph are probabilistic, all transitions evolve to the next subgraph are delay transitions, and transitions with reset lead to a state in the first subgraph. For conciseness, the location v_9 stands for all nodes that may be reached by a Markovian transition yet cannot reach an accepting node.

Proposition 1 ([10]). For CTMC \mathcal{C} and DTA \mathcal{A} , $\Pr(\mathcal{C} \models \mathcal{A})$ is measurable and

$$\Pr(\mathcal{C} \models \mathcal{A}) = \Pr^{\mathcal{C} \otimes \mathcal{G}(\mathcal{A})} \{ \text{Path}^{\mathcal{C} \otimes \mathcal{G}(\mathcal{A})}(\diamond \overline{Q_F}) \}.$$

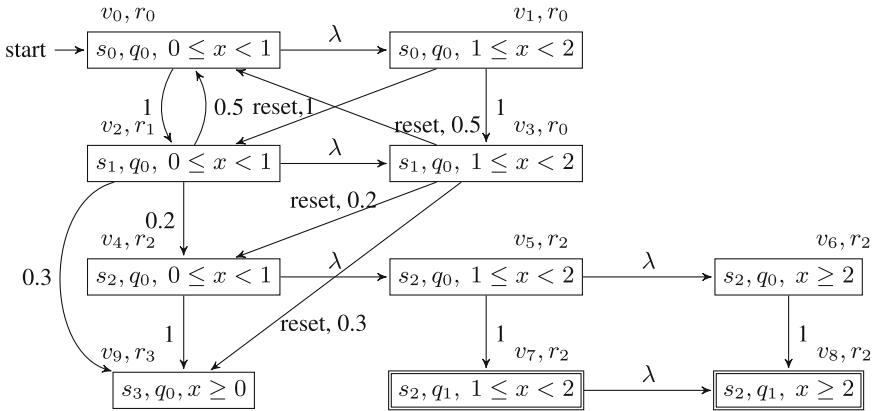


Fig. 4. Product region graph $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$ of CTMC \mathcal{C} in Example 1 and DTA \mathcal{A} in Example 2

When treated as a stochastic process, $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$ can be interpreted as a PDP. In this way, computing the reachability probability of Q_F in $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$ can be reduced to computing the time-unbounded reachability probability in the EPDP of $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$.

Definition 7 (EPDP, [7]). Given $\mathcal{C} \otimes \mathcal{G}(\mathcal{A}) = (\mathcal{X}, V, \alpha', V_F, \rightarrow, \Lambda)$, the EPDP $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}$ is a tuple $(\mathcal{X}, V, \text{Inv}, \phi, \Lambda, \mu)$ where for any $v = (s, (q, \Theta)) \in V$

- $\text{Inv}(v) = \Theta$, $\mathbb{S} = \{(v, \boldsymbol{\eta}) \mid v \in V, \boldsymbol{\eta} \models \text{Inv}(v)\}$ is the state space;
- $\phi(v, \boldsymbol{\eta}, t) = \boldsymbol{\eta} + t$ for $\boldsymbol{\eta} \models \text{Inv}(v)$;
- $\Lambda(v, \boldsymbol{\eta}) = \Lambda(v)$ is the exit rate of $(v, \boldsymbol{\eta})$;
- Boundary jump: for each delay transition $v \xrightarrow{\lambda} v'$ in $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, $\mu(\xi, \{\xi'\}) = 1$ whenever $\xi = (v, \boldsymbol{\eta})$, $\xi' = (v', \boldsymbol{\eta})$ and $\boldsymbol{\eta} \models \partial \text{Inv}(v)$;
- Markovian transition jump: for each Markovian transition $v \xrightarrow{p, X} v''$ in $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, $\mu(\xi, \{\xi''\}) = p$ whenever $\xi = (v, \boldsymbol{\eta})$, $\boldsymbol{\eta} \models \text{Inv}(v)$ and $\xi'' = (v'', \boldsymbol{\eta}[X := 0])$.

The flow function here describes that all clocks increase with a uniform rate (i.e., $\dot{x}_1 = 1, \dots, \dot{x}_n = 1$, or simply $\dot{\mathcal{X}} = 1$) at all locations. The original reachability problem is then reduced to the reachability probability of the set $\{(v, \boldsymbol{\eta}) \mid v \in V_F, \boldsymbol{\eta} \models \text{Inv}(v)\}$, given the initial state $(v_0, \mathbf{0})$ and the EPDP $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}$. Let $Pr_v^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta})$ stand for the probability to reach the final states $(V_F \times *)$ from $(v, \boldsymbol{\eta})$ in $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}$. Thus, $Pr_v^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta})$ can be computed recursively by

$$Pr_v^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta}) = \begin{cases} Pr_{v, \lambda}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta}) + \sum_{v \xrightarrow{p, X} v'} Pr_{v, v'}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta}) & \text{if } v \notin V_F \\ 1, & v \in V_F \wedge \boldsymbol{\eta} \models \text{Inv}(v) \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Let $t_z^*(v, \boldsymbol{\eta})$ denote the minimal time for $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}$ to reach $\partial \text{Inv}(v)$ from $(v, \boldsymbol{\eta})$. More precisely,

$$t_z^*(v, \boldsymbol{\eta}) = \inf\{t \mid \phi(v, \boldsymbol{\eta}, t) \models \text{Inv}(v)\}.$$

$Pr_{v, \lambda}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta})$ is the probability from $(v, \boldsymbol{\eta})$ with a delay and then a forced jump to $(v', \boldsymbol{\eta} + t_z^*(v, \boldsymbol{\eta}))$, onwards evolves to an accepting state, which can be recursively computed by

$$Pr_{v, \lambda}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta}) = \exp(-\Lambda(v)t_z^*(v, \boldsymbol{\eta})) \cdot Pr_{v'}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta} + t_z^*(v, \boldsymbol{\eta})).$$

$Pr_{v, v'}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta})$ is the probability that a Markovian transition $v \xrightarrow{p, X} v'$ happens within $t_z^*(v, \boldsymbol{\eta})$ time units, onwards involves to an accepted state, which can be recursively computed by

$$Pr_{v, v'}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta}) = \int_0^{t_z^*(v, \boldsymbol{\eta})} p \cdot \Lambda(v) \exp(-\Lambda(v)s) \cdot Pr_{v'}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\boldsymbol{\eta} + s[X := 0]) ds.$$

$Pr(\mathcal{C} \models \mathcal{A})$ is reduced to compute $Pr_{v_0}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\mathbf{0})$, equivalent to computing the least fixed point of the Eq. (1). That is,

Theorem 1. [10] For CTMC \mathcal{C} and DTA \mathcal{A} , $Pr(\mathcal{C} \models \mathcal{A}) = Pr^{\mathcal{C} \otimes \mathcal{A}}\{\text{Path}^{\mathcal{C} \otimes \mathcal{A}}(\diamond \overline{Q_F})\}$ is the least fixed point of (1).

Remark 2. Generally, it is difficult to solve a recursive equation like (1). As an alternative, we discuss the augmented EPDP of $\mathcal{Q}^{C \otimes \mathcal{A}}$ by replacing \mathcal{A} with a bounded DTA resulting from \mathcal{A} . As a consequence, using the extended generator of the augmented EPDP, we can induce a partial differential equation (PDE) whose solution is the reachability probability. We will elaborate the idea in the subsequent section.

4 Approximating the Reachability Probability of EPDP

In this section, we present a numerical method to approximate $Pr_{v_0}^{\mathcal{Q}^{C \otimes \mathcal{A}}}(\mathbf{0})$, as we discussed previously that exactly computing is impossible, at least too expensive, in general. We will first introduce the basic idea of our approach in detail, then discuss its time complexity and convergence property. A key point is that our approach exploits the observation that the flow function of $\mathcal{Q}^{C \otimes \mathcal{A}}$ is linear, only related to time t , and remains the same at all locations. This enables to reduce computing $Pr_{v_0}^{\mathcal{Q}^{C \otimes \mathcal{A}}}(\mathbf{0})$ to solving an ODE system.

4.1 Reduction to a PDE System

In this subsection, we first show that $Pr_{v_0}^{\mathcal{Q}^{C \otimes \mathcal{A}}}(\mathbf{0})$ can be approximated by that of the EPDP of \mathcal{C} and a bounded DTA derived from \mathcal{A} , i.e., the length of all its paths is bounded. Then show that the latter can be reduced to solving a PDE system.

Given a DTA \mathcal{A} , we construct a bounded DTA $\mathcal{A}[t_f]$ by introducing a new clock y , adding a timing constraint $y < t_f$ to the guard of each transition of \mathcal{A} ingoing to an accepting state in Q_F , and never resetting y , where $t_f \in \mathbb{N}$ is a parameter. So, the length of all accepting paths of $\mathcal{A}[t_f]$ is time-bounded by t_f . Obviously, $Path^C(\mathcal{A}[t_f])$ is a subset of $Path^C(\mathcal{A})$. As $Pr(\mathcal{C} \models \mathcal{A})$ is measurable and $\mathcal{Q}^{C \otimes \mathcal{A}}$ is Borel right continuous, we have the following proposition.

Proposition 2. *Given a CTMC \mathcal{C} , a DTA \mathcal{A} , and $t_f \in \mathbb{N}$,*

$$\lim_{t_f \rightarrow \infty} Pr(\mathcal{C} \models \mathcal{A}[t_f]) = Pr(\mathcal{C} \models \mathcal{A}). \quad (2)$$

Moreover, if \mathcal{C} is weakly irreducible or satisfies some conditions (please refer to Chap. 4 of [26] for details), then there exist positive constants $K, K_0 \in \mathbb{R}_{\geq 0}$ such that

$$Pr(\mathcal{C} \models \mathcal{A}) - Pr(\mathcal{C} \models \mathcal{A}[t_f]) \leq K \exp\{-K_0 t_f\}. \quad (3)$$

Remark 3. (2) was first observed in [7], thereof the authors pointed out the feasibility of using a bounded system to approximate the original unbounded system in order to simplify a verification obligation. (3) further indicates that such approximation is exponentially convergent w.r.t. $-t_f$ if the CTMC is weakly irreducible.

For a path starting in a state $(v, \boldsymbol{\eta})$ at time y , we use $Path_{(v, \boldsymbol{\eta})}^y[t]$ to denote the set of its locations at time t , and $\bar{h}_v(y, \boldsymbol{\eta}) = Pr(Path_{(v, \boldsymbol{\eta})}^y[t_f] \in V_F) = \mathbb{E}(\mathbf{1}_{Path_{(v, \boldsymbol{\eta})}^y[t_f] \in V_F})$ as the probability of a path reaching V_F within t_f time units, where $\mathbf{1}_{Path_{(v, \boldsymbol{\eta})}^y[t_f] \in V_F}$ is the indicator function of $Path_{(v, \boldsymbol{\eta})}^y[t_f] \in V_F$. Then, $\bar{h}_{v_0}(0, \mathbf{0}) = Pr(\mathcal{C} \models \mathcal{A}[t_f])$ is the probability to reach the set of accepting states from the initial state $(0, \mathbf{0})$, which satisfies the following system of PDEs.

Theorem 2. *Given a CTMC \mathcal{C} , a bounded DTA $\mathcal{A}[t_f]$, and the EPDP $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{G}(\mathcal{A}[t_f])} = (\mathcal{X}, V, Inv, \phi, \Lambda, \mu)$, $\bar{h}_{v_0}(0, \mathbf{0})$ is the unique solution of the following system of PDEs:*

$$\frac{\partial \bar{h}_v(y, \boldsymbol{\eta})}{\partial y} + \sum_{i=1}^{|\mathcal{X}|} \frac{\partial \bar{h}_v(y, \boldsymbol{\eta})}{\partial \boldsymbol{\eta}^{(i)}} + \Lambda(v) \cdot \sum_{v \xrightarrow{p, X} v'} p \cdot (\bar{h}_{v'}(y, \boldsymbol{\eta}[X := 0]) - \bar{h}_v(y, \boldsymbol{\eta})) = 0, \quad (4)$$

where $v \in V \setminus V_F$, $\boldsymbol{\eta} \models Inv(v)$, $\boldsymbol{\eta}^{(i)}$ is the i -th clock variable and $y \in [0, t_f]$. The boundary conditions are:

- (i) $\bar{h}_v(y, \boldsymbol{\eta}) = \bar{h}_{v'}(y, \boldsymbol{\eta})$, for every $\boldsymbol{\eta} \models \partial Inv(v)$ and transition $v \xrightarrow{\lambda} v'$;
- (ii) $\bar{h}_v(y, \boldsymbol{\eta}) = 1$, for every vertex $v \in V_F$, $\boldsymbol{\eta} \models Inv(v)$, and $y \in [0, t_f]$;
- (iii) $\bar{h}_v(t_f, \boldsymbol{\eta}) = 0$, for every vertex $v \in V \setminus V_F$ and $\boldsymbol{\eta} \models Inv(v) \cup \partial Inv(v)$.

Remark 4. Note that the PDE system (4) in Theorem 2 is different from the one presented in [10] for reducing $Pr_{v_0}^{\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}}}(\mathbf{0})$. In particular, the boundary condition in [10] has been corrected here.

4.2 Reduction to an ODE System

There are several classical methods to solve PDEs. *Finite element method*, which is a numerical technique for solving PDEs as well as integral equations, is a prominent one, of which different versions have been established to solve different PDEs with specific properties. Other numerical methods include finite difference method and finite volume method and so on, the reader is referred to [20, 21] for details. Thanks to the special form of the Eq. (4), we are able to obtain a numerical solution in a more efficient way.

The fact that the flow function (which is the solution to the ODE system $\bigwedge_{x \in \mathcal{X}} \dot{x} = 1 \wedge \dot{y} = 1$) is the same at all locations of the EPDP $\mathcal{Q}^{\mathcal{C} \otimes \mathcal{A}[t_f]}$ suggests that the partial derivatives of $\boldsymbol{\eta}$ and y in the left side of (4) evolve with the same pace. Thus, we can view all clocks as an array, and reformulate (4) as

$$\left[\frac{\partial \bar{h}_v(y, \boldsymbol{\eta})}{\partial y}, \frac{\partial \bar{h}_v(y, \boldsymbol{\eta})}{\partial \boldsymbol{\eta}^{(1)}}, \dots, \frac{\partial \bar{h}_v(y, \boldsymbol{\eta})}{\partial \boldsymbol{\eta}^{(|\mathcal{X}|)}} \right] \bullet \mathbf{1} + \Lambda(v) \cdot \sum_{v \xrightarrow{p, X} v'} p \cdot (\bar{h}_{v'}(y, \boldsymbol{\eta}[X := 0]) - \bar{h}_v(y, \boldsymbol{\eta})) = 0, \quad (5)$$

where \bullet stands for the inner product of two vectors of the same dimension, e.g.,

$$(a_1, \dots, a_n) \bullet (b_1, \dots, b_n) = \sum_{i=1}^n a_i b_i, \text{ and } \mathbf{1} \text{ for the vector } \overbrace{(1, \dots, 1)}^{n \text{ times}}.$$

By Theorem 2, there exist v_0, y_0 and η_0 such that $v_0 \in V_F$, $y_0 = t_f$, and $\eta_0 \models \text{Inv}(v) \vee \partial \text{Inv}(v)$. Besides, by the definition of $\mathcal{Q}^{C \otimes A[t_f]}$, it follows $\frac{\partial z}{\partial t} = 1$, which implies $dz = dt$, for any $z \in \{y\} \cup \mathcal{X}$. Hence, we can simplify (5) as the following ODE system:

$$\begin{aligned} \frac{d\hbar_v((y_0, \eta_0) + t)}{dt} + \Lambda(v) \cdot \\ \sum_{v \xrightarrow{p, X} v'} p \cdot (\hbar_{v'}((y_0, \eta_0) + t)[X := 0]) - \hbar_v(y_0, \eta_0) = 0, \quad (6) \end{aligned}$$

with the initial condition $v_0 \in V_F$, $y_0 = t_f$, and $\eta_0 \models \text{Inv}(v) \vee \partial \text{Inv}(v)$, where $v \in V \setminus V_F$. Note that we compute the reachability probability by (6) backwards.

4.3 Numerical Solution

Since $\hbar_v((y_0, \eta_0) + t)$ satisfies an ODE equation, we can apply a discretization method to (6) and obtain an approximation efficiently. To this end, the remaining obstacle is how to deal with the reset part $\hbar_{v'}(y_0 + t, (\eta_0 + t)[X := 0])$. Notice that $X \neq \emptyset \Rightarrow \text{sum}((\eta_0 + t)[X := 0]) + (t_f - y_0 - t) < \text{sum}(\eta_0 + t) + (t_f - t_0 - t)$, where $\text{sum}(\eta) = \sum_{x \in \mathcal{X}} \eta(x)$. So we just need to solve the ODE system starting from (t_f, η_0) using the descending order over $\text{sum}(\eta)$ in a backward manner. In this way, all of the reset values needed for the current iteration have been computed in the previous iterations. Therefore for each iteration, the derivation is fixed and easy to calculate.

We denote by δ the length of discretization step, the number of total discretization steps is $\lceil \frac{t_f}{\delta} \rceil \in \mathbb{N}$. An approximate solution to (4) can be computed efficiently by the following algorithm.

Line 4 in Algorithm 1 computes a numerical solution to (6) on $[t_f - t, t_f]$ by discretizing $\frac{d\hbar_v((y_0, \eta_0) + t)}{dt}$ with $\frac{1}{\delta}(\hbar_v((y_0, \eta_0) + (t + \delta)) - \hbar_v((y_0, \eta_0) + t))$. A pictorial illustration to Algorithm 1 for the two-dimensional setting is shown in Fig. 5. The blue polyhedron covers all the points we need to calculate. The algorithm starts from $(0, 0, t_f)$, where $\text{sum}(\eta) = x_1 + x_2 = 0$. Then $\text{sum}(\eta)$ is incremented until $2t_f$ in a stepwise manner. For each fixed $\text{sum}(\eta)$, for example $\text{sum}(\eta) = t_f$, the algorithm calculates all discrete points in the gray plane following the direction $(-1, -1, -1)$, and finally reaches the two reset lines. The red line reaching the origin provides the final result.

Algorithm 1. Finding numerical solution to (4)

Input: $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, the region graph of the product of CTMC \mathcal{C} and DTA \mathcal{A} ; t_f , the time bound

Output: A numerical solution for $\bar{h}_{v_0}(0, 0)$, an approximation of $\Pr(\mathcal{C} \models \mathcal{A}[t_f])$

```

1: for  $n \leftarrow 0$  to  $|\mathcal{X}| \cdot t_f$  by  $\delta$  do
2:   for each  $\eta$  in  $\{\eta' \mid \text{sum}(\eta') = n \wedge \forall i \in \{1, \dots, |\mathcal{X}|\} 0 \leq \eta^{(i)} \leq t_f\}$  do
3:     for  $t$  from 0 down to  $-\min(t_f, \eta)$  do
4:       Compute numerical solution to (6) with  $(y_0, \eta_0) = (t_f, \eta)$  on  $[t_f - t, t_f]$ 
5:     end for
6:   end for
7: end for
8: return numerical solution for  $\bar{h}_{v_0}(0, 0)$ 

```

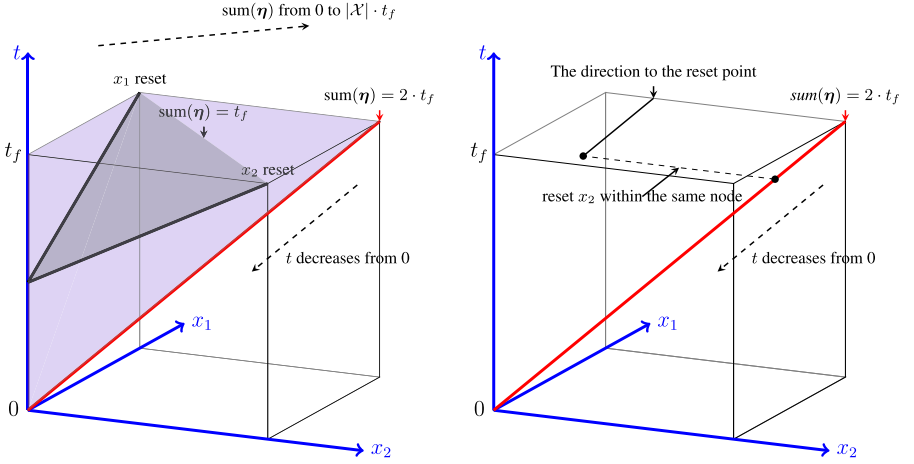


Fig. 5. Illustrating Algorithm 1 (left) and Algorithm 2 (right) for the 2-dimensional setting (Color figure online)

Example 4. Consider the product $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$ shown in Example 3 (in page 8). For state v_3 in which clock x is 1 and y is arbitrary, the corresponding PDE is

$$\frac{\partial \bar{h}_{v_3}(y, 1)}{\partial y} + \frac{\partial \bar{h}_{v_3}(y, 1)}{\partial x} + r_0[0.5 \cdot \bar{h}_{v_0}(y, 0) + 0.2 \cdot \bar{h}_{v_4}(y, 0) + 0.4 \cdot \bar{h}_{v_9}(y, 0) - \bar{h}_{v_3}(y, 0)] = 0.$$

Since $\text{sum}(y, 0) = y < y + 1 = \text{sum}(y, 1)$, the value for $\bar{h}_{v_0}(y, 0)$, $\bar{h}_{v_4}(y, 0)$ and $\bar{h}_{v_3}(y, 0)$ have been calculated in the previous iterations, thus the value for $\bar{h}_{v_3}(y, 1)$ can be computed.

To optimize Algorithm 1 for multi-clock objects, we exploit the idea of “lazy computation”. In Algorithm 1, in order to determine the reset part for (6), we calculate all discretized points generated by all ODEs. The efficiency is influenced since the amount of ODEs is quite large (the same as the number of states in product automaton). However in Algorithm 2, we only compute

the reset part that we need for computing $\bar{h}_{v_0}(0, \mathbf{0})$. If we meet a reset part $\bar{h}_v(y, \boldsymbol{\eta}[X := 0])$ which has not been decided yet, we suspend the equation we are computing now and switch to compute the equation leading to the undecided point following the direction of $(-1, \dots, -1)$. The algorithm terminates since the number of points it computes is no more than that of Algorithm 1. A pseudo-code is described in Algorithm 2.

Algorithm 2. The lazy computation to find numerical solution to (4)

Input: $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, the region graph of the product of CTMC \mathcal{C} and DTA \mathcal{A} ; t_f , the time bound

Output: A numerical solution for $\bar{h}_{v_0}(0, \mathbf{0})$, an approximation of $Pr(\mathcal{C} \models \mathcal{A}[t_f])$

Procedure dhv($y, \boldsymbol{\eta}$) //Computing numerical solution for $(y, \boldsymbol{\eta})$

```

1: for  $t$  from 0 down to  $-\min(t_f, \boldsymbol{\eta})$  by  $\delta$  do
2:   for  $v \in V$  do
3:     Check if  $\boldsymbol{\eta}$  satisfies initial and boundary condition from Theorem 2
4:     for each Markovian transition  $v \xrightarrow{p, X} v'$  do
5:        $up = (-t - \delta) \cdot \mathbf{1} + ((t + \delta) \cdot \mathbf{1})[X := 0]$ 
6:       if reset exists and  $\boldsymbol{\eta}[X := 0] + up$  is undecided then
7:         call dhv( $t_f, \boldsymbol{\eta}[X := 0] + up$ )
8:       end if
9:       compute  $\bar{h}_v$ 
10:    end for
11:  end for
12:  execute  $\lambda$ -transition according to Theorem 2
13:  compute  $\bar{h}_v((y_0, \boldsymbol{\eta}_0) + t)$  by equation (6)
14: end for
15: mark  $\boldsymbol{\eta}$  decided

```

End Procedure

```

1: Call dhv( $v_0, t_f, (t_f)$ )
2: return numerical solution for  $\bar{h}_{v_0}(0, \mathbf{0})$ 

```

4.4 Complexity Analysis

Let $|S|$ be the number of the states of the CTMC, and n the number of the clocks of the DTA. The worst-case time complexity of Algorithms 1 and 2 lies in $\mathcal{O}(|V| \cdot \lceil \frac{t_f}{\delta} \rceil^{(n+1)})$, where $|V|$ is the number of the equations in (4), i.e., the number of the locations in the product region graph, that are not accepting. The number of states in the region graph of the DTA is bounded by $n! \cdot 2^{n-1} \cdot \prod_{x \in \mathcal{X}} (c_x + 1)$, denoted by C_b , where c_x is the maximum constant occurring in the guards that constrain x . Note that C_b differs from the bound given in [1], since the boundaries of a region do not matter in our setting and hence can be merged into the region. Thus, the number of states in the product region graph, as well as the number of PDE equations in Theorem 2, is at most $C_b \cdot |S|$. So the total complexity is $\mathcal{O}(C_b \cdot |S| \cdot \lceil \frac{t_f}{\delta} \rceil^{(n+1)})$.

Let $\bar{h}_{v,n}(y_0, \boldsymbol{\eta}_0)$ denote the numerical solution to ODE (6) with $t = -n\delta$, and $A_{max} = \max\{A(v_i) \mid 0 \leq i \leq |S|\}$. Let $N = \lceil \frac{t_f}{\delta} \rceil$. By Proposition 2, $\lim_{t_f \rightarrow +\infty} \bar{h}_v(0, \mathbf{0}) = Pr(\mathcal{C} \models \mathcal{A})$ and $\bar{h}_v(0, \mathbf{0})$ is monotonically increasing for t_f . In

the following proposition, for simplicity of discussion, we assume t_f equal to $N\delta$. Then, the error caused by discretization can be estimated as follows:

Proposition 3. For $N \in \mathbb{N}^+$ and $\delta = \frac{t_f}{N}$,

$$|\bar{h}_{v_0, N}(t_f, t_f \cdot \mathbf{1}) - \bar{h}_{v_0}(0, \mathbf{0})| = \mathcal{O}(\delta)$$

For function $f(\delta)$, f is of the magnitude $\mathcal{O}(\delta)$ if $\lim_{\delta \rightarrow 0} \left| \frac{f(\delta)}{\delta} \right| = C$, where C is a constant. From Proposition 3, if we view Λ_{\max} and t_f as constants, then the error is $\mathcal{O}(\delta)$ to the step length δ . By Proposition 2, the numerical solution generated by Algorithm 1 converges to the reachability probability of $\mathcal{C} \otimes \mathcal{A}$, and the error can be as small as we expect if we decrease the size of discretization δ , and increase the time bound t_f .

5 Experimental Results

We implemented a prototype including Algorithms 1 and 2 in C and a tool taking a CTMC \mathcal{C} and a DTA \mathcal{A} as input and generating a .c file to store their product in Python, which is used as an input to Algorithms 1 and 2. The first two examples (Examples 5 and 6) come from [10] to show the feasibility of our tool. The last case study is an example of robot navigation from [7]. In order to demonstrate the scalability of our approach, we revise the example with different real-time requirements, which require DTA with different number of clocks. The examples are executed in Linux 16.04 LTS with Intel(R) Core(TM) i7-4710HQ 2.50 GHz CPU and 16 G RAM. The column “time” reports the running time for Algorithm 1, and “time (lazy)” reports the running time for Algorithm 2. All time is counted in seconds.

Example 5. Consider Example 3 with $r_i = 1$, $i = 0, \dots, 3$ and $\delta = 0.01$, experimental result is shown in Table 1. The relevant error when $t_f = 30$ and $t_f = 40$ is 5×10^{-7} .

Table 1. The experimental results for Examples 5 and 6

| t_f | Example 5 | | | Example 6 | | |
|-------|--------------------------------|--------|-------------|--------------------------------|--------|-------------|
| | $\bar{h}_{v_0}(0, \mathbf{0})$ | time | time (lazy) | $\bar{h}_{v_0}(0, \mathbf{0})$ | time | time (lazy) |
| 20 | 0.110791 | 0.8070 | 0.7232 | 0.999999 | 0.1685 | 0.0002 |
| 30 | 0.110792 | 1.7246 | 1.6260 | 0.999999 | 0.3453 | 0.0003 |
| 40 | 0.110792 | 3.0344 | 2.8760 | 0.999999 | 0.6265 | 0.0003 |

Example 6. Consider the reachability probability for the product of a CTMC and a DTA as shown in Fig. 6. A part of its region graph is shown in Fig. 7. Set $r_0 = r_1 = 1$, $\delta = 0.1$, the experimental result is given in Table 1. The relevant error when $t_f = 30$ and $t_f = 40$ is 1×10^{-7} . Note that even for this simple example, none of existing tools can handle it.

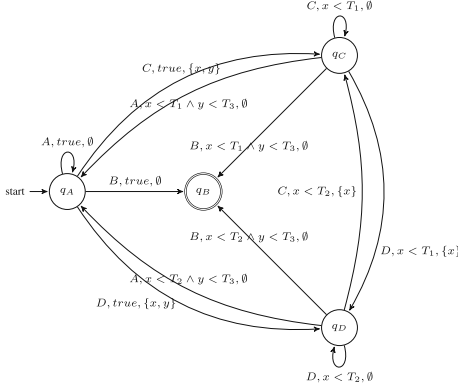


Fig. 10. A DTA with two clocks for $P_1 \wedge P_2$

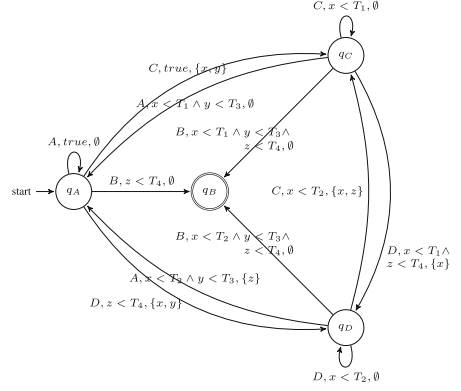


Fig. 11. A DTA with three clocks for $P_1 \wedge P_2 \wedge P_3$

Table 2. Experimental results for the robot example with $\delta = 0.1$, running time longer than 2700s is denoted by ‘TO’ (timeout), the column “#(P)” counts the number of states in the product automaton $\mathcal{C} \otimes \mathcal{G}(\mathcal{A})$, “time([7])” is the running time of prototype in [7] when precision = 0.01, $T_1 = T_2 = 3$, $T_3 = 5$, $T_4 = 7$

| | | One clock | | | | Two clocks | | | Three clocks | | |
|----|-------|-----------|-------|-------------|-----------|------------|--------|-------------|--------------|-------|-------------|
| N | t_f | #(P) | time | time (lazy) | time([7]) | #(P) | time | time (lazy) | #(P) | time | time (lazy) |
| 4 | 10 | 39 | 0.027 | 0.027 | 0.011 | 139 | 2.583 | 1.746 | 733 | 525.7 | 141.4 |
| | 15 | | 0.049 | 0.043 | | | 7.117 | 3.445 | | TO | 257.35 |
| | 20 | | 0.070 | 0.071 | | | 12.88 | 5.49 | | TO | 583.76 |
| 10 | 10 | 232 | 0.167 | 0.164 | 0.087 | 968 | 39.41 | 25.92 | 5134 | TO | 1039.7 |
| | 15 | | 0.278 | 0.278 | | | 108.48 | 53.28 | | TO | TO |
| | 20 | | 0.417 | 0.421 | | | 226.56 | 89.50 | | TO | TO |
| 20 | 10 | 940 | 1.142 | 0.909 | 1.23 | 4000 | 250.1 | 180.7 | | TO | TO |
| | 15 | | 1.65 | 1.54 | | | 672.8 | 375.6 | | TO | TO |
| | 20 | | 2.54 | 2.41 | | | 1326.8 | 616.1 | | TO | TO |
| 30 | 10 | 2125 | 2.38 | 2.45 | 6.84 | 9120 | 812.9 | 380.5 | | TO | TO |
| | 15 | | 4.45 | 5.42 | | | 2058.1 | 770.8 | | TO | TO |
| | 20 | | 7.45 | 7.28 | | | TO | 1283.4 | | TO | TO |
| 40 | 10 | 3820 | 5.62 | 6.52 | 20.31 | 16395 | 1484.3 | 759.8 | | TO | TO |
| | 15 | | 11.97 | 11.02 | | | TO | 1619.9 | | TO | TO |
| | 20 | | 15.26 | 16.17 | | | TO | 2661.3 | | TO | TO |

results reported in [7] for the case of one clock in this case study (when the precision is set to be 10^{-2}), our result is as fast as theirs, but their tool cannot handle the cases of multiple clocks. In contrast, our approach can handle DTA with multiple clocks as indicated in the verification of P_2 and P_3 . Algorithm 2 is much more faster than Algorithm 1 when the number of clocks grows up. To

the best of our knowledge, this is the first prototypical tool verifying CTMCs against multi-clock DTA.

6 Concluding Remarks

In this paper, we present a practical approach to verify CTMCs against DTA objectives. First, the desired probability can be reduced to the reachability probability of the product region graph in the form of PDPs. Then we use the augmented PDP to approximate the reachability probability, in which the reachability probability coincides with the solution to a PDE system at the starting point. We further propose a numerical solution to the PDE system by reduction it to a ODE system. The experimental results indicate the efficiency and scalability compared with existing work, as it can handle DTA with multiple clocks.

As a future work, it deserves to investigate whether our approach also works in the verification of CTMCs against more complicated real-time properties, either expressed by timed automata and MTL as considered in [9], or by linear duration invariants as considered in [8].

Acknowledgements. This research is partly funded by the Sino-German Center for Research Promotion as part of the project CAP (GZ 1023), from Yijun Feng, Haokun Li and Bican Xia is partly funded by NSFC under grant No. 61732001 and 61532019, from Joost-Pieter Katoen is partly funded by the DFG Research Training Group 2236 UnRAVeL, from Naijun Zhan is funded partly by NSFC under grant No. 61625206 and 61732001, by “973 Program” under grant No. 2014CB340701 and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994)
2. Amparore, E.G., Beccuti, M., Donatelli, S.: (Stochastic) model checking in Great-SPN. In: Ciardo, G., Kindler, E. (eds.) *PETRI NETS 2014*. LNCS, vol. 8489, pp. 354–363. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07734-5_19
3. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Log.* **1**(1), 162–170 (2000)
4. Baier, C., Cloth, L., Haverkort, B.R., Kuntz, M., Siegle, M.: Model checking Markov chains with actions and state labels. *IEEE Trans. Softw. Eng.* **33**(4), 209–224 (2007)
5. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Softw. Eng.* **29**(6), 524–541 (2003)
6. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
7. Barbot, B., Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Efficient CTMC model checking of linear real-time objectives. In: Abdulla, P.A., Leino, K.R.M. (eds.) *TACAS 2011*. LNCS, vol. 6605, pp. 128–142. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_12

8. Chen, T., Diciolla, M., Kwiatkowska, M., Mereacre, A.: Verification of linear duration properties over continuous-time Markov chains. *ACM Trans. Comput. Log.* **14**(4), 33 (2013)
9. Chen, T., Diciolla, M., Kwiatkowska, M., Mereacre, A.: Time-bounded verification of CTMCs against real-time specifications. In: Fahrenberg, U., Tripakis, S. (eds.) *FORMATS 2011*. LNCS, vol. 6919, pp. 26–42. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_4
10. Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Quantitative model checking of continuous-time Markov chains against timed automata specifications. In: *LICS*, pp. 309–318 (2009)
11. Chen, T., Han, T., Katoen, J., Mereacre, A.: Model checking of continuous-time Markov chains against timed automata specifications. *Log. Methods Comput. Sci.* **7**(1) (2011)
12. Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Observing continuous-time MDPs by 1-clock timed automata. In: Delzanno, G., Potapov, I. (eds.) *RP 2011*. LNCS, vol. 6945, pp. 2–25. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24288-5_2
13. Dang, V.H., Zhou, C.: Probabilistic duration calculus for continuous time. *Formal Aspects Comput.* **11**(1), 21–44 (1999)
14. Davis, M.H.: *Markov Models and Optimization*, vol. 49. CRC Press, Boca Raton (1993)
15. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A STORM is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) *CAV 2017*. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
16. Donatelli, S., Haddad, S., Sproston, J.: Model checking timed and stochastic properties with CSL^{TA}. *IEEE Trans. Softw. Eng.* **35**(2), 224–240 (2009)
17. Feller, W.: *An Introduction to Probability Theory and Its Applications*, vol. 3. Wiley, New York (1968)
18. Fu, H.: Approximating acceptance probabilities of CTMC-paths on multi-clock deterministic timed automata. In: *HSCC*, pp. 323–332. ACM (2013)
19. Gao, Y., Xu, M., Zhan, N., Zhang, L.: Model checking conditional CSL for continuous-time Markov chains. *Inf. Process. Lett.* **113**(1–2), 44–50 (2013)
20. Grossmann, C., Roos, H.-G., Stynes, M.: *Numerical Treatment of Partial Differential Equations*, vol. 154. Springer, Heidelberg (2007)
21. Johnson, C.: *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Courier Corporation, Chelmsford (2012)
22. 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)
23. 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). https://doi.org/10.1007/978-3-642-22110-1_47
24. Mikeev, L., Neuhäuser, M.R., Spieler, D., Wolf, V.: On-the-fly verification and optimization of DTA-properties for large Markov chains. *Formal Methods Syst. Des.* **43**(2), 313–337 (2013)
25. Wisniewski, R., Sloth, C., Bujorianu, M.L., Piterman, N.: Safety verification of piecewise-deterministic Markov processes. In: *HSCC*, pp. 257–266. ACM (2016)

26. Yin, G.G., Zhang, Q.: Continuous-Time Markov Chains and Applications: A Two-Time-Scale Approach, vol. 37. Springer, New York (2012). <https://doi.org/10.1007/978-1-4614-4346-9>
27. Zhang, L., Jansen, D.N., Nielson, F., Hermanns, H.: Efficient CSL model checking using stratification. *Log. Methods Comput. Sci.* **8**(2:17), 1–18 (2012)
28. Zhou, C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. *Inf. Process. Lett.* **40**(5), 269–276 (1991)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Start Pruning When Time Gets Urgent: Partial Order Reduction for Timed Systems

Frederik M. Bønneland, Peter Gjøøl Jensen,
Kim Guldstrand Larsen, Marco Muñiz,
and Jiří Srba^(✉)

Department of Computer Science,
Aalborg University, Aalborg, Denmark
{frederikb,pgj,kgl,muniz,srba}@cs.aau.dk



Abstract. Partial order reduction for timed systems is a challenging topic due to the dependencies among events induced by time acting as a global synchronization mechanism. So far, there has only been a limited success in finding practically applicable solutions yielding significant state space reductions. We suggest a working and efficient method to facilitate stubborn set reduction for timed systems with urgent behaviour. We first describe the framework in the general setting of timed labelled transition systems and then instantiate it to the case of timed-arc Petri nets. The basic idea is that we can employ classical untimed partial order reduction techniques as long as urgent behaviour is enforced. Our solution is implemented in the model checker TAPAAL and the feature is now broadly available to the users of the tool. By a series of larger case studies, we document the benefits of our method and its applicability to real-world scenarios.

1 Introduction

Partial order reduction techniques for untimed systems, introduced by Godefroid, Peled, and Valmari in the nineties (see e.g. [6]), have since long proved successful in combating the notorious state space explosion problem. For *timed* systems, the success of partial order reduction has been significantly challenged by the strong dependencies between events caused by time as a global synchronizer. Only recently—and moreover in combination with *approximate* abstraction techniques—stubborn set techniques have demonstrated a true reduction potential for systems modelled by timed automata [23].

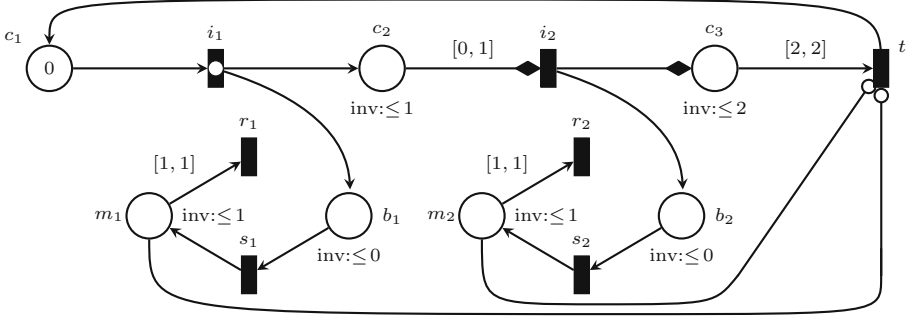
We pursue an orthogonal solution to the current partial order approaches for timed systems and, based on a stubborn set reduction [28, 39], we target a general class of timed systems with *urgent behaviour*. In a modular modelling approach for timed systems, urgency is needed to realistically model behaviour in a component that should be unobservable to other components [36]. Examples of such instantaneously evolving behaviours include, among others, cases like behaviour detection in a part of a sensor (whose duration is assumed to be

negligible) or handling of release and completion of periodic tasks in a real-time operating system. We observe that focusing on the urgent part of the behaviour of a timed system allows us to exploit the full range of partial order reduction techniques already validated for untimed systems. This leads to an exact and broadly applicable reduction technique, which we shall demonstrate on a series of industrial case studies showing significant space and time reduction. In order to highlight the generality of the approach, we first describe our reduction technique in the setting of timed labelled transition systems. We shall then instantiate it to timed-arc Petri nets and implement and experimentally validate it in the model checker TAPAAL [19].

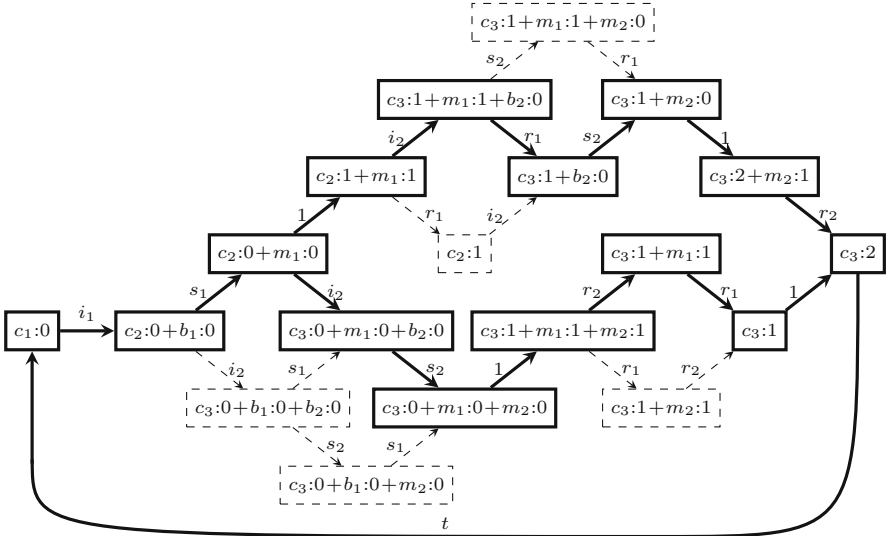
Let us now briefly introduce the model of timed-arc Petri nets and explain our reduction ideas. In timed-arc Petri nets, each token is associated with a nonnegative integer representing its age and input arcs to transitions contain intervals, restricting the ages of tokens available for transition firing (if an interval is missing, we assume the default interval $[0, \infty]$ that accepts all token ages). In Fig. 1a we present a simple monitoring system modelled as a timed-arc Petri net. The system consists of two identical sensors where sensor i , $i \in \{1, 2\}$, is represented by the places b_i and m_i , and the transitions s_i and r_i . Once a token of age 0 is placed into the place b_i , the sensor gets started by executing the transition s_i and moving the token from place b_i to m_i where the monitoring process starts. As the place b_i has an associated age invariant ≤ 0 , meaning that all tokens in b_i must be of age at most 0, no time delay is allowed and the firing of s_i becomes urgent. In the monitoring place m_i we have to delay one time unit before the transition r_i reporting the reading of the sensor becomes enabled. Due to the age invariant ≤ 1 in the place m_i , we cannot wait longer than one time unit, after which r_i becomes also urgent.

The places c_1 , c_2 and c_3 together with the transitions i_1 , i_2 and t are used to control the initialization of the sensors. At the execution start, only the transition i_1 is enabled and because it is an urgent transition (denoted by the white circle), no delay is initially possible and i_1 must be fired immediately while removing the token of age 0 from c_1 and placing a new token of age 0 into c_2 . At the same time, the first sensor gets started as i_1 also places a fresh token of age 0 into b_1 . Now the control part of the net can decide to fire without any delay the transition i_2 and start the second sensor, or it can delay one unit of time after which i_2 becomes urgent due to the age invariant ≤ 1 as the token in c_2 is now of age 1. If i_2 is fired now, it will place a fresh token of age 0 into b_2 . However, the token that is moved from c_2 to c_3 by the pair of transport arcs with the diamond-shaped arrow tips preserves its age 1, so now we have to wait precisely one more time unit before t becomes enabled. Moreover, before t can be fired, the places m_1 and m_2 must be empty as otherwise the firing of t is disabled due to inhibitor arcs with circle-shaped arrow tips.

In Fig. 1b we represent the reachable state space of the simple monitoring system where markings are represented using the notation like $c_3 : 1 + b_2 : 2$ that stands for one token of age 1 in place c_3 and one token of age 2 in place b_2 . The dashed boxes represent the markings that can be avoided during the state space exploration when we apply our partial order reduction method for checking if



(a) TAPN model of a simple monitoring system



(b) Reachable state space generated by the net in Figure 1a

Fig. 1. Simple monitoring system

the termination transition t can become enabled from the initial marking. We can see that the partial order reduction is applied such that it preserves at least one path to all configurations where our goal is reached (transition t is enabled) and where time is not urgent anymore (i.e. to the configurations that allow the delay of 1 time unit). The basic idea of our approach is to apply the stubborn set reduction on the commutative diamonds where time is not allowed to elapse.

Related Work. Our stubborn set reduction is based on the work of Valmari et al. [28, 39]. We formulate their stubborn set method in the abstract framework of labelled transition systems with time and add further axioms for time elapsing in order to guarantee preservation of the reachability properties.

For Petri nets, Yoneda and Schlingloff [41] apply a partial order reduction to one-safe time Petri nets, however, as claimed in [38], the method is mainly suitable for small to medium models due to a computational overhead, confirmed also in [29]. The experimental evaluation in [41] shows only one selected example. Sloan and Buy [38] try to improve on the efficiency of the method, at the expense of considering only a rather limited model of *simple time Petri nets* where each transition has a statically assigned duration. Lilius [29] suggests to instead use alternative semantics of timed Petri nets to remove the issues related to the global nature of time, allowing him to apply directly the untimed partial order approaches. However, the semantics is nonstandard and no experiments are reported. Another approach is by Virbitskaite and Pokozy [40], who apply a partial order method on the *region graph* of bounded time Petri nets. Region graphs are in general not an efficient method for state space representation and the method is demonstrated only on a small buffer example with no further experimental validation. Recently, partial order techniques were suggested by André et al. for parametric time Petri nets [5], however, the approach is working only for safe and acyclic nets. Boucheneb and Barkaoui [12–14] discuss a partial order reduction technique for timed Petri nets based on *contracted state class graphs* and present a few examples on a prototype implementation (the authors do not refer to any publicly available tool). Their method is different from ours as it aims at adding timing constraints to the independence relation, but it does not exploit urgent behaviour. Moreover, the models of time Petri nets and timed-arc Petri nets are, even on the simplest nets, incomparable due to the different way to modelling time.

The fact that we are still lacking a practically applicable method for the time Petri net model is documented by a missing implementation of the technique in leading tools for time Petri net model checking like TINA [9] and Romeo [22]. We are not aware of any work on partial order reduction technique for the class of timed-arc Petri nets that we consider in this paper. This is likely because this class of nets provides even more complex timing behaviour, as we consider unbounded nets where each token carries its timing information (and needs a separate clock to remember the timing), while in time Petri nets timing is associated only to a priori fixed number of transitions in the net.

In the setting of timed automata [3], early work on partial order reduction includes Bengtsson et al. [8] and Minea [32] where they introduce the notion of local as well as global clocks but provide no experimental evaluation. Dams et al. [18] introduce the notion of *covering* in order to generalize dependencies but also here no empirical evaluation is provided. Lugiez, Niebert et al. [30, 34] study the notion of *event zones* (capturing time-durations between events) and use it to implement Mazurkiewicz-trace reductions. Salah et al. [37] introduce and implement an exact method based on merging zones resulting from different interleavings. The method achieves performance comparable with the approximate convex-hull abstraction which is by now superseded by the exact LU-abstraction [7]. Most recently, Hansen et al. [23] introduce a variant of stubborn sets for reducing an *abstracted zone graph*, thus in general offering overapproximate analysis. Our technique is orthogonal to the other approaches mentioned

above; not only is the model different but also the application of our reduction gives exact results and is based on new reduction ideas. Finally, the idea of applying partial order reduction for independent events that happen at the same time appeared also in [15] where the authors, however, use a static method that declares actions as independent only if they do not communicate, do not emit signals and do not access any shared variables. Our realization of the method to the case of timed-arc Petri nets applies a dynamic (on-the-fly) reduction, while executing a detailed timing analysis that allows us to declare more transitions as independent—sometimes even in the case when they share resources.

2 Partial Order Reduction for Timed Systems

We shall now describe the general idea of our partial order reduction technique (based on stubborn sets [28, 39]) in terms of timed transition systems. We consider real-time delays in the rest of this section, as these results are not specific only to discrete time semantics. Let A be a given set of actions such that $A \cap \mathbb{R}_{\geq 0} = \emptyset$ where $\mathbb{R}_{\geq 0}$ stands for the set of nonnegative real numbers.

Definition 1 (Timed Transition System). *A timed transition system is a tuple (S, s_0, \rightarrow) where S is a set of states, $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times (A \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation.*

If $(s, \alpha, s') \in \rightarrow$ we write $s \xrightarrow{\alpha} s'$. We implicitly assume that if $s \xrightarrow{0} s'$ then $s = s'$, i.e. zero time delays do not change the current state. The set of *enabled actions* at a state $s \in S$ is defined as $\text{En}(s) \stackrel{\text{def}}{=} \{a \in A \mid \exists s' \in S. s \xrightarrow{a} s'\}$. Given a sequence of actions $w = \alpha_1 \alpha_2 \alpha_3 \dots \alpha_n \in (A \cup \mathbb{R}_{\geq 0})^*$ we write $s \xrightarrow{w} s'$ iff $s \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s'$. If there is a sequence w of length n such that $s \xrightarrow{w} s'$, we also write $s \rightarrow^n s'$. Finally, let \rightarrow^* be the reflexive and transitive closure of the relation \rightarrow such that $s \rightarrow^* s'$ iff there is $\alpha \in \mathbb{R}_{\geq 0} \cup A$ and $s \xrightarrow{\alpha} s'$.

For the rest of this section, we assume a fixed transition system (S, s_0, \rightarrow) and a set of goal states $G \subseteq S$. The *reachability problem*, given a timed transition system (S, s_0, \rightarrow) and a set of goal states G , is to decide whether there is $s' \in G$ such that $s_0 \rightarrow^* s'$.

We now develop the theoretical foundations of stubborn sets for timed transition systems. A state $s \in S$ is *zero time* if time can not elapse at s . We denote the zero time property of a state s by the predicate $\text{zt}(s)$ and define it as $\text{zt}(s)$ iff for all $s' \in S$ and all $d \in \mathbb{R}_{\geq 0}$ if $s \xrightarrow{d} s'$ then $d = 0$. A *reduction* of a timed transition system is a function $\text{St} : S \rightarrow 2^A$. A reduction defines a reduced transition relation $\xrightarrow{\text{St}} \subseteq \rightarrow$ such that $s \xrightarrow{\text{St}} s'$ iff $s \xrightarrow{\alpha} s'$ and $\alpha \in \text{St}(s) \cup \mathbb{R}_{\geq 0}$. For a given state $s \in S$ we define $\overline{\text{St}(s)} \stackrel{\text{def}}{=} A \setminus \text{St}(s)$ as the set of all actions that are not in $\text{St}(s)$.

Definition 2 (Reachability Conditions). *A reduction St on a timed transition system (S, s_0, \rightarrow) is reachability preserving if it satisfies the following four conditions.*

- (\mathcal{Z}) $\forall s \in S. \neg \text{zt}(s) \implies \text{En}(s) \subseteq \text{St}(s)$
- (\mathcal{D}) $\forall s, s' \in S. \forall w \in \overline{\text{St}(s)}^*. \text{zt}(s) \wedge s \xrightarrow{w} s' \implies \text{zt}(s')$
- (\mathcal{R}) $\forall s, s' \in S. \forall w \in \overline{\text{St}(s)}^*. \text{zt}(s) \wedge s \xrightarrow{w} s' \wedge s \notin G \implies s' \notin G$
- (\mathcal{W}) $\forall s, s' \in S. \forall w \in \overline{\text{St}(s)}^*. \forall a \in \text{St}(s). \text{zt}(s) \wedge s \xrightarrow{wa} s' \implies s \xrightarrow{aw} s'$

Condition \mathcal{Z} declares that in a state where a delay is possible, all enabled actions become stubborn actions. Condition \mathcal{D} guarantees that in order to enable a time delay from a state where delaying is not allowed, a stubborn action must be executed. Similarly, Condition \mathcal{R} requires that a stubborn action must be executed before a goal state can be reached from a non-goal state. Finally, Condition \mathcal{W} allows us to commute stubborn actions with non-stubborn actions. The following theorem shows that reachability preserving reductions generate pruned transition systems where the reachability of goal states is preserved.

Theorem 1 (Shortest-Distance Reachability Preservation). *Let St be a reachability preserving reduction satisfying \mathcal{Z} , \mathcal{D} , \mathcal{R} and \mathcal{W} . Let $s \in S$. If $s \rightarrow^n s'$ for some $s' \in G$ then also $s \xrightarrow[\text{St}]^m s''$ for some $s'' \in G$ where $m \leq n$.*

Proof. We proceed by induction on n . *Base step.* If $n = 0$, then $s = s'$ and $m = n = 0$. *Inductive step.* Let $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_{n+1}$ where $s_0 \notin G$ and $s_{n+1} \in G$. Without loss of generality we assume that for all i , $0 \leq i \leq n$, we have $\alpha_i \neq 0$ (otherwise we can simply skip these 0-delay actions and get a shorter sequence). We have two cases. Case $\neg \text{zt}(s_0)$: by condition \mathcal{Z} we have $\text{En}(s_0) \subseteq \text{St}(s_0)$ and by the definition of $\xrightarrow[\text{St}]$ we have $s_0 \xrightarrow[\text{St}]{\alpha_0} s_1$ since $\alpha_0 \in \text{En}(s_0) \cup \mathbb{R}_{\geq 0}$. By the induction hypothesis we have $s_1 \xrightarrow[\text{St}]^m s''$ with $s'' \in G$ and $m \leq n$ and $m + 1 \leq n + 1$. Case $\text{zt}(s_0)$: let $w = \alpha_0 \alpha_1 \dots \alpha_n$ and α_i be such that $\alpha_i \in \text{St}(s_0)$ and for all $k < i$ holds that $\alpha_k \notin \text{St}(s_0)$, i.e. α_i is the first stubborn action in w . Such an α_i has to exist otherwise $s_{n+1} \notin G$ due to condition \mathcal{R} . Because of condition \mathcal{D} we get $\text{zt}(s_k)$ for all k , $0 \leq k < i$, otherwise α_i cannot be the first stubborn action in w . We can split w as $w = u \alpha_i v$ with $u \in \overline{\text{St}(s_0)}^*$. Since all states in the path to s_i are zero time, by \mathcal{W} we can swap α_i as $s_0 \xrightarrow{\alpha_i} s'_1 \xrightarrow{u} s_i \xrightarrow{v} s'$ with $|uv| = n$. Since $\alpha_i \in \text{St}(s_0)$ we get $s_0 \xrightarrow[\text{St}]{\alpha_i} s'_1$ and by the induction hypothesis we have $s'_1 \xrightarrow[\text{St}]^m s''$ where $s'' \in G$, $m \leq n$, and $m + 1 \leq n + 1$. \square

3 Timed-Arc Petri Nets

We shall now define the model of timed-arc Petri nets (as informally described in the introduction) together with a reachability logic and a few technical lemmas needed later on. Let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ and $\mathbb{N}_0^\infty = \mathbb{N}_0 \cup \{\infty\}$. We define the set of *well-formed closed time intervals* as $\mathcal{I} \stackrel{\text{def}}{=} \{[a, b] \mid a \in \mathbb{N}_0, b \in \mathbb{N}_0^\infty, a \leq b\}$ and its subset $\mathcal{I}^{\text{inv}} \stackrel{\text{def}}{=} \{[0, b] \mid b \in \mathbb{N}_0^\infty\}$ used in age invariants.

Definition 3 (Timed-Arc Petri Net). *A timed-arc Petri net (TAPN) is a 9-tuple $N = (P, T, T_{\text{urg}}, IA, OA, g, w, \text{Type}, I)$ where*

- P is a finite set of places,
- T is a finite set of transitions such that $P \cap T = \emptyset$,
- $T_{urg} \subseteq T$ is the set of urgent transitions,
- $IA \subseteq P \times T$ is a finite set of input arcs,
- $OA \subseteq T \times P$ is a finite set of output arcs,
- $g : IA \rightarrow \mathcal{I}$ is a time constraint function assigning guards (time intervals) to input arcs s.t.
 - if $(p, t) \in IA$ and $t \in T_{urg}$ then $g((p, t)) = [0, \infty]$,
- $w : IA \cup OA \rightarrow \mathbb{N}$ is a function assigning weights to input and output arcs,
- $Type : IA \cup OA \rightarrow \mathbf{Types}$ is a type function assigning a type to all arcs where $\mathbf{Types} = \{Normal, Inhib\} \cup \{Transport_j \mid j \in \mathbb{N}\}$ such that
 - if $Type(z) = Inhib$ then $z \in IA$ and $g(z) = [0, \infty]$,
 - if $Type((p, t)) = Transport_j$ for some $(p, t) \in IA$ then there is exactly one $(t, p') \in OA$ such that $Type((t, p')) = Transport_j$,
 - if $Type((t, p')) = Transport_j$ for some $(t, p') \in OA$ then there is exactly one $(p, t) \in IA$ such that $Type((p, t)) = Transport_j$,
 - if $Type((p, t)) = Transport_j = Type((t, p'))$ then $w((p, t)) = w((t, p'))$,
- $I : P \rightarrow \mathcal{I}^{inv}$ is a function assigning age invariants to places.

Note that for transport arcs we assume that they come in pairs (for each type $Transport_j$) and that their weights match. Also for inhibitor arcs and for input arcs to urgent transitions, we require that the guards are $[0, \infty]$.

Before we give the formal semantics of the model, let us fix some notation.

Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPN. We denote by $\bullet x \stackrel{\text{def}}{=} \{y \in P \cup T \mid (y, x) \in IA \cup OA, Type((y, x)) \neq Inhib\}$ the preset of a transition or a place x . Similarly, the postset is defined as $x^\bullet \stackrel{\text{def}}{=} \{y \in P \cup T \mid (x, y) \in (IA \cup OA)\}$. We denote by ${}^\circ t \stackrel{\text{def}}{=} \{p \in P \mid (p, t) \in IA \wedge Type((p, t)) = Inhib\}$ the inhibitor preset of a transition t . The inhibitor postset of a place p is defined as $p^\circ \stackrel{\text{def}}{=} \{t \in T \mid (p, t) \in IA \wedge Type((p, t)) = Inhib\}$. Let $\mathcal{B}(\mathbb{R}^{\geq 0})$ be the set of all finite multisets over $\mathbb{R}^{\geq 0}$. A marking M on N is a function $M : P \rightarrow \mathcal{B}(\mathbb{R}^{\geq 0})$ where for every place $p \in P$ and every token $x \in M(p)$ we have $x \in I(p)$, in other words all tokens have to satisfy the age invariants. The set of all markings in a net N is denoted by $\mathcal{M}(N)$.

We write (p, x) to denote a token at a place p with the age $x \in \mathbb{R}^{\geq 0}$. Then $M = \{(p_1, x_1), (p_2, x_2), \dots, (p_n, x_n)\}$ is a multiset representing a marking M with n tokens of ages x_i in places p_i . We define the size of a marking as $|M| = \sum_{p \in P} |M(p)|$ where $|M(p)|$ is the number of tokens located in the place p . A marked TAPN (N, M_0) is a TAPN N together with an initial marking M_0 with all tokens of age 0.

Definition 4 (Enabledness). Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPN. We say that a transition $t \in T$ is enabled in a marking M by the multisets of tokens $In = \{(p, x_p^1), (p, x_p^2), \dots, (p, x_p^{w((p, t))}) \mid p \in \bullet t\} \subseteq M$ and $Out = \{(p', x_{p'}^1), (p', x_{p'}^2), \dots, (p', x_{p'}^{w((t, p'))}) \mid p' \in t^\bullet\}$ if

- for all input arcs except the inhibitor arcs, the tokens from In satisfy the age guards of the arcs, i.e.

$$\forall p \in \bullet t. x_p^i \in g((p, t)) \text{ for } 1 \leq i \leq w((p, t))$$

- for any inhibitor arc pointing from a place p to the transition t , the number of tokens in p is smaller than the weight of the arc, i.e.

$$\forall (p, t) \in IA.Type((p, t)) = Inhib \Rightarrow |M(p)| < w((p, t))$$

- for all input arcs and output arcs which constitute a transport arc, the age of the input token must be equal to the age of the output token and satisfy the invariant of the output place, i.e.

$$\begin{aligned} \forall (p, t) \in IA. \forall (t, p') \in OA. Type((p, t)) = Type((t, p')) = Transport_j \\ \Rightarrow (x_p^i = x_{p'}^i \wedge x_{p'}^i \in I(p')) \text{ for } 1 \leq i \leq w((p, t)) \end{aligned}$$

- for all normal output arcs, the age of the output token is 0, i.e.

$$\forall (t, p') \in OA. Type((t, p')) = Normal \Rightarrow x_{p'}^i = 0 \text{ for } 1 \leq i \leq w((t, p')).$$

A given marked TAPN (N, M_0) defines a timed transition system $T(N) \stackrel{\text{def}}{=} (\mathcal{M}(N), M_0, \rightarrow)$ where the states are markings and the transitions are as follows.

- If $t \in T$ is enabled in a marking M by the multisets of tokens In and Out then t can *fire* and produce the marking $M' = (M \setminus In) \uplus Out$ where \uplus is the multiset sum operator and \setminus is the multiset difference operator; we write $M \xrightarrow{t} M'$ for this action transition.
- A time *delay* $d \in \mathbb{N}_0$ is allowed in M if
 - $(x + d) \in I(p)$ for all $p \in P$ and all $x \in M(p)$, i.e. by delaying d time units no token violates any of the age invariants, and
 - if $M \xrightarrow{t} M'$ for some $t \in T_{urg}$ then $d = 0$, i.e. enabled urgent transitions disallow time passing.

By delaying d time units in M we reach the marking M' defined as $M'(p) = \{x + d \mid x \in M(p)\}$ for all $p \in P$; we write $M \xrightarrow{d} M'$ for this delay transition.

Note that the semantics above defines the discrete-time semantics as the delays are restricted to nonnegative integers. It is well known that for timed-arc Petri nets with nonstrict intervals, the marking reachability problem on discrete and continuous time nets coincide [31]. This is, however, not the case for more complex properties like liveness that can be expressed in the CTL logic (for counter examples that can be expressed in CTL see e.g. [25]).

3.1 Reachability Logic and Interesting Sets of Transitions

We now describe a logic for expressing the properties of markings based on the number of tokens in places and transition enabledness, inspired by the logic

Table 1. Interesting transitions of φ (assuming $M \not\models \varphi$, otherwise $A_M(\varphi) = \emptyset$)

| Formula φ | $A_M(\varphi)$ | $A_M(\neg\varphi)$ |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <i>deadlock</i> | $(\bullet t) \bullet \cup \bullet (\circ t)$ for some $t \in \text{En}(M)$ | \emptyset |
| t | $\bullet p$ for some $p \in \bullet t$ where $M(p) < w((p, t))$ or $p \bullet$ for some $p \in \circ t$ where $M(p) \geq w((p, t))$ | $(\bullet t) \bullet \cup \bullet (\circ t)$ |
| $e_1 < e_2$ | $\text{decr}_M(e_1) \cup \text{incr}_M(e_2)$ | $A_M(e_1 \geq e_2)$ |
| $e_1 \leq e_2$ | $\text{decr}_M(e_1) \cup \text{incr}_M(e_2)$ | $A_M(e_1 > e_2)$ |
| $e_1 > e_2$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_2)$ | $A_M(e_1 \leq e_2)$ |
| $e_1 \geq e_2$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_2)$ | $A_M(e_1 < e_2)$ |
| $e_1 = e_2$ | $\text{decr}_M(e_1) \cup \text{incr}_M(e_2)$ if $\text{eval}_M(e_1) > \text{eval}_M(e_2)$ $\text{incr}_M(e_1) \cup \text{decr}_M(e_2)$ if $\text{eval}_M(e_1) < \text{eval}_M(e_2)$ | $A_M(e_1 \neq e_2)$ |
| $e_1 \neq e_2$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_1) \cup \text{incr}_M(e_2) \cup \text{decr}_M(e_2)$ | $A_M(e_1 = e_2)$ |
| $\varphi_1 \wedge \varphi_2$ | $A_M(\varphi_i)$ for some $i \in \{1, 2\}$ where $M \not\models \varphi_i$ | $A_M(\neg\varphi_1 \vee \neg\varphi_2)$ |
| $\varphi_1 \vee \varphi_2$ | $A_M(\varphi_1) \cup A_M(\varphi_2)$ | $A_M(\neg\varphi_1 \wedge \neg\varphi_2)$ |

Table 2. Increasing and decreasing transitions of expression e

| Expression e | $\text{incr}_M(e)$ | $\text{decr}_M(e)$ |
|----------------|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| c | \emptyset | \emptyset |
| p | $\bullet p$ | $p \bullet$ |
| $e_1 + e_2$ | $\text{incr}_M(e_1) \cup \text{incr}_M(e_2)$ | $\text{decr}_M(e_1) \cup \text{decr}_M(e_2)$ |
| $e_1 - e_2$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_2)$ | $\text{decr}_M(e_1) \cup \text{incr}_M(e_2)$ |
| $e_1 * e_2$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_1) \cup$ $\text{incr}_M(e_2) \cup \text{decr}_M(e_2)$ | $\text{incr}_M(e_1) \cup \text{decr}_M(e_1) \cup$ $\text{incr}_M(e_2) \cup \text{decr}_M(e_2)$ |

used in the Model Checking Contest (MCC) Property Language [27]. Let $N = (P, T, T_{\text{urg}}, IA, OA, g, w, \text{Type}, I)$ be a TAPN. The formulae of the logic are given by the abstract syntax:

$\varphi ::= \text{deadlock} \mid t \mid e_1 \bowtie e_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi$

$e ::= c \mid p \mid e_1 \oplus e_2$

where $t \in T$, $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $c \in \mathbb{Z}$, $p \in P$, and $\oplus \in \{+, -, *\}$. Let Φ be the set of all such formulae and let E_N be the set of arithmetic expressions over the net N . The semantics of φ in a marking $M \in \mathcal{M}(N)$ is given by

$$\begin{array}{ll}
 M \models \text{deadlock} & \text{if } \text{En}(M) = \emptyset \\
 M \models t & \text{if } t \in \text{En}(M) \\
 M \models e_1 \bowtie e_2 & \text{if } \text{eval}_M(e_1) \bowtie \text{eval}_M(e_2)
 \end{array}$$

assuming a standard semantics for Boolean operators and where the semantics of arithmetic expressions in a marking M is as follows: $\text{eval}_M(c) = c$, $\text{eval}_M(p) = |M(p)|$, and $\text{eval}_M(e_1 \oplus e_2) = \text{eval}_M(e_1) \oplus \text{eval}_M(e_2)$.

Let φ be a formula. We are interested in the question, whether we can reach from the initial marking some of the goal markings from $G_\varphi = \{M \in \mathcal{M}(N) \mid M \models \varphi\}$. In order to guide the reduction such that transitions that lead to the goal markings are included in the generated stubborn set, we define the notion of *interesting transitions* for a marking M relative to φ , and we let $A_M(\varphi) \subseteq T$ denote the set of interesting transitions. Formally, we shall require that whenever $M \xrightarrow{w} M'$ via a sequence of transitions $w = t_1 t_2 \dots t_n \in T^*$ where $M \notin G_\varphi$ and $M' \in G_\varphi$, then there must exist i , $1 \leq i \leq n$, such that $t_i \in A_M(\varphi)$.

Table 1 gives a possible definition of $A_M(\varphi)$. Let us remark that the definition is at several places nondeterministic, allowing for a variety of sets of interesting transitions. Table 1 uses the functions $incr_M : E_N \rightarrow 2^T$ and $decr_M : E_N \rightarrow 2^T$ defined in Table 2. These functions take as input an expression e , and return all transitions that can possibly, when fired, increase resp. decrease the evaluation of e . The following lemma formally states the required property of the functions $incr_M$ and $decr_M$.

Lemma 1. *Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPN and $M \in \mathcal{M}(N)$ a marking. Let $e \in E_N$ and let $M \xrightarrow{w} M'$ where $w = t_1 t_2 \dots t_n \in T^*$.*

- *If $eval_M(e) < eval_{M'}(e)$ then there is i , $1 \leq i \leq n$, such that $t_i \in incr_M(e)$.*
- *If $eval_M(e) > eval_{M'}(e)$ then there is i , $1 \leq i \leq n$, such that $t_i \in decr_M(e)$.*

We finish this section with the main technical lemma, showing that at least one interesting transition must be fired before we can reach a marking satisfying a given reachability formula.

Lemma 2. *Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPN, let $M \in \mathcal{M}(N)$ be its marking and let $\varphi \in \Phi$ be a given formula. If $M \not\models \varphi$ and $M \xrightarrow{w} M'$ where $w \in \overline{A_M(\varphi)}^*$ then $M' \not\models \varphi$.*

4 Partial Order Reductions for TAPN

We are now ready to state the main theorem that provides sufficient syntax-driven conditions for a reduction in order to guarantee preservation of reachability. Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPN, let $M \in \mathcal{M}(N)$ be a marking of N , and let $\varphi \in \Phi$ be a formula. We recall that $A_M(\varphi)$ is the set of interesting transitions as defined earlier.

Theorem 2 (Reachability Preserving Closure). *Let St be a reduction such that for all $M \in \mathcal{M}(N)$ it satisfies the following conditions.*

- 1 *If $\neg \mathbf{zt}(M)$ then $\mathbf{En}(M) \subseteq \mathbf{St}(M)$.*
- 2 *If $\mathbf{zt}(M)$ then $A_M(\varphi) \subseteq \mathbf{St}(M)$.*
- 3 *If $\mathbf{zt}(M)$ then either*
 - (a) *there is $t \in T_{urg} \cap \mathbf{En}(M) \cap \mathbf{St}(M)$ where $\bullet(^{\circ}t) \subseteq \mathbf{St}(M)$, or*
 - (b) *there is $p \in P$ where $I(p) = [a, b]$ and $b \in M(p)$ such that $t \in \mathbf{St}(M)$ for every $t \in p^\bullet$ where $b \in g((p, t))$.*

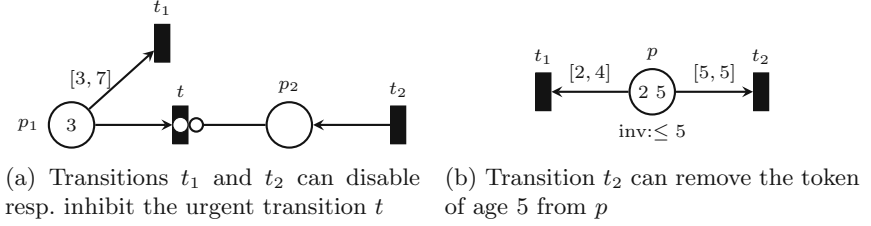


Fig. 2. Cases for Condition 3

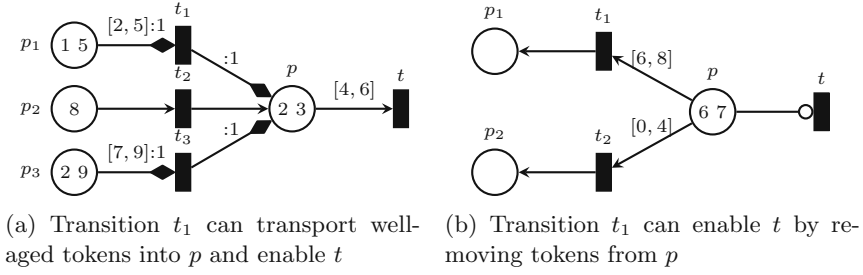
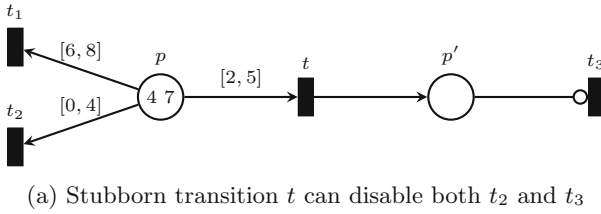
- 4 For all $t \in \text{St}(M) \setminus \text{En}(M)$ either
 - (a) there is $p \in \bullet t$ such that $|\{x \in M(p) \mid x \in g((p, t))\}| < w((p, t))$ and
 - $t' \in \text{St}(M)$ for all $t' \in \bullet p$ where there is $p' \in \bullet t'$ with $\text{Type}((t', p)) = \text{Type}((p', t')) = \text{Transport}_j$ and where $g((p', t')) \cap g((p, t)) \neq \emptyset$, and
 - if $0 \in g((p, t))$ then also $\bullet p \subseteq \text{St}(M)$, or
 - (b) there is $p \in {}^\circ t$ where $|M(p)| \geq w((p, t))$ such that
 - $t' \in \text{St}(M)$ for all $t' \in p^\bullet$ where $M(p) \cap g((p, t')) \neq \emptyset$.
- 5 For all $t \in \text{St}(M) \cap \text{En}(M)$ we have
 - (a) $t' \in \text{St}(M)$ for every $t' \in p^\bullet$ where $p \in \bullet t$ and $g((p, t)) \cap g((p, t')) \neq \emptyset$, and
 - (b) $(t^\bullet)^\circ \subseteq \text{St}(M)$.

Then St satisfies \mathcal{Z} , \mathcal{D} , \mathcal{R} , and \mathcal{W} .

Let us now briefly discuss the conditions of Theorem 2. Clearly, Condition 1 ensures that if time can elapse, we include all enabled transitions into the stubborn set and Condition 2 guarantees that all interesting transitions (those that can potentially make the reachability proposition true) are included as well.

Condition 3 makes sure that if time elapsing is disabled then any transition that can possibly enable time elapsing will be added to the stubborn set. There are two situations how time progress can be disabled. Either, there is an urgent enabled transition, like the transition t in Fig. 2a. Since t_2 can add a token to p_2 and by that inhibit t , Condition 3a makes sure that t_2 is added into the stubborn set in order to satisfy \mathcal{D} . As t_1 can remove the token of age 3 from p_1 and hence disable t , we must add t_1 to the stubborn set too (guaranteed by Condition 5a). The other situation when time gets stopped is when a place with an age invariant contains a token that disallows time passing, like in Fig. 2b where time is disabled because the place p has a token of age 5, which is the maximum possible age of tokens in p due to the age invariant. Since t_2 can remove the token of age 5 from p , we include it to the stubborn set due to Condition 3b. On the other hand t_1 does not have to be included in the stubborn set as its firing cannot remove the token of age 5 from p .

Condition 4 makes sure that an disabled stubborn transition can never be enabled by a non-stubborn transition. There are two reasons why a transition is disabled. Either, as in Fig. 3a where t is disabled, there is an insufficient number of tokens of appropriate age to fire the transition. In this case, Condition 4a

**Fig. 3.** Cases for Condition 4**Fig. 4.** Cases for Condition 5

makes sure that transitions that can add tokens of a suitable age via transport arcs are included in the stubborn set. This is the case for the transition t_1 in our example, as $[2, 5]$ has a nonempty intersection with $[4, 6]$. On the other hand, t_3 does not have to be added. As the transition t_2 only adds fresh tokens of age 0 to p via normal arcs, there is no need to add t_2 into the stubborn set either. The other reason for a transition to be disabled is due to inhibitor arcs, as shown on the transition t in Fig. 3b. Condition 4b makes sure that t_1 is added to the stubborn set, as it can enable t (the interval $[6, 8]$ has a nonempty intersection with the tokens of age 6 and 7 in the place p). As this is not the case for t_2 , this transition can be left out from the stubborn set.

Finally, Condition 5 guarantees that enabled stubborn transitions can never disable any non-stubborn transitions. For an illustration, take a look at Fig. 4a and assume that t is an enabled stubborn transition. Firing of t can remove the token of age 4 from p and disable t_2 , hence t_2 must become stubborn by Condition 5a in order to satisfy \mathcal{W} . On the other hand, the intervals $[6, 8]$ and $[2, 5]$ have empty intersection, so there is no need to declare t_1 as a stubborn transition. Moreover, firing of t can also disable the transition t_3 due to the inhibitor arc, so we must add t_3 to the stubborn set by Condition 5b.

The conditions of Theorem 2 can be turned into an iterative saturation algorithm for the construction of stubborn sets as shown in Algorithm 1. When running this algorithm for the net in our running example, we can reduce the state space exploration for fireability of the transition t as depicted in Fig. 1b. Our last theorem states that the algorithm returns stubborn subsets of enabled

Algorithm 1. Construction of a reachability preserving stubborn set

```

input      :  $N = (P, T, T_{urg}, IA, OA, g, w, Type, I), M \in \mathcal{M}(N), \varphi \in \Phi$ 
output    :  $St(M) \cap En(M)$ 
1 if  $\neg zt(M)$  then
2   return  $En(M)$ ;
3  $X := \emptyset; Y := A_M(\varphi)$ ;
4 if  $T_{urg} \cap En(M) \neq \emptyset$  then
5   pick any  $t \in T_{urg} \cap En(M)$ ;
6   if  $t \notin Y$  then
7      $Y := Y \cup \{t\}$ ;
8    $Y := Y \cup \bullet(t)$ ;
9 else
10  pick any  $p \in P$  where  $I(p) = [a, b]$  and  $b \in M(p)$ 
11  forall  $t \in p^\bullet$  do
12    if  $b \in g((p, t))$  then
13       $Y := Y \cup \{t\}$ ;
14 while  $Y \neq \emptyset$  do
15   pick any  $t \in Y$ ;
16   if  $t \notin En(M)$  then
17     if  $\exists p \in \bullet t. |\{x \in M(p) \mid x \in g((p, t))\}| < w((p, t))$  then
18       pick any such  $p$ ;
19       forall  $t' \in p^\bullet \setminus X$  do
20         forall  $p' \in \bullet t'$  do
21           if  $Type((t', p)) = Type((p', t')) =$   

 $Transport_j \wedge g((p', t')) \cap g((p, t)) \neq \emptyset$  then
22              $Y := Y \cup \{t'\}$ ;
23       if  $0 \in g((p, t))$  then
24          $Y := Y \cup (\bullet p \setminus X)$ ;
25     else
26       pick any  $p \in {}^\circ t$  s.t.  $|M(p)| \geq w((p, t))$ ;
27       forall  $t' \in p^\bullet \setminus X$  do
28         if  $M(p) \cap g((p, t')) \neq \emptyset$  then
29            $Y := Y \cup \{t'\}$ ;
30   else
31     forall  $p \in \bullet t$  do
32        $Y := Y \cup (\{t' \in p^\bullet \mid g((p, t)) \cap g((p, t')) \neq \emptyset\} \setminus X)$ ;
33      $Y := Y \cup ((t^\bullet)^\circ \setminus X)$ ;
34    $Y := Y \setminus \{t\}$ ;
35    $X := X \cup \{t\}$ ;
36 return  $X \cap En(M)$ ;

```

transitions that satisfy the four conditions of Theorem 1 and hence we preserve the reachability property as well as the minimum path to some reachable goal.

Theorem 3. *Algorithm 1 terminates and returns $\text{St}(M) \cap \text{En}(M)$ for some reduction St that satisfies \mathcal{Z} , \mathcal{D} , \mathcal{R} , and \mathcal{W} .*

5 Implementation and Experiments

We implemented our partial order method in C++ and integrated it within the model checker TAPAAL [19] and its discrete time engine `verifydtapn` [4, 11]. We evaluate our partial order reduction on a wide range of case studies.

PatientMonitoring. The patient monitoring system [17] models a medical system that through sensors periodically scans patient’s vital functions, making sure that abnormal situations are detected and reported within given deadlines. The timed-arc Petri net model was described in [17] for two sensors monitoring patient’s pulse rate and oxygen saturation level. We scale the case study by adding additional sensors. *BloodTransfusion.* This case study models a larger blood transfusion workflow [16], the benchmarking case study of the little-JIL language. The timed-arc Petri net model was described in [10] and we verify that the workflow is free of deadlocks (unless all sub-workflows correctly terminate). The problem is scaled by the number of patients receiving a blood transfusion. *FireAlarm.* This case study uses a modified (due to trade secrets) fire alarm system owned by a German company [20, 21]. It models a four-channel round-robin frequency-hopping transmission scheduling in order to ensure a reliable communication between a number of wireless sensors (by which the case study is scaled) and a central control unit. The protocol is based on time-division multiple access (TDMA) channel access and we verify that for a given frequency-jammer, it takes never more than three cycles before a fire alarm is communicated to the central unit. *BAwPC.* Business Activity with Participant Completion (BAwPC) is a web-service coordination protocol from WS-BA specification [33] that ensures a consistent agreement on the outcome of long-running distributed applications. In [26] it was shown that the protocol is flawed and a correct, enhanced variant was suggested. We model check this enhanced protocol and scale it by the capacity of the communication buffer. *Fischer.* Here we consider a classical Fischer’s protocol for ensuring mutual exclusion for a number of timed processes. The timed-arc Petri net model is taken from [2] and it is scaled by the number of processes. *LynchShavit.* This is another timed-based mutual exclusion algorithm by Lynch and Shavit, with the timed-arc Petri net model taken from [1] and scaled by the number of processes. *MPEG2.* This case study describes the workflow of the MPEG-2 video encoding algorithm run on a multicore processor (the timed-arc Petri net model was published in [35]) and we verify the maximum duration of the workflow. The model is scaled by the number of B frames in the IBⁿP frame sequence. *AlternatingBit.* This is a classical case study of alternating bit protocol, based on the timed-arc Petri net model given in [24]. The purpose of the protocol is to ensure a safe communication between a sender and a receiver over an unreliable medium. Messages are time-stamped in order to compensate

Table 3. Experiments with and without partial order reduction (POR)

| Model | Time (seconds) | | Markings $\times 1000$ | | Reduction | |
|---------------------|----------------|---------|------------------------|--------|-----------|-----------|
| | NORMAL | POR | NORMAL | POR | %Time | %Markings |
| PatientMonitoring 3 | 5.88 | 0.35 | 333 | 28 | 94 | 92 |
| PatientMonitoring 4 | 22.06 | 0.48 | 1001 | 36 | 98 | 96 |
| PatientMonitoring 5 | 80.76 | 0.65 | 3031 | 44 | 99 | 99 |
| PatientMonitoring 6 | 305.72 | 0.85 | 9248 | 54 | 100 | 99 |
| PatientMonitoring 7 | 5516.93 | 5.75 | 130172 | 318 | 100 | 100 |
| BloodTransfusion 2 | 0.32 | 0.41 | 48 | 43 | -28 | 11 |
| BloodTransfusion 3 | 7.88 | 6.45 | 792 | 546 | 18 | 31 |
| BloodTransfusion 4 | 225.18 | 109.30 | 14904 | 7564 | 51 | 49 |
| BloodTransfusion 5 | 5256.01 | 1611.14 | 248312 | 94395 | 69 | 62 |
| FireAlarm 10 | 28.95 | 14.17 | 796 | 498 | 51 | 37 |
| FireAlarm 12 | 116.97 | 17.51 | 1726 | 526 | 85 | 70 |
| FireAlarm 14 | 598.89 | 21.65 | 5367 | 554 | 96 | 90 |
| FireAlarm 16 | 5029.25 | 29.48 | 19845 | 582 | 99 | 97 |
| FireAlarm 18 | 27981.90 | 34.55 | 77675 | 610 | 100 | 99 |
| FireAlarm 20 | 154495.29 | 41.47 | 308914 | 638 | 100 | 100 |
| FireAlarm 80 | >2 days | 602.71 | — | 1522 | — | — |
| FireAlarm 125 | >2 days | 1957.00 | — | 2260 | — | — |
| BAwPC 2 | 0.21 | 0.41 | 19 | 16 | -95 | 15 |
| BAwPC 4 | 3.45 | 4.04 | 193 | 125 | -17 | 35 |
| BAwPC 6 | 23.01 | 17.08 | 900 | 452 | 26 | 50 |
| BAwPC 8 | 73.73 | 39.29 | 2294 | 952 | 47 | 58 |
| BAwPC 10 | 135.62 | 60.66 | 3819 | 1412 | 55 | 63 |
| BAwPC 12 | 173.09 | 73.53 | 4736 | 1665 | 58 | 65 |
| Fischer-9 | 3.24 | 2.37 | 281 | 233 | 27 | 17 |
| Fischer-11 | 12.68 | 8.73 | 923 | 738 | 31 | 20 |
| Fischer-13 | 42.52 | 28.53 | 2628 | 2041 | 33 | 22 |
| Fischer-15 | 121.31 | 77.50 | 6700 | 5066 | 36 | 24 |
| Fischer-17 | 313.69 | 198.36 | 15622 | 11536 | 37 | 26 |
| Fischer-19 | 748.52 | 456.30 | 33843 | 24469 | 39 | 28 |
| Fischer-21 | 1622.69 | 985.07 | 68934 | 48904 | 39 | 29 |
| LynchShavit 9 | 3.98 | 3.31 | 282 | 234 | 17 | 17 |
| LynchShavit 11 | 15.73 | 12.19 | 925 | 740 | 23 | 20 |
| LynchShavit 13 | 51.08 | 37.97 | 2631 | 2043 | 26 | 22 |
| LynchShavit 15 | 146.63 | 103.63 | 6703 | 5069 | 29 | 24 |
| LynchShavit 17 | 384.52 | 258.09 | 15626 | 11540 | 33 | 26 |
| LynchShavit 19 | 907.60 | 597.68 | 33848 | 24474 | 34 | 28 |
| LynchShavit 21 | 2011.58 | 1307.72 | 68940 | 48910 | 35 | 29 |
| MPEG2 3 | 13.17 | 15.43 | 2188 | 2187 | -17 | 0 |
| MPEG2 4 | 109.62 | 125.45 | 15190 | 15180 | -14 | 0 |
| MPEG2 5 | 755.54 | 840.84 | 87568 | 87478 | -11 | 0 |
| MPEG2 6 | 4463.19 | 5092.58 | 435023 | 434354 | -14 | 0 |
| AlternatingBit 20 | 9.17 | 9.51 | 617 | 617 | -4 | 0 |
| AlternatingBit 30 | 48.20 | 49.13 | 2804 | 2804 | -2 | 0 |
| AlternatingBit 40 | 161.18 | 162.94 | 8382 | 8382 | -1 | 0 |
| AlternatingBit 50 | 408.34 | 408.86 | 19781 | 19781 | 0 | 0 |

(via retransmission) for the possibility of losing messages. The case study is scaled by the maximum number of messages in transfer.

All experiments were run on AMD Opteron 6376 Processors with 500 GB memory. In Table 3 we compare the time to verify a model without (NORMAL) and with (POR) partial order reduction, the number of explored markings (in thousands) and the percentage of time and memory reduction. We can observe clear benefits of our technique on PatientMonitoring, BloodTransfusion and FireAlarm where we are both exponentially faster and explore only a fraction of all reachable markings. For example in FireAlarm, we are able to verify its correctness for all 125 sensors, as it is required by the German company [21]. This would be clearly unfeasible without the use of partial order reduction.

In BAwPC, we can notice that for the smallest instances, there is some computation overhead from computing the stubborn sets, however, it clearly pays off for the larger instances where the percentages of reduced state space are closely followed by the percentages of the verification times and in fact improve with the larger instances. Fischer and LynchShavit case studies demonstrate that even moderate reductions of the state space imply considerable reduction in the running time and computing the stubborn sets is well worth the extra effort.

MPEG2 is an example of a model that allows only negligible reduction of the state space size, and where we observe an actual slowdown in the running time due to the computation of the stubborn sets. Nevertheless, the overhead stays constant in the range of about 15%, even for increasing instance sizes. Finally, AlternatingBit protocol does not allow for any reduction of the state space (even though it contains age invariants) but the overhead in the running time is negligible.

We observed similar performance of our technique also for the cases where the reachability property does not hold and a counter example can be generated.

6 Conclusion

We suggested a simple, yet powerful and application-ready partial order reduction for timed systems. The reduction comes into effect as soon as the timed system enters an urgent configuration where time cannot elapse until a nonempty sequence of transitions gets executed. The method is implemented and fully integrated, including GUI support, into the open-source tool TAPAAL. We demonstrated its practical applicability on several case studies and conclude that computing the stubborn sets causes only a minimal overhead while providing large benefits for reducing the state space in numerous models. The method is not specific to stubborn reduction technique only and it preserves the shortest execution sequences. Moreover, once the time gets urgent, other classical (untimed) partial order approaches should be applicable too. Our method was instantiated to (unbounded) timed-arc Petri nets with discrete time semantics, however, we claim that the technique allows for general application to other modelling formalisms like timed automata and timed Petri nets, as well as an extension to continuous time. We are currently working on adapting the theory and providing

an efficient implementation for UPPAAL-style timed automata with continuous time semantics.

Acknowledgements. We thank Mads Johannsen for his help with the GUI support for partial order reduction. The work was funded by the center IDEA4CPS, Innovation Fund Denmark center DiCyPS and ERC Advanced Grant LASSO. The last author is partially affiliated with FI MU in Brno.

References

1. Abdulla, P., Deneux, J., Mahata, P., Nylén, A.: Using forward reachability analysis for verification of timed Petri nets. *Nord. J. Comput.* **14**, 1–42 (2007)
2. Abdulla, P.A., Nylén, A.: Timed Petri nets and BQOs. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 53–70. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45740-2_5
3. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
4. Andersen, M., Gatten Larsen, H., Srba, J., Grund Sørensen, M., Haahr Taankvist, J.: Verification of liveness properties on closed timed-arc Petri nets. In: Kučera, A., Henzinger, T.A., Nešetřil, J., Vojnar, T., Antoš, D. (eds.) MEMICS 2012. LNCS, vol. 7721, pp. 69–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36046-6_8
5. André, E., Chatain, T., Rodríguez, C.: Preserving partial-order runs in parametric time Petri nets. *ACM Trans. Embed. Comput. Syst.* **16**(2), 43:1–43:26 (2017)
6. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
7. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. *STTT* **8**(3), 204–215 (2006)
8. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055643>
9. Berthomieu, B., Vernadat, F.: Time Petri nets analysis with TINA. In: Third International Conference on Quantitative Evaluation of Systems, pp. 123–124. IEEE Computer Society (2006)
10. Bertolini, C., Liu, Z., Srba, J.: Verification of timed healthcare workflows using component timed-arc Petri nets. In: Weber, J., Perseil, I. (eds.) FHIES 2012. LNCS, vol. 7789, pp. 19–36. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39088-3_2
11. Viesmose Birch, S., Stig Jacobsen, T., Jon Jensen, J., Moesgaard, C., Nørgaard Samuelsen, N., Srba, J.: Interval abstraction refinement for model checking of timed-arc Petri nets. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 237–251. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10512-3_17
12. Boucheneb, H., Barkaoui, K.: Reducing interleaving semantics redundancy in reachability analysis of time Petri nets. *ACM Trans. Embed. Comput. Syst.* **12**(1), 7:1–7:24 (2013)
13. Boucheneb, H., Barkaoui, K.: Stubborn sets for time Petri nets. *ACM Trans. Embed. Comput. Syst.* **14**(1), 11:1–11:25 (2015)

14. Boucheneb, H., Barkaoui, K.: Delay-dependent partial order reduction technique for real time systems. *Real-Time Syst.* **54**, 278–306 (2017)
15. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) *SFM-RT 2004*. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_8
16. Christov, S., Avrunin, G., Clarke, A., Osterweil, L., Henneman, E.: A benchmark for evaluating software engineering techniques for improving medical processes. In: *SEHC 2010*, pp. 50–56. ACM (2010)
17. Cicirelli, F., Furfaro, A., Nigro, L.: Model checking time-dependent system specifications using time stream Petri nets and UPPAAL. *Appl. Math. Comput.* **218**(16), 8160–8186 (2012)
18. Dams, D., Gerth, R., Knaack, B., Kuiper, R.: Partial-order reduction techniques for real-time model checking. *Form. Asp. Comput.* **10**(5–6), 469–482 (1998)
19. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: Flanagan, C., König, B. (eds.) *TACAS 2012*. LNCS, vol. 7214, pp. 492–497. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_36
20. Feo-Arenis, S., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A.S.: The wireless fire alarm system: ensuring conformance to industrial standards through formal verification. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) *FM 2014*. LNCS, vol. 8442, pp. 658–672. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_44
21. Feo-Arenis, S., Westphal, B., Dietsch, D., Muñiz, M., Andisha, S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal verification. *Form. Asp. Comput.* **28**(3), 499–527 (2016)
22. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Romeo: a tool for analyzing time Petri nets. In: Etesami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_41
23. Hansen, H., Lin, S.-W., Liu, Y., Nguyen, T.K., Sun, J.: Diamonds are a girl's best friend: partial order reduction for timed automata with abstractions. In: Biere, A., Bloem, R. (eds.) *CAV 2014*. LNCS, vol. 8559, pp. 391–406. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_26
24. Jacobsen, L., Jacobsen, M., Møller, M.H., Srba, J.: Verification of timed-arc Petri Nets. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Kráľović, R., Vukolić, M., Wolf, S. (eds.) *SOFSEM 2011*. LNCS, vol. 6543, pp. 46–72. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18381-2_4
25. Jensen, P., Larsen, K., Srba, J.: Discrete and continuous strategies for timed-arc Petri net games. *Int. J. Softw. Tools Technol. Transf. (STTT)*, 1–18 (2017, to appear). Online since September 2017
26. Marques Jr., A., Ravn, A., Srba, J., Vighio, S.: Model-checking web services business activity protocols. *Int. J. Softw. Tools Technol. Transf. (STTT)* **15**(2), 125–147 (2012)
27. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Chiardo, G., Hamez, A., Jezequel, L., Miner, A., Meijer, J., Paviot-Adet, E., Racordon, D., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Trinh, G., Wolf, K.: Complete Results for the 2016 Edition of the Model Checking Contest, June 2016. <http://mcc.lip6.fr/2016/results.php>
28. Kristensen, L.M., Schmidt, K., Valmari, A.: Question-guided stubborn set methods for state properties. *Form. Methods Syst. Des.* **29**(3), 215–251 (2006)
29. Lilius, J.: Efficient state space search for time Petri nets. *Electron. Notes Theo. Comput. Sci.* **18**, 113–133 (1998). MFCS 1998 Workshop on Concurrency

30. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. *Theor. Comput. Sci.* **345**(1), 27–59 (2005)
31. Mateo, J., Srba, J., Sørensen, M.: Soundness of timed-arc workflow nets in discrete and continuous-time semantics. *Fundam. Inform.* **140**(1), 89–121 (2015)
32. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) *CONCUR 1999*. LNCS, vol. 1664, pp. 431–446. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_30
33. Newcomer, E., Robinson, I.: Web services business activity (WS-businessactivity) version 1.2 (2009). <http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html>
34. Niebert, P., Qu, H.: Adding invariants to event zone automata. In: Asarin, E., Bouyer, P. (eds.) *FORMATS 2006*. LNCS, vol. 4202, pp. 290–305. Springer, Heidelberg (2006). https://doi.org/10.1007/11867340_21
35. Pelayo, F., Cuartero, F., Valero, V., Macia, H., Pelayo, M.: Applying timed-arc Petri nets to improve the performance of the MPEG-2 encoding algorithm. In: 10th International Multimedia Modelling Conference, pp. 49–56. IEEE Computer Society (2004)
36. Perin, M., Faure, J.: Coupling timed plant and controller models with urgent transitions without introducing deadlocks. In: 17th International Conference on Emerging Technologies and Factory Automation (ETFA 2012), pp. 1–9. IEEE (2012)
37. Salah, R.B., Bozga, M., Maler, O.: On interleaving in timed automata. In: Baier, C., Hermanns, H. (eds.) *CONCUR 2006*. LNCS, vol. 4137, pp. 465–476. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_31
38. Sloan, R.H., Buy, U.: Stubborn sets for real-time Petri nets. *Form. Methods Syst. Des.* **11**(1), 23–40 (1997)
39. Valmari, A., Hansen, H.: Stubborn set intuition explained. In: Koutny, M., Kleijn, J., Penczek, W. (eds.) *Transactions on Petri Nets and Other Models of Concurrency XII*. LNCS, vol. 10470, pp. 140–165. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-55862-1_7
40. Virbitskaite, I., Pokozy, E.: A partial order method for the verification of time Petri nets. In: Ciobanu, G., Păun, G. (eds.) *FCT 1999*. LNCS, vol. 1684, pp. 547–558. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48321-7_46
41. Yoneda, T., Schlingloff, B.-H.: Efficient verification of parallel real-time systems. *Form. Methods Syst. Des.* **11**(2), 187–215 (1997)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





A Counting Semantics for Monitoring LTL Specifications over Finite Traces

Ezio Bartocci^{1(✉)}, Roderick Bloem², Dejan Nickovic³, and Franz Roeck²

¹ TU Wien, Vienna, Austria
ezio.bartocci@tuwien.ac.at

² Graz University of Technology, Graz, Austria

³ Austrian Institute of Technology GmbH, Vienna, Austria

Abstract. We consider the problem of monitoring a Linear Time Logic (LTL) specification that is defined on infinite paths, over finite traces. For example, we may need to draw a verdict on whether the system satisfies or violates the property “p holds infinitely often.” The problem is that there is always a continuation of a finite trace that satisfies the property and a different continuation that violates it.

We propose a two-step approach to address this problem. First, we introduce a counting semantics that computes the number of steps to witness the satisfaction or violation of a formula for each position in the trace. Second, we use this information to make a prediction on inconclusive suffixes. In particular, we consider a *good* suffix to be one that is shorter than the longest witness for a satisfaction, and a *bad* suffix to be shorter than or equal to the longest witness for a violation. Based on this assumption, we provide a verdict assessing whether a continuation of the execution on the same system will presumably satisfy or violate the property.

1 Introduction

Alice is a verification engineer and she is presented with a new exciting and complex design. The requirements document coming with the design already incorporates functional requirements formalized in Linear Temporal Logic (LTL) [13]. The design contains features that are very challenging for exhaustive verification and her favorite model checking tool does not terminate in reasonable time.

This work was partially supported by the European Union (IMMORTAL project, grant no. 644905), the Austrian FWF (National Research Network RiSE/SHiNE S11405-N23 and S11406-N23), the SeCludE project (funded by UnivPM) and the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Unions HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

Runtime Verification. Alice decides to tackle this problem using runtime verification (RV) [3], a light, yet rigorous verification method. RV drops the exhaustiveness of model checking and analyzes individual traces generated by the system. Thus, it scales much better to the industrial-size designs. RV enables automatic generation of monitors from formalized requirements and thus provides a systematic way to check if the system traces satisfy (violate) the specification.

Motivating Example. In particular, Alice considers the following specification:

$$\psi \equiv \mathbf{G}(\text{request} \rightarrow \mathbf{F} \text{ grant})$$

This LTL formula specifies that every request coming from the environment must be granted by the design in some finite (but unbounded) future. Alice realizes that she is trying to check a *liveness* property over a set of *finite* traces. She looks closer at the executions and identifies the two interesting examples trace τ_1 and trace τ_2 , depicted in Table 1.

The monitoring tool reports that both τ_1 and τ_2 presumably violate the unbounded response property. This verdict is against Alice’s intuition. The evaluation of trace τ_1 seems right to her – the request at Cycle 1 is followed by a grant at Cycle 3, however the request at Cycle 4 is never granted during that execution. There are good reasons to suspect a bug in the design. Then she looks at τ_2 and observes that after every **request** the **grant** is given exactly after 2 cycles. It is true that the last request at Cycle 7 is not followed by a grant, but this seems to happen because the execution ends at that cycle – the past trace observations give reason to think that this request would be followed by a grant in cycle 9 if the execution was continued. Thus, Alice is not satisfied by the second verdict.

Alice looks closer at the way that the LTL property is evaluated over finite traces. She finds out that temporal operators are given *strength* – *eventually* and *until* are declared as *strong* operators, while *always* and *weak until* are defined to be *weak* [9]. A strong temporal operator requires all outstanding obligations to be met before the end of the trace. In contrast, a weak temporal operator must not witness any outstanding obligation violation before the end of the trace. Under this interpretation, both τ_1 and τ_2 violate the unbounded response property.

Alice explores another popular approach to evaluate future temporal properties over finite traces – the 3-valued semantics for LTL [4]. In this setting, the Boolean set of verdicts is extended with a third **unknown** (or **maybe**) value. A finite trace satisfies (violates) the 3-valued LTL formula if and only if all the infinite extensions of the trace satisfy (violate) the same LTL formula under its classical interpretation. In all other cases, we say that the satisfaction of the formula by the trace is **unknown**. Alice applies the 3-valued interpretation of LTL on the traces τ_1 and τ_2 to evaluate the unbounded response property. In

Table 1. Unbounded response property example.

| trace | time | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---------|---|---|---|---|---|---|---|
| τ_1 | request | T | – | – | T | – | – | – |
| | grant | – | – | T | – | – | – | – |
| τ_2 | request | T | – | – | T | – | – | T |
| | grant | – | – | T | – | – | T | – |

We use “–” instead of “ \perp ” to improve the trace readability.

both situations, she ends up with the **unknown** verdict. Once again, this is not what she expects and it does not meet her intuition about the satisfaction of the formula by the observed traces.

Alice desires a semantics that evaluates LTL properties on finite traces by taking previous observations into account.

Contributions. In this paper, we study the problem of LTL evaluation over finite traces encountered by Alice and propose a solution. We introduce a new counting semantics for LTL that takes into account the intuition illustrated by the example from Table 1. This semantics computes for every position of a trace two values – the distances to the nearest satisfaction and violation of the co-safety, respectively safety, part of the specification. We use this quantitative information to make *predictions* about the (infinite) suffixes of the finite observations. We infer from these values the maximum time that we expect for a future obligation to be fulfilled. We compare it to the value that we have for an open obligation at the end of the trace. If the latter is greater (smaller) than the expected maximum value, we have a good indication of a *presumed violation (satisfaction)* that we report to the user. In particular, our approach will indicate that τ_1 is likely to violate the specification and should be further inspected. In contrast, it will evaluate that τ_2 most likely satisfies the unbounded response property.

Organization of the Paper. The rest of the paper is organized as follows. We discuss the related work in Sect. 2 and we provide the preliminaries in Sect. 3. In Sect. 4 we present our new counting semantics for LTL and we show how to make *predictions* about (infinite) suffixes of the finite observations. Section 5 shows the application of our approach to some examples. Finally in Sect. 6 we draw our conclusions.

2 Related Work

The finitary interpretation of LTL was first considered in [11], where the authors propose to enrich the logic with the *weak* next operator that is dual to the (strong) next operator defined on infinite traces. While the strong next requires the existence of a next state, the weak next trivially evaluates to true at the end of the trace. In [9], the authors propose a more semantic approach with *weak* and *strong* views for evaluating future obligations at the end of the trace. In essence the empty word satisfies (violates) every formula according to the weak (strong) view. These two approaches result in the violation of the specification ψ by both traces τ_1 and τ_2 .

The authors in [4] propose a 3-valued finitary LTL interpretation of LTL, in which the set $\{\text{true}, \text{false}\}$ of verdicts is extended with a third **inconclusive** verdict. According to the 3-valued LTL, a finite trace satisfies (violates) a specification iff all its infinite extensions satisfy (violate) the same property under the classical LTL interpretation. Otherwise, it evaluates to **inconclusive**. The main disadvantage of the 3-valued semantics is the dominance of the **inconclusive** verdict in

the evaluation of many interesting LTL formulas. In fact, both τ_1 and τ_2 from Table 1 evaluate to **inconclusive** against the unbounded response specification ψ .

In [5], the authors combine the weak and strong operators with the 3-valued semantics to refine the **inconclusive** with **{presumably true, presumably false}**. The strength of the remaining future obligation dictates the **presumable** verdict. The authors in [12] propose a finitary semantics for each of the LTL (safety, liveness, persistence and recurrence) hierarchy classes that asymptotically converges to the infinite traces semantics of the logic. In these two works, the specification ψ also evaluates to the same verdict for both the traces τ_1 and τ_2 .

To summarize, none of the related work handles the unbounded response example from Table 1 in a satisfactory manner. This is due to the fact that these approaches decide about the verdict based on the specification and its remaining future obligations at the end of the trace. In contrast, we propose an approach in which the past observations within the trace are used to predict the future and derive the appropriate verdict. In particular, the application of our semantics for the evaluation of ψ over τ_1 and τ_2 results in **presumably true** and **presumably false** verdicts.

In [17], the authors propose another predictive semantics for LTL. In essence, this work assumes that at every point in time the monitor is able to precisely predict a segment of the trace that it has not observed yet and produce its outcome accordingly. In order to ensure such predictive power, this approach requires a white-box setting in which instrumentation and some form of static analysis of the systems are needed in order to foresee in advance the upcoming observations. This is in contrast to our work, in which the monitor remains a passive participant and predicts its verdict only based on the past observations.

In a different research thread [15], the authors introduce the notion of *monitorable* specifications that can be positively or negatively determined by a finite trace. The monitorability of LTL is further studied in [6, 14]. This classification of specifications is orthogonal to our work. We focus on providing a sensible evaluation to all LTL properties, including the non-monitorable ones (e.g., $\text{GF } p$).

We also mention the recent work on statistical model checking for LTL [8]. In this work, the authors assume a gray-box setting, where the system-under-test (SUT) is a Markov chain with the known minimum transition probability. This is in contrast to our work, in which we passively observe existing finite traces generated by the SUT, i.e., we have a blackbox setting.

In [1], the authors propose extending LTL with a discounting operator and study the properties of the augmented logic. The LTL specification formalism is extended with path-accumulation assertions in [7]. These LTL extensions are motivated by the need for a more quantitative and refined analysis of the systems. In our work, the motivation for the counting semantics is quite different. We use the quantitative information that we collect during the execution of the trace to predict the future behavior of the system and thus improve the quality of the monitoring verdict.

3 Preliminaries

We first introduce *traces* and Linear Temporal Logic (LTL) that we interpret over 3-valued semantics.

Definition 1 (Trace). *Let P a finite set of propositions and let $\Pi = 2^P$. A (finite or infinite) trace π is a sequence $\pi_1, \pi_2, \dots \in \Pi^* \cup \Pi^\omega$. We denote by $|\pi| \in \mathbb{N} \cup \{\infty\}$ the length of π . We denote by $\pi \cdot \pi'$ the concatenation of $\pi \in \Pi^*$ and $\pi' \in \Pi^* \cup \Pi^\omega$.*

Definition 2 (Linear Temporal Logic). *In this paper, we consider linear temporal logic (LTL) and we define its syntax by the grammar:*

$$\phi := p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathbf{X}\phi \mid \phi_1 \mathbf{U} \phi_2,$$

where $p \in P$. We denote by Φ the set of all LTL formulas.

From the basic definition we can derive other standard Boolean and temporal operators as follows:

$$\top = p \vee \neg p, \quad \perp = \neg\top, \quad \phi \wedge \psi = \neg(\neg\phi \vee \neg\psi), \quad \mathbf{F}\phi = \top \mathbf{U} \phi, \quad \mathbf{G}\phi = \neg\mathbf{F}\neg\phi$$

Let $\pi \in \Pi^\omega$ be an infinite trace and ϕ an LTL formula. The satisfaction relation $(\pi, i) \models \phi$ is defined inductively as follows

$$\begin{aligned} (\pi, i) &\models p && \text{iff } p \in \pi_i, \\ (\pi, i) &\models \neg\phi && \text{iff } (\pi, i) \not\models \phi, \\ (\pi, i) &\models \phi_1 \vee \phi_2 && \text{iff } (\pi, i) \models \phi_1 \text{ or } (\pi, i) \models \phi_2, \\ (\pi, i) &\models \mathbf{X}\phi && \text{iff } (\pi, i+1) \models \phi, \\ (\pi, i) &\models \phi_1 \mathbf{U} \phi_2 && \text{iff } \exists j \geq i \text{ s.t. } (\pi, j) \models \phi_2 \text{ and } \forall i \leq k < j, (\pi, k) \models \phi_1. \end{aligned}$$

We now recall the 3-valued semantics from [4]. We denote by $[\pi \models_3 \phi]$ the evaluation of ϕ with respect to the trace $\pi \in \Pi^*$ that yields a value in $\{\top, \perp, ?\}$.

$$[\pi \models_3 \phi] = \begin{cases} \top & \forall \pi' \in \Pi^\omega, \pi \cdot \pi' \models \phi, \\ \perp & \forall \pi' \in \Pi^\omega, \pi \cdot \pi' \not\models \phi, \\ ? & \text{otherwise.} \end{cases}$$

We now restrict LTL to a fragment without explicit \top and \perp symbols and with the explicit \mathbf{F} operator that we add to the syntax. We provide an alternative 3-valued semantics for this fragment, denoted by $\mu_\pi(\phi, i)$ where $i \in \mathbb{N}_{>0}$ indicates a position in or outside the trace. We assume the order $\perp < ? < \top$, and extend the Boolean operations to the 3-valued domain with the rules $\neg_3\top = \perp$, $\neg_3\perp = \top$ and $\neg_3? = ?$ and $\phi_1 \vee_3 \phi_2 = \max(\phi_1, \phi_2)$. We define the semantics inductively as follows:

$$\begin{aligned}
\mu_\pi(p, i) &= \begin{cases} \top & \text{if } i \leq |\pi| \text{ and } p \in \pi_i, \\ \perp & \text{else if } i \leq |\pi| \text{ and } p \notin \pi_i, \\ ? & \text{otherwise,} \end{cases} \\
\mu_\pi(\neg\phi, i) &= \neg_3 \mu_\pi(\phi, i), \\
\mu_\pi(\phi_1 \vee \phi_2, i) &= \mu_\pi(\phi_1, i) \vee_3 \mu_\pi(\phi_2, i), \\
\mu_\pi(\mathbf{X}\phi, i) &= \mu_\pi(\phi, i + 1), \\
\mu_\pi(\mathbf{F}\phi, i) &= \begin{cases} \mu_\pi(\phi, i) \vee_3 \mu_\pi(\mathbf{X}\mathbf{F}\phi, i) & \text{if } i \leq |\pi|, \\ \mu_\pi(\phi, i) & \text{if } i > |\pi|, \end{cases} \\
\mu_\pi(\phi_1 \mathbf{U} \phi_2, i) &= \begin{cases} \mu_\pi(\phi_2, i) \vee_3 (\mu_\pi(\phi_1, i) \wedge_3 \mu_\pi(\mathbf{X}(\phi_1 \mathbf{U} \phi_2), i)) & \text{if } i \leq |\pi|, \\ \mu_\pi(\phi_2, i) & \text{if } i > |\pi|. \end{cases}
\end{aligned}$$

We note that the adapted semantics allows evaluating a finite trace in polynomial time, in contrast to $[\pi \models_3 \phi]$, which requires a PSPACE-complete algorithm. This improvement in complexity comes at a price – the adapted semantics cannot semantically characterize tautologies and contradiction. We have for example that $\mu_\pi(p \vee \neg p, 1)$ for the empty word evaluates to $?$, despite the fact that $p \vee \neg p$ is semantically equivalent to \top . The novel semantics that we introduce in the following sections make the same tradeoff.

In the following lemma, we relate the two three-valued semantics.

Lemma 3. *Given an LTL formula and a trace $\pi \in \Pi^*$, $|\pi| \neq 0$, we have that*

$$\begin{aligned}
\mu_\pi(\phi, 1) = \top &\Rightarrow [\pi \models_3 \phi] = \top, \\
\mu_\pi(\phi, 1) = \perp &\Rightarrow [\pi \models_3 \phi] = \perp.
\end{aligned}$$

Proof. These two statements can be proven by induction on the structure of the LTL formula (see Appendix A.1 in [2]). $[\pi \models_3 \phi] = ? \Rightarrow \mu_\pi(\phi, 1) = ?$ is the consequence of the first two.

4 Counting Finitary Semantics for LTL

In this section, we introduce the counting semantics for LTL. We first provide necessary definitions in Sect. 4.1, we present the new semantics in Sect. 4.2 and finally propose a predictive mapping that transforms the counting semantics into a qualitative 5-valued verdict in Sect. 4.3.

4.1 Definitions

Let $\mathbb{N}_+ = \mathbb{N}_0 \cup \{\infty, -\}$ be the set of *natural* numbers (incl. 0) extended with the two special symbols ∞ (infinite) and $-$ (impossible) such that $\forall n \in \mathbb{N}_0$, we define $n < \infty < -$. We define the addition \oplus of two elements $a, b \in \mathbb{N}_+$ as follows.

Definition 4 (Operator \oplus). We define the binary operator $\oplus : \mathbb{N}_+ \times \mathbb{N}_+ \rightarrow \mathbb{N}_+$ s. t. for $a \oplus b$ with $a, b \in \mathbb{N}_+$ we have $a + b$ if $a, b \in \mathbb{N}_0$ and $\max\{a, b\}$ otherwise.

We denote by (s, f) a pair of two extended numbers $s, f \in \mathbb{N}_+$. In Definition 5, we introduce several operations on pairs: (1) the *swap* between the two values (\sim), (2) the increment by 1 of both values ($\oplus 1$), (3) the *minmax* binary operation (\sqcup) that gives the pair consisting of the minimum first value and the maximum second value, and (4) the *maxmin* binary operation (\sqcap) that is symmetric to (\sqcup).

Definition 7 introduces the counting semantics for LTL that for a finite trace π and LTL formula ϕ gives a pair $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$. We call s and f *satisfaction* and *violation witness counts*, respectively. Intuitively, the s (f) value denotes the minimal number of additional steps that is needed to witness the satisfaction (violation) of the formula. The value ∞ is used to denote that the property can be satisfied (violated) only in an infinite number of steps, while $-$ means the property cannot be satisfied (violated) by any continuation of the trace.

Definition 5 (Operations $\sim, \oplus 1, \sqcup, \sqcap$). Given two pairs $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$ and $(s', f') \in \mathbb{N}_+ \times \mathbb{N}_+$, we have:

$$\begin{aligned}\sim (s, f) &= (f, s), \\ (s, f) \oplus 1 &= (s \oplus 1, f \oplus 1), \\ (s, f) \sqcup (s', f') &= (\min(s, s'), \max(f, f')), \\ (s, f) \sqcap (s', f') &= (\max(s, s'), \min(f, f')).\end{aligned}$$

Example 6. Given the pairs $(0, 0)$, $(\infty, 1)$ and $(7, -)$ we have the following:

$$\begin{aligned}\sim (0, 0) &= (0, 0), & \sim (\infty, 1) &= (1, \infty), \\ (0, 0) \oplus 1 &= (1, 1), & (\infty, 1) \oplus 1 &= (\infty, 2), \\ (0, 0) \sqcup (\infty, 1) &= (0, 1), & (\infty, 1) \sqcup (7, -) &= (7, -), \\ (0, 0) \sqcap (\infty, 1) &= (\infty, 0), & (\infty, 1) \sqcap (7, -) &= (\infty, 1).\end{aligned}$$

Remark. Note that $\mathbb{N}_+ \times \mathbb{N}_+$ forms a lattice where $(s, f) \leq (s', f')$ when $s \geq s'$ and $f \leq f'$ with join \sqcup and meet \sqcap . Intuitively, larger values are closer to true.

4.2 Semantics

We now present our finitary semantics.

Definition 7 (Counting finitary semantics). Let $\pi \in \Pi^*$ be a finite trace, $i \in \mathbb{N}_{>0}$ be a position in or outside the trace and $\phi \in \Phi$ be an LTL formula. We define the counting finitary semantics of LTL as the function $d_\pi : \Phi \times \Pi^* \times \mathbb{N}_{>0} \rightarrow \mathbb{N}_+ \times \mathbb{N}_+$ such that:

$$\begin{aligned}
d_\pi(p, i) &= \begin{cases} (0, -) & \text{if } i \leq |\pi| \wedge p \in \pi_i, \\ (-, 0) & \text{if } i \leq |\pi| \wedge p \notin \pi_i, \\ (0, 0) & \text{if } i > |\pi|, \end{cases} \\
d_\pi(\neg\phi, i) &= \sim d_\pi(\phi, i), \\
d_\pi(\phi_1 \vee \phi_2, i) &= d_\pi(\phi_1, i) \sqcup d_\pi(\phi_2, i), \\
d_\pi(\mathbf{X}\phi, i) &= d_\pi(\phi, i+1) \oplus 1, \\
d_\pi(\phi \mathbf{U} \psi, i) &= \begin{cases} d_\pi(\psi, i) \sqcup \left(d_\pi(\phi, i) \sqcap d_\pi(\mathbf{X}(\phi \mathbf{U} \psi), i) \right) & \text{if } i \leq |\pi|, \\ d_\pi(\psi, i) \sqcup \left(d_\pi(\phi, i) \sqcap (-, \infty) \right) & \text{if } i > |\pi|, \end{cases} \\
d_\pi(\mathbf{F}\phi, i) &= \begin{cases} d_\pi(\phi, i) \sqcup d_\pi(\mathbf{X}\mathbf{F}\phi, i) & \text{if } i \leq |\pi|, \\ d_\pi(\phi, i) \sqcup (-, \infty) & \text{if } i > |\pi|. \end{cases}
\end{aligned}$$

We now provide some motivations behind the above definitions.

Proposition. A proposition is either evaluated before or after the end of the trace. If it is evaluated before the end of the trace and the proposition holds, the satisfaction and violations witness counts are trivially 0 and $-$, respectively. In the case that the proposition does not hold, we have the symmetric witness counts. Finally, we take an optimistic view in case of evaluating a proposition after the end of the trace: The trace can be extended to a trace with i steps s.t. either p holds or p does not hold.

Negation. Negating a formula simply swaps the witness counts. If we witness the satisfaction of ϕ in n steps, we witness the violation of $\neg\phi$ in n steps, and vice versa.

Disjunction. We take the shorter satisfaction witness count, because the satisfaction of one subformula is enough to satisfy the property. And we take the longer violation witness count, because both subformulas need to be violated to violate the property.

Next. The next operator naturally increases the witness counts by one step.

Eventually. We use the rewriting rule $\mathbf{F}\phi \equiv \phi \vee \mathbf{X}\mathbf{F}\phi$ to define the semantics of the eventually operator. When evaluating the formula after the end of the trace, we replace the remaining obligation ($\mathbf{X}\mathbf{F}\phi$) by $(-, \infty)$. Thus, $\mathbf{F}\phi$ evaluated on the empty word is satisfied by a suffix that satisfies ϕ , and it is violated only by infinite suffixes.

Until. We use the same principle for defining the until semantics that we used for the eventually operator. We use the rewriting rule $\phi \mathbf{U} \psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi \mathbf{U} \psi))$. On the empty word, $\phi \mathbf{U} \psi$ is satisfied (in the shortest way) by a suffix that satisfies ψ , and it is violated by a suffix that violates both ϕ and ψ .

Example 8. We refer to our motivating example from Table 1 and evaluate the trace τ_2 with respect to the specification ψ . We present the outcome in Table 2. We see that every proposition evaluates to $(0, -)$ when true. The satisfaction of a proposition that holds at time i is immediately witnessed and it cannot be violated by any suffix. Similarly, a proposition evaluates to $(-, 0)$ when false. The valuations of $\mathbf{F}g$ count the number of steps to positions in which g holds. For instance, the first time at which g holds is $i = 3$, hence $\mathbf{F}g$ evaluates to

$(2, -)$ at time 1, $(1, -)$ at time 2 and $(0, -)$ at time 3. We also note that Fg evaluates to $(0, \infty)$ at the end of the trace – it could be immediately satisfied with the continuation of the trace with g that holds, but could be violated only by an infinite suffix in which g never holds. We finally observe that $G(r \rightarrow Fg)$ evaluates to (∞, ∞) at all positions – the property can be both satisfied and violated only with infinite suffixes.

Table 2. Unbounded response property example: $d_\pi(\phi, i)$ with the trace $\pi = \tau_2$.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | EOT |
|---------------------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| r | \top | $-$ | $-$ | \top | $-$ | $-$ | \top | |
| g | $-$ | $-$ | \top | $-$ | $-$ | \top | $-$ | |
| $d_\pi(r, i)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(0, 0)$ |
| $d_\pi(g, i)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(0, 0)$ |
| $d_\pi(\neg r, i)$ | $(-, 0)$ | $(0, -)$ | $(0, -)$ | $(-, 0)$ | $(0, -)$ | $(0, -)$ | $(-, 0)$ | $(0, 0)$ |
| $d_\pi(Fg, i)$ | $(2, -)$ | $(1, -)$ | $(0, -)$ | $(2, -)$ | $(1, -)$ | $(0, -)$ | $(1, \infty)$ | $(0, \infty)$ |
| $d_\pi(r \rightarrow Fg, i)$ | $(2, -)$ | $(0, -)$ | $(0, -)$ | $(2, -)$ | $(0, -)$ | $(0, -)$ | $(1, \infty)$ | $(0, \infty)$ |
| $d_\pi(G(r \rightarrow Fg), i)$ | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) |

We use “ $-$ ” instead of “ \perp ” in the traces r and g to improve the readability.

Not all pairs $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$ are possible according to the counting semantics. We present the possible pairs in Lemma 9.

Lemma 9. *Let $\pi \in \Pi^*$ be a finite trace, ϕ an LTL formula and $i \in \mathbb{N}_0$ an index. We have that $d_\pi(\phi, i)$ is of the form $(a, -)$, $(-, a)$, (b_1, b_2) , (b_1, ∞) , (∞, b_2) or (∞, ∞) , where $a \leq |\pi| - i$ and $b_j > |\pi| - i$ for $j \in \{1, 2\}$.*

Proof. The proof can be obtained using structural induction on the LTL formula (see Appendix A.2 in [2]).

Finally, we relate our counting semantics to the three valued semantics in Lemma 10.

Lemma 10. *Given an LTL formula and a trace $\pi \in \Pi^*$ where $i \in \mathbb{N}_{>0}$ is an index and ϕ is an LTL formula, we have that*

$$\begin{aligned}
 d_\pi(\phi, i) = (a, -) &\leftrightarrow \mu_\pi(\phi, i) = \top, \\
 &\text{and } \nexists x < a. \pi' = \pi_i \cdot \pi_{i+1} \cdot \dots \pi_{i+x}, \mu_{\pi'}(\phi, 1) = \top \\
 d_\pi(\phi, i) = (-, a) &\leftrightarrow \mu_\pi(\phi, i) = \perp, \\
 &\text{and } \nexists x < a. \pi' = \pi_i \cdot \pi_{i+1} \cdot \dots \pi_{i+x}, \mu_{\pi'}(\phi, 1) = \perp \\
 d_\pi(\phi, i) = (b_1, b_2) &\leftrightarrow \mu_\pi(\phi, i) = ?,
 \end{aligned}$$

where $a \leq |\pi| - i$ and b_j is either ∞ or $b_j > |\pi| - i$ for $j \in \{1, 2\}$.

Intuitively, Lemma 10 holds because we only introduce the symbol “ $-$ ” within the trace when a satisfaction (violation) is observed. And the values of a pair only propagate into the past (and never into the future).

4.3 Evaluation

We now propose a mapping that predicts a qualitative verdict from our counting semantics. We adopt a 5-valued set consisting of **true** (\top), **presumably true** (\top_P), **inconclusive** ($?$), **presumably false** (\perp_P) and **false** (\perp) verdicts. We define the following order over these five values: $\perp < \perp_P < ? < \top_P < \top$. We equip this 5-valued domain with the negation (\neg) and disjunction (\vee) operations, letting $\neg\top = \perp$, $\neg\top_P = \perp_P$, $\neg? = ?$, $\neg\perp_P = \top_P$, $\neg\perp = \top$ and $\phi_1 \vee \phi_2 = \max\{\phi_1, \phi_2\}$. We define other Boolean operators such as conjunction by the usual logical equivalences ($\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$, etc.).

We evaluate a property on a trace to \top (\perp) when the satisfaction (violation) can be fully determined from the trace, following the definition of the three-valued semantics μ . Intuitively, this takes care of the case in which the safety (co-safety) part of a formula has been violated (satisfied), at least for properties that are intentionally safe (intentionally co-safe, resp.) [10].

Whenever the truth value is not determined, we distinguish whether $d_\pi(\phi, i)$ indicates the possibility for a satisfaction, respective violation, in finite time or not. For possible satisfactions, respective violations, in finite time we make a prediction on whether past observations support the believe that the trace is going to satisfy or violate the property. If the predictions are not inconclusive and not contradicting, then we evaluate the trace to the (presumable) truth value \top_P or \perp_P . If we cannot make a prediction to a truth value, we compute the truth value recursively based on the operator in the formula and the truth values of the subformulas (with temporal operators unrolled).

We use the predicate pred_π to give the prediction based on the observed witnesses for satisfaction. The predicate $\text{pred}_\pi(\phi, i)$ becomes $?$ when no witness for satisfaction exists in the past. When there exists a witness that requires at least the same amount of additional steps as the trace under evaluation then the predicate evaluates to \top . If all the existing witnesses (and at least one exists) are shorter than the current trace, then the predicate evaluates to \perp . For a prediction on the violation we make a prediction on the satisfaction of $d_\pi(\neg\phi, i)$, i.e., we compute $\text{pred}_\pi(\neg\phi, i)$.

Definition 11 (Prediction predicate). *Let s, f denote natural numbers and let $s_\pi(\phi, i), f_\pi(\phi, i) \in \mathbb{N}_+$ such that $d_\pi(\phi, i) = (s_\pi(\phi, i), f_\pi(\phi, i))$. We define the 3-valued predicate pred_π as*

$$\text{pred}_\pi(\phi, i) = \begin{cases} \top & \text{if } \exists j < i. d_\pi(\phi, j) = (s', -) \text{ and } s_\pi(\phi, i) \leq s', \\ ? & \text{if } \nexists j < i. d_\pi(\phi, j) = (s', -), \\ \perp & \text{if } \exists j < i. d_\pi(\phi, j) = (s', -) \text{ and } \\ & s_\pi(\phi, i) > \max_{0 \leq j < i} \{s' \mid d_\pi(\phi, j) = (s', -)\}, \end{cases}$$

For the evaluation we consider a case split among the possible combinations of values in the pairs.

Definition 12 (Predictive evaluation). We define the predictive evaluation function $e_\pi(\phi, i)$, with $a \leq |\pi| - i$ and $b_j > |\pi| - i$ for $j \in \{1, 2\}$ and $a, b_j \in \mathbb{N}_0$, for the different cases of $d_\pi(\phi, i)$:

| $d_\pi(\phi, i)$ | $e_\pi(\phi, i)$ |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $(a, -)$ | \top |
| (b_1, b_2) | $\text{if } \text{pred}_\pi(\phi, i) > \text{pred}_\pi(\neg\phi, i) \quad \top_P$ $\text{if } \text{pred}_\pi(\phi, i) = \text{pred}_\pi(\neg\phi, i) \quad r_\pi(\phi, i)$ $\text{if } \text{pred}_\pi(\phi, i) < \text{pred}_\pi(\neg\phi, i) \quad \perp_P$ |
| (b_1, ∞) | $\text{if } \text{pred}_\pi(\phi, i) = \top \quad \top_P$ $\text{if } \text{pred}_\pi(\phi, i) = ? \quad r_\pi(\phi, i)$ $\text{if } \text{pred}_\pi(\phi, i) = \perp \quad \perp_P$ |
| (∞, b_1) | $e_\pi(\neg\phi, i)$ |
| (∞, ∞) | $r_\pi(\phi, i)$ |
| $(-, a)$ | \perp |

where $r_\pi(\phi, i)$ is an auxiliary function defined inductively as follows:

$$\begin{aligned}
r_\pi(p, i) &= ? \\
r_\pi(\neg\phi, i) &= \neg e_\pi(\phi, i) \\
r_\pi(\phi_1 \vee \phi_2, i) &= e_\pi(\phi_1, i) \vee e_\pi(\phi_2, i) \\
r_\pi(X^n \phi, i) &= e_\pi(\phi, i + n) \\
r_\pi(F \phi, i) &= \begin{cases} e_\pi(\phi, i) \vee r_\pi(XF \phi, i) & \text{if } i \leq |\pi| \\ e_\pi(\phi, i) & \text{if } i > |\pi| \end{cases} \\
r_\pi(\phi_1 \text{ U } \phi_2, i) &= \begin{cases} e_\pi(\phi_2, i) \vee (e_\pi(\phi_2, i) \wedge e_\pi(X(\phi_1 \text{ U } \phi_2), i)) & \text{if } i \leq |\pi| \\ e_\pi(\phi_2, i) & \text{if } i > |\pi| \end{cases}
\end{aligned}$$

The predictive evaluation function is symmetric. Hence, $e_\pi(\phi, i) = \neg e_\pi(\neg\phi, i)$ holds.

Example 13. The outcome of evaluating τ_2 from Table 1 is shown in Table 3. Subformula $r \rightarrow Fg$ is predicted to be \top_P at $i = 7$ because there exists a longer witness for satisfaction in the past (e.g., at $i = 1$). Thus, the trace evaluates to \top_P , as expected.

In Fig. 1 we visualize the evaluation of a pair $d_\pi(\phi, i) = (s, f)$ for a fixed ϕ and a fixed position i . On the x-axis is the witness count s for a satisfaction and on the y-axis is the witness count f for a violation. For a value s , respectively f , that is smaller than the length of the suffix starting at position i (with the other value of the pair always being $-$), the evaluation is either \top or \perp . Otherwise the evaluation depends on the values s_{max} and f_{max} . These two values

Table 3. Unbounded response property example with $\pi = \tau_2$.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | EOT |
|---------------------------------------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| r | \top | $-$ | $-$ | \top | $-$ | $-$ | \top | |
| g | $-$ | $-$ | \top | $-$ | $-$ | \top | $-$ | |
| $d_\pi(r, i)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(0, 0)$ |
| $e_\pi(r, i)$ | \top | \perp | \perp | \top | \perp | \perp | \top | $?$ |
| $d_\pi(g, i)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(-, 0)$ | $(0, -)$ | $(-, 0)$ | $(0, 0)$ |
| $e_\pi(g, i)$ | \perp | \perp | \top | \perp | \perp | \top | \perp | $?$ |
| $d_\pi(\mathbf{F}g, i)$ | $(2, -)$ | $(1, -)$ | $(0, -)$ | $(2, -)$ | $(1, -)$ | $(0, -)$ | $(1, \infty)$ | $(0, \infty)$ |
| $e_\pi(\mathbf{F}g, i)$ | \top | \top | \top | \top | \top | \top | \top_P | \top_P |
| $d_\pi(r \rightarrow \mathbf{F}g, i)$ | $(2, -)$ | $(0, -)$ | $(0, -)$ | $(2, -)$ | $(0, -)$ | $(0, -)$ | $(1, \infty)$ | $(0, \infty)$ |
| $e_\pi(r \rightarrow \mathbf{F}g, i)$ | \top | \top | \top | \top | \top | \top | \top_P | \top_P |
| $d_\pi(\mathbf{G}(r \rightarrow \mathbf{F}g), i)$ | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) | (∞, ∞) |
| $e_\pi(\mathbf{G}(r \rightarrow \mathbf{F}g), i)$ | \top_P | \top_P | \top_P | \top_P | \top_P | \top_P | \top_P | \top_P |

We use “ $-$ ” instead of “ \perp ” in the traces r and g to improve the readability.

represent the largest witness counts for a satisfaction and a violation in the past, i.e., for positions smaller than i in the trace. Based on the prediction function $\text{pred}_\pi(\phi, i)$ the evaluation becomes \top_P , $?$ or \perp_P , where $?$ indicates that the auxiliary function $r_\pi(\phi, i)$ has to be applied. Starting at an arbitrary point in the diagram and moving to the right increases the witness count for a satisfaction while the witness count for a violation remains constant. Thus, moving to the right makes the pair “more false”. The same holds when keeping the witness count for a satisfaction constant and moving up in the diagram as this decrease the witness count for a violation. Analogously, moving down and/or left makes the pair “more true” as the witness count for a violation gets larger and/or the witness count for a satisfaction gets smaller.

Our 5-valued predictive evaluation refines the 3-valued LTL semantics.

Theorem 14. *Let ϕ be an LTL formula, $\pi \in \Pi^*$ and $i \in \mathbb{N}_{>0}$. We have*

$$\begin{aligned}
\mu_\pi(\phi, i) = \top &\leftrightarrow e_\pi(\phi, i) = \top, \\
\mu_\pi(\phi, i) = \perp &\leftrightarrow e_\pi(\phi, i) = \perp, \\
\mu_\pi(\phi, i) = ? &\leftrightarrow e_\pi(\phi, i) \in \{\top_P, \perp_P, ?\}.
\end{aligned}$$

Theorem 14 holds, because the evaluation to \top and \perp is simply the mapping of a pair that contains the symbol “ $-$ ”, which we have shown in Lemma 10.

Remember that $\mathbb{N}_+ \times \mathbb{N}_+$ is partially ordered by \preceq . We now show that having a trace that is “more true” than another is correctly reflected in our finitary semantics. To define “more true”, we first need the polarity of a proposition in an LTL formula.

Example 15. Note that g has positive polarity in $\phi = \mathbf{G}(r \rightarrow \mathbf{F}g)$. If we define τ'_2 to be as τ_2 , except that $g \in \tau'_2(i)$ for $i \in \{1, \dots, 6\}$, we have $e_{\tau'_2}(\phi, i) = \perp_P$, whereas $e_{\tau_2}(\phi, i) = \top_P$.

Theorem 18 holds, because we have that replacing an arbitrary observed value in π by one with positive polarity in π' always results with $d_\pi(\phi, 1) = (s, f)$ and $d_{\pi'}(\phi, 1) = (s', f')$ in $s' \leq s$ and $f' \geq f$, as with $\pi \sqsubseteq_\phi \pi'$ we have that π' witnesses a satisfaction of ϕ not later than π and π' also witness a violation of ϕ not earlier than π .

Table 4. Making a system “more true”.

| ϕ | π | $d_\pi(\phi, 1)$ | $e_\pi(\phi, 1)$ | ϕ | π | $d_\pi(\phi, 1)$ | $e_\pi(\phi, 1)$ |
|--------------------------|--------------------------------------------------------------|------------------------------------------------------|-------------------------------------------------|------------------------|----------------------------------------------------------------------------|---------------------------------------------------------------------|--------------------------------------------------|
| p | $\begin{array}{c} \top \\ \top \end{array}$ | $\begin{array}{c} (-, 0) \\ (0, -) \end{array}$ | $\begin{array}{c} \perp \\ \top \end{array}$ | $\text{F G } p$ | $\begin{array}{c} \top - \top - \top \\ \top - \top \top \top \end{array}$ | $\begin{array}{c} (\infty, \infty) \\ (\infty, \infty) \end{array}$ | $\begin{array}{c} \perp_P \\ \top_P \end{array}$ |
| $p \wedge \text{X F } p$ | $\begin{array}{c} \top - - \\ \top - - \end{array}$ | $\begin{array}{c} (-, 0) \\ (3, \infty) \end{array}$ | $\begin{array}{c} \perp \\ \perp_P \end{array}$ | $\text{G F } p$ | $\begin{array}{c} - - \top - - \\ \top - \top - - \end{array}$ | $\begin{array}{c} (\infty, \infty) \\ (\infty, \infty) \end{array}$ | $\begin{array}{c} \top_P \\ \perp_P \end{array}$ |
| $\text{G } p$ | $\begin{array}{c} - \top \top \\ \top \top \top \end{array}$ | $\begin{array}{c} (-, 0) \\ (\infty, 3) \end{array}$ | $\begin{array}{c} \perp \\ \top_P \end{array}$ | $p \vee \text{X G } p$ | $\begin{array}{c} - \top \top \\ \top \top \top \end{array}$ | $\begin{array}{c} (\infty, 3) \\ (0, -) \end{array}$ | $\begin{array}{c} \top_P \\ \top \end{array}$ |
| $\text{F } p$ | $\begin{array}{c} - - - \\ \top - - \end{array}$ | $\begin{array}{c} (3, \infty) \\ (0, -) \end{array}$ | $\begin{array}{c} \perp_P \\ \top \end{array}$ | | | | |

In Table 4 we give examples to illustrate the transition of one evaluation to another one. Note that it is possible to change from \top_P to \perp_P . However, this is only the predicated truth value that becomes “worse”, because we have strengthened the prefix on which the prediction is based on, the values of $d_\pi(\phi, i)$ do not change and remain the same in such a case.

5 Examples

We demonstrate the strengths and weaknesses of our approach on the examples of LTL specifications and traces shown in Table 5. We fully develop these examples in Appendix B in [2].

Table 5. Examples of LTL specifications and traces

| Specifications | Traces | |
|-----------------------------------------------------------------------------------|-------------------------------------------------------|--------------------------------------------------------------------------------------|
| $\psi_1 \equiv \text{F X } g$ | $\pi_1 : g : \perp \perp \perp \perp$ | $\pi_5 : r : \perp \top \top \top \top \perp \top \top$ |
| $\psi_2 \equiv \text{G X } g$ | $\pi_2 : g : \top \top \top \top$ | $g : \perp \top \perp \perp \perp \top \perp$ |
| $\psi_3 \equiv \text{G}(r \rightarrow \text{F } g)$ | $\pi_3 : r : \perp \top \perp \perp \top \perp$ | $\pi_6 : g : \top \top \perp \perp \top \top \perp \perp \top \top \perp \perp \top$ |
| $\psi_4 \equiv \bigwedge_{i \in \{1,2\}} \text{G}(r_i \rightarrow \text{F } g_i)$ | $g : \perp \perp \top \perp \perp \perp$ | $\pi_7 : g : \top \top \perp \perp \top \top \perp \perp \top \top \top \top \top$ |
| $\psi_5 \equiv \text{G}((\text{X } r) \cup (\text{X X } g))$ | $\pi_4 : r_1 : \top \perp \top \perp \top \perp \top$ | $\pi_8 : r : \top \top \top \top \perp \perp$ |
| $\psi_6 \equiv \text{F G } g \vee \text{F G } \neg g$ | $g_1 : \perp \top \perp \top \perp \top \perp$ | $g : \top \perp \top \perp \top \perp$ |
| $\psi_7 \equiv \text{G}(\text{F } r \vee \text{F } g)$ | $r_2 : \perp \top \perp \top \perp \top \perp$ | |
| $\psi_8 \equiv \text{G F}(r \vee g)$ | $g_2 : \top \perp \top \perp \top \perp \top$ | |
| $\psi_9 \equiv \text{G F } r \vee \text{G F } g$ | | |

Table 6 summarizes the evaluation of our examples. The first and the second column denote the evaluated specification and trace. We use these examples to compare LTL with counting semantics (c-LTL) presented in this paper, to the other two popular finitary LTL interpretations, the 3-valued LTL semantics [4] (3-LTL) and LTL on truncated paths [9] (t-LTL). We recall that in t-LTL there is a distinction between a weak and a strong next operator. We denote by t-LTL-s (t-LTL-w) the specifications from our examples in which X is interpreted as the strong (weak) next operator and assume that we always give a strong interpretation to U and F and a weak interpretation to G .

Table 6. Comparison of different verdicts with different semantics

| Spec. | Trace | c-LTL | 3-LTL | t-LTL-s | t-LTL-w |
|----------|---------|-----------|-------|---------|---------|
| ψ_1 | π_1 | \perp_P | ? | \perp | \top |
| ψ_2 | π_2 | \top_P | ? | \perp | \top |
| ψ_3 | π_3 | \perp_P | ? | \perp | \perp |
| ψ_4 | π_4 | \top_P | ? | \perp | \perp |
| ψ_5 | π_5 | \top_P | ? | \perp | \top |

| Spec. | Trace | c-LTL | 3-LTL | t-LTL-s | t-LTL-w |
|----------|---------|-----------|-------|---------|---------|
| ψ_6 | π_6 | \perp_P | ? | \top | \top |
| ψ_6 | π_7 | \top_P | ? | \top | \top |
| ψ_7 | π_8 | \perp_P | ? | \perp | \perp |
| ψ_8 | π_8 | \perp_P | ? | \perp | \perp |
| ψ_9 | π_8 | \top_P | ? | \perp | \perp |

There are two immediate observations that we can make regarding the results presented in Table 6. First, the 3-valued LTL gives for all the examples an *inconclusive* verdict, a feedback that after all has little value to a verification engineer. The second observation is that the verdicts from c-LTL and t-LTL can differ quite a lot, which is not very surprising given the different strategies to interpret the unseen future. We now further comment on these examples, explaining in more details the results and highlighting the intuitive outcomes of c-LTL for a large class of interesting LTL specifications.

Effect of Nested Next. We evaluate with ψ_1 and ψ_2 the effect of nesting X in an F and an G formula, respectively. We make a prediction on Xg at the end of the trace before evaluating F and G . As a consequence, we find that (ψ_1, π_1) evaluates to *presumably false*, while (ψ_2, π_2) evaluates to *presumably true*. In t-LTL, this class of specification is very sensitive to the weak/strong interpretation of next, as we can see from the verdicts.

Request/Grants. We evaluate the request/grant property ψ_3 from the motivating example on the trace π_3 . We observe that r at cycle 2 is followed by g at cycle 3, while r at cycle 5 is not followed by g at cycle 6. Hence, (ψ_3, π_3) evaluates to *presumably false*.

Concurrent Request/Grants. We evaluate the specification ψ_4 against the trace π_4 . In this example r_1 is triggered at even time stamps and r_2 is triggered at odd time stamps. Every request is granted in one cycle. It follows that regardless of

the time when the trace ends, there is one request that is not granted yet. We note that ψ_4 is a conjunction of two basic request/grant properties and we make independent predictions for each conjunct. Every basic request/grant property is evaluated to **presumably true**, hence (ψ_4, π_4) evaluates to **presumably true**. At this point, we note that in **t-LTL**, every request that is not granted by the end of the trace results in the property violation, regardless of the past observations.

Until. We use the specification ψ_5 and the trace π_5 to evaluate the effect of **U** on the predictions. The specification requires that $\mathbf{X}r$ continuously holds until $\mathbf{X}\mathbf{X}g$ becomes true. We can see that in π_5 $\mathbf{X}r$ is witnessed at cycles 1 – 4, while $\mathbf{X}\mathbf{X}g$ is witnessed at cycle 5. We can also see that $\mathbf{X}r$ is again witnessed from cycle 6 until the end of the trace at cycle 8. As a consequence, (ψ_5, π_5) is evaluated to **presumably true**.

Stabilization. The specification ψ_6 says that the value of g has to eventually stabilize to either true or false. We evaluate the formula on two traces π_6 and π_7 . In the trace π_6 , g alternates between true and false every two cycles and becomes true in the last cycle. Hence, there is no sufficiently long witness of trace stabilization (ψ_6, π_6) evaluates to **presumably false**. In the trace π_7 , g also alternates between true and false every two cycles, but in the last four cycles g remains continuously true. As a consequence, (ψ_6, π_7) evaluates to **presumably true**. This example also illustrates the importance of when the trace truncation occurs. If both π_6 and π_7 were truncated at cycle 5, both (ψ_6, π_6) and (ψ_6, π_7) would evaluate to **presumably false**. We note that ψ_6 is satisfied by all traces in **t-LTL**.

Sub-formula Domination. The specification ψ_7 exposes a weakness of our approach. It requires that in every cycle, either r or g is witnessed in some unbounded future. With our approach, (ψ_7, π_8) evaluates to **presumably false**. This is against our intuition because we have observed that g becomes regularly true very second time step. However, in this example our prediction for $\mathbf{F}r$ dominates over the prediction for $\mathbf{F}g$, leading to the unexpected **presumably false** verdict. On the other hand, **t-LTL** interpretation of the same specification is dependent only on the last value of r and g .

Semantically Equivalent Formulas. We now demonstrate that our approach may give different answers for semantically equivalent formulas. For instance, both ψ_8 and ψ_9 are semantically equivalent to ψ_7 . We have that (ψ_8, π_8) evaluates to **presumably false**, while (ψ_9, π_8) evaluates to **presumably true**. We note that **t-LTL** verdicts are stable for semantically different formulas.

6 Conclusion

We have presented a novel finitary semantics for LTL that uses the history of satisfaction and violation in a finite trace to predict whether the co-safety

and safety aspects of a formula will be satisfied in the extension of the trace to an infinite one. We claim that the semantics closely follow human intuition when predicting the truth value of a trace. The presented examples (incl. non-monitorable LTL properties) illustrate our approach and support this claim.

Our definition of the semantics is trace-based, but it is easily extended to take an entire database of traces into account, which may make the approach more precise. Our approach currently uses a very simple form of learning to predict the future. We would like to consider more sophisticated statistical methods to make better predictions. In particular, we plan to apply nonparametric statistical methods (i.e., the Wilcoxon signed-rank test [16]), in combination with our counting semantics, to identify and quantify the traces that are outliers.

References

1. Almagor, S., Boker, U., Kupferman, O.: Discounting in LTL. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 424–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_37
2. Bartocci, E., Bloem, R., Nickovic, D., Roeck, F.: A counting semantics for monitoring LTL specifications over finite traces. CoRR, abs/1804.03237 (2018)
3. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-75632-5>
4. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006). https://doi.org/10.1007/11944836_25
5. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşiran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
7. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. ACM Trans. Comput. Logic **15**(4), 27:1–27:25 (2014)
8. Daca, P., Henzinger, T.A., Křetínský, J., Petrov, T.: Faster statistical model checking for unbounded temporal properties. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 112–129. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_7
9. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_3
10. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Form. Methods Syst. Des. **19**(3), 291–314 (2001)
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, Heidelberg (1992). <https://doi.org/10.1007/978-1-4612-0931-7>
12. Morgenstern, A., Gesell, M., Schneider, K.: An asymptotically correct finite path semantics for LTL. In: Björner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol.

- 7180, pp. 304–319. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_24
13. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57 (1977)
 14. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_38
 15. Viswanathan, M., Kim, M.: Foundations for the run-time monitoring of reactive systems – fundamentals of the MaC language. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 543–556. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_38
 16. Wilcoxon, F.: Individual comparisons by ranking methods. *Biom. Bull.* **1**(6), 80–83 (1945)
 17. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 418–432. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_37

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Tools



Rabinizer 4: From LTL to Your Favourite Deterministic Automaton

Jan Křetínský^(✉), Tobias Meggendorfer^(ID),
Salomon Sickert^(ID), and Christopher Ziegler



Technical University of Munich, Munich, Germany
jan.kretinsky@gmail.com, {meggendo,sickert}@in.tum.de

Abstract. We present Rabinizer 4, a tool set for translating formulae of linear temporal logic to different types of deterministic ω -automata. The tool set implements and optimizes several recent constructions, including the first implementation translating the frequency extension of LTL. Further, we provide a distribution of PRISM that links Rabinizer and offers model checking procedures for probabilistic systems that are not in the official PRISM distribution. Finally, we evaluate the performance and in cases with any previous implementations we show enhancements both in terms of the size of the automata and the computational time, due to algorithmic as well as implementation improvements.

1 Introduction

Automata-theoretic approach [VW86] is a key technique for verification and synthesis of systems with linear-time specifications, such as formulae of linear temporal logic (LTL) [Pnu77]. It proceeds in two steps: first, the formula is translated into a corresponding automaton; second, the product of the system and the automaton is further analyzed. The size of the automaton is important as it directly affects the size of the product and thus largely also the analysis time, particularly for deterministic automata and probabilistic model checking in a very direct proportion. For verification of non-deterministic systems, mostly non-deterministic Büchi automata (NBA) are used [EH00,SB00,GO01,GL02,BKRS12,DLLF+16] since they are typically very small and easy to produce.

Probabilistic LTL model checking cannot profit directly from NBA. Even the qualitative question, whether a formula holds with probability 0 or 1, requires automata with at least a restricted form of determinism. The prime example are the limit-deterministic (also called semi-deterministic) Büchi automata (LDBA) [CY88] and the generalized LDBA (LDGBA). However, for the general quantitative questions, where the probability of satisfaction is computed, general limit-determinism is not sufficient. Instead, deterministic Rabin automata (DRA) have

This work has been partially supported by the Czech Science Foundation grant No. P202/12/G061 and the German Research Foundation (DFG) project KR 4890/1-1 “Verified Model Checkers” (317422601). A part of the frequency extension has been implemented within Google Summer of Code 2016.

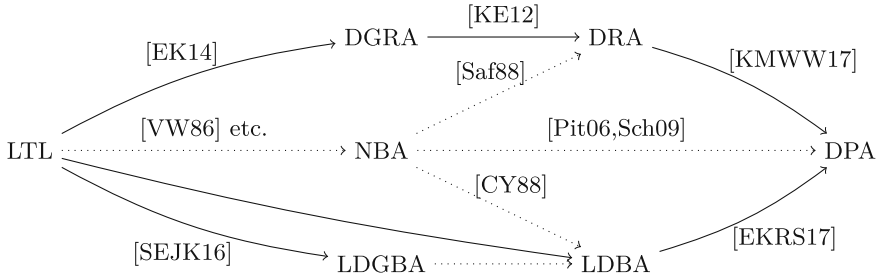


Fig. 1. LTL translations to different types of automata. Translations implemented in Rabinizer 4 are indicated with a solid line. The traditional approaches are depicted as dotted arrows. The determinization of NBA to DRA is implemented in *ltl2dstar* [Kle], to LDDBA in *Seminator* [BDK+17] and to (mostly) DPA in *spot* [DLLF+16].

been mostly used [KNP11] and recently also deterministic generalized Rabin automata (DGRA) [CGK13]. In principle, all standard types of deterministic automata are applicable here except for deterministic Büchi automata (DBA), which are not as expressive as LTL. However, other types of automata, such as deterministic Muller and deterministic parity automata (DPA) are typically larger than DGRA in terms of acceptance condition or the state space, respectively.¹ Recently, several approaches with specific LDDBA were proved applicable to the quantitative setting [HLS+15, SEJK16] and competitive with DGRA. Besides, model checking MDP against LTL properties involving frequency operators [BDL12] also allows for an automata-theoretic approach, via deterministic generalized Rabin mean-payoff automata (DGRMA) [FKK15].

LTL synthesis can also be solved using the automata-theoretic approach. Although DRA and DGRA transformed into games can be used here, the algorithms for the resulting Rabin games [PP06] are not very efficient in practice. In contrast, DPA may be larger, but in this setting they are the automata of choice due to the good practical performance of parity-game solvers [FL09, ML16, JBB+17].

Types of Translations. The translations of LTL to NBA, e.g., [VW86], are typically “*semantic*” in the sense that each state is given by a set of logical formulae and the language of the state can be captured in terms of semantics of these formulae. In contrast, the determinization of Safra [Saf88] or its improvements [Pit06, Sch09, TD14, FL15] are not “*semantic*” in the sense that they ignore the structure and produce trees as the new states that, however, lack the logical interpretation. As a result, if we apply Safra’s determinization on semantically created NBA, we obtain DRA that lack the structure and, moreover, are unnecessarily large since the construction cannot utilize the original structure. In contrast, the

¹ Note that every DGRA can be written as a Muller automaton on the same state space with an exponentially-sized acceptance condition, and DPA are a special case of DRA and thus DGRA.

recent works [KE12,KLG13,EK14,KV15,SEJK16,EKRS17,MS17,KV17] provide “semantic” constructions, often producing smaller automata. Furthermore, various transformations such as degeneralization [KE12], index appearance record [KMWW17] or determinization of limit-deterministic automata [EKRS17] preserve the semantic description, allowing for further optimizations of the resulting automata.

Our Contribution. While all previous versions of Rabinizer [GKE12,KLG13,KK14] featured only the translation $LTL \rightarrow DGRA \rightarrow DRA$, Rabinizer 4 now implements all the translations depicted by the solid arrows in Fig. 1. It improves all these translations, both algorithmically and implementation-wise, and moreover, features the first implementation of the translation of a frequency extension of LTL [FKK15].

Further, in order to utilize the resulting automata for verification, we provide our own distribution² of the PRISM model checker [KNP11], which allows for model checking MDP against LTL using not only DRA and DGRA, but also using LDBA and against frequency LTL using DGRMA. Finally, the tool can turn the produced DPA into parity games between the players with input and output variables. Therefore, when linked to parity-game solvers, Rabinizer 4 can be also used for LTL synthesis.

Rabinizer 4 is freely available at <http://rabinizer.model.in.tum.de> together with an on-line demo, visualization, usage instructions and examples.

2 Functionality

We recall that the previous version Rabinizer 3 has the following functionality:

- It translates LTL formulae into equivalent DGRA or DRA.
- It is linked to PRISM, allowing for probabilistic verification using DGRA (previously PRISM could only use DRA).

2.1 Translations

Rabinizer 4 inputs formulae of LTL and outputs automata in the standard HOA format [BBD+15], which is used, e.g., as the input format in PRISM. Automata in the HOA format can be directly visualized, displaying the “semantic” description of the states. Rabinizer 4 features the following command-line tools for the respective translations depicted as the solid arrows in Fig. 1:

ltl2dgra and **ltl2dra** correspond to the original functionality of Rabinizer 3, i.e., they translate LTL (now with the extended syntax, including all common temporal operators) to DGRA and DRA [EK14], respectively.

² Merging these features into the public release of PRISM as well as linking the new version of Rabinizer is subject to current collaboration with the authors of PRISM.

ltl2ldgba and **ltl2ldba** translate LTL to LDGBA using the construction of [SEJK16] and to LDBA, respectively. The latter is our modification of the former, which produces smaller automata than chaining the former with the standard degeneralization.

ltl2dpa translates LTL to DPA using two modes:

- The default mode uses the translation to LDBA, followed by a LDBA-to-DPA determinization [EKRS17] specially tailored to LDBA with the “semantic” labelling of states, avoiding additional exponential blow-up of the resulting automaton.
- The alternative mode uses the translation to DRA, followed by our improvement of the index appearance record of [KMWW17].

ftl2dgrma translates the frequency extension of $LTL_{\setminus GU}$, i.e. $LTL_{\setminus GU}$ [KLG13] with $G^{\sim\rho}$ operator³, to DGRMA using the construction of [FKK15].

2.2 Verification and Synthesis

The resulting automata can be used for model checking probabilistic systems and for LTL synthesis. To this end, we provide our own distribution of the probabilistic model checker PRISM as well as a procedure transforming automata into games to be solved.

Model checking: PRISM distribution. For model checking Markov chains and Markov decision processes, PRISM [KNP11] uses DRA and recently also more efficient DGRA [CGK13, KK14]. Our distribution, which links Rabinizer, additionally features model checking using the LDBA [SEJK16, SK16] that are created by our **ltl2ldba**.

Further, the distribution provides an implementation of frequency $LTL_{\setminus GU}$ model checking, using DGRMA. To the best of our knowledge, there are no other implemented procedures for logics with frequency. Here, techniques of linear programming for multi-dimensional mean-payoff satisfaction [CKK15] and the model-checking procedure of [FKK15] are implemented and applied.

Synthesis: Games. The automata-theoretic approach to LTL synthesis requires to transform the LTL formula into a game of the input and output players. We provide this transformer and thus an end-to-end LTL synthesis solution, provided a respective game solver is linked. Since current solutions to Rabin games are not very efficient we implemented a transformation of DPA into parity games and a serialization to the format of PG Solver [FL09]. Due to the explicit serialization, we foresee the main use in quick prototyping.

³ The *frequential globally* construct [BDL12, BMM14] $G^{\sim\rho}\varphi$ with $\sim \in \{\geq, >, \leq, <\}$, $\rho \in [0, 1]$ intuitively means that the fraction of positions satisfying φ satisfies $\sim\rho$. Formally, the fraction on an infinite run is defined using the long-run average [BMM14].

3 Optimizations, Implementation, and Evaluation

Compared to the theoretical constructions and previous implementations, there are numerous improvements, heuristics, and engineering enhancements. We evaluate the improvements both in terms of the size of the resulting automaton as well as the running time. When comparing with respect to the original Rabinizer functionality, we compare our implementation **ltl2dgra** to the previous version Rabinizer 3.1, which is already a significantly faster [EKS16] re-implementation of the official release Rabinizer 3 [KK14]. All of the benchmarks have been executed on a host with i7-4700MQ CPU (4x2.4 GHz), running Linux 4.9.0-5-amd64 and the Oracle JRE 9.0.4+11 JVM. Due to the start-up time of JVM, all times below 2s are denoted by <2 and not specified more precisely. All experiments were given a time-out of 900s and mem-out of 4GB, denoted by $-$.

Algorithmic improvements and heuristics for each of the translations:

ltl2dgra and **ltl2dra**. These translations create a master automaton monitoring the satisfaction of the given formula and a dedicated slave automaton for each subformula of the form $\mathbf{G}\psi$ [EK14]. We (i) simplify several classes of slaves and (ii) “suspend” (in the spirit of [BBDL+13]) some so that they appear in the final product only in some states. The effect on the size of the state space is illustrated in Table 1 on a nested formula. Further, (iii) the acceptance condition is considered separately for each strongly connected component (SCC) and then combined. On a concrete example of Table 2, the automaton for $i = 8$ has 31 atomic propositions, whereas the number of atomic propositions relevant in each component of the master automaton is constant, which we utilize and thus improve performance on this family both in terms of size and time.

ltl2ldb. This translation is based on breakpoints for subformulae of the form $\mathbf{G}\psi$. We provide a heuristic that avoids breakpoints when ψ is a safety or co-safety subformula, see Table 3.

Besides, we add an option to generate a non-deterministic initial component for the LDBA instead of a deterministic one. Although the LDBA is then no more suitable for quantitative probabilistic model checking, it still is for qualitative model checking. At the same time, it can be much smaller, see Table 4 which shows a significant improvement on the particular formula.

ltl2dpa. Both modes inherit the improvements of the respective **ltl2ldb** and **ltl2dgra** translations. Further, since complementing DPA is trivial, we can run in parallel both the translation of the input formula and of its negation, returning the smaller of the two results. Finally, we introduce several heuristics to optimize the treatment of safety subformulae of the input formula.

dra2dpa. The index appearance record of [KMWW17] keeps track of a permutation (ordering) of Rabin pairs. To do so, all ties between pairs have to be resolved. In our implementation, we keep a pre-order instead, where irrelevant

ties are not resolved. Consequently, it cannot happen that an irrelevant tie is resolved in two different ways like in [KMWW17], thus effectively merging such states.

Table 1. Effect of simplifications and suspension for **ltl2dgra** on the formulae $\psi_i = \mathbf{G}\phi_i$ where $\phi_1 = a_1, \phi(i) = (a_i \mathbf{U}(\mathbf{X}\phi_{i-1}))$, and $\psi'_i = \mathbf{G}\phi'_i$ where $\phi'_1 = a_1, \phi'_1 = (\phi'_{i-1} \mathbf{U}(\mathbf{X}^i a_i))$, displaying execution time in seconds/#states.

| | ψ_2 | ψ_3 | ψ_4 | ψ_5 | ψ_6 |
|------------------------------|-----------|-----------|-----------|-----------|-----------|
| Rabinizer 3.1 [EKS16] | <2/4 | <2/16 | <2/73 | 3/332 | 60/1463 |
| ltl2dgra | <2/3 | <2/7 | <2/35 | 3/199 | 13/1155 |
| | ψ'_2 | ψ'_3 | ψ'_4 | ψ'_5 | ψ'_6 |
| Rabinizer 3.1 [EKS16] | <2/4 | <2/16 | 2/104 | 128/670 | — |
| ltl2dgra | <2/3 | <2/10 | <2/38 | 7/175 | 239/1330 |

Table 2. Effect of computing acceptance sets per SCC on formulae $\psi_1 = x_1 \wedge \phi_1$, $\psi_2 = (x_1 \wedge \phi_1) \vee (\neg x_1 \wedge \phi_2)$, $\psi_3 = (x_1 \wedge x_2 \wedge \phi_1) \vee (\neg x_1 \wedge x_2 \wedge \phi_2) \vee (x_1 \wedge \neg x_2 \wedge \phi_3), \dots$, where $\phi_i = \mathbf{XG}((a_i \mathbf{U} b_i) \vee (c_i \mathbf{U} d_i))$, displaying execution time in seconds/#acceptance sets.

| | ψ_1 | ψ_2 | ψ_3 | ψ_4 | ψ_5 | ... | ψ_8 |
|------------------------------|----------|----------|----------|----------|----------|-----|----------|
| Rabinizer 3.1 [EKS16] | <2/2 | <2/7 | <2/19 | — | — | | — |
| ltl2dgra | <2/1 | <2/1 | <2/1 | <2/1 | <2/1 | | <2/1 |

Table 3. Effect of break-point elimination for **ltl2ldb**a on safety formulae $s(n, m) = \bigwedge_{i=1}^n \mathbf{G}(a_i \vee \mathbf{X}^m b_i)$ and for **ltl2ldg**ba on liveness formulae $l(n, m) = \bigwedge_{i=1}^n \mathbf{GF}(a_i \wedge \mathbf{X}^m b_i)$, displaying #states (#Büchi conditions)

| | $s(1, 3)$ | $s(2, 3)$ | $s(3, 3)$ | $s(4, 3)$ | $s(1, 4)$ | $s(2, 4)$ | $s(3, 4)$ | $s(4, 4)$ |
|-------------------|-----------|-----------|--------------------|---------------------|-----------|-----------|------------|------------|
| [SEJK16] | 20 (1) | 400 (2) | $8 \cdot 10^3$ (3) | $16 \cdot 10^4$ (4) | 48 (1) | 2304 (2) | 110592 (3) | — |
| ltl2ldb a | 8 (1) | 64 (1) | 512 (1) | 4096 (1) | 16 (1) | 256 (1) | 4096 (1) | 65536 (1) |
| | $l(1, 1)$ | $l(2, 1)$ | $l(3, 1)$ | $l(4, 1)$ | $l(1, 4)$ | $l(2, 4)$ | $l(3, 4)$ | $l(4, 4)$ |
| [SEJK16] | 3 (1) | 9 (2) | 27 (3) | 81 (4) | 10 (1) | 100 (2) | 10^3 (3) | 10^4 (4) |
| ltl2ldg ba | 3 (1) | 5 (2) | 9 (3) | 17 (4) | 3 (1) | 5 (2) | 9 (3) | 17 (4) |

Table 4. Effect of non-determinism of the initial component for **ltl2ldb**a on formulae $f(i) = \mathbf{F}(a \wedge \mathbf{X}^i \mathbf{G}b)$, displaying #states (#Büchi conditions)

| | $f(1)$ | $f(2)$ | $f(3)$ | $f(4)$ | $f(5)$ | $f(6)$ |
|------------------|--------|--------|--------|--------|--------|--------|
| [SEJK16] | 4 (1) | 6 (1) | 10 (1) | 18 (1) | 34 (1) | 66 (1) |
| ltl2ldb a | 2 (1) | 3 (1) | 4 (1) | 5 (1) | 6 (1) | 7 (1) |

Table 5. Comparison of the average performance with the previous version of Rabinizer. The statistics are taken over a set of 200 standard formulae [KMS18] used, e.g., in [BKS13,EKS16], run in a batch mode for both tools to eliminate the effect of the JVM start-up overhead.

| Tool | Avg # states | Avg # acc. sets | Avg runtime |
|------------------------------|--------------|-----------------|-------------|
| Rabinizer 3.1 [EKS16] | 6.3 | 6.7 | 0.23 |
| ltl2dgra | 6.2 | 4.4 | 0.12 |

Implementation. The main performance bottleneck of the older implementations is that explicit data structures for the transition system are not efficient for larger alphabets. To this end, Rabinizer 3.1 provided symbolic (BDD) representation of states and edge labels. On the top, Rabinizer 4 represents the transition function symbolically, too.

Besides, there are further engineering improvements on issues such as storing the acceptance condition only as a local edge labelling, caching, data-structure overheads, SCC-based divide-and-conquer constructions, or the introduction of parallelization for batch inputs.

Average Performance Evaluation. We have already illustrated the improvements on several hand-crafted families of formulae. In Tables 1 and 2 we have even seen the respective running-time speed-ups. As the basis for the overall evaluation of the improvements, we use some established datasets from literature, see [KMS18], altogether two hundred formulae. The results in Table 5 indicate that the performance improved also on average among the more realistic formulae.

4 Conclusion

We have presented Rabinizer 4, a tool set to translate LTL to various deterministic automata and to use them in probabilistic model checking and in synthesis. The tool set extends the previous functionality of Rabinizer, improves on previous translations, and also gives the very first implementations of frequency LTL translation as well as model checking. Finally, the tool set is also more user-friendly due to richer input syntax, its connection to PRISM and PG Solver, and the on-line version with direct visualization, which can be found at <http://rabinizer.model.in.tum.de>.

References

- [BBD+15] Babiak, T., et al.: The hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31

- [BBDL+13] Babiak, T., Badie, T., Duret-Lutz, A., Křetínský, M., Strejček, J.: Compositional approach to suspension and other improvements to LTL translation. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 81–98. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39176-7_6
- [BDK+17] Blahoudek, F., Duret-Lutz, A., Klokočka, M., Křetínský, M., Strejček, J.: Seminar: a tool for semi-determinization of omega-automata. In: LPAR, pp. 356–367 (2017)
- [BDL12] Bollig, B., Decker, N., Leucker, M.: Frequency linear-time temporal logic. In: TASE, pp. 85–92 (2012)
- [BKŘS12] 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). https://doi.org/10.1007/978-3-642-28756-5_8
- [BKS13] Blahoudek, F., Křetínský, M., Strejček, J.: Comparison of LTL to deterministic rabin automata translators. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 164–172. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_12
- [BMM14] Bouyer, P., Markey, N., Matteplackel, R.M.: Averaging in LTL. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 266–280. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_19
- [CGK13] Chatterjee, K., Gaiser, A., Křetínský, J.: Automata with generalized rabin pairs for probabilistic model checking and LTL synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 559–575. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_37
- [CKK15] Chatterjee, K., Komárková, Z., Křetínský, J.: Unifying two views on multiple mean-payoff objectives in Markov decision processes. In: LICS, pp. 244–256 (2015)
- [CY88] Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: FOCS, pp. 338–345 (1988)
- [DLLF+16] Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω -automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
- [EH00] Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–168. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_13
- [EK14] Esparza, J., Křetínský, J.: From LTL to deterministic automata: a safrless compositional approach. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 192–208. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_13
- [EKRS17] Esparza, J., Křetínský, J., Raskin, J.-F., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 426–442. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_25

- [EKS16] Esparza, J., Křetínský, J., Sickert, S.: From LTL to deterministic automata - a safrless compositional approach. *Formal Methods Syst. Des.* **49**(3), 219–271 (2016)
- [FKK15] Forejt, V., Krčál, J., Křetínský, J.: Controller synthesis for MDPs and frequency LTL\GU. In: LPAR, pp. 162–177 (2015)
- [FL09] Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_15
- [FL15] Fisman, D., Lustig, Y.: A modular approach for büchi determinization. In: CONCUR, pp. 368–382 (2015)
- [GKE12] Gaiser, A., Křetínský, J., Esparza, J.: Rabinizer: small deterministic automata for LTL(**F,G**). In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 72–76. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_7
- [GL02] Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36135-9_20
- [GO01] 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). https://doi.org/10.1007/3-540-44585-4_6. <http://www.lsv.ens-cachan.fr/gastin/ltl2ba/>
- [HLS+15] Hahn, E.M., Li, G., Schewe, S., Turrini, A., Zhang, L.: Lazy probabilistic model checking without determinisation. In: CONCUR. LIPIcs, vol. 42, pp. 354–367 (2015)
- [JBB+17] Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J.-F., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): benchmarks, participants & results. CoRR, abs/1711.11439 (2017)
- [KE12] 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). https://doi.org/10.1007/978-3-642-31424-7_7
- [KK14] Komárková, Z., Křetínský, J.: Rabinizer 3: safrless translation of LTL to small deterministic automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 235–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_17
- [Kle] Klein, J.: ltl2dstar - LTL to deterministic Streett and Rabin automata. <http://www.ltl2dstar.de/>
- [KLG13] Křetínský, J., Garza, R.L.: Rabinizer 2: Small Deterministic Automata for LTL\GU. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 446–450. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_32
- [KMS18] Křetínský, J., Meggendorfer, T., Sickert, S.: LTL store: repository of LTL formulae from literature and case studies. CoRR, abs/1807.03296 (2018)
- [KMWW17] Křetínský, J., Meggendorfer, T., Waldmann, C., Weininger, M.: Index appearance record for transforming rabin automata into parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 443–460. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_26

- [KNP11] 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). https://doi.org/10.1007/978-3-642-22110-1_47
- [KV15] Kini, D., Viswanathan, M.: Limit deterministic and probabilistic automata for LTL\GU. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 628–642. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_57
- [KV17] Kini, D., Viswanathan, M.: Optimal translation of LTL to limit deterministic automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 113–129. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_7
- [ML16] Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 262–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_17
- [MS17] Müller, D., Sickert, S.: LTL to deterministic Emerson-Lei automata. In: GandALF, pp. 180–194 (2017)
- [Pit06] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: LICS, pp. 255–264 (2006)
- [Pnu77] Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
- [PP06] Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: LICS, pp. 275–284 (2006)
- [Saf88] Safra, S.: On the complexity of omega-automata. In: FOCS, pp. 319–327 (1988)
- [SB00] 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). https://doi.org/10.1007/10722167_21
- [Sch09] 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). https://doi.org/10.1007/978-3-642-00596-1_13
- [SEJK16] Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17
- [SK16] Sickert, S., Křetínský, J.: MoChiBA: probabilistic LTL model checking using limit-deterministic Büchi automata. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 130–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_9
- [TD14] Tian, C., Duan, Z.: Buchi determinization made tighter. Technical report abs/1404.1436, [arXiv.org](https://arxiv.org/abs/1404.1436) (2014)
- [VW86] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, pp. 332–344 (1986)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Strix: Explicit Reactive Synthesis Strikes Back!

Philipp J. Meyer¹, Salomon Sickert²,
and Michael Luttenberger

Technical University of Munich, Munich, Germany
{meyerphi,sickert,luttenbe}@in.tum.de



Abstract. STRIX is a new tool for reactive LTL synthesis combining a direct translation of LTL formulas into deterministic parity automata (DPA) and an efficient, multi-threaded explicit state solver for parity games. In brief, STRIX (1) decomposes the given formula into simpler formulas, (2) translates these on-the-fly into DPAs based on the queries of the parity game solver, (3) composes the DPAs into a parity game, and at the same time already solves the intermediate games using strategy iteration, and (4) finally translates the winning strategy, if it exists, into a Mealy machine or an AIGER circuit with optional minimization using external tools. We experimentally demonstrate the applicability of our approach by a comparison with PARTY, BOsY, and LTLsYNT using the SYNTCOMP2017 benchmarks. In these experiments, our prototype can compete with BOsY and LTLsYNT with only PARTY performing slightly better. In particular, our prototype successfully synthesizes the full and unmodified LTL specification of the AMBA protocol for $n = 2$ masters.

1 Introduction

Reactive synthesis refers to the problem of finding for a formal specification of an input-output relation, in our case a *linear temporal logic (LTL)*, a matching implementation [22], e.g. a *Mealy machine* or an *and-inverter-graph (AIG)*. Since the automata-theoretic approach to synthesis involves the construction of a potentially double exponentially sized automaton (in the length of the specification) [13], most existing tools focus on symbolic and bounded methods in order to combat the state-space explosion [5, 9, 11, 18]. A beneficial side effect of these approaches is that they tend to yield succinct implementations.

In contrast to these approaches, we present a prototype implementation of an LTL synthesis tool which follows the automata theoretic approach using parity games as an intermediate step. STRIX¹ uses the LTL-to-DPA translation

This work was partially funded and supported by the German Research Foundation (DFG) projects “Game-based Synthesis for Industrial Automation” (253384115) and “Verified Model Checkers” (317422601).

¹ <https://strix.model.in.tum.de/>

presented in [10, 23] and the multi-threaded explicit-state parity game solver presented in [14, 20]: First, the given formula is decomposed into much simpler requirements, often resulting in a large number of safety and co-safety conditions and only a few requiring Büchi or parity acceptance conditions, which is comparable to the approach of [5, 21]. These requirements are then translated on-the-fly into automata, keeping the invariant that the parity game solver can easily compose the actual parity game. Further, by querying only for states that are actually required for deciding the winner, the implementation avoids unnecessary work.

The parity game solver is based on the *strategy iteration* of [19] which iteratively improves non-deterministic strategies, i.e. strategies that can allow several actions for a given state as long as they all are guaranteed to lead to the specified system behaviour. When translating the winning strategy into a Mealy automaton or an AIG this non-determinism can be used similarly to “don’t cares” when minimizing boolean circuits. Strategy iteration offers us two additional advantages, first, we can directly take advantage of multi-core systems; second, we can reuse the winning strategies which have been computed for the intermediate arenas.

Related Work and Experimental Evaluation. From the tools submitted to SYNTCOMP2017, LTLSYNT [15] is closest to our approach: it also combines an LTL-to-DPA-translation with an explicit-state parity game solver, but it does not intertwine the two steps, instead it uses a different approach for the translation leading to one monolithic DPA which is then turned in a parity game. In contrast, the two best performing tools from SYNTCOMP2017, BoSY and PARTY, use bounded synthesis, by reduction either to SAT, SMT, or safety games.

In order to give a realistic estimation of how our tool would have fared at SYNTCOMP2017 (TLSF/LTL track), we tried to re-create the benchmark environment of SYNTCOMP2017 as close as possible on our hardware: in its current state, our tool would have been ranked below PARTY, but before LTLSYNT and BoSY. Due to time and resource constraints, we could only do an in-depth comparison with the current version of LTLSYNT; in particular we used the TLSF specification of the complete² AMBA protocol for $n = 2$ as a benchmark. We refer to Sect. 3 for details on the benchmarking procedure.

2 Design and Implementation

STRIX is implemented in Java and C++. It supports LTL and TLSF [16] (only the reduced *basic* variant) as input languages, while the latter one is preferred, since it contains more information about the specification. We describe the main steps of the tool in the following paragraphs with examples given in Fig. 1.

² i.e. no decomposition in masters and clients or structural properties were used.

the external SAT-based minimizer MEMIN [1] for Mealy machines and extract a more compact description.

AIGER Circuit Generation and Minimization. We translate the minimized Mealy machine with the tool SPECULOOS³ into an AIGER circuit. In parallel, we also construct an AIGER circuit out of the non-minimized Mealy machine, since this can sometimes result in smaller circuits. The two AIGER circuits are then further compressed using ABC [6], and the smaller one is returned.

3 Experimental Evaluation

We evaluate STRIX on the TLFS/LTL-track benchmark of the SYNTCOMP2017 competition, which consists of 177 realizable and 67 unrealizable temporal logic synthesis specifications [15]. The experiment was run on a server with an Intel E5-2630 v4 clocked at 2.2 GHz (boost disabled). To mimic SYNTCOMP2017 we imposed a limit of 8 threads for parallelization, a memory limit of 32 GB and a timeout of one hour for each specification. Every specification for that a tool correctly decides realizability within these limits is counted as solved for the category **Realizability**, and every specification for that it can additionally produce an AIGER circuit that is successfully verified is counted as solved for the category **Synthesis**. For this we verified the circuits with an additional time limit of one hour using the NUXMV model checker [7] with the `check.ltlspec` and `check.ltlspec.klive` routines in parallel.

We compared STRIX with LTLSTYNT in the latest available release (version 2.5) at time of writing. This version differs from the one used during SYNTCOMP2017 as it contains several improvements, but also performs worse in a few cases and exhibits erroneous behaviour: for **Realizability**, it produced one wrong answer, and for **Synthesis**, it failed in 72 cases to produce AIGER circuits due to a program error.

Additionally, we compare our results with the best configuration of the top tools competing in SYNTCOMP2017: PARTY (portfolio), LTLSTYNT and BoSY (spot). Due to the difficulty of recreating the SYNTCOMP2017 hardware setup⁴, we compiled the results for these tools in Table 1 from the SYNTCOMP2017 webpage⁵ combining them with our results.

³ <https://github.com/romainbrenuguier/Speculoos>

⁴ SYNTCOMP2017 was run on an Intel E3-1271 v3 (4 cores/8 threads) at 3.6 GHz with 32 GB of RAM available for the tools. As stated above, we imposed the same constraints regarding timeout, maximal number of threads, and memory limit; but the Intel E3-1271 v3 runs at 3.6 GHz (with boost 4.0 GHz), while the Intel E5-2630 v4 used by us runs at only 2.2 GHz (boost disabled) resulting in a lower per-thread-performance (potentially 30% slower); on the other hand our system has a larger cache and a theoretically much higher memory bandwidth from up to 68.3 GB/s compared to 25.6 GB/s (for random reads, as in the case of dynamically generated parity games, these numbers are much closer). It seems therefore likely that for some benchmark-tool combinations our system is faster while for others it is slower.

⁵ <http://syntcomp.cs.uni-saarland.de/syntcomp2017/experiments/>

The **Quality** rating compares the size of the solutions according to the SYNTCOMP2017 formula, where a tool gets $2 - \log_{10} \frac{n+1}{r+1}$ quality points for each verified solution of size n for a specification with reference size r . We now move on to a detailed discussion of the results and their interpretation.

Table 1. Results for STRIX compared with LTL SYNT and selected results from SYNTCOMP2017 on the TLSF/LTL-track benchmark and on notable instances. We mark timeouts by TIME, memouts by MEM, and errors by ERR.

| | | Our system | | SYNTCOMP2017 | | |
|---------------------------|-----------------------|------------|----------------|--------------|----------|------|
| | | STRIX | LTL SYNT (2.5) | PARTY | LTL SYNT | BoSY |
| Solved | Realizability | 214 | 204 | 224 | 195 | 181 |
| | Synthesis | 197 | 123 | 203 | 182 | 181 |
| | Quality | 330 | 136 | 308 | 180 | 298 |
| | Avg. Quality | 1.68 | 1.10 | 1.52 | 0.99 | 1.64 |
| Time (s) Realizability | full_arbiter_7 | 11.34 | MEM | 8.77 | MEM | TIME |
| | prioritized_arbiter_7 | 58.53 | TIME | 372.95 | TIME | TIME |
| | round_robin_arbiter_6 | 8.45 | 158.33 | TIME | 733.92 | TIME |
| | 1t12dba_E_10 | 6.79 | 324.84 | TIME | TIME | TIME |
| | 1t12dba_Q_8 | 2.13 | 346.12 | TIME | TIME | TIME |
| Size (AIG) | amba_..._encode_12 | 89 | ERR | 1040 | 3251 | 369 |
| | full_arbiter_5 | 531 | ERR | 2257 | 7393 | TIME |
| | full_arbiter_6 | 626 | ERR | 7603 | 26678 | TIME |
| | 1t12dba_E_4 | 7 | 406 | 243 | 406 | TIME |
| | 1t12dba_E_6 | 11 | 3952 | 1955 | 3952 | TIME |

Realizability. We were able to correctly decide realizability for 163 and unrealizability for 51 specifications, resulting in 214 solved instances. We solve five instances that were previously unsolved in SYNTCOMP2017.

Synthesis. We produced AIGER circuits for 148 of the realizable specifications. In 15 cases, we only constructed a Mealy machine, but the subsequent steps (MEMIN for minimization or SPECULOOS for circuit generation) reached the time or memory limit. We were able to verify correctness for 146 of the circuits, reaching the model checking time limit in two case. Together with the 51 specifications for which we determined unrealizability, this results in 197 solved instances.

Quality. We produced 36 solutions that are smaller than any solution during SYNTCOMP2017. The most significant reductions are for the AMBA encoder and the full arbiter, with reductions of over 75%, and for 1t12dba_E.4 and 1t12dba_E.6, where we produce indeed the smallest implementation there is.

3.1 Effects of Minimization

We could reduce the size of the Mealy machine in 80 cases, and on average by 45%. However the data showed that this did not always reduce the size of the generated AIGER circuit: in 13 cases (most notably for several arbiter specifications) the size of the circuit generated from the Mealy machine actually increased when applying minimization (on average by 190%), while it decreased in 62 cases (on average by 55%).

We conjecture that the structure of the product-arena is sometimes amenable to compact representation in an AIGER circuit, while after the (SAT-based) minimization this is lost. In these cases the SAT/SMT-based bounded synthesis tools such as BOSY and PARTY also have difficulties producing a small solution, if any at all.

3.2 Synthesis of Complete AMBA AHB Arbiter

To test maturity and scalability of our tool, we synthesized the AMBA AHB arbiter [2], a common case study for reactive synthesis. We used the parameterized specification from [17] for $n = 2$ masters, which was also part of SYNT-COMP2016, but was left unsolved by any tool. With a memory limit of 128 GB, we could decide realizability within 26 min and produce a Mealy machine with 83 states after minimization. While specialised GR(1) solvers [2, 4, 12] or decompositional approaches [3] are able to synthesize the specification in a matter of minutes, to the best of our knowledge we are the first full LTL synthesis tool that can handle the complete non-decomposed specification in a reasonable amount of time. For comparison, LTLSYNT (2.5) needs more than 2.5 days on our system and produces a Mealy machine with 340 states.

3.3 Discussion

The LTLSYNT tool is part of Spot [8], which uses a Safra-style determinization procedure for NBAs. Conceptually, it also uses DPAs and a parity game solver as a decision procedure. However, as shown in [10] the produced automata tend to be larger compared to our translation, which probably results in the lower quality score. Our approach has similar performance and scales better on certain cases. The instances where LTLSYNT performs better than STRIX are specifications that we cannot split efficiently and the DPA construction becomes the bottleneck.

Bounded synthesis approaches (BOSY, PARTY) tend to produce smaller Mealy machines and to be able to handle larger alphabets. However, they fail when the minimal machine implementing the desired property is large, even if there is a compact implementation as a circuit. In our approach, we can often solve these cases and still regain compactness of the implementation through minimization afterwards. The strength of the PARTY portfolio is the combination of traditional bounded synthesis and a novel approach by reduction to safety games, which results in a large number of solved instances, but reduces the avg. quality score.

Future Work. STRIX combines Java (LTL simplification and automata translations) and C++ (parity game construction and solving). We believe that a pure C++ implementation will further improve the overall runtime and reduce the memory footprint. Next, there are several algorithmic questions we want to investigate going forward, especially expanding parallelization of the tool. Furthermore, we want to reduce the dependency on external tools for circuit generation in order to be able to fine-tune this step better. Especially replacing SPECULOOS is important, since it turned out that it was unable to handle complex transition systems.

References

1. Abel, A., Reineke, J.: MeMin: SAT-based exact minimization of incompletely specified mealy machines. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, 2–6 November 2015, pp. 94–101 (2015). <https://doi.org/10.1109/ICCAD.2015.7372555>
2. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: hardware from PSL. *Electr. Notes Theor. Comput. Sci.* **190**(4), 3–16 (2007). <https://doi.org/10.1016/j.entcs.2007.09.004>
3. Bloem, R., Jacobs, S., Khalimov, A.: Parameterized synthesis case study: AMBA AHB. In: Proceedings of the 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, 23–24 July 2014, pp. 68–83 (2014). <https://doi.org/10.4204/EPTCS.157.9>
4. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.* **78**(3), 911–938 (2012). <https://doi.org/10.1016/j.jcss.2011.08.007>
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.-F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_45
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Cavada, R., et al.: The nuXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
8. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω -automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
9. 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). https://doi.org/10.1007/978-3-642-19835-9_25
10. Esparza, J., Křetínský, J., Raskin, J.-F., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 426–442. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_25

11. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 325–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_17
12. Godhal, Y., Chatterjee, K., Henzinger, T.A.: Synthesis of AMBA AHB from formal specification: a case study. STTT **15**(5–6), 585–601 (2013). <https://doi.org/10.1007/s10009-011-0207-9>
13. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-36387-4>
14. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 455–459. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_34
15. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): benchmarks, participants and results. [arXiv:1711.11439](https://arxiv.org/abs/1711.11439) [cs.LO] (2017)
16. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Proceedings of the Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, 17–18 July 2016, pp. 112–132 (2016). <https://doi.org/10.4204/EPTCS.229.10>
17. Jobstmann, B.: Applications and optimizations for LTL synthesis. Ph.D. thesis, Graz University of Technology (2007)
18. Khalimov, A., Jacobs, S., Bloem, R.: PARTY parameterized synthesis of token rings. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 928–933. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_66
19. Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games. [arXiv:0806.2923](https://arxiv.org/abs/0806.2923) [cs.GT] (2008)
20. Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 262–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_17
21. Morgenstern, A., Schneider, K.: Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In: Proceedings of the First Symposium on Games, Automata, Logic, and Formal Verification, GANDALF 2010, Minori (Amalfi Coast), Italy, 17–18 June 2010, pp. 89–102 (2010). <https://doi.org/10.4204/EPTCS.25.11>
22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1989, pp. 179–190. ACM, New York (1989). <https://doi.org/10.1145/75277.75293>
23. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17




Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





BTOR2 , BtorMC and Boolector 3.0

Aina Niemetz^{1,2} , Mathias Preiner^{1,2} ,
Clifford Wolf³, and Armin Biere¹ 

¹ Johannes Kepler University Linz, Linz, Austria

² Stanford University, Stanford, USA

niemetz@cs.stanford.edu

³ Symbiotic EDA, Vienna, Austria



Abstract. We describe BTOR2, a word-level model checking format for capturing models of hardware and potentially software in a bit-precise manner. This simple, line-based and easy to parse format can be seen as a sorted extension of the word-level format BTOR. It uses design principles from the bit-level format AIGER and follows semantics of the SMT-LIB logics of bit-vectors with arrays. This intermediate format can be used in various verification flows and is perfectly suited to establish a word-level model checking competition. It is supported by our new open source model checker BtorMC, which is built on top of version 3.0 of our SMT solver Boolector. We further provide new word-level benchmarks on which these open source tools are evaluated.

Our format BTOR2 generalizes and extends the BTOR [5] format, which can be seen as a word-level generalization of the initial version of the bit-level format AIGER [2]. BTOR is a format for quantifier-free formulas over bit-vectors and arrays with SMT-LIB [1] semantics but also provides sequential extensions for specifying word-level model checking problems with registers and memories. In contrast to BTOR, which is tailored towards bit-vectors and one-dimensional bit-vector arrays, BTOR2 has explicit sort declarations. It further allows to explicitly initialize registers and memories (instead of implicit initialization in BTOR) and extends the set of sequential features with witnesses, invariant and fairness constraints, and liveness properties. All of these are word-level variants lifted from corresponding features in the latest AIGER format [4], the input format of the hardware model checking competition (HWMCC) [3, 6] since 2011. We provide an open source BTOR2 tool suite, which includes a generic parser, random simulator and witness checker. We further implemented a reference bounded model checker BtorMC on top of our SMT solver Boolector. We consider BTOR2 as an ideal candidate to establish a word-level hardware model checking competition.

1 Format Description

The syntax of BTOR2 is shown in Fig. 1. The `sort` keyword is used to define arbitrary bit-vector and array sorts. This not only allows to specify multi-dimensional

Supported by Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

© The Author(s) 2018

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 587–595, 2018.

https://doi.org/10.1007/978-3-319-96145-3_32

| | | |
|----------------------------------|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{num} \rangle$ | ::= | positive unsigned integer (greater than zero) |
| $\langle \text{uint} \rangle$ | ::= | unsigned integer (including zero) |
| $\langle \text{string} \rangle$ | ::= | sequence of whitespace and printable characters without '\n' |
| $\langle \text{symbol} \rangle$ | ::= | sequence of printable characters without '\n' |
| $\langle \text{comment} \rangle$ | ::= | ';' $\langle \text{string} \rangle$ |
| $\langle \text{nid} \rangle$ | ::= | $\langle \text{num} \rangle$ |
| $\langle \text{sid} \rangle$ | ::= | $\langle \text{num} \rangle$ |
| $\langle \text{const} \rangle$ | ::= | 'const' $\langle \text{sid} \rangle$ [0-1]+ |
| $\langle \text{constd} \rangle$ | ::= | 'constd' $\langle \text{sid} \rangle$ ['-'](uint) |
| $\langle \text{consth} \rangle$ | ::= | 'consth' $\langle \text{sid} \rangle$ [0-9a-fA-F]+ |
| $\langle \text{input} \rangle$ | ::= | ('input' 'one' 'ones' 'zero') $\langle \text{sid} \rangle$ $\langle \text{const} \rangle$ $\langle \text{constd} \rangle$ $\langle \text{consth} \rangle$ |
| $\langle \text{state} \rangle$ | ::= | 'state' $\langle \text{sid} \rangle$ |
| $\langle \text{bitvec} \rangle$ | ::= | 'bitvec' $\langle \text{num} \rangle$ |
| $\langle \text{array} \rangle$ | ::= | 'array' $\langle \text{sid} \rangle$ $\langle \text{sid} \rangle$ |
| $\langle \text{node} \rangle$ | ::= | $\langle \text{sid} \rangle$ 'sort' ($\langle \text{array} \rangle$ $\langle \text{bitvec} \rangle$) $\langle \text{nid} \rangle$ ($\langle \text{input} \rangle$ $\langle \text{state} \rangle$) $\langle \text{nid} \rangle$ $\langle \text{opidx} \rangle$ $\langle \text{sid} \rangle$ $\langle \text{nid} \rangle$ $\langle \text{uint} \rangle$ [$\langle \text{uint} \rangle$] $\langle \text{nid} \rangle$ $\langle \text{op} \rangle$ $\langle \text{sid} \rangle$ $\langle \text{nid} \rangle$ [$\langle \text{nid} \rangle$ [$\langle \text{nid} \rangle$]] $\langle \text{nid} \rangle$ ('init' 'next') $\langle \text{sid} \rangle$ $\langle \text{nid} \rangle$ $\langle \text{nid} \rangle$ $\langle \text{nid} \rangle$ ('bad' 'constraint' 'fair' 'output') $\langle \text{nid} \rangle$ $\langle \text{nid} \rangle$ 'justice' $\langle \text{num} \rangle$ ($\langle \text{nid} \rangle$)+ |
| $\langle \text{line} \rangle$ | ::= | $\langle \text{comment} \rangle$ $\langle \text{node} \rangle$ [$\langle \text{symbol} \rangle$] [$\langle \text{comment} \rangle$] |
| $\langle \text{btor} \rangle$ | ::= | ($\langle \text{line} \rangle$ '\n')+ |

Fig. 1. Syntax of BTOR2. Non-terminals $\langle \text{opidx} \rangle$ and $\langle \text{op} \rangle$ are indexed and non-indexed operators as defined in Table 1 (sequential part in red). (Color figure online)

arrays but can be extended to support (uninterpreted) functions, floating points and other sorts. As a consequence, BTOR2 is not backwards compatible with BTOR. For clarity, in Fig. 1 we distinguish between node (line) identifiers $\langle \text{nid} \rangle$ and sort identifiers $\langle \text{sid} \rangle$, and do not allow an identifier to occur in both sets. Introducing sorts renders type specific keywords such as `var`, `array` and `acond` from BTOR obsolete. Instead, BTOR2 uses the keyword `input` to declare bit-vector and array variables of a given sort. Bit-vector constants are created as in BTOR with the keywords `const[dh]`, `one`, `ones` and `zero`.

Bit-vector and array operators as supported by BTOR2 and their respective sorts are shown in Table 1. We use \mathcal{B}^n for a bit-vector sort of width n , and \mathcal{I} and \mathcal{E} for the index and element sorts of an array sort $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}}$. Note that some bit-vector operators can be interpreted as *signed* or *unsigned*. In signed context, as in SMT-LIB, bit-vectors are represented in two's complement.

2 Sequential Extension

As shown in Fig. 1, the sequential extension of BTOR2 introduces a `state` keyword, which allows to specify registers and memories. In contrast to BTOR, where registers are implicitly zero-initialized and memories are uninitialized, BTOR2 provides a keyword `init` to explicitly define initialization functions for states. This enables us to also model partial initialization. For example, initializing a memory with a bit-vector constant `zero`, zero-initializes the whole memory, whereas

Table 1. Operators supported by BTOR2, where \mathcal{B}^n represents a bit-vector sort of size n and $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}}$ represents an array sort with index sort \mathcal{I} and element sort \mathcal{E} .

| | | |
|----------------------------------------|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| indexed | | |
| [su]ext w | (un)signed extension | $\mathcal{B}^n \rightarrow \mathcal{B}^{n+w}$ |
| slice $u \ l$ | extraction, $n > u \geq l$ | $\mathcal{B}^n \rightarrow \mathcal{B}^{u-l+1}$ |
| unary | | |
| not | bit-wise | $\mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| inc, dec, neg | arithmetic | $\mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| redand, redor, redxor | reduction | $\mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| binary | | |
| iff, implies | Boolean | $\mathcal{B}^1 \times \mathcal{B}^1 \rightarrow \mathcal{B}^1$ |
| eq, neq | (dis)equality | $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{B}^1$ |
| [su]gt, [su]gte, [su]lt, [su]lte | (un)signed inequality | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| and, nand, nor, or, xnor, xor | bit-wise | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| rol, ror, sll, sra, srl | rotate, shift | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| add, mul, [su]div, smod, [su]rem, sub | arithmetic | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| [su]addo, [su]divo, [su]mulo, [su]subo | overflow | $\mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^1$ |
| concat | concatenation | $\mathcal{B}^n \times \mathcal{B}^m \rightarrow \mathcal{B}^{n+m}$ |
| read | array read | $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}} \times \mathcal{I} \rightarrow \mathcal{E}$ |
| ternary | | |
| ite | conditional | $\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ |
| write | array write | $\mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}} \times \mathcal{I} \times \mathcal{E} \rightarrow \mathcal{A}^{\mathcal{I} \rightarrow \mathcal{E}}$ |

partially initializing a register can be achieved by applying a bit-mask to an uninitialized register.

Transition functions for both registers and memories are defined with the `next` keyword. It takes the current and next states as arguments. A state variable without associated next function is treated as a *primary* input, i.e., it has the same behaviour as inputs defined via keyword `input`. Note that BTOR provides a `next` keyword for registers and an `anext` keyword for memories. Using sorts in BTOR2 avoids such sort specific keyword variants.

As in the latest version of AIGER [4], BTOR2 supports *bad* state properties, which are essentially negations of safety properties. Multiple properties can be specified by simply adding multiple *bad* state properties. Invariant constraints can be introduced via the `constraint` keyword and are assumed to hold globally. A witness for a bad state property is an initialized finite path, which reaches (actually, contains) a bad state and satisfies all invariant constraints.

Again as in AIGER [4], keywords `fair` and `justice` allow to specify (global) fairness constraints and (negations of) liveness properties. Each *justice* property consists of a set of Büchi conditions. A witness for a justice property is an infinite initialized path on which all Büchi conditions and all global fairness constraints

are satisfied infinitely often. In addition, all global invariant constraints have to hold. The **justice** keyword takes a number (the number of Büchi conditions) and an arbitrary number of nodes (the Büchi conditions) as arguments.

3 Witness Format

The syntax of the BTOR2 witness format is shown in Fig. 2. A BTOR2 witness consists of a sequence of valid input assignments grouped by (time) frames. It starts with **'sat'** followed by a list of properties that are satisfied by the witness. A property is identified by a prefix **'b'** (for **bad**) and **'j'** (for **justice**) followed by a number i , which ranges over the number of defined *bad* and *justice* properties starting from 0. For example, **'b0 j0'** refers to the first bad and first justice property in the order as they occur in the BTOR2 input. The list of properties is followed by a sequence of $k + 1$ frames at time $t \in \{0, \dots, k\}$. A *frame* is divided into a state and input part. The *state* part starts with **'#t'** and is mandatory for the first frame ($t = 0$) and optional for later frames ($t > 0$). It contains state assignments at time t . The *input* part starts with **'@t'** and consists of input assignments of the transition from time t to $t + 1$. If states are uninitialized (no **init**), their initial assignment is required to be specified in frame **'#0'**. The state part is usually omitted for $t > 0$ since state assignments can be computed from states and inputs at time $t - 1$. While don't care inputs can be omitted, our witness checker assumes that they are zero. Input and state assignments use the same numbering scheme as properties, i.e., states and inputs are numbered separately in the order they are defined, starting from 0. For example, 0 in frame **'#t'** (or **'@t'**) refers to the first state (or input) as defined in the BTOR2 input. For justice properties we assume the witness to be lasso shaped, i.e., the next state, which can be computed from the last state and inputs at time k , is identical to one of the previous states at time $t = 0 \dots k$. As in AIGER, a BTOR2 witness is terminated with **'.'** on a separate line.

| | | |
|---------------------------------|-----|---------------------------------------------------------------------------|
| <u><binary-string></u> | ::= | [0-1]+ |
| <u><bv-assignment></u> | ::= | <binary-string> |
| <u><array-assignment></u> | ::= | '[' <binary-string> ']' <binary-string> |
| <u><assignment></u> | ::= | <uint> (<bv-assignment> <array-assignment>) [<u><symbol></u>] |
| <u><model></u> | ::= | (<comment> '\n' <assignment> '\n')+ |
| <u><state part></u> | ::= | '#' <uint> '\n' <model> |
| <u><input part></u> | ::= | '@' <uint> '\n' <model> |
| <u><frame></u> | ::= | [<state part>] <input part> |
| <u><prop></u> | ::= | ('b' 'j') <uint> |
| <u><header></u> | ::= | 'sat' \n' (<prop>)+ '\n' |
| <u><witness></u> | ::= | (<comment> '\n')+ (<header>) (<frame>)+ '.' |

Fig. 2. BTOR2 model and witness format syntax (sequential part in **red**). (Color figure online)

Figure 3 illustrates a simple C program (left), the corresponding BTOR2 model with the negation of the assertion as a **bad** property (center), and a

```

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
static bool read_bool () {
    int ch = getc (stdin);
    if (ch == '0') return false;
    if (ch == '1') return true;
    exit (0);
}
int main () {
    bool turn;           // input
    unsigned a = 0, b = 0; // states
    for (;;) {
        turn = read_bool ();
        assert (!(a == 3 && b == 3));
        if (turn) a = a + 1;
        else      b = b + 1;
    }
}

```

| | | |
|----|----------------|------------|
| 1 | sort bitvec 1 | sat |
| 2 | sort bitvec 32 | b0 |
| 3 | input 1 turn | #0 |
| 4 | state 2 a | @0 |
| 5 | state 2 b | 0 1 turn@0 |
| 6 | zero 2 | @1 |
| 7 | init 2 4 6 | 0 0 turn@1 |
| 8 | init 2 5 6 | @2 |
| 9 | one 2 | 0 0 turn@2 |
| 10 | add 2 4 9 | @3 |
| 11 | add 2 5 9 | 0 0 turn@3 |
| 12 | ite 2 3 4 10 | @4 |
| 13 | ite 2 -3 5 11 | 0 1 turn@4 |
| 14 | next 2 4 12 | @5 |
| 15 | next 2 5 13 | 0 1 turn@5 |
| 16 | constd 2 3 | @6 |
| 17 | eq 1 4 16 | 0 0 turn@6 |
| 18 | eq 1 5 16 | |
| 19 | and 1 17 18 | |
| 20 | bad 19 | |

Fig. 3. Example C program with corresponding BTOR2 model and witness.

BTOR2 witness for the violated property (right). The BTOR2 model defines one bad property ($a == 3 \ \&\& \ b == 3$), which is satisfied in frame 6. The corresponding witness identifies this property as bad property ‘b0’ (first bad property defined in the model). All states are initialized, hence ‘#0’ is empty, and ‘@0’ to ‘@6’ indicate the assignments of input 0 (**turn**, the first input defined in the model) in frames 0 to 6, e.g., **turn** = 1 at $t = 0$, **turn** = 0 at $t = 1$ and so on. In frame 6, both states **a** and **b** reach value 3, and therefore property ‘b0’ is satisfied.

4 Tools

We provide a generic stand-alone parser for BTOR2, which features basic type checking and consists of approx. 1,500 lines of C code. We implemented a reference bounded model checker BtorMC, which currently supports checking safety (aka. bad state) properties for models with registers and memories and produces witnesses for satisfiable properties. Unrolling the model is performed by symbolic simulation, i.e., symbolic substitution of current state expressions into next state functions, and incremental SMT solving. We also implemented a simulator for randomly simulating BTOR2 models. It further supports checking BTOR2 witnesses. The model checker is tightly integrated into our SMT solver Boolector [18], an award-winning SMT solver for the theory of fixed-size bit-vectors with arrays and uninterpreted functions. Since the last major version [18], we extended Boolector with several new features. Most notably, Boolector 3.0 now comes with support for quantified bit-vectors [24] and two different local search strategies for quantifier-free bit-vector formulas that don’t rely on but can be combined with bit-blasting [19, 21, 22]. It further provides support for BTOR2. In contrast to previous versions of Boolector, Boolector 3.0 and all BTOR2 tools

are released under the MIT open source license and the source code is hosted on GitHub¹.

5 Experiments

We collected ten real-world (System)Verilog designs with safety properties from various open source projects [11, 26–28]. The majority of these designs include memories. We used the open synthesis suite Yosys [29] to synthesize these designs into BTOR2 and SMT-LIB. For BTOR2, Yosys directly generates the models from a circuit description. For SMT-LIB, since the language does not support describing model checking problems, we used Yosys in combination with Yosys-SMTBMC to produce unrolled (incremental) problems.

We compared BtorMC against the most recent versions of Boolector (3.0) and Yices [10] (2.5.4), the two best solvers of the QF_ABV division of the SMT competition 2017. The BTOR2 models serve as input for BtorMC, and the incremental SMT-LIB benchmarks serve as input for Boolector and Yices. All benchmarks, synthesis scripts, generated files, log files and the source code of our tools for this evaluation are available at <http://fmv.jku.at/cav18-btor2>.

The results in Table 2 show that our flow using BTOR2 as intermediate format is competitive with simple unrolling. Note that our model checker BtorMC issues incremental calls to Boolector. However, in Boolector, sophisticated word-level rewriting is currently disabled in incremental mode. We expect a major performance boost by fully supporting incremental word-level preprocessing.

Table 2. BtorMC/BTOR2 vs. unrolled SMT-LIB with a time limit of 3600 s, where k is the bound and #bad is the number of bad properties.

| Benchmark | k | #bad | BtorMC time[s] | Boolector time[s] | Yices time[s] |
|---------------------------|-----|------|----------------|-------------------|---------------|
| picorv32-check | 30 | 23 | 4.8 | 18.9 | 10.8 |
| picorv32-pregs | 20 | 3 | 63.0 | 293.0 | TO |
| ponylink-slaveTXlen-sat | 230 | 1 | 305.5 | 406.8 | 145.6 |
| ponylink-slaveTXlen-unsat | 231 | 1 | 183.8 | 131.4 | 71.4 |
| VexRiscv-regch0-15 | 17 | 2 | 9.6 | 48.3 | 12.2 |
| VexRiscv-regch0-20 | 22 | 2 | 528.8 | 520.7 | 2232.2 |
| VexRiscv-regch0-30 | 32 | 2 | TO | TO | TO |
| zipcpu-busdelay | 100 | 50 | 157.0 | 287.0 | 181.2 |
| zipcpu-pfcache | 100 | 39 | 17.4 | 19.9 | 32.5 |
| zipcpu-zipmmu | 30 | 57 | 86.0 | 412.9 | 46.5 |

¹ <https://github.com/boolector>.

6 Conclusion

We propose BTOR2, a new word-level model-checking and witness format. For this format we provide a generic parser implementation, a simulator that also checks witnesses, and a reference bounded model checker BtorMC, which is tightly integrated with our SMT solver Boolector. These open source tools are evaluated on new real-world benchmarks, which we synthesized from open source hardware (System) Verilog models into BTOR2 and SMT-LIB with Yosys. The tool Verilog2SMV [14] translates Verilog into model-checking problems in several formats, including nuXmv [7] and BTOR. However, its translation to BTOR is incomplete and development discontinued.

We plan to provide a translator from BTOR2 into SALLY [25], and VMT [8], which are both extensions of SMT-LIB to model symbolic transition systems. It might also be interesting to translate incremental SMT-LIB benchmarks and horn clause models (as handled by, e.g., μZ [13]) into BTOR2 and vice versa. We hope other compilers and model checkers such as SAL [9], EBMC [15] and ABC [12, 16] will provide support to produce and read BTOR2 models. We want to extend the format to other logics, in particular to support lambdas as in [23]. There is also a need for fuzzing [20] and delta-debugging tools [17].

Last but not least, we want to use this format to bootstrap a word-level model checking competition, which of course needs more benchmarks.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
2. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr 69, 4040 Linz, Austria (2007)
3. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, p. 9. IEEE (2017). <https://doi.org/10.23919/FMCAD.2017.8102233>
4. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr 69, 4040 Linz, Austria (2011)
5. Brummayer, R., Biere, A., Lonsing, F.: BTOR: bit-precise modelling of word-level problems for model checking. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, SMT 2008/BPR 2008, pp. 33–38. ACM, New York, USA (2008). <http://doi.acm.org/10.1145/1512464.1512472>
6. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K.: Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. *J. Satisf. Boolean Model. Comput.* **9**, 135–172 (2014). Published 2016

7. Cavada, R., et al.: The nuXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
8. Cimatti, A., Roveri, M., Griggio, A., Irfan, A.: Verification modulo theories. <http://es.fbk.eu/projects/vmt-lib/>
9. De Moura, L., Owre, S., Shankar, N.: The SAL language manual. Technical report CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park (2003)
10. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
11. Gisselquist, D.: ZipCPU. <https://github.com/ZipCPU/zipcpu>
12. Ho, Y., Mishchenko, A., Brayton, R.K.: Property directed reachability with word-level abstraction. In: FMCAD, pp. 132–139. IEEE (2017)
13. Hoder, K., Bjørner, N., de Moura, L.: μZ - an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_36
14. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: a tool for word-level verification. In: DATE, pp. 1156–1159. IEEE (2016)
15. Kroening, D.: Computing over-approximations with bounded model checking. *Electr. Notes Theor. Comput. Sci.* **144**(1), 79–92 (2006)
16. Long, J., Ray, S., Sterin, B., Mishchenko, A., Brayton, R.K.: Enhancing ABC for stabilization verification of systemverilog/VHDL models. In: Proceedings of the CEUR Workshop DIFTS@FMCAD, vol. 832. CEUR-WS.org (2011)
17. Niemetz, A., Biere, A.: ddSMT: a delta debugger for the SMT-LIB v2 format. In: Bruttomesso, R., Griggio, A. (eds.) Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013, Affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, 8–9 July 2013, pp. 36–45 (2013)
18. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. *JSAT* **9**, 53–58 (2015)
19. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 199–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_11
20. Niemetz, A., Preiner, M., Biere, A.: Model-based API testing for SMT solvers. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017, Affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, 24–28 July 2017, p. 10 (2017)
21. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. *Formal Methods Syst. Des.* **51**(3), 608–636 (2017). <https://doi.org/10.1007/s10703-017-0295-6>
22. Niemetz, A., Preiner, M., Biere, A., Fröhlich, A.: Improving local search for bit-vector logics in SMT with path propagation. In: Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems, Austin, USA, 26–27 September 2015, pp. 1–10 (2015)
23. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. In: Proceedings of the CEUR Workshop DIFTS@FMCAD, vol. 1130. CEUR-WS.org (2013)
24. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 264–280. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_15

25. SRI International's Computer Science Laboratory: Sally - a model checker for infinite-state systems. <https://github.com/SRI-CSL/sally>
26. Wolf, C.: PicoRV32. <https://github.com/cliffordwolf/picorv32>
27. Wolf, C.: PonyLink. <https://github.com/cliffordwolf/PonyLink>
28. Wolf, C.: riscv-formal. <https://github.com/cliffordwolf/riscv-formal>
29. Wolf, C.: Yosys. <https://github.com/YosysHQ/yosys>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Nagini: A Static Verifier for Python

Marco Eilers^(✉)  and Peter Müller 

Department of Computer Science, ETH Zurich,
Zurich, Switzerland
{marco.eilers,peter.mueller}@inf.ethz.ch



Abstract. We present Nagini, an automated, modular verifier for statically-typed, concurrent Python 3 programs, built on the Viper verification infrastructure. Combining established concepts with new ideas, Nagini can verify memory safety, functional properties, termination, deadlock freedom, and input/output behavior. Our experiments show that Nagini is able to verify non-trivial properties of real-world Python code.

1 Introduction

Dynamic languages have become widely used because of their expressiveness and ease of use. The Python language in particular is popular in domains like teaching, prototyping, and more recently data science. Python’s lack of safety guarantees can be problematic when, as is increasingly the case, it is used for critical applications with high correctness demands. The Python community has reacted to this trend by integrating type annotations and optional static type checking into the language [20]. However, there is currently virtually no tool support for reasoning about Python programs beyond type safety.

We present Nagini, a sound verifier for statically-typed, concurrent Python programs. Nagini can prove memory safety, data race freedom, and user-supplied assertions. Nagini performs *modular* verification, which is important for verification to scale and to be able to verify libraries, and *automates* the verification process for programs annotated with specifications.

Nagini builds on many techniques established in existing tools: (1) Like VeriFast [10] and other tools [4, 19, 22], it uses separation logic style permissions [16] in order to locally reason about concurrent programs. (2) Like .NET Code Contracts [7], it uses a contract library to enable users to write code-level specifications. (3) Like many verification tools [2, 6, 11, 13], it verifies programs by encoding the program and its specification into an intermediate verification language [1, 8], namely Viper [14], for which automatic verifiers already exist.

Nagini combines these techniques with new ideas in order to verify advanced properties and handle the dynamic aspects of Python. In particular, Nagini implements a comprehensive system for verifying finite blocking [5] and input/output behavior [18], and builds on Mypy [12] to verify safety while also supporting important dynamic language features. Nagini is intended for verifying substantial, real-world code, and is currently used to verify the Python

implementation of the SCION internet architecture [3]. To our knowledge, it is the first tool to enable automatic verification of Python code. Existing tools for JavaScript [21, 24] also target a dynamic language, but focus on faithfully modeling JavaScript’s complex semantics rather than practical verification of high-level properties.

Due to its wide range of verifiable properties, Nagini has applications in many domains: In addition to memory safety, programmers can choose to prove that a server implementation will stay responsive, that data science code has desired functional properties, or that algorithms terminate and preserve certain invariants, for example in a teaching context. Nagini is open-source and available online¹, and can be used from the popular PyCharm IDE via a prototype plugin.

In this paper, we describe Nagini’s supported Python subset and specification language, give an overview of its implementation and the encoding from Python to Viper, and provide an experimental evaluation of Nagini on real-world code.

2 Language and Specifications

Python Subset: Nagini requires input programs to comply to the static, nominal type system defined in PEP 484 [20] as implemented in the Mypy type checker [12], which requires type annotations for function parameters and return types, but can normally infer types of local variables. Nagini fully supports the non-gradual part of Mypy’s type system, including generics and union types.

The Python subset accepted by Mypy and Nagini can accommodate most real Python programs, potentially via some workarounds like using union types instead of structural typing. While our subset is statically typed, it includes many features and potential pitfalls not found in static languages, such as dynamic addition and removal fields from objects. Some other features like reflection and dynamic code generation are not supported.

Where compromises are necessary, Nagini aims for modularity, performance, and completeness for features typically found in user code over general support for all language features. As an example, Nagini works with a simplified model of Python’s object attribute lookup behavior: A simple attribute access in Python leads to the invocation of several “magic” methods, which, if modelled correctly, would result in an overhead that would likely make automatic verification intractable. Nagini exploits the fact that these methods are mostly used to implement decorators, metaclasses, and system libraries, but rarely in user code. It assumes the default behavior of those methods, and implements direct support for frequently-used decorators and metaclasses that change their behavior. Importantly, Nagini flags an error if verified programs override these methods or are otherwise outside the supported subset, and is therefore sound.

Specification Language: Nagini includes a library of specification functions similar to .NET Code Contracts [7] to express pre- and postconditions, loop invariants, and other assertions. Calls to these functions are interpreted as specifications by Nagini, but can be automatically removed before execution. Users can

¹ <https://github.com/marcoeilers/nagini>.

```

1  from nagini_contracts.contracts import *
2  from typing import List
3  import db
4
5  class Ticket:
6      def __init__(self, show: int, row: int, seat: int) -> None:
7          self.show_id = show
8          self.row, self.seat = row, seat
9          Fold(self.state())
10         Ensures(self.state() and MayCreate(self, 'discount_code'))
11
12     @Predicate
13     def state(self) -> bool:
14         return Acc(self.show_id) and Acc(self.row) and Acc(self.seat)
15
16 def order_tickets(num: int, show_id: int, code: str=None) -> List[Ticket]:
17     Requires(num > 0)
18     Exsures(SoldoutException, True)
19     seats = db.get_seats(show_id, num)
20     res = [] # type: List[Ticket]
21     for row, seat in seats:
22         Invariant(list_pred(res))
23         Invariant(Forall(res, lambda t: t.state() and
24                        Implies(code is not None, Acc(t.discount_code))))
25         Invariant(MustTerminate(len(seats) - len(res)))
26         ticket = Ticket(show_id, row, seat)
27         if code:
28             ticket.discount_code = code
29         res.append(ticket)
30     return res

```

Fig. 1. Example program demonstrating Nagini’s specification language. Contract functions are highlighted in italics. Note that functional specifications and postconditions are largely omitted to highlight the different specification constructs.

annotate Mypy-style type stub files for external libraries with specifications; the program will then be verified assuming they are correct. A detailed explanation of the specification language can be found in Nagini’s Wiki².

An example of an annotated program is shown in Fig. 1. The first two lines import the contract library and Python’s library for type annotations. Pre- and postconditions are declared via calls to the contract functions *Requires* and *Ensures* in lines 17 and 10, respectively. The arguments of these functions are interpreted as assertions, which can be side-effect free boolean Python expressions or calls to other contract functions. Similarly, loops must be annotated with invariants (line 22), and special *exceptional* postconditions specify which exceptions a method may raise, and what postconditions must hold in this case. The *Exsures* annotation in line 18 states that a *SoldoutException* may be raised and makes no guarantees in this case. The invariant *MustTerminate* in line 25 specifies that the loop terminates; the argument represents a ranking function [5].

Like the underlying Viper language, Nagini uses Implicit Dynamic Frames (IDF) [23], a variation of separation logic [16], to achieve framing and allow local reasoning in the presence of concurrency. IDF establishes a system of *permissions* for heap locations that roughly corresponds to separation logic’s points-to predicates. Methods may only read or write heap locations they currently hold a permission for, and can specify which permissions they require from and give

² <https://github.com/marcoeilers/nagini/wiki>.

back to their caller in their pre- and postconditions. Since there is only ever a single permission per heap location, holding a permission guarantees that neither other threads nor called methods can modify the respective location.

In Nagini, a permission is created when a field is assigned to for the first time; e.g., when executing line 9, the `__init__` method will have permission to three fields. Permission assertions are expressed using the `Acc` function (line 14). Assertions can be abstracted over using predicates [17], declared in Nagini by using annotated functions (line 12). In the example, the constructor of `Ticket` bundles all available permissions in the predicate `state` using the ghost statement `Fold` in line 9 and subsequently returns this predicate to its caller via its postcondition.

In addition, Nagini offers a second kind of permission that allows *creating* a field that does not currently exist, but cannot be used for reading (since that would cause a runtime error). Constructors implicitly get this kind of permission for every field mentioned in a class; in the example, such a permissions is returned to the caller (line 10) and used in line 28. The loop invariant contains the permission to modify the `res` list using one of several built-in predicates for Python’s standard data types (line 22) as well as permissions to the fields of all objects in the list (line 23). This kind of *quantified permission* [15], corresponding to separation logic’s iterated separating conjunction, is one of two supported ways to express permissions over unbounded numbers of heap locations.

Other contract functions allow specifying, e.g., I/O behavior, and some have variations for advanced users, e.g., the `Forall` function can take trigger expressions to specify when the underlying SMT solver should instantiate the quantifier.

Verified properties: Nagini verifies some safety properties by default: Verified programs will not raise runtime errors or undeclared exceptions. The permission system guarantees that verified code is memory safe and free of data races. Nagini also verifies some properties that Mypy only checks optimistically, e.g., that referenced names are defined before they are used. As an example, if the `Ticket` class were defined after the `order_tickets` function, Nagini would not allow calls to the function *before* the class definition, because of the call in line 26.

Beyond this, Nagini can verify (1) functional properties, (2) input/output properties, i.e., which I/O operations may or must occur, using a generalization of the method by Penninckx et al. [18], and (3) finite blocking [5], i.e., that no thread blocks indefinitely when trying to acquire a lock or join another thread, which includes deadlock freedom and termination. Verification is modular in the sense that adding code to a program only requires verifying the added parts; any code that verified before is guaranteed to still verify. Top level statements are an exception and have to be reverified when any part of the program changes, since Python’s import mechanism is inherently non-modular.

3 Implementation

Nagini’s verification workflow is depicted in Fig. 2. After parsing, Nagini invokes the Mypy type checker on the input and rejects the program if errors are found.

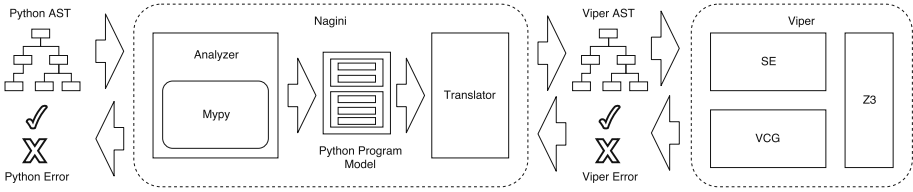


Fig. 2. Nagini verification workflow.

It then analyzes the input program and extracts structural information into an internal model, which is then encoded into a Viper program. The program is verified using one of the two Viper backends, based on either symbolic execution (SE) or verification condition generation (VCG), respectively. Any resulting Viper-level error messages are mapped back to a Python-level error.

Encoding: Nagini encodes Python programs into Viper programs that verify only if the original program was correct. At the top level, Viper programs consist of *methods*, whose bodies contain imperative code, side-effect free *functions*, and the aforementioned *predicates*, as well as *domains*, which can be used to declare and axiomatize custom data types. The structure of a created Viper program roughly follows the structure of the Python program: Each function in the Python program corresponds to either a method, a function, or a predicate in the Viper program, depending on its annotation. Additional Viper methods are generated to check proof obligations like behavioral subtyping and to model the execution of all top level statements.

Nagini maintains various kinds of ghost state, e.g., for verifying finite blocking and to represent which names are currently defined. It models Python’s type system using a Viper domain axiomatized to reflect subtype relations. Nagini desugars complex Python language constructs into simple ones that exist in Viper, but subtle language differences often require additional effort in the encoding. As an example, Viper distinguishes references from primitive values whereas Python does not, requiring boxing and unboxing operations in the encoding.

Tool interaction: Nagini is invoked on an annotated Python file, and verifies this file and all (transitive) imports without user interaction. It then outputs either a success message or Python-level error messages that indicate type or verification errors, use of unsupported features, or invalid specifications, along with the source location. As an example, removing the `Fold` statement in line 9 of Fig. 1 yields the error message “Postcondition of `__init__` might not hold. There might be insufficient permission to access `self.state()`. (example.py@10.16)”.

4 Evaluation

In addition to having a comprehensive test suite of over 12,500 lines of code, we have evaluated Nagini on a set of examples containing (parts of) implemen-

| | Example | LOC / Spec. | Viper LOC | SF | FC | FB | IO | T_{Seq} | T_{Par} |
|----|------------------------------|-------------|-----------|----|----|----|----|-----------|-----------|
| 1 | rosetta/quicksort | 31 / 10 | 635 | ✓ | - | ✓ | - | 8.48 | 8.31 |
| 2 | interactivepython/bst | 145 / 65 | 947 | ✓ | ✓ | - | - | 57.44 | 41.80 |
| 3 | keon/knapsack | 33 / 10 | 864 | ✓ | - | - | - | 19.39 | 14.49 |
| 4 | wikipedia/duck_typing | 19 / 0 | 486 | ✓ | - | - | - | 1.82 | 1.92 |
| 5 | scion/path_store | 207 / 94 | 2133 | ✗ | - | - | - | 51.37 | 35.26 |
| 6 | example | 40 / 19 | 736 | ✓ | - | ✓ | - | 6.11 | 5.91 |
| 7 | verifast/brackets_checker | 143 / 82 | 1081 | ✓ | ✓ | ✓ | ✓ | 7.66 | 6.63 |
| 8 | verifast/putchar_with_buffer | 139 / 88 | 865 | ✓ | - | ✓ | ✓ | 4.74 | 4.29 |
| 9 | chalice2viper/watchdog | 66 / 22 | 769 | ✓ | - | ✓ | - | 3.66 | 3.41 |
| 10 | parkinson/recell | 46 / 25 | 561 | ✓ | ✓ | - | - | 2.09 | 2.07 |

Fig. 3. Experiments. For each example, we list the lines of code (excluding whitespace and comments), the number of those lines that are used for specifications, the length of the resulting Viper program, properties (SF = safety, FC = functional correctness, FB = finite blocking, IO = input/output behavior) that could be verified (✓), could not be verified (✗) or were not attempted (-), and the verification times with Viper’s SE backend, sequential and parallelized, in seconds.

tations of standard algorithms from the internet³, the example from Fig. 1, a class from the SCION implementation, as well as examples from other verifiers translated to Python. Figure 3 shows the examples and which properties were verified; the functional property we proved for the binary search tree implementation is that it maintains a sorted tree. The examples cover language features like inheritance (example 10), comprehensions (3), dynamic field addition (6), operator overloading (3), union types (4), threads and locks (9), as well as specification constructs like quantified permissions (6) and predicate families (10). Nagini correctly finds an error in the SCION example and successfully verifies all other examples.

The runtimes shown in Fig. 3 were measured by averaging over ten runs on a Lenovo Thinkpad T450s running Ubuntu 16.04, Python 3.5 and OpenJDK 8 on a warmed-up JVM. They show that Nagini can effectively verify non-trivial properties of real-life Python programs in reasonable time. Due to modular verification, parts of a program can be verified independently and in parallel (which Nagini does by default), so that larger programs will not inherently lead to performance problems. This is demonstrated by the speedup achieved via parallelization on the two larger examples; for the smaller ones, verification time is dominated by a single complex method. Additionally, the annotation overhead is well within the range of other verification tools [9].

Acknowledgements. Thanks to Vytautas Astrauskas, Samuel Hitz, and Fábio Pakk Selmi-Dei for their contributions to Nagini. We gratefully acknowledge support from the Zurich Information Security and Privacy Center (ZISC).

³ We chose examples that do not make use of dynamic features or external libraries from rosettacode.org, interactivepython.org and github.com/keon/algorithms.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. *Commun. ACM* **54**(6), 81–91 (2011)
3. Barrera, D., Chuat, L., Perrig, A., Reischuk, R.M., Szalachowski, P.: The scion internet architecture. *Commun. ACM* **60**(6), 56–65 (2017)
4. Berdine, J., Calcagno, C., O’Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_6
5. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. In: Boyland, J.T. (ed.) European Conference on Object-Oriented Programming (ECOOP). LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl (2015)
6. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: contract-based modular verification of concurrent C. In: 2009 31st International Conference on Software Engineering - Companion Volume, pp. 429–430, May 2009
7. Fähndrich, M., Barnett, M., Logozzo, F.: Code contracts (2008). <http://research.microsoft.com/contracts>
8. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
9. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: end-to-end security via automated full-system verification. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, Broomfield, CO, USA, 6–8 October 2014, pp. 165–181 (2014)
10. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
11. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. *Formal Aspects Comput.* **27**(3), 573–609 (2015)
12. Lehtosalo, J., et al.: Mypy - optional static typing for python (2017). <http://mypy-lang.org>
13. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
14. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
15. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 405–425. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_22

16. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
17. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 247–258. ACM, New York (2005)
18. Penninckx, W., Jacobs, B., Piessens, F.: Sound, modular and compositional verification of the input/output behavior of programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 158–182. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_7
19. Piskac, R., Wies, T., Zufferey, D.: GRASShopper: complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 124–139. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_9
20. van Rossum, G., Lehtosalo, J., Langa, L.: Type Hints (2014). <https://www.python.org/dev/peps/pep-0484/>
21. Santos, J.F., Maksimovic, P., Naudziuniene, D., Wood, T., Gardner, P.: JaVert: JavaScript verification toolchain. PACMPL **2**(POPL), 50:1–50:33 (2018)
22. Smans, J., Jacobs, B., Piessens, F.: VeriCool: an automatic verifier for a concurrent object-oriented language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_14
23. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (May 2012)
24. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, Part of SPLASH 2016, Amsterdam, The Netherlands, 30 October–4 November 2016, pp. 74–91 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





PEREGRINE: A Tool for the Analysis of Population Protocols

Michael Blondin¹, Javier Esparza²,
and Stefan Jaax³

Technische Universität München, Munich, Germany
{blondimi, esparza, jaax}@in.tum.de



Abstract. We introduce PEREGRINE, the first tool for the analysis and parameterized verification of population protocols. Population protocols are a model of computation very much studied by the distributed computing community, in which mobile anonymous agents interact stochastically to achieve a common task. PEREGRINE allows users to design protocols, to simulate them both manually and automatically, to gather statistics of properties such as convergence speed, and to verify correctness automatically. This paper describes the features of PEREGRINE and their implementation.

Keywords: Population protocols · Distributed computing
Parameterized verification · Simulation

1 Introduction

Population protocols [1, 3, 4] are a model of distributed computing in which replicated, mobile agents with limited computational power interact stochastically to achieve a common task. They provide a simple and elegant formalism to model, e.g., networks of passively mobile sensors [1, 5], trust propagation [13], evolutionary dynamics [14], and chemical systems, under the name chemical reaction networks [12, 16, 19].

Population protocols are parameterized: the number of agents does not change during the execution of the protocol, but is *a priori* unbounded. A protocol is correct if it behaves correctly for all of its infinitely many initial configurations. For this reason, it is challenging to design correct and efficient protocols.

In this paper we introduce PEREGRINE¹, the first tool for the parameterized analysis of population protocols. PEREGRINE is intended for use by researchers in distributed computing and systems biology. It allows the user to specify protocols either through an editor or as simple scripts, and to analyze them via a

M. Blondin was supported by the Fonds de recherche du Québec – Nature et technologies (FRQNT).

¹ PEREGRINE can be found at <https://peregrine.model.in.tum.de>.

graphical interface. The analysis features of PEREGRINE include manual step-by-step simulation; automatic sampling; statistics generation of average convergence speed; detection of incorrect executions through simulation; and formal verification of correctness. The first four features are supported for all protocols, while verification is supported for silent protocols, a large subclass of protocols [6]. Verification is performed automatically over *all* of the infinitely many initial configurations using the recent approach of [6] for solving the so-called well-specification problem.

Related Work. The problem of automatically verifying that a population protocol conforms to its specification for *one fixed initial configuration* has been considered in [10, 11, 17, 20]. In [10], *ad hoc* search algorithms are used. In [11, 17], the authors show how to model the problem in the probabilistic model checker PRISM, and under certain conditions in SPIN. In [20], the problem is modeled with the PAT toolkit for model checking under fairness assumptions. All these tools increase our confidence in the correctness of a protocol. However, compared to PEREGRINE, they are not visual tools, they do not offer simulation capabilities, and they can only verify the correctness of a protocol for a finite number of initial configurations, with typically a small number of agents. PEREGRINE proves correctness for all of the infinitely many initial configurations, with an arbitrarily large number of agents.

As mentioned in the introduction, population protocols are isomorphic to chemical reaction networks (CRNs), a popular model in natural computing. Cardelli et al. have recently developed model checking techniques and analysis algorithms for *stochastic* CRNs [7–9]. The problems studied therein are incomparable to the parameterized questions addressed by PEREGRINE.

The verification algorithm of PEREGRINE is based on [6], where a novel approach for the parameterized verification of silent population protocols has been presented. The command-line tool of [6] only offers support for proving correctness, with no functionality for visualization or simulation. Further, contrary to PEREGRINE, the tool cannot produce counterexamples when correctness fails.

2 Population Protocols

We introduce population protocols through a simple example and then briefly formalize the model. We refer the reader to [4] for a more thorough but still intuitive presentation. Suppose anonymous and mobile agents wish to take a majority vote. Intuitively, *anonymous* means that agents have no identity, and *mobile* that agents are “wandering around”, and can only interact whenever they bump into each other. In order to vote, all agents conduct the following protocol. Each agent is in one out of four states $\{Y, N, y, n\}$. Initially all agents are in the states Y or N , corresponding to how they want to vote (states y, n are auxiliary states). Agents repeatedly interact pairwise according to the following rules:

$$a: YN \mapsto yn \quad b: Yn \mapsto Yy \quad c: Ny \mapsto Nn \quad d: yn \mapsto yy$$

For example, if the population initially has two agents of opinion “yes” and one agent of opinion “no”, then a possible execution is:

$$\langle \underline{Y}, Y, \underline{N} \rangle \xrightarrow{a} \langle y, \underline{Y}, \underline{n} \rangle \xrightarrow{b} \langle y, Y, y \rangle, \quad (1)$$

where e.g. $\langle Y, Y, N \rangle$ denotes the multiset with two agents in state Y and one agent in state N .

The goal of every population protocol is to ensure that the agents eventually reach a lasting consensus, i.e., a multiset in which (1) either all agents are in “yes”-states, or all agents are in “no”-states, and (2) further interactions do not destroy the consensus. On top of this universal specification, each protocol has an individual goal, determining which initial configurations should reach the “yes” and the “no” lasting consensus. In the majority protocol above, the agents should reach a “yes”-consensus iff 50% or more agents vote “yes”.

Execution (1) above leads to a lasting “yes”-consensus; further, the consensus is the right one, since 2 out of 3 agents voted “yes”. In fact, assuming agents interact uniformly and independently at random, the above protocol is correct: executions almost surely reach a correct lasting consensus.

More formally, a population protocol is a tuple (Q, T, I, O) where Q is a finite set of *states*, $T \subseteq Q^2 \times Q^2$ is a set of *transitions*, $I \subseteq Q$ are the *initial states* and $O: Q \rightarrow \{0, 1\}$ is the *output mapping*. A *configuration* is a non-empty multiset over Q , an *initial configuration* is a non-empty multiset over I , and a configuration is *terminal* if it cannot be altered by any transition. A configuration is in a *consensus* if all of its states map to the same output under O .

An *execution* is a finite or infinite sequence $C_0 \xrightarrow{t_1} C_1 \xrightarrow{t_2} \dots$ such that C_i is obtained from applying transition t_i to C_{i-1} . A *fair execution* is either a finite execution that reaches a terminal configuration, or an infinite execution such that if $\{i \in \mathbb{N} : C_i \xrightarrow{*} D\}$ is infinite, then $\{i \in \mathbb{N} : C_i = D\}$ is infinite for any configuration D . In other words, fairness ensures that a configuration cannot be avoided forever if it is reachable infinitely often. Fairness is an abstraction of the random interactions occurring within a population. A configuration C is in a *lasting consensus* if every execution from C only leads to configurations of the same consensus.

If for every initial configuration C , all fair executions from C lead to a lasting consensus $\varphi(C) \in \{0, 1\}$, then we say that the protocol *computes* the predicate φ . For example, the above majority protocol with $O(Y) = O(y) = 1$ and $O(N) = O(n) = 0$ computes the predicate $C[Y] \geq C[N]$, where $C[x]$ denotes the number of occurrences of state x in C . A protocol does not necessarily compute a predicate. For example, if we alter the majority protocol by removing transition d , then $\langle Y, N \rangle \xrightarrow{a} \langle y, n \rangle$ is a fair execution, but $\langle y, n \rangle$ is not in a consensus. In other words, transition d acts as a tie-breaker which allows to reach the consensus configuration $\langle y, y \rangle$. A protocol that computes a predicate is said to be *well-specified*. It is well-known that well-specified population protocols compute precisely the predicates definable in Presburger arithmetic [3]. On top of different *majority protocols* for the predicate $C[x] \geq C[y]$, the literature contains, e.g., different families of so-called *flock-of-birds protocols* for the predicates $C[x] \geq c$,

where c is an integer constant, and families of *threshold protocols* for the predicates $a_1 \cdot C[x_1] + \dots + a_n \cdot C[x_n] \geq c$, where a_1, \dots, a_n, c are integer constants and x_1, \dots, x_n are initial states.

3 Analyzing Population Protocols

PEREGRINE is a web tool with a JavaScript frontend and a Haskell backend. The backend makes use of the SMT solver Z3 [15] to test satisfiability of Presburger arithmetic formulas. The user has access to four main features through the graphical frontend. We present these features in the remainder of the section.

Protocol Description. PEREGRINE offers a description language for both single protocols and families of protocols depending on some parameters. Single protocols are described either through a graphical editor or as simple Python scripts. Families of protocols (called parametric protocols) can only be specified as scripts, but PEREGRINE assists the user by generating a code skeleton.

Simulation. Population protocols can be simulated through a graphical player depicted in Fig. 1. The user can pick an initial configuration and simulate the protocol by either manual selection of interactions, or by letting a scheduler pick interactions uniformly at random. The simulator keeps a history of the execution which can be rewound at any time, making it easy to experiment with the different behaviours of a protocol. Configurations can be displayed in two ways: either as explicit populations, as illustrated in Fig. 1, or as bar charts of the states count, more convenient for large populations.



Fig. 1. Simulation of the majority protocol from the initial configuration $\{5 \cdot Y, 10 \cdot N\}$.

Statistics. PEREGRINE can generate statistics from batch simulations. The user provides four parameters: s_{\min} , s_{\max} , m and n . PEREGRINE generates n random executions as follows. For each execution, a number s is picked uniformly at random from $[s_{\min}, s_{\max}]$, and an initial configuration of size s is then picked uniformly at random. Each step of an execution is picked uniformly at random

among enabled interactions. If no terminal configuration is reached within m steps, then the simulation halts. In the end, n executions of length at most m are gathered. PEREGRINE classifies the generated executions according to their consensus, and computes statistics on the convergence speed (see the next two paragraphs). The results can be visualized in different ways, and the raw data can be exported as a JSON file.

Consensus. For each random execution, PEREGRINE checks whether the last configuration of an execution is in a consensus and, if so, whether the consensus corresponds to the expected output of the protocol. PEREGRINE reports which percentage of the executions reach a consensus, and whether the consensus is correct and/or lasting. In normal mode, PEREGRINE only classifies an execution as lasting consensus if it ends in a terminal configuration. In the *increased accuracy* mode, if the execution ends in a configuration C of consensus $b \in \{0, 1\}$, then the model checker LOLA [18] is used to determine whether there exists a configuration C' such that $C \xrightarrow{*} C'$ and C' is not of consensus b . If it is not the case, then PEREGRINE concludes that C is in a lasting consensus. PEREGRINE plots the percentage of executions in each category as a function of the population size, as illustrated on the left of Fig. 2.

Average Convergence Speed. PEREGRINE also provides statistics on the convergence speed of a protocol. Let $C_0 \xrightarrow{t_1} C_1 \xrightarrow{t_2} \dots \xrightarrow{t_\ell} C_\ell$ be an execution such that C_ℓ is in a consensus $b \in \{0, 1\}$. The *number of steps to convergence* of the execution is defined as 0 if all configurations are of consensus b , and otherwise as $i+1$, where i is the largest index such that C_i is not in consensus b . For each population size, PEREGRINE computes the average number of steps to convergence of all consensus executions of that population size, and plots the information as illustrated on the right of Fig. 2.

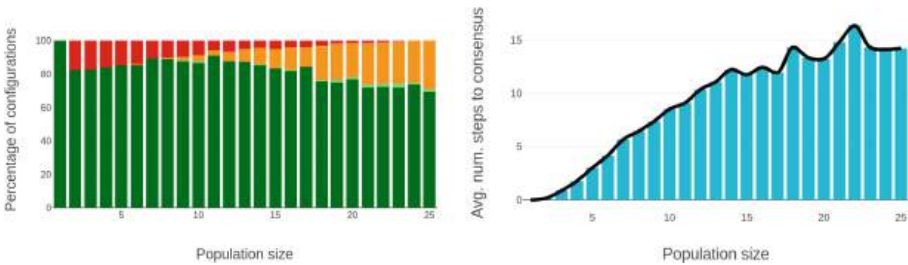


Fig. 2. Statistics for 5000 random executions of the approximate majority protocol of [2], of length at most 40, from initial configurations of size at most 25. The left plot shows the percentage of executions reaching a consensus (dark green: lasting correct, light green: correct, light red: incorrect, dark red: lasting incorrect) and no consensus (orange). In this example the occurrences of light red are negligible. The right plot shows the average number of steps to convergence. (Color figure online)



The protocol does not satisfy correctness.

Peregrine found a finite execution π from initial configuration C_0 to configuration C_1 that violates correctness. The protocol should reach consensus *true* from C_0 , but instead π reaches C_1 which is terminal and not in a consensus. Configurations C_0 and C_1 contain 2 agents, and execution π has length 1.

SHOW COUNTER-EXAMPLE ▼



EXPORT

You may replay execution π :



Fig. 3. Verification of the majority protocol of Sect. 2 without transition $d: yn \mapsto yy$.

Verification. PEREGRINE can automatically verify that a population protocol computes a given predicate. Predicates can be specified by the user in quantifier-free Presburger arithmetic extended with the family of predicates $\{x \equiv y \pmod{c}\}_{c \geq 2}$, which is equivalent to Presburger arithmetic. For example, for the majority protocol of Sect. 2, the user simply specifies $C[Y] \geq C[N]$.

PEREGRINE implements the approach of [6] to verify correctness of protocols which are silent. A protocol is said to be *silent* if from every initial configuration, every fair execution leads to a terminal configuration. The majority protocol of Sect. 2 and most existing protocols from the literature are silent [6]. We briefly describe the approach of [6] and how it is integrated into PEREGRINE.

Suppose we are given a population protocol \mathcal{P} and we wish to determine whether it computes a predicate φ . The procedure first tries to prove that \mathcal{P} is silent. This is done by verifying a more restricted condition called *layered termination*. Verifying the latter property reduces to testing satisfiability of a Presburger arithmetic formula. If this formula holds, then the protocol is silent, otherwise no conclusion is derived. However, essentially all existing silent protocols satisfy layered termination [6].

Once \mathcal{P} is proven to be silent, the procedure attempts to prove that no “bad execution” exists. More precisely, it checks whether there exist configurations C_0 and C_1 such that $C_0 \xrightarrow{*} C_1$, C_0 is initial, C_1 is terminal, and C_1 is not in consensus $\varphi(C_0) \in \{0, 1\}$. Since reachability is not definable in Presburger arithmetic, a Presburger-definable over-approximation $\xrightarrow{*}$ of reachability, borrowed from Petri net theory, is used instead. We obtain the following formula $\Phi_{\text{bad-exec}}$:

$$\exists C_0, C_1: C_0 \xrightarrow{*} C_1 \wedge \bigwedge_{q \notin I} C_0[q] = 0 \wedge \bigwedge_{t \in T} \text{succ}(C_1, t) \subseteq \{C_1\} \wedge \bigvee_{q \in C_1} (O(q) = \neg \varphi(C_0)).$$

If $\Phi_{\text{bad-exec}}$ is unsatisfiable, then \mathcal{P} is correct. Otherwise, no conclusion is reached, and $\Phi_{\text{bad-exec}}$ is iteratively strengthened by enriching the over-approximation $\xrightarrow{*}$. Whenever $\Phi_{\text{bad-exec}}$ is satisfied by (C_0, C_1) , PEREGRINE calls the model-checker LOLA to test whether C_1 is indeed reachable from C_0 . If so, then PEREGRINE reports \mathcal{P} to be incorrect, and generates a counter-example execution, which can be replayed or exported as a JSON file (see Fig. 3).

Currently PEREGRINE can verify protocols with up to a hundred states and a few thousands transitions. The bottleneck is the size of the constraint system. Due to lack of space, we refer the reader to [6] for detailed experimental results.

References

1. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 290–299 (2004). <https://doi.org/10.1145/1011767.1011810>
2. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. *Distrib. Comput.* **21**(2), 87–102 (2008). <https://doi.org/10.1007/s00446-008-0059-z>
3. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. *Distrib. Comput.* **20**(4), 279–304 (2007). <https://doi.org/10.1007/s00446-007-0040-2>
4. Aspnes, J., Ruppert, E.: An introduction to population protocols. In: Garbinato, B., Miranda, H., Rodrigues, L. (eds.) *Middleware for Network Eccentric and Mobile Applications*, pp. 97–120. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-89707-1_5
5. Beauquier, J., Blanchard, P., Burman, J., Delaët, S.: Tight complexity analysis of population protocols with cover times - the ZebraNet example. *Theor. Comput. Sci.* **512**, 15–27 (2013). <https://doi.org/10.1016/j.tcs.2012.10.032>
6. Blondin, M., Esparza, J., Jaax, S., Meyer, P.J.: Towards efficient verification of population protocols. In: Proceedings of the 36th ACM Symposium on Principles of Distributed Computing (PODC), pp. 423–430 (2017). <https://doi.org/10.1145/3087801.3087816>
7. Cardelli, L., Češka, M., Fränzle, M., Kwiatkowska, M., Laurenti, L., Paoletti, N., Whitby, M.: Syntax-guided optimal synthesis for chemical reaction networks. In: Majumdar, R., Kunčák, V. (eds.) *CAV 2017. LNCS*, vol. 10427, pp. 375–395. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_20
8. Cardelli, L., Kwiatkowska, M., Laurenti, L.: Stochastic analysis of chemical reaction networks using linear noise approximation. *Biosystems* **149**, 26–33 (2016). <https://doi.org/10.1016/j.biosystems.2016.09.004>
9. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Syntactic Markovian bisimulation for chemical reaction networks. In: Aceto, L., et al. (eds.) *Models, Algorithms, Logics and Tools. LNCS*, vol. 10460, pp. 466–483. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_23
10. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Algorithmic verification of population protocols. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) *SSS 2010. LNCS*, vol. 6366, pp. 221–235. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16023-3_19
11. Clément, J., Delporte-Gallet, C., Fauconnier, H., Sighireanu, M.: Guidelines for the verification of population protocols. In: *ICDCS*, pp. 215–224. IEEE Computer Society (2011). <https://doi.org/10.1109/ICDCS.2011.36>
12. Cummings, R., Doty, D., Soloveichik, D.: Probability 1 computation with chemical reaction networks. *Nat. Comput.* **15**(2), 245–261 (2016). <https://doi.org/10.1007/s11047-015-9501-x>

13. Diamadi, Z., Fischer, M.J.: A simple game for the study of trust in distributed systems. *Wuhan Univ. J. Nat. Sci.* **6**(1), 72–82 (2001). <https://doi.org/10.1007/BF03160228>
14. Moran, P.A.P.: Random processes in genetics. *Math. Proc. Cambridge Philos. Soc.* **54**(1), 60–71 (1958). <https://doi.org/10.1017/S0305004100033193>
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24. z3 is available at <https://github.com/Z3Prover/z3>
16. Navlakha, S., Bar-Joseph, Z.: Distributed information processing in biological and computational systems. *Commun. ACM* **58**(1), 94–102 (2014). <https://doi.org/10.1145/2678280>
17. Pang, J., Luo, Z., Deng, Y.: On automatic verification of self-stabilizing population protocols. In: *Proceedings of the 2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 185–192 (2008). <https://doi.org/10.1109/TASE.2008.8>
18. Schmidt, K.: LoLA a low level analyser. In: Nielsen, M., Simpson, D. (eds.) *ICATPN 2000*. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44988-4_27. LoLA is available at <http://service-technology.org/lola/>
19. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with finite stochastic chemical reaction networks. *Nat. Comput.* **7**(4), 615–633 (2008). <https://doi.org/10.1007/s11047-008-9067-y>
20. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009*. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_59

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





ADAC: Automated Design of Approximate Circuits

Milan Češka^(✉), Jiří Matyáš, Vojtech Mrazek,
Lukas Sekanina, Zdenek Vasicek,
and Tomáš Vojnar

Faculty of Information Technology,
IT4Innovations Centre of Excellence,
Brno University of Technology,
Brno, Czech Republic
`ceskam@fit.vutbr.cz`



Abstract. Approximate circuits with relaxed requirements on functional correctness play an important role in the development of resource-efficient computer systems. Designing approximate circuits is a very complex and time-demanding process trying to find optimal trade-offs between the approximation error and resource savings. In this paper, we present ADAC—a novel framework for automated design of approximate arithmetic circuits. ADAC integrates in a unique way efficient simulation and formal methods for approximate equivalence checking into a search-based circuit optimisation. To make ADAC easily accessible, it is implemented as a module of the ABC tool: a state-of-the-art system for circuit synthesis and verification. Within several hours, ADAC is able to construct high-quality Pareto sets of complex circuits (including even 32-bit multipliers), providing useful trade-offs between the resource consumption and the error that is formally guaranteed. This demonstrates outstanding performance and scalability compared with other existing approaches.

1 Introduction

In the recent years, reduction of power consumption of computer systems and mobile devices has become one of the biggest challenges in the computer industry. *Approximate computing* has been established as a new research field aiming at reducing system resource demands (and, in particular, power demands) by relaxing the requirement that all computations are always performed correctly. Approximate computing exploits the fact that many applications, including image and multimedia processing, signal processing, data mining, machine learning, neural networks, and scientific computations, are *error resilient*, i.e.

This work was supported by the IT4Innovations excellence in science project No. LQ1602.

produce acceptable results even though the underlying computations are performed with a certain error. Therefore, the error can be used as a design metric and traded for chip area, power consumption, or runtime. Chippa et al. [7] claims that almost 80% of runtime is spent in procedures that could be approximated.

Approximate computing can be conducted at different system levels with arithmetic circuit approximation being one of the most popular as such circuits are frequently used in the core computations. In our work, we focus on functional approximation where the original circuit is replaced by a less complex one which exhibits some errors but improves non-functional circuit parameters such as power consumption or chip area. Circuit approximation can be formulated as an optimisation problem where the error and non-functional circuit parameters are conflicting design objectives. Designing complex approximate circuits is a time-demanding and error-prone process. Moreover, its automation is challenging too since the design space including candidate solutions is huge and checking that a candidate solution has the required error is itself a computationally demanding task, especially if formal guarantees on the error have to be ensured.

In this tool paper, we present *ADAC*¹—a novel framework for automated design of approximate circuits. The framework implements a design loop including (i) a *generator* of candidate solutions employing genetic search algorithms, (ii) an *evaluator* estimating non-functional parameters of a candidate solution, and (iii) a *verifier* checking that the candidate solution does not exceed the permissible error. ADAC is integrated as a new module into the ABC tool—a state-of-the-art and widely used system for circuit synthesis and verification [1]. The framework takes as the inputs:

- a golden combinational circuit in Verilog implementing the correct functionality,
- an error metric (such as the worst-case error, mean error, Hamming distance, etc.),
- a threshold on the error metric representing the maximal permissible error,
- a time limit on the overall design process, and
- a file specifying sizes of gates available to the design process.

With these inputs, ADAC searches for an approximate circuit satisfying the error threshold and having the minimal estimated chip area. Previous works [3, 14, 20, 22] confirmed that the chip area is a good optimization objective as it highly correlates with power consumption, which is a crucial target in approximate computing.

The results of [21] clearly demonstrate that search algorithms based on *Cartesian Genetic Programming* (CGP) [12] are well capable of generating high-quality approximate circuits. For complex circuits, however, a high number of candidate solutions has to be generated and evaluated, which significantly limits the scalability of the design process. Our framework implements several approaches for error evaluation suitable for different error metrics and application domains. They include both *SAT* and *BDD-based techniques* for

¹ <https://github.com/imatyas/ADAC>.

approximate equivalence checking providing *formal error guarantees* as well as a *bit-parallel circuit simulation* utilising the computing power of modern processors. We also implement a novel search strategy that drives the search towards *promptly verifiable approximate circuits*, which significantly accelerates the design process in many cases [3]. As such, the framework offers a unique integration of techniques based on simulation, formal reasoning, and evolutionary circuit optimisation. Our extensive experimental evaluation demonstrates that ADAC offers outstanding performance and scalability compared with existing methods and tools and paves a way towards an automated design process of complex provably-correct circuit approximations.

2 Architecture and Implementation

The ADAC framework has a modular architecture illustrated in Fig. 1.

The setup phase is responsible mainly for preparing a chromosome representation of the golden circuit. The circuit is given in a high-level Verilog format, which is first translated to a gate-level representation using the tool Yosys [25], and then the chromosome representation is obtained using our V2CH script. The setup phase is also responsible for generating a configuration file controlling the main design loop. It is generated from the user inputs and optional parameters for CGP and search strategies.

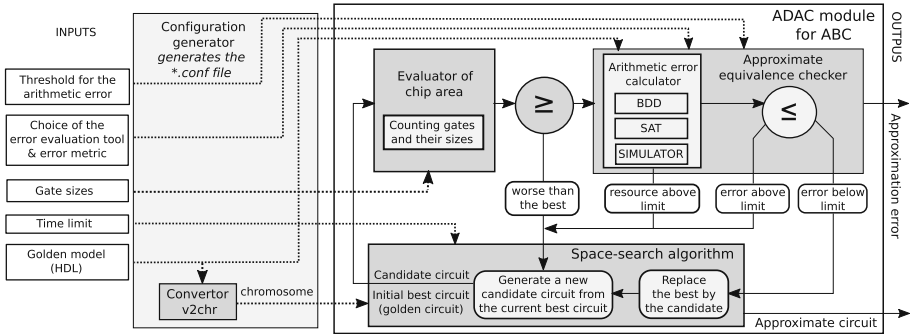


Fig. 1. A scheme of the ADAC architecture.

The design loop consists of three components: (i) a generator of candidate designs, (ii) an evaluator of non-functional parameters of the candidate circuit (currently estimating the chip area), and (iii) a verifier evaluating the candidate error. The chip area and the error form a basis of the *fitness function*, whose value is minimised via our search strategy. In particular, the fitness is infinity if the circuit error exceeds the given threshold, and the chip area otherwise. In the future, we plan to support a more general specification of the fitness. As an additional feature, ADAC can also quantify the difference (in the given metric) between two given circuits.

The real values of non-functional parameters, such as the chip area or the power-delay product (PDP), depend on the target technology, and the synthesis of an optimal implementation of the given circuit using the target technology is highly time-consuming. Therefore, our design loop currently uses the *chip area* as the sole non-functional parameter. The chip area is estimated as the sum of the sizes of the gates of the circuit, which are given as one of the inputs of ADAC. The chip area is typically a good estimate of the power consumption [3, 14, 20, 22]. The output of ADAC (in the gate-level Verilog format) can be passed to industrial circuit design tools to obtain accurate circuit parameters for the target technology. In our experiments, we report PDP for the 45 nm technology synthesised by the Synopsys Design Compiler [19].

We now briefly describe the candidate circuit generator and three methods for error evaluation that are currently supported in ADAC.

The *candidate circuit generator* is based on CGP where a candidate solution is encoded as a chromosome describing an oriented acyclic graph, given as a 2-dimensional array of 2-input nodes. Every node is numbered and is encoded by 3 integers where the first two numbers denote the inputs and the third represents the function of the node. New candidate circuits are obtained using a mutation operator that performs random changes in the chromosome. The mutations can either modify the node interconnection or functionality. The area of candidate circuits is reduced by making some nodes unreachable (such nodes, however, are removed only at the very end, and so they can still be mutated and even become reachable again). The candidates are evaluated, and the one with the best one is used in the next iteration of the design loop. The whole loop starts with the golden circuit and iteratively generates approximate solutions with better fitness values until a termination criterion (typically a given time limit) is met. Optionally, user can provide approximate circuit satisfying the threshold on the error as a seed to start with.

The *bit-parallel circuit simulation* supports all common error metrics, including the worst-case error (WCE), the mean error, the error rate representing the number of inputs leading to an incorrect output, and the Hamming distance. It utilises the power of modern processors by simulating the circuit on multiple inputs vectors (e.g. 64 inputs for 64-bit processors) in a single pass through the circuit [24]. However, despite the parallel processing that significantly accelerates the simulation, for circuits with arguments of larger bit-widths (beyond 12 bits), it is not feasible to simulate the circuits on all possible inputs, and so statistical guarantees on the approximation error are provided only.

The *BDD-based evaluation* also supports all common error metrics, and, unlike simulation, it is able to provide formal error guarantees for circuits with larger input bit-widths. For the purpose of the evaluation, the original correct circuit and its approximation are interconnected into an auxiliary circuit called a *miter* such that the error can be deduced from its output (e.g. to compute the error rate, the outputs of the golden and candidate circuits are subtracted, and the result is compared with 0). The miter is encoded as a BDD on which the circuit error is evaluated using BDD operations [22, 23]. However, this technique

does not scale well with the complexity of the circuits in terms of the number of their gates as the resulting BDD representation becomes prohibitively huge. Hence, this approach works well for large adders and similar circuits, but, it fails, e.g., for multipliers beyond 12-bits.

The *SAT-based evaluation* currently supports WCE only, but it provides formal guarantees and a superior performance to the BDD-based technique. ADAC implements a novel miter construction based on subtracting the output of the golden and approximate circuit, followed by a comparison with the error threshold [3]. The construction is optimised for SAT-based evaluation by avoiding long XOR chains known to cause poor performance of state-of-the-art SAT solvers [5, 9]. This allows us to exploit the ABC engine *improve*, designed originally for miter-based exact circuit equivalence checking, to quickly evaluate WCE.

The final ingredient of the design process is the *search strategy*. Apart from the standard evolutionary strategies based solely on the fitness function, ADAC also implements a novel verifiability-driven approach [3] combined with the SAT-based evaluation.

The *verifiability-driven search strategy* uses a limit L on the resources available to the underlying SAT decision procedure. The limit effectively controls the time the SAT solver can use. We require that every improving candidate has to be verifiable using the resource limit L . Therefore the strategy drives the search towards candidates that improve the fitness and can be promptly evaluated. As the result, we can evaluate in the given time a much larger set of candidate circuits. Our experiments indicate that this strategy often leads to a higher number of improving solutions and thus finds circuits having a smaller chip area meeting the permissible error. On the other hand, it can happen that, for a limit L , no improving sequence exists, while it exists for a slightly greater resource limit. We are currently implementing auto-adaptive techniques that should automatically select the adequate resource limit for the given circuit.

Integration to the ABC Tool. To make ADAC easily accessible, it is implemented as a new module for the ABC tool. ABC allows us to support an important subset of the Verilog specification and implementation language. We also utilize ABC to translate the circuits among different intermediate representations used for constructing miters. As mentioned before, we employ the *improve* engine in our SAT-based method for evaluating the WCE. Note that *improve* uses MiniSat [18] as the SAT solver. Despite the fact that ABC supports a BDD-based circuit representation and manipulation, we implemented our own BDD component (based on the BuDDy library [2]) that is tailored for evolutionary circuit approximation.

Extensibility. Due to its modular architecture, ADAC can be easily extended. Apart from the extensions mentioned above, we are working on a new component for error evaluation based on SAT counting methods (e.g. #SAT [4]) that could offer formal guarantees and a better scalability for the mean error and error-rate metrics, and on new candidate circuit generators counter-examples produced

during the verification of candidate circuits. In a long term perspective, we plan to generalise the underlying methods and support also design of approximate sequential circuits.

3 Evaluation, Related Works, and Applications

We first compare the performance of the different methods of circuit error evaluation supported in ADAC. For that, we use results from adder approximation obtained from 10 runs, each for 5 min. The table in Fig. 2 shows average runtimes of a single error evaluation using the bit-parallel simulation, the BDD-based approach, and the SAT-based approach. The reported speedups are with respect to the simulation. We can see that the simulation provides the best performance for small bit-widths only, but it does not scale well. The SAT-based method offers the best scalability and dominates for larger circuits, but it supports the WCE evaluation only. The BDD-based method, like simulation, supports all metrics and significantly outperforms the simulation for larger circuits. Note that, for more complex circuits such as multipliers, we would observe similar results with a worse relative performance of the BDD-based approach.

There indeed exist also other known methods for computing approximation errors for arithmetic circuits, including methods based on BDDs [6] or a SAT-based miter solution [5]. Comparing to ADAC, these methods are less scalable, which is demonstrated by the fact that they have been used for approximating multipliers limited to 8-bit operands and adders limited to 16-bit operands only. Apart from that, there are efficient methods for *exact* equivalence checking based on algebraic computations [8, 16]. However, they are so far not known for approximate equivalence checking.

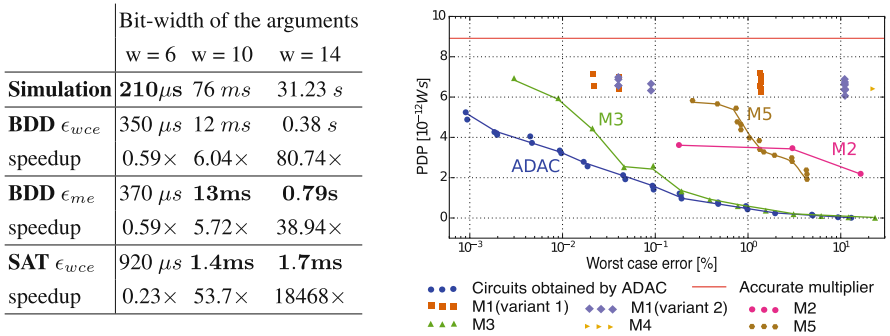


Fig. 2. (Left) Performance of error evaluation methods for adders. (Right) A comparison of 16-bit approximate multipliers designed by ADAC vs. the best known solutions.

Next, we compare the quality of approximate circuits obtained using ADAC with circuits that appeared in the literature. We consider 16-bit multipliers since existing approaches are not able to handle larger and more complex circuits. The different points in Fig. 2 correspond to circuits with different trade-

offs between WCE in % and the power-delay product (PDP²), which is a key non-functional circuit characteristic. These circuits were obtained using various existing approaches including: (M1) configurable circuits from the lpACLib library [17], (M2) the bit-significance-driven logic compression [15], (M3) the bit-width truncation [10], (M4) compositional techniques [11], and (M5) circuits from the EvoApproxLib library [13]. We can see that just the bit-width truncation can provide a quality of results comparable with ADAC (in terms of the PDP reduction for the given WCE), but for large target errors (20% WCE or more) only. For small target errors, ADAC clearly dominates.

Note that, for each target WCE, we performed 30 independent runs of CGP to obtain statistically significant results. For each run, ADAC was executed for 2 h on an Intel Xeon X5670 2.4 GHz processor using a single core. Also note that the individual runs are independent and thus can be easily parallelised.

Further, Fig. 3 presents approximate multipliers up to 32 bits obtained by ADAC. It shows Pareto fronts representing circuits with different compromises between WCE in % and PDP, and demonstrates that ADAC goes beyond capabilities of existing methods and tools. For each target WCE, ADAC was executed for 4 hours in the case of the 24-bit instances and for 6 hours in the case of the larger instances. Note that a 32-bit exact multiplier requires over 6,300 gates, and, to the best of our knowledge, ADAC is the first tool that is able to approximate such complex circuits with formal error guarantees.

Besides the approaches mentioned above, there also exist general-purpose methods, such as SALSA [14] or SASIMI [15], approximating circuits independently of their structure. We were unable to perform a direct comparison with them due to their implementation is not available, but based on the published results, ADAC is able to provide a significantly better scalability.

Practical Impacts. The following list briefly characterises several resource-aware applications that build on approximate circuits. The circuits were obtained using prototype implementations of the above mentioned approaches that are now integrated in ADAC.

Approximate multipliers for convolutional neural networks [14]. In such networks, millions of multiplications have to be performed. The usage of application-specific approximate multipliers led to 90% savings in terms of power consumption of the data path for a negligible drop in classification accuracy.

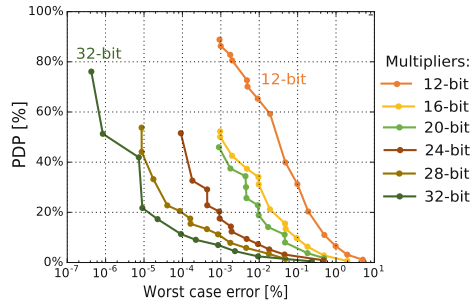


Fig. 3. Approximate multipliers designed by ADAC. 100% refers to PDP of the accurate circuits for the given bit-width.

² PDP characterises both the speed and energy efficiency of the circuit.

Approximate Adders and Subtractors for a Discrete Convolutional Transformation [22]. These adders and subtractors were designed to reduce the power consumption in video compression for the High Efficiency Video Coding (HEVC) standard. They show better quality/power trade-offs than implementations available in the literature. For example, a 25% power reduction for the same error was obtained in comparison with a recent highly-optimised implementation.

Approximate Adders and Multipliers for Image Processing [20]. These circuits were used in the development of efficient hardware implementations of filters and edge detectors. A 50% reduction was observed in the number of look-up tables used in a field programmable gate array for a negligible drop in the image visual quality.

References

1. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
2. BuDDy: A BDD package, January 2018. <http://buddy.sourceforge.net/manual/main.html>
3. Česka, M., Matyáš, J., Mrazek, V., Sekanina, L., Vasicek, Z., Vojnar, T.: Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In: Proceedings of the ICCAD 2017, pp. 416–423. IEEE (2017)
4. Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate probabilistic inference via word-level counting. In: Proceedings of the AAAI 2016, pp. 3218–3224. AAAI Press (2016)
5. Chandrasekharan, A., Soeken, M., Große, D., Drechsler, R.: Precise error determination of approximated components in sequential circuits with model checking. In: Proceedings of the DAC 2016, pp. 129:1–129:6. ACM (2016)
6. Chandrasekharan, A., Soeken, M., et al.: Approximation-aware rewriting of AIGs for error tolerant applications. In: Proceedings of the ICCAD 2016, pp. 83:1–83:8. ACM (2016)
7. Chippa, V.K., Chakradhar, S.T., Roy, K., Raghunathan, A.: Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of the DAC 2013, pp. 1–9. IEEE (2013)
8. Ciesielski, M., Yu, C., Brown, W., Liu, D., Rossi, A.: Verification of gate-level arithmetic circuits by function extraction. In: Proceedings of the DAC 2015. ACM (2015)
9. Han, C.-S., Jiang, J.-H.R.: When boolean satisfiability meets gaussian elimination in a simplex way. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 410–426. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_31
10. Jiang, H., Liu, C., Liu, L., Lombardi, F., Han, J.: A review, classification, and comparative evaluation of approximate arithmetic circuits. J. Emerg. Technol. Comput. Syst. **13**(4), 60:1–60:34 (2017)
11. Kulkarni, P., Gupta, P., Ercegovac, M.D.: Trading accuracy for power in a multiplier architecture. J. Low Power Electron. **7**(4), 490–501 (2011)
12. Miller, J.F.: Cartesian Genetic Programming. Springer, Berlin (2011). <https://doi.org/10.1007/978-3-642-17310-3>

13. Mrazek, V., Hrbacek, R., et al.: EvoApprox8B: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Proceedings of the DATE 2017, pp. 258–261. EDAA (2017)
14. Mrazek, V., Sarwar, S.S., Sekanina, L., Vasicek, Z., Roy, K.: Design of power-efficient approximate multipliers for approximate artificial neural networks. In: Proceedings of the ICCAD 2016, pp. 81:1–81:7. ACM (2016)
15. Qiqieh, I., Shafik, R., et al.: Energy-efficient approximate multiplier design using bit significance-driven logic compression. In: Proceedings of the DATE 2017. EDAA (2017)
16. Sayed-Ahmed, A., Große, D., et al.: Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In: Proceedings of the DATE 2016, pp. 1048–1053. IEEE (2016)
17. Shafique, M., Ahmad, W., et al.: A low latency generic accuracy configurable adder. In: Proceedings of the DAC 2015, pp. 86:1–86:6. ACM (2015)
18. Sorensson, N., Een, N.: MiniSat v1.13 – a sat solver with conflict-clause minimization. SAT 2005, no. 53, pp. 1–2 (2005)
19. Synopsys design compiler, January 2018. <https://www.synopsys.com/>
20. Vasicek, Z., Mrazek, V., Sekanina, L.: Evolutionary functional approximation of circuits implemented into FPGAs. In: Proceedings of the SSCI 2016, pp. 1–8. IEEE (2016)
21. Vasicek, Z., Sekanina, L.: Evolutionary approach to approximate digital circuits design. Trans. Evol. Comput. **19**(3), 432–444 (2015)
22. Vasicek, Z., Mrazek, V., Sekanina, L.: Towards low power approximate DCT architecture for HEVC standard. In: Proceedings of the DATE 2017, pp. 1576–1581. EDAA (2017)
23. Vasicek, Z., Sekanina, L.: Evolutionary design of complex approximate combinational circuits. Genet. Program Evolvable Mach. **17**(2), 169–192 (2016)
24. Vašíček, Z., Slaný, K.: Efficient phenotype evaluation in cartesian genetic programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) EuroGP 2012. LNCS, vol. 7244, pp. 266–278. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29139-5_23
25. Wolf, C.: Yosys open synthesis suite, January 2018. <http://www.clifford.at/yosys/>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Probabilistic Systems



Value Iteration for Simple Stochastic Games: Stopping Criterion and Learning Algorithm

Edon Kelmendi, Julia Krämer, Jan Křetínský^(✉),
and Maximilian Weininger

Technical University of Munich, Munich, Germany
jan.kretinsky@tum.de



Abstract. Simple stochastic games can be solved by value iteration (VI), which yields a sequence of under-approximations of the value of the game. This sequence is guaranteed to converge to the value only in the limit. Since no stopping criterion is known, this technique does not provide any guarantees on its results. We provide the first stopping criterion for VI on simple stochastic games. It is achieved by additionally computing a convergent sequence of *over-approximations* of the value, relying on an analysis of the game graph. Consequently, VI becomes an anytime algorithm returning the approximation of the value and the current error bound. As another consequence, we can provide a simulation-based asynchronous VI algorithm, which yields the same guarantees, but without necessarily exploring the whole game graph.

1 Introduction

Simple Stochastic Game. (SG) [Con92] is a zero-sum two-player game played on a graph by Maximizer and Minimizer, who choose actions in their respective vertices (also called states). Each action is associated with a probability distribution determining the next state to move to. The objective of Maximizer is to maximize the probability of reaching a given target state; the objective of Minimizer is the opposite.

Stochastic games constitute a fundamental problem for several reasons. From the theoretical point of view, the complexity of this problem¹ is known to be in $\text{UP} \cap \text{coUP}$ [HK66], but no polynomial-time algorithm is known. Further,

This research was funded in part by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement No. 291763 for TUM – IAS, the Studienstiftung des deutschen Volkes project “Formal methods for analysis of attack-defence diagrams”, the Czech Science Foundation grant No. 18-11193S, TUM IGSSE Grant 10.06 (PARSEC), and the German Research Foundation (DFG) project KR 4890/2-1 “Statistical Unbounded Verification”.

¹ Formally, the problem is to decide, for a given $p \in [0, 1]$ whether Maximizer has a strategy ensuring probability at least p to reach the target.

several other important problems can be reduced to SG, for instance parity games, mean-payoff games, discounted-payoff games and their stochastic extensions [CF11]. The task of solving SG is also polynomial-time equivalent to solving perfect information Shapley, Everett and Gillette games [AM09]. Besides, the problem is practically relevant in verification and synthesis. SG can model reactive systems, with players corresponding to the controller of the system and to its environment, where quantified uncertainty is explicitly modelled. This is useful in many application domains, ranging from smart energy management [CFK+13a] to autonomous urban driving [CKSW13], robot motion planning [LaV00] to self-adaptive systems [CMG14]; for various recent case studies, see e.g. [SK16]. Finally, since Markov decision processes (MDP) [Put14] are a special case with only one player, SG can serve as abstractions of large MDP [KKNP10].

Solution Techniques. There are several classes of algorithms for solving SG, most importantly strategy iteration (SI) algorithms [HK66] and value iteration (VI) algorithms [Con92]. Since the repetitive evaluation of strategies in SI is often slow in practice, VI is usually preferred, similarly to the special case of MDPs [KM17]. For instance, the most used probabilistic model checker PRISM [KNP11] and its branch PRISM-Games [CFK+13a] use VI for MDP and SG as the default option, respectively. However, while SI is in principle a precise method, VI is an approximative method, which converges only in the limit. Unfortunately, there is no known stopping criterion for VI applied to SG. Consequently, there are no guarantees on the results returned in finite time. Therefore, current tools stop when the difference between the two most recent approximations is low, and thus may return arbitrarily imprecise results [HM17].

Value Iteration with Guarantees. In the special case of MDP, in order to obtain bounds on the imprecision of the result, one can employ a *bounded* variant of VI [MLG05, BCC+14] (also called *interval iteration* [HM17]). Here one computes not only an under-approximation, but also an over-approximation of the actual value as follows. On the one hand, iterative computation of the least fixpoint of Bellman equations yields an under-approximating sequence converging to the value. On the other hand, iterative computation of the greatest fixpoint yields an over-approximation, which, however, does not converge to the value. Moreover, it often results in the trivial bound of 1. A solution suggested for MDPs [BCC+14, HM17] is to modify the underlying graph, namely to collapse end components. In the resulting MDP there is only one fixpoint, thus the least and greatest fixpoint coincide and both approximating sequences converge to the actual value. In contrast, for general SG no procedure where the greatest fixpoint converges to the value is known. In this paper we provide one, yielding a stopping criterion. We show that the pre-processing approach of collapsing is not applicable in general and provide a solution on the original graph. We also characterize SG where the fixpoints coincide and no processing is needed. The main technical challenge is that states in an end component in SG can have different values, in contrast to the case of MDP.

Practical Efficiency Using Guarantees. We further utilize the obtained guarantees to practically improve our algorithm. Similar to the MDP case [BCC+14], the quantification of the error allows for ignoring parts of the state space, and thus a speed up without jeopardizing the correctness of the result. Indeed, we provide a technique where some states are not explored and processed at all, but their potential effect is still taken into account. The information is further used to decide the states to be explored next and to be analyzed in more detail. To this end, simulations and learning are used as tools. While for MDP this idea has already demonstrated speed ups in orders of magnitude [BCC+14, ACD+17], this paper provides the first technique of this kind for SG. **Our contribution** is summarized as follows

- We introduce a VI algorithm yielding both under- and over-approximation sequences, both of which converge to the value of the game. Thus we present the first stopping criterion for VI on SG and the first anytime algorithm with guaranteed precision. We also characterize when a simpler solution is sufficient.
- We provide a learning-based algorithm, which preserves the guarantees, but is in some cases more efficient since it avoids exploring the whole state space.
- We evaluate the running times of the algorithms experimentally, concluding that obtaining guarantees requires an overhead that is either negligible or mitigated by the learning-based approach.

Related Work. The works closest to ours are the following. As mentioned above, [BCC+14, HM17] describe the solution to the special case of MDP. While [BCC+14] also provides a learning-based algorithm, [HM17] discusses the convergence rate and the exact solution. The basic algorithm of [HM17] is implemented in PRISM [BKL+17] and the learning approach of [BCC+14] in STORM [DJKV17a]. The extension for SG where the interleaving of players is severely limited (every end component belongs to one player only) is discussed in [Ujm15].

Further, in the area of probabilistic planning, bounded real-time dynamic programming [MLG05] is related to our learning-based approach. However, it is limited to the setting of stopping MDP where the target sink or the non-target sink is reached almost surely under any pair of strategies and thus the fixpoints coincide. Our algorithm works for general SG, not only for stopping ones, without any blowup.

For SG, the tools implementing the standard SI and/or VI algorithms are PRISM-games [CFK+13a], GAVS+ [CKLB11] and GIST [CHJR10]. The latter two are, however, neither maintained nor accessible via the links provided in their publications any more.

Apart from fundamental algorithms to solve SG, there are various practically efficient heuristics that, however, provide none or weak guarantees, often based on some form of learning [BT00, LL08, WT16, TT16, AY17, BBS08]. Finally, the only currently available way to obtain any guarantees through VI is to perform γ^2 iterations and then round to the nearest multiple of $1/\gamma$, yielding the value of the game with precision $1/\gamma$ [CH08]; here γ cannot be freely chosen, but it

is a fixed number, exponential in the number of states and the used probability denominators. However, since the precision cannot be chosen and the number of iterations is always exponential, this approach is infeasible even for small games.

Organization of the Paper. Section 2 introduces the basic notions and revises value iteration. Section 3 explains the idea of our approach on an example. Section 4 provides a full technical treatment of the method as well as the learning-based variation. Section 5 discusses experimental results and Sect. 6 concludes. The appendix (available in [KKKW18]) gives technical details on the pseudocode as well as the conducted experiments and provides more extensive proofs to the theorems and lemmata; in this paper, there are only proof sketches and ideas.

2 Preliminaries

2.1 Basic Definitions

A probability distribution on a finite set X is a mapping $\delta : X \rightarrow [0, 1]$, such that $\sum_{x \in X} \delta(x) = 1$. The set of all probability distributions on X is denoted by $\mathcal{D}(X)$. Now we define stochastic games, in literature often referred as simple stochastic games or stochastic two-player games with a reachability objective.

Definition 1 (SG). A stochastic game (SG) is a tuple $(S, S_{\square}, S_{\circ}, s_0, A, Av, \delta, \mathbf{1}, \mathbf{o})$, where S is a finite set of states partitioned into the sets S_{\square} and S_{\circ} of states of the player Maximizer and Minimizer, respectively, $s_0, \mathbf{1}, \mathbf{o} \in S$ is the initial state, target state, and sink state, respectively, A is a finite set of actions, $Av : S \rightarrow 2^A$ assigns to every state a set of available actions, and $\delta : S \times A \rightarrow \mathcal{D}(S)$ is a transition function that given a state s and an action $a \in Av(s)$ yields a probability distribution over successor states.

A Markov decision process (MDP) is a special case of SG where $S_{\circ} = \emptyset$.

We assume that SGs are non-blocking, so for all states s we have $Av(s) \neq \emptyset$. Further, $\mathbf{1}$ and \mathbf{o} only have one action and it is a self-loop with probability 1. Additionally, we can assume that the SG is preprocessed so that all states with no path to $\mathbf{1}$ are merged with \mathbf{o} .

For a state s and an available action $a \in Av(s)$, we denote the set of successors by $Post(s, a) := \{s' \mid \delta(s, a, s') > 0\}$. Finally, for any set of states $T \subseteq S$, we use T_{\square} and T_{\circ} to denote the states in T that belong to Maximizer and Minimizer, whose states are drawn in the figures as \square and \circ , respectively.

The semantics of SG is given in the usual way by means of strategies and the induced Markov chain and the respective probability space, as follows. An *infinite path* ρ is an infinite sequence $\rho = s_0 a_0 s_1 a_1 \cdots \in (S \times A)^\omega$, such that for every $i \in \mathbb{N}$, $a_i \in Av(s_i)$ and $s_{i+1} \in Post(s_i, a_i)$. *Finite paths* are defined analogously as elements of $(S \times A)^* \times S$. Since this paper deals with the reachability objective, we can restrict our attention to memoryless strategies, which are optimal for this objective. We still allow randomizing strategies, because they are needed for the learning-based algorithm later on. A *strategy* of Maximizer or Minimizer is a function $\sigma : S_{\square} \rightarrow \mathcal{D}(A)$ or $S_{\circ} \rightarrow \mathcal{D}(A)$, respectively, such that $\sigma(s) \in \mathcal{D}(Av(s))$

for all \mathbf{s} . We call a strategy *deterministic* if it maps to Dirac distributions only. Note that there are finitely many deterministic strategies. A pair (σ, τ) of strategies of Maximizer and Minimizer induces a Markov chain $\mathbf{G}^{\sigma, \tau}$ where the transition probabilities are defined as $\delta(\mathbf{s}, \mathbf{s}') = \sum_{\mathbf{a} \in \text{Av}(\mathbf{s})} \sigma(\mathbf{s}, \mathbf{a}) \cdot \delta(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ for states of Maximizer and analogously for states of Minimizer, with σ replaced by τ . The Markov chain induces a unique probability distribution $\mathbb{P}_{\mathbf{s}}^{\sigma, \tau}$ over measurable sets of infinite paths [BK08, Chap. 10].

We write $\Diamond \mathbf{1} := \{\rho \mid \exists i \in \mathbb{N}. \rho(i) = \mathbf{1}\}$ to denote the (measurable) set of all paths which eventually reach $\mathbf{1}$. For each $\mathbf{s} \in S$, we define the *value* in \mathbf{s} as

$$V(\mathbf{s}) := \sup_{\sigma} \inf_{\tau} \mathbb{P}_{\mathbf{s}}^{\sigma, \tau}(\Diamond \mathbf{1}) = \inf_{\tau} \sup_{\sigma} \mathbb{P}_{\mathbf{s}}^{\sigma, \tau}(\Diamond \mathbf{1}),$$

where the equality follows from [Mar75]. We are interested not only in $V(\mathbf{s}_0)$, but also its ε -approximations and the corresponding (ε) -optimal strategies for both players.

Now we recall a fundamental tool for analysis of MDP called end components. We introduce the following notation. Given a set of states $T \subseteq S$, a state $\mathbf{s} \in T$ and an action $\mathbf{a} \in \text{Av}(\mathbf{s})$, we say that (\mathbf{s}, \mathbf{a}) exits T if $\text{Post}(\mathbf{s}, \mathbf{a}) \not\subseteq T$. We define an end component of a SG as the end component of the underlying MDP with both players unified.

Definition 2 (EC). *A non-empty set $T \subseteq S$ of states is an end component (EC) if there is a non-empty set $B \subseteq \bigcup_{\mathbf{s} \in T} \text{Av}(\mathbf{s})$ of actions such that*

1. *for each $\mathbf{s} \in T, \mathbf{a} \in B \cap \text{Av}(\mathbf{s})$ we do not have (\mathbf{s}, \mathbf{a}) exits T ,*
2. *for each $\mathbf{s}, \mathbf{s}' \in T$ there is a finite path $\mathbf{w} = \mathbf{s}\mathbf{a}_0 \dots \mathbf{a}_n \mathbf{s}' \in (T \times B)^* \times T$, i.e. the path stays inside T and only uses actions in B .*

Intuitively, ECs correspond to bottom strongly connected components of the Markov chains induced by possible strategies, so for some pair of strategies all possible paths starting in the EC remain there. An end component T is a *maximal end component (MEC)* if there is no other end component T' such that $T \subseteq T'$. Given an SG \mathbf{G} , the set of its MECs is denoted by $\text{MEC}(\mathbf{G})$ and can be computed in polynomial time [CY95].

2.2 (Bounded) Value Iteration

The value function V satisfies the following system of equations, which is referred to as the *Bellman equations*:

$$V(\mathbf{s}) = \begin{cases} \max_{\mathbf{a} \in \text{Av}(\mathbf{s})} V(\mathbf{s}, \mathbf{a}) & \text{if } \mathbf{s} \in S_{\square} \\ \min_{\mathbf{a} \in \text{Av}(\mathbf{s})} V(\mathbf{s}, \mathbf{a}) & \text{if } \mathbf{s} \in S_{\bigcirc} \\ 1 & \text{if } \mathbf{s} = \mathbf{1} \\ 0 & \text{if } \mathbf{s} = \mathbf{o} \end{cases} \quad (1)$$

where²

$$V(s, a) := \sum_{s' \in S} \delta(s, a, s') \cdot V(s') \quad (2)$$

Moreover, V is the *least* solution to the Bellman equations, see e.g. [CH08]. To compute the value of V for all states in an SG, one can thus utilize the iterative approximation method *value iteration* (VI) as follows. We start with a lower bound function $L_0: S \rightarrow [0, 1]$ such that $L_0(\mathbf{1}) = 1$ and, for all other $s \in S$, $L_0(s) = 0$. Then we repetitively apply Bellman updates (3) and (4)

$$L_n(s, a) := \sum_{s' \in S} \delta(s, a, s') \cdot L_{n-1}(s') \quad (3)$$

$$L_n(s) := \begin{cases} \max_{a \in \text{Av}(s)} L_n(s, a) & \text{if } s \in S_{\square} \\ \min_{a \in \text{Av}(s)} L_n(s, a) & \text{if } s \in S_{\bigcirc} \end{cases} \quad (4)$$

until convergence. Note that convergence may happen only in the limit even for such a simple game as in Fig. 1 on the left. The sequence is monotonic, at all times a *lower* bound on V , i.e. $L_i(s) \leq V(s)$ for all $s \in S$, and the least fixpoint satisfies $L^* := \lim_{n \rightarrow \infty} L_n = V$.

Unfortunately, there is no known stopping criterion, i.e. no guarantees how close the current under-approximation is to the value [HM17]. The current tools stop when the difference between two successive approximations is smaller than a certain threshold, which can lead to arbitrarily wrong results [HM17].

For the special case of MDP, it has been suggested to also compute the greatest fixpoint [MLG05] and thus an *upper* bound as follows. The function $G: S \rightarrow [0, 1]$ is initialized for all states $s \in S$ as $G_0(s) = 1$ except for $G_0(o) = 0$. Then we repetitively apply updates (3) and (4), where L is replaced by G . The resulting sequence G_n is monotonic, provides an upper bound on V and the greatest fixpoint $G^* := \lim_n G_n$ is the greatest solution to the Bellman equations on $[0, 1]^S$.

This approach is called *bounded value iteration* (BVI) (or *bounded real-time dynamic programming* (BRTDP) [MLG05, BCC+14] or *interval iteration* [HM17]). If $L^* = G^*$ then they are both equal to V and we say that *BVI converges*. BVI is guaranteed to converge in MDP if the only ECs are those of $\mathbf{1}$ and o [BCC+14]. Otherwise, if there are non-trivial ECs they have to be “collapsed”³. Computing the greatest fixpoint on the modified MDP results in another sequence U_i of upper bounds on V , converging to $U^* := \lim_n U_n$. Then BVI converges even for general MDPs, $U^* = V$ [BCC+14], when transformed this way. The next section illustrates this difficulty and the solution through collapsing on an example.

² Throughout the paper, for any function $f: S \rightarrow [0, 1]$ we overload the notation and also write $f(s, a)$ meaning $\sum_{s' \in S} \delta(s, a, s') \cdot f(s')$.

³ All states of an EC are merged into one, all leaving actions are preserved and all other actions are discarded. For more detail see [KKKW18, Appendix A.1].

In summary, all versions of BVI discussed so far and later on in the paper follow the pattern of Algorithm 1. In the naive version, UPDATE just performs the Bellman update on L and U according to Eqs. (3) and (4).⁴ For a general MDP, U does not converge to V , but to G^* , and thus the termination criterion may never be met if $G^*(s_0) - V(s_0) > 0$. If the ECs are collapsed in pre-processing then U converges to V .

For the general case of SG, the collapsing approach fails and this paper provides another version of BVI where U converges to V , based on a more detailed structural analysis of the game.

Algorithm 1. Bounded value iteration algorithm

```

1: procedure BVI(precision  $\epsilon > 0$ )
2:   for  $s \in S$  do      \* Initialization * \
3:      $L(s) = 0$          \* Lower bound * \
4:      $U(s) = 1$          \* Upper bound * \
5:    $L(\mathbf{1}) = 1$         \* Value of sinks is determined a priori * \
6:    $U(\mathbf{o}) = 0$ 
7:   repeat
8:     UPDATE( $L, U$ )      \* Bellman updates or their modification * \
9:   until  $U(s_0) - L(s_0) < \epsilon$   \* Guaranteed error bound * \

```

3 Example

In this section, we illustrate the issues preventing BVI convergence and our solution on a few examples. Recall that G is the sequence converging to the greatest solution of the Bellman equations, while U is in general any sequence over-approximating V that one or another BVI algorithm suggests.

Firstly, we illustrate the issue that arises already for the special case of MDP. Consider the MPD of Fig. 1 on the left. Although $V(s) = V(t) = 0.5$, we have $G_i(s) = G_i(t) = 1$ for all i . Indeed, the upper bound for t is always updated as the maximum of $G_i(t, c)$ and $G_i(t, b)$. Although $G_i(t, c)$ decreases over time, $G_i(t, b)$ remains the same, namely equal to $G_i(s)$, which in turn remains equal to $G_i(s, a) = G_i(t)$. This cyclic dependency lets both s and t remain in an “illusion” that the value of the other one is 1.

The solution for MDP is to remove this cyclic dependency by collapsing all MECs into singletons and removing the resulting purely self-looping actions. Figure 1 in the middle shows the MDP after collapsing the EC $\{s, t\}$. This turns the MDP into a stopping one, where $\mathbf{1}$ or \mathbf{o} is under any strategy reached with probability 1. In such MDP, there is a unique solution to the Bellman equations. Therefore, the greatest fixpoint is equal to the least one and thus to V .

⁴ For the straightforward pseudocode, see [KKKW18, Appendix A.2].

Secondly, we illustrate the issues that additionally arise for general SG. It turns out that the collapsing approach can be extended only to games where all states of each EC belong to one player only [Ujm15]. In this case, both Maximizer's and Minimizer's ECs are collapsed the same way as in MDP.

However, when both players are present in an EC, then collapsing may not solve the issue. Consider the SG of Fig. 2. Here α and β represent the values of the respective actions.⁵ There are three cases:

First, let $\alpha < \beta$. If the bounds converge to these values we eventually observe $G_i(q, e) < L_i(r, f)$ and learn the induced inequality. Since \mathbf{p} is a Minimizer's state it will never pick the action leading to the greater value of β . Therefore, we can safely merge \mathbf{p} and \mathbf{q} , and remove the action leading to \mathbf{r} , as shown in the second subfigure.

Second, if $\alpha > \beta$, \mathbf{p} and \mathbf{r} can be merged in an analogous way, as shown in the third subfigure.

Third, if $\alpha = \beta$, both previous solutions as well as collapsing all three states as in the fourth subfigure is possible. However, since the approximants may only converge to α and β in the limit, we may not know in finite time which of these cases applies and thus cannot decide for any of the collapses.

Consequently, the approach of collapsing is not applicable in general. In order to ensure BVI convergence, we suggest a different method, which we call *deflating*. It does not involve changing the state space, but rather decreasing the upper bound U_i to the least value that is currently provable (and thus still correct). To this end, we analyze the exiting actions, i.e. with successors outside of the EC, for the following reason. If the play stays in the EC forever, the target is never reached and Minimizer wins. Therefore, Maximizer needs to pick some exiting action to avoid staying in the EC.

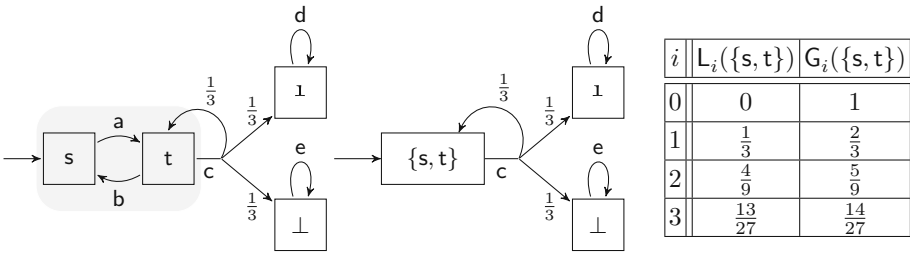


Fig. 1. Left: An MDP (as special case of SG) where BVI does not converge due to the grayed EC. Middle: The same MDP where the EC is collapsed, making BVI converge. Right: The approximations illustrating the convergence of the MDP in the middle.

⁵ Precisely, we consider them to stand for a probabilistic branching with probability α (or β) to 1 and with the remaining probability to 0. To avoid clutter in the figure, we omit this branching and depict only the value.

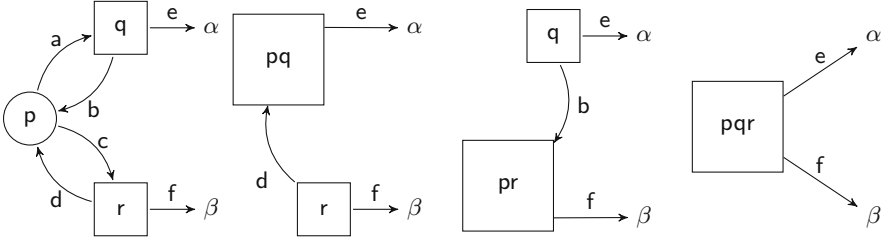


Fig. 2. Left: Collapsing ECs in SG may lead to incorrect results. The Greek letters on the leaving arrows denote the values of the exiting actions. Right three figures: Correct collapsing in different cases, depending on the relationship of α and β . In contrast to MDP, some actions of the EC exiting the collapsed part have to be removed.

For the EC with the states s and t in Fig. 1, the only exiting action is c . In this example, since c is the only exiting action, $U_i(t, c)$ is the highest possible upper bound that the EC can achieve. Thus, by decreasing the upper bound of all states in the EC to that number⁶, we still have a safe upper bound. Moreover, with this modification BVI converges in this example, intuitively because now the upper bound of t depends on action c as it should.

For the example in Fig. 2, it is correct to decrease the upper bound to the maximal exiting one, i.e. $\max\{\hat{\alpha}, \hat{\beta}\}$, where $\hat{\alpha} := U_i(a)$, $\hat{\beta} := U_i(b)$ are the current approximations of α and of β . However, this itself does not ensure BVI convergence. Indeed, if for instance $\hat{\alpha} < \hat{\beta}$ then deflating all states to $\hat{\beta}$ is not tight enough, as values of p and q can even be bounded by $\hat{\alpha}$. In fact, we have to find a certain sub-EC that corresponds to $\hat{\alpha}$, in this case $\{p, q\}$ and set all its upper bounds to $\hat{\alpha}$. We define and compute these sub-ECs in the next section.

In summary, the general structure of our convergent BVI algorithm is to produce the sequence \mathbf{U} by application of Bellman updates and occasionally find the relevant sub-ECs and deflate them. The main technical challenge is that states in an EC in SG can have different values, in contrast to the case of MDP.

4 Convergent Over-Approximation

In Sect. 4.1, we characterize SGs where Bellman equations have more solutions. Based on the analysis, subsequent sections show how to alter the procedure computing the sequence \mathbf{G}_i over-approximating \mathbf{V} so that the resulting tighter sequence \mathbf{U}_i still over-approximates \mathbf{V} , but also converges to \mathbf{V} . This ensures that thus modified BVI converges. Section 4.4 presents the learning-based variant of our BVI.

⁶ We choose the name “deflating” to evoke decreasing the overly high “pressure” in the EC until it equalizes with the actual “pressure” outside.

4.1 Bloated End Components Cause Non-convergence

As we have seen in the example of Fig. 2, BVI generally does not converge due to ECs with a particular structure of the exiting actions. The analysis of ECs relies on the extremal values that can be achieved by exiting actions (in the example, α and β). Given the value function V or just its current over-approximation U_i , we define the most profitable exiting action for Maximizer (denoted by \square) and Minimizer (denoted by \circ) as follows.

Definition 3 (bestExit). *Given a set of states $T \subseteq S$ and a function $f : S \rightarrow [0, 1]$ (see footnote 2), the f -value of the best T -exiting action of Maximizer and Minimizer, respectively, is defined as*

$$\begin{aligned} \text{bestExit}_f^\square(T) &= \max_{\substack{s \in T_\square \\ (s,a) \text{ exits } T}} f(s, a) \\ \text{bestExit}_f^\circ(T) &= \min_{\substack{s \in T_\circ \\ (s,a) \text{ exits } T}} f(s, a) \end{aligned}$$

with the convention that $\max_\emptyset = 0$ and $\min_\emptyset = 1$.

Example 1. In the example of Fig. 2 on the left with $T = \{p, q, r\}$ and $\alpha < \beta$, we have $\text{bestExit}_V^\square(T) = \beta$, $\text{bestExit}_V^\circ(T) = 1$. It is due to $\beta < 1$ that BVI does not converge here. We generalize this in the following lemma. \triangle

Lemma 1. *Let T be an EC. For every m satisfying $\text{bestExit}_V^\square(T) \leq m \leq \text{bestExit}_V^\circ(T)$, there is a solution $f : S \rightarrow [0, 1]$ to the Bellman equations, which on T is constant and equal to m .*

Proof (Idea). Intuitively, such a constant m is a solution to the Bellman equations on T for the following reasons. As both players prefer getting m to exiting and getting “only” the values of their respective **bestExit**, they both choose to stay in the EC (and the extrema in the Bellman equations are realized on non-exiting actions). On the one hand, Maximizer (Bellman equations with max) is hoping for the promised m , which is however not backed up by any actions actually exiting towards the target. On the other hand, Minimizer (Bellman equations with min) does not realize that staying forever results in her optimal value 0 instead of m . \square

Corollary 1. *If $\text{bestExit}_V^\circ(T) > \text{bestExit}_V^\square(T)$ for some EC T , then $G^* \neq V$.*

Proof. Since there are m_1, m_2 such that $\text{bestExit}_V^\square(T) < m_1 < m_2 < \text{bestExit}_V^\circ(T)$, by Lemma 1 there are two different solutions to the Bellman equations. In particular, $G^* > L^* = V$, and BVI does not converge. \square

In accordance with our intuition that ECs satisfying the above inequality should be deflated, we call them bloated.

Definition 4 (BEC). An EC T is called a bloated end component (BEC), if $\text{bestExit}_V^\circ(T) > \text{bestExit}_V^\square(T)$.

Example 2. In the example of Fig. 2 on the left with $\alpha < \beta$, the ECs $\{p, q\}$ and $\{p, q, r\}$ are BECs. \triangle

Example 3. If an EC T has no exiting actions of Minimizer (or no Minimizer's states at all, as in an MDP), then $\text{bestExit}_V^\circ(T) = 1$ (the case with \min_\emptyset). Hence all numbers between $\text{bestExit}_V^\square(T)$ and 1 are a solution to the Bellman equations and $G^*(s) = 1$ for all states $s \in T$.

Analogously, if Maximizer does not have any exiting action in T , then it holds that $\text{bestExit}_V^\square(T) = 0$ (the case with \max_\emptyset), T is a BEC and all numbers between 0 and $\text{bestExit}_V^\circ(T)$ are a solution to the Bellman equations.

Note that in MDP all ECs belong to one player, namely Maximizer. Consequently, all ECs are BECs except for ECs where Maximizer has an exiting action with value 1; all other ECs thus have to be collapsed (or deflated) to ensure BVI convergence in MDPs. Interestingly, all non-trivial ECs in MDPs are a problem, while in SGs through the presence of the other player some ECs can converge, namely if both players want to exit (See e.g. [KKKW18, Appendix A.3]). \triangle

We show that BECs are indeed the only obstacle for BVI convergence.

Theorem 1. If the SG contains no BECs except for $\{o\}$ and $\{1\}$, then $G^* = V$.

Proof (Sketch). Assume, towards a contradiction, that there is some state s with a positive difference $G^*(s) - V(s) > 0$. Consider the set D of states with the maximal difference. D can be shown to be an EC. Since it is not a BEC there has to be an action exiting D and realizing the optimum in that state. Consequently, this action also has the maximal difference, and all its successors, too. Since some of the successors are outside of D , we get a contradiction with the maximality of D . \square

In Sect. 4.2, we show how to eliminate BECs by collapsing their “core” parts, called below MSECs (maximal simple end components). Since MSECs can only be identified with enough information about V , Sect. 4.3 shows how to avoid direct *a priori* collapsing and instead dynamically deflate candidates for MSECs in a conservative way.

4.2 Static MSEC Decomposition

Now we turn our attention to SG with BECs. Intuitively, since in a BEC all Minimizer's exiting actions have a higher value than what Maximizer can achieve, Minimizer does not want to use any of his own exiting actions and prefers staying in the EC (or steering Maximizer towards his worse exiting actions). Consequently, only Maximizer wants to take an exiting action. In the MDP case he can pick any desirable one. Indeed, he can wait until he reaches a state where it is available. As a result, in MDP all states of an EC have the *same value*

and can all be collapsed into one state. In the SG case, he may be restricted by Minimizer's behaviour or even not given any chance to exit the EC at all. As a result, a BEC may contain several parts (below denoted MSECs), each with different value, intuitively corresponding to different exits. Thus instead of MECs, we have to decompose into finer MSECs and only collapse these.

Definition 5 (*Simple EC*). An EC T is called simple (SEC), if for all $s \in T$ we have $V(s) = \text{bestExit}_V^\square(T)$.

A SEC C is maximal (MSEC) if there is no SEC C' such that $C \subsetneq C'$.

Intuitively, an EC is simple, if Minimizer cannot keep Maximizer away from his bestExit . Independently of Minimizer's decisions, Maximizer can reach the bestExit almost surely, unless Minimizer decides to leave, in which case Maximizer could achieve an even higher value.

Example 4. Assume $\alpha < \beta$ in the example of Fig. 2. Then $\{p, q\}$ is a SEC and an MSEC. Further observe that action c is sub-optimal for Minimizer and removing it does not affect the value of any state, but simplifies the graph structure. Namely, it destructs the whole EC into several (here only one) SECs and some non-EC states (here r). \triangle

Algorithm 2, called FIND_MSEC, shows how to compute MSECs. It returns the set of all MSECs if called with parameter V . However, later we also call this function with other parameters $f : S \rightarrow [0, 1]$. The idea of the algorithm is the following. The set X consists of Minimizer's sub-optimal actions, leading to a higher value. As such they cannot be a part of any SEC and thus should be ignored when identifying SECs. (The previous example illustrates that ignoring X is indeed safe as it does not change the value of the game.) We denote the game G where the available actions Av are changed to the new available actions Av' (ignoring the Minimizer's sub-optimal ones) as $G_{[\text{Av}/\text{Av}]}$. Once removed, Minimizer has no choices to affect the value and thus each EC is simple.

Algorithm 2. FIND_MSEC

```

1: function FIND_MSEC( $f : S \rightarrow [0, 1]$ )
2:    $X \leftarrow \{(s, \{a \in \text{Av}(s) \mid f(s, a) > f(s)\}) \mid s \in S_\circ\}$ 
3:    $\text{Av}' \leftarrow \text{Av} \setminus X$            \* Minimizer's  $f$ -suboptimal actions removed * \
4:   return MEC( $G_{[\text{Av}/\text{Av}]}$ )       \* MEC( $G_{[\text{Av}/\text{Av}]}$ ) are MSECs of the original  $G$  * \

```

Lemma 2 (Correctness of Algorithm 2). $T \in \text{FIND_MSEC}(V)$ if and only if T is a MSEC.

Proof (Sketch). “If”: As T is an MSEC, all states in T have the value $\text{bestExit}_V^\square(T)$, and hence also all actions that stay inside T have this value. Thus, no action that stays in T is removed by Line 3 and it is still a MEC in the modified game.

“Only if”: If $T \in \text{FIND.MSEC}(\mathbf{V})$, then T is a MEC of the game where the suboptimal available actions (those in X) of Minimizer have been removed. Hence for all $s \in T : \mathbf{V}(s) = \text{bestExit}_{\mathbf{V}}^{\square}(T)$, because intuitively Minimizer has no possibility to influence the value any further, since all actions that could do so were in X and have been removed. Since T is a MEC in the modified game, it certainly is an EC in the original game. Hence T is a SEC. The inclusion maximality follows from the fact that we compute MECs in the modified game. Thus T is an MSEC. \square

Remark 1 (Algorithm with an oracle). In Sect. 3, we have seen that collapsing MECs does not ensure BVI convergence. Collapsing does not preserve the values, since in BECs we would be collapsing states with different values. Hence we want to collapse only MSECs, where the values are the same. If, moreover, we remove X in such a collapsed SG, then there are no (non-sink) ECs and BVI converges on this SG to the original value.

The difficulty with this algorithm is that it requires an oracle to compare values, for instance a sufficiently precise approximation of \mathbf{V} . Consequently, we cannot pre-compute the MSECs, but have to find them while running BVI. Moreover, since the approximations converge only in the limit we may never be able to conclude on simplicity of some ECs. For instance, if $\alpha = \beta$ in Fig. 2, and if the approximations converge at different speeds, then Algorithm 2 always outputs only a part of the EC, although the whole EC on $\{p, q, r\}$ is simple.

In MDPs, all ECs are simple, because there is no second player to be resolved and all states in an EC have the same value. Thus for MDPs it suffices to collapse all MECs, in contrast to SG.

4.3 Dynamic MSEC Decomposition

Since MSECs cannot be identified from approximants of \mathbf{V} for sure, we refrain from collapsing⁷ and instead only decrease the over-approximation in the corresponding way. We call the method *deflating*, by which we mean decreasing the upper bound of all states in an EC to its $\text{bestExit}_{\mathbf{U}}^{\square}$, see Algorithm 3. The procedure DEFLATE (called on the current upper bound \mathbf{U}_i) decreases this upper bound to the minimum possible value according to the current approximation and thus prevents states from only depending on each other, as in SECs. Intuitively, it gradually approximates SECs and performs the corresponding adjustments, but does not commit to any of the approximations.

Algorithm 3. DEFLATE

```

1: function DEFLATE(EC  $T$ ,  $f : S \rightarrow [0, 1]$ )
2:   for  $s \in T$  do
3:      $f(s) \leftarrow \min(f(s), \text{bestExit}_f^{\square}(T))$     \* Decrease the upper bound * \
4:   return  $f$ 
```

⁷ Our subsequent method can be combined with local collapsing whenever the lower and upper bounds on \mathbf{V} are conclusive.

Lemma 3 (DEFLATE is sound). *For any $f : S \rightarrow [0, 1]$ such that $f \geq V$ and any EC T , $\text{DEFLATE}(T, f) \geq V$.*

This allows us to define our BVI algorithm as the naive BVI with only the additional lines 3–4, see Algorithm 4.

Algorithm 4. UPDATE procedure for bounded value iteration on SG

```

1: procedure UPDATE( $L : S \rightarrow [0, 1]$ ,  $U : S \rightarrow [0, 1]$ )
2:    $L, U$  get updated according to Eq. (3) and (4)    \(* Bellman updates * \
3:   for  $T \in \text{FIND\_MSEC}(L)$  do                      \(* Use lower bound to find ECs * \
4:      $U \leftarrow \text{DEFLATE}(T, U)$                     \(* and deflate the upper bound there * \

```

Theorem 2 (Soundness and completeness). *Algorithm 1 (calling Algorithm 4) produces monotonic sequences L under- and U over-approximating V , and terminates.*

Proof (Sketch). The crux is to show that U converges to V . We assume towards a contradiction, that there exists a state s with $\lim_{n \rightarrow \infty} U_n(s) - V(s) > 0$. Then there exists a nonempty set of states X where the difference between $\lim_{n \rightarrow \infty} U_n$ and V is maximal. If the upper bound of states in X depends on states outside of X , this yields a contradiction, because then the difference between upper bound and value would decrease in the next Bellman update. So X must be an EC where all states depend on each other. However, if that is the case, calling **DEFLATE** decreases the upper bound to something depending on the states outside of X , thus also yielding a contradiction. \square

Summary of Our Approach:

1. We cannot collapse MECs, because we cannot collapse BECs with non-constant values.
2. If we remove X (the sub-optimal actions of Minimizer) we can collapse MECs (now actually MSECs with constant values).
3. Since we know neither X nor SECs we gradually deflate SEC approximations.

4.4 Learning-Based Algorithm

Asynchronous value iteration selects in each round a subset $T \subseteq S$ of states and performs the Bellman update in that round only on T . Consequently, it may speed up computation if “important” states are selected. However, using the standard VI it is even more difficult to determine the current error bound. Moreover, if some states are not selected infinitely often the lower bound may not even converge.

In the setting of bounded value iteration, the current error bound is known for each state and thus convergence can easily be enforced. This gave rise to

asynchronous VI, such as BRTDP (bounded real time dynamic programing) in the setting of stopping MDPs [MLG05], where the states are selected as those that appear on a simulation run. Very similar is the adaptation for general MDP [BCC+14]. In order to simulate a run, the transition probabilities determine how to resolve the probabilistic choice. In order to resolve the non-deterministic choice of Maximizer, the “most promising action” is taken, i.e., with the highest U . This choice is derived from a reinforcement algorithm called delayed Q-learning and ensures convergence while practically performing well [BCC+14].

In this section, we harvest our convergence results and BVI algorithm for SG, which allow us to trivially extend the asynchronous learning-based approach of BRTDP to SGs. On the one hand, the only difference to the MDP algorithm is how to resolve the choice for Minimizer. Since the situation is dual, we again pick the “most promising action”, in this case with the lowest L . On the other hand, the only difference to Algorithm 1 calling Algorithm 4 is that the Bellman updates of U and L are performed on the states of the simulation run only, see lines 2–3 of Algorithm 5.

Algorithm 5. Update procedure for the learning/BRTDP version of BVI on SG

```

1: procedure UPDATE( $L : S \rightarrow [0, 1]$ ,  $U : S \rightarrow [0, 1]$ )
2:    $\rho \leftarrow$  path  $s_0, s_1, \dots, s_\ell$  of length  $\ell \leq k$ , obtained by simulation where the
     successor of  $s$  is  $s'$  with probability  $\delta(s, a, s')$  and  $a$  is sampled randomly from
      $\arg \max_a U(s, a)$  and  $\arg \min_a L(s, a)$  for  $s \in S_\square$  and  $s \in S_\circ$ , respectively
3:    $L, U$  get updated by Eq. (3) and (4) on states  $s_\ell, s_{\ell-1}, \dots, s_0 \quad \backslash^* \text{ all } s \in \rho^* \backslash$ 
4:   for  $T \in \text{FIND\_MSEC}(L)$  do
5:     DEFLATE( $T, U$ )

```

If $\mathbf{1}$ or $\mathbf{0}$ is reached in a simulation, we can terminate it. It can happen that the simulation cycles in an EC. To that end, we have a bound k on the maximum number of steps. The choice of k is discussed in detail in [BCC+14] and we use $2 \cdot |S|$ to guarantee the possibility of reaching sinks as well as exploring new states. If the simulation cycles in an EC, the subsequent call of DEFLATE ensures that next time there is a positive probability to exit this EC. Further details can be found in [KKKW18, Appendix A.4].

5 Experimental Results

We implemented both our algorithms as an extension of PRISM-games [CFK+13a], a branch of PRISM [KNP11] that allows for modelling SGs, utilizing previous work of [BCC+14, Ujm15] for MDP and SG with single-player ECs. We tested the implementation on the SGs from the PRISM-games case studies [gam] that have reachability properties and one additional model from [CKJ12] that was also used in [Ujm15]. We compared the results with both

the explicit and the hybrid engine of PRISM-games, but since the models are small both of them performed similar and we only display the results of the hybrid engine in Table 1.

Furthermore we ran experiments on MDPs from the PRISM benchmark suite [KNP12]. We compared our results there to the hybrid and explicit engine of PRISM, the interval iteration implemented in PRISM [HM17], the hybrid engine of STORM [DJKV17a] and the BRTDP implementation of [BCC+14].

Recall that the aim of the paper is not to provide a faster VI algorithm, but rather the first guaranteed one. Consequently, the aim of the experiments is not to show any speed ups, but to experimentally estimate the overhead needed for computing the guarantees.

For information on the technical details of the experiments, all the models and the tables for the experiments on MDPs we refer to [KKKW18, Appendix B]. Note that although some of the SG models are parametrized they could only be scaled by manually changing the model file, which complicates extensive benchmarking.

Although our approaches compute the additional upper bound to give the convergence guarantees, for each of the experiments one of our algorithms performed similar to PRISM-games. Table 1 shows this result for three of the four SG models in the benchmarking set. On the fourth model, PRISM’s pre-computations already solve the problem and hence it cannot be used to compare the approaches. For completeness, the results are displayed in [KKKW18, Appendix B.5].

Table 1. Experimental results for the experiments on SGs. The left two columns denote the model and the given parameters, if present. Columns 3 to 5 display the verification time in seconds for each of the solvers, namely PRISM-games (referred as PRISM), our BVI algorithm (BVI) and our learning-based algorithm (BRTDP). The next two columns compare the number of states that BRTDP explored (#States_B) to the total number of states in the model. The rightmost column shows the number of MSECs in the model.

| Model | Parameters | PRISM | BVI | BRTDP | #States_B | #States | #MSECs |
|-------|------------|-------|-----|-------|-----------|---------|--------|
| mdsm | prop = 1 | 8 | 8 | 17 | 767 | 62,245 | 1 |
| | prop = 2 | 4 | 4 | 29 | 407 | 62,245 | 1 |
| cdmsn | | 2 | 2 | 3 | 1,212 | 1,240 | 1 |
| cloud | N = 5 | 3 | 7 | 15 | 1,302 | 8,842 | 4,421 |
| | N = 6 | 6 | 59 | 4 | 570 | 34,954 | 17,477 |

Whenever there are few MSECs, as in mdsm and cdmsn, BVI performs like PRISM-games, because only little time is used for deflating. Apparently the additional upper bound computation takes very little time in comparison to the other tasks (e.g. parsing, generating the model, pre-computation) and does not

slow down the verification significantly. For cloud, BVI is slower than PRISM-games, because there are thousands of MSECs and deflating them takes over 80% of the time. This comes from the fact that we need to compute the expensive end component decomposition for each deflating step. BRTDP performs well for cloud, because in this model, as well as generally often if there are many MECs [BCC+14], only a small part of the state space is relevant for convergence. For the other models, BRTDP is slower than the deterministic approaches, because the models are so small that it is faster to first construct them completely than to explore them by simulation.

Our more extensive experiments on MDPs compare the guaranteed approaches based on collapsing (i.e. learning-based from [BCC+14] and deterministic from [HM17]) to our guaranteed approaches based on deflating (so BRTDP and BVI). Since both learning-based approaches as well as both deterministic approaches perform similarly (see Table 2 in [KKKW18, Appendix B]), we conclude that collapsing and deflating are both useful for practical purposes, while the latter is also applicable to SGs. Furthermore we compared the usual unguaranteed value iteration of PRISM’s explicit engine to BVI and saw that our guaranteed approach did not take significantly more time in most cases. This strengthens the point that the overhead for the computation of the guarantees is negligible.

6 Conclusions

We have provided the first stopping criterion for value iteration on simple stochastic games and an anytime algorithm with bounds on the current error (guarantees on the precision of the result). The main technical challenge was that states in end components in SG can have different values, in contrast to the case of MDP. We have shown that collapsing is in general not possible, but we utilized the analysis to obtain the procedure of *deflating*, a solution on the original graph. Besides, whenever a SEC is identified for sure it can be collapsed and the two techniques of collapsing and deflating can thus be combined.

The experiments indicate that the price to pay for the overhead to compute the error bound is often negligible. For each of the available models, at least one of our two implementations has performed similar to or better than the standard approach that yields no guarantees. Further, the obtained guarantees open the door to (e.g. learning-based) heuristics which treat only a part of the state space and can thus potentially lead to huge improvements. Surprisingly, already our straightforward adaptation of such an algorithm for MDP to SG yields interesting results, palliating the overhead of our non-learning method, despite the most naive implementation of deflating. Future work could reveal whether other heuristics or more efficient implementation can lead to huge savings as in the case of MDP [BCC+14].

References

- [ACD+17] Ashok, P., Chatterjee, K., Dacá, P., Křetínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 201–221. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_10
- [AM09] Andersson, D., Miltersen, P.B.: The complexity of solving stochastic games on graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 112–121. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_13
- [AY17] Arslan, G., Yüksel, S.: Decentralized Q-learning for stochastic teams and games. *IEEE Trans. Autom. Control* **62**(4), 1545–1558 (2017)
- [BBS08] Busoniu, L., Babuska, R., De Schutter, B.: A comprehensive survey of multi-agent reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part C* **38**(2), 156–172 (2008)
- [BCC+14] Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
- [BK08] Baier, C., Katoen, J.-P.: *Principles of Model Checking* (2008)
- [BKL+17] Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
- [BT00] Brafman, R.I., Tennenholtz, M.: A near-optimal polynomial time algorithm for learning in certain classes of stochastic games. *Artif. Intell.* **121**(1–2), 31–47 (2000)
- [CF11] Chatterjee, K., Fijalkow, N.: A reduction from parity games to simple stochastic games. In: GandALF, pp. 74–86 (2011)
- [CFK+13a] Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_13
- [CH08] Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_7
- [CHJR10] Chatterjee, K., Henzinger, T.A., Jobstmann, B., Radhakrishna, A.: GIST: a solver for probabilistic games. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 665–669. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_57
- [CKJ12] Calinescu, R., Kikuchi, S., Johnson, K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 303–329. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_16

- [CKLB11] Cheng, C.-H., Knoll, A., Luttenberger, M., Buckl, C.: GAVS+: an open platform for the research of algorithmic game solving. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 258–261. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_22
- [CKSW13] Chen, T., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: Synthesis for multi-objective stochastic games: an application to autonomous urban driving. In: Joshi, K., Siegle, M., Stoelinga, M., D’Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 322–337. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_28
- [CMG14] Cámara, J., Moreno, G.A., Garlan, D.: Stochastic game analysis and latency awareness for proactive self-adaptation. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, 2–3 June 2014, pp. 155–164 (2014)
- [Con92] Condon, A.: The complexity of stochastic games. *Inf. Comput.* **96**(2), 203–224 (1992)
- [CY95] Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. *J. ACM* **42**(4), 857–907 (1995)
- [DJKV17a] Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A STORM is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
- [gam] PRISM-games Case Studies. prismmodelchecker.org/games/casestudies.php. Accessed 18 Sept 2017
- [HK66] Hoffman, A.J., Karp, R.M.: On nonterminating stochastic games. *Manag. Sci.* **12**(5), 359–370 (1966)
- [HM17] Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. *Theor. Comput. Sci.* **735**, 111–131 (2018). <https://doi.org/10.1016/j.tcs.2016.12.003>
- [KKKW18] Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. Technical report abs/1804.04901, [arXiv.org](https://arxiv.org/abs/1804.04901) (2018)
- [KKNP10] Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods Syst. Des.* **36**(3), 246–280 (2010)
- [KM17] Křetínský, J., Meggendorfer, T.: Efficient strategy iteration for mean payoff in Markov decision processes. In: D’Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 380–399. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_25
- [KNP11] 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). https://doi.org/10.1007/978-3-642-22110-1_47
- [KNP12] Kwiatkowska, M., Norman, G., Parker, D.: The prism benchmark suite. In: 9th International Conference on Quantitative Evaluation of Systems (QEST 2012), pp. 203–204. IEEE (2012)
- [LaV00] LaValle, S.M.: Robot motion planning: a game-theoretic foundation. *Algorithmica* **26**(3–4), 430–465 (2000)
- [LL08] Li, J., Liu, W.: A novel heuristic Q-learning algorithm for solving stochastic games. In: IJCNN, pp. 1135–1144 (2008)

- [Mar75] Martin, D.A.: Borel determinacy. *Ann. Math.* **102**, 363–371 (1975)
- [MLG05] McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: *ICML 2005*, pp. 569–576 (2005)
- [Put14] Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, Hoboken (2014)
- [SK16] Svorenová, M., Kwiatkowska, M.: Quantitative verification and strategy synthesis for stochastic games. *Eur. J. Control* **30**, 15–30 (2016)
- [TT16] Tcheukam, A., Tembine, H.: One swarm per queen: a particle swarm learning for stochastic games. In: *SASO*, pp. 144–145 (2016)
- [Ujm15] Ujma, M.: On verification and controller synthesis for probabilistic systems at runtime. Ph.D. thesis, Wolfson College, Oxford (2015)
- [WT16] Wen, M., Topcu, U.: Probably approximately correct learning in stochastic games with temporal logic specifications. In: *IJCAI*, pp. 3630–3636 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Sound Value Iteration

Tim Quatmann^(✉) and Joost-Pieter Katoen

RWTH Aachen University, Aachen, Germany
tim.quatmann@cs.rwth-aachen.de



Abstract. Computing reachability probabilities is at the heart of probabilistic model checking. All model checkers compute these probabilities in an iterative fashion using value iteration. This technique approximates a fixed point from below by determining reachability probabilities for an increasing number of steps. To avoid results that are significantly off, variants have recently been proposed that converge from both below and above. These procedures require starting values for both sides. We present an alternative that does not require the a priori computation of starting vectors and that converges faster on many benchmarks. The crux of our technique is to give tight and safe bounds—whose computation is cheap—on the reachability probabilities. Lifting this technique to expected rewards is trivial for both Markov chains and MDPs. Experimental results on a large set of benchmarks show its scalability and efficiency.

1 Introduction

Markov decision processes (MDPs) [1, 2] have their roots in operations research and stochastic control theory. They are frequently used for stochastic and dynamic optimization problems and are widely applicable in, e.g., stochastic scheduling and robotics. MDPs are also a natural model in randomized distributed computing where coin flips by the individual processes are mixed with non-determinism arising from interleaving the processes’ behaviors. The central problem for MDPs is to find a policy that determines what action to take in the light of what is known about the system at the time of choice. The typical aim is to optimize a given objective, such as minimizing the expected cost until a given number of repairs, maximizing the probability of being operational for 1,000 steps, or minimizing the probability to reach a “bad” state.

Probabilistic model checking [3, 4] provides a scalable alternative to tackle these MDP problems, see the recent surveys [5, 6]. The central computational issue in MDP model checking is to solve a system of linear inequalities. In absence of non-determinism—the MDP being a Markov Chain (MC)—a linear equation system is obtained. After appropriate pre-computations, such as determining the states for which no policy exists that eventually reaches the goal state, the (in)equation system has a unique solution that coincides with the extremal value

This work is partially supported by the Sino-German Center project CAP (GZ 1023).

© The Author(s) 2018

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 643–661, 2018.

https://doi.org/10.1007/978-3-319-96145-3_37

that is sought for. Possible solution techniques to compute such solutions include policy iteration, linear programming, and value iteration. Modern probabilistic model checkers such as **PRISM** [7] and **Storm** [8] use value iteration by default. This approximates a fixed point from below by determining the probabilities to reach a target state within k steps in the k -th iteration. The iteration is typically stopped if the difference between the value vectors of two successive (or vectors that are further apart) is below the desired accuracy ε .

This procedure however can provide results that are significantly off, as the iteration is stopped prematurely, e.g., since the probability mass in the MDP only changes slightly in a series of computational steps due to a “slow” movement. This problem is not new; similar problems, e.g., occur in iterative approaches to compute long-run averages [9] and transient measures [10] and pop up in statistical model checking to decide when to stop simulating for unbounded reachability properties [11]. As recently was shown, this phenomenon does not only occur for hypothetical cases but affects practical benchmarks of MDP model checking too [12]. To remedy this, Haddad and Monmege [13] proposed to iteratively approximate the (unique) fixed point from both below and above; a natural termination criterion is to halt the computation once the two approximations differ less than $2 \cdot \varepsilon$. This scheme requires two starting vectors, one for each approximation. For reachability probabilities, the conservative values zero and one can be used. For expected rewards, it is non-trivial to find an appropriate upper bound—how to “guess” an adequate upper bound to the expected reward to reach a goal state? Baier *et al.* [12] recently provided an algorithm to solve this issue.

This paper takes an alternative perspective to obtaining a sound variant of value iteration. *Our approach does not require the a priori computation of starting vectors and converges faster on many benchmarks.* The crux of our technique is to give tight and safe bounds—whose computation is cheap and that are obtained during the course of value iteration—on the reachability probabilities. The approach is simple and can be lifted straightforwardly to expected rewards. The central idea is to split the desired probability for reaching a target state into the sum of

- (i) the probability for reaching a target state *within* k steps and
- (ii) the probability for reaching a target state *only after* k steps.

We obtain (i) via k iterations of (standard) value iteration. A second instance of value iteration computes the probability that a target state is still reachable after k steps. We show that from this information safe lower and upper bounds for (ii) can be derived. We illustrate that the same idea can be applied to expected rewards, topological value iteration [14], and Gauss-Seidel value iteration. We also discuss in detail its extension to MDPs and provide extensive experimental evaluation using our implementation in the model checker **Storm** [8]. Our experiments show that on many practical benchmarks we need significantly fewer iterations, yielding a speed-up of about 20% on average. More importantly though, is the conceptual simplicity of our approach.

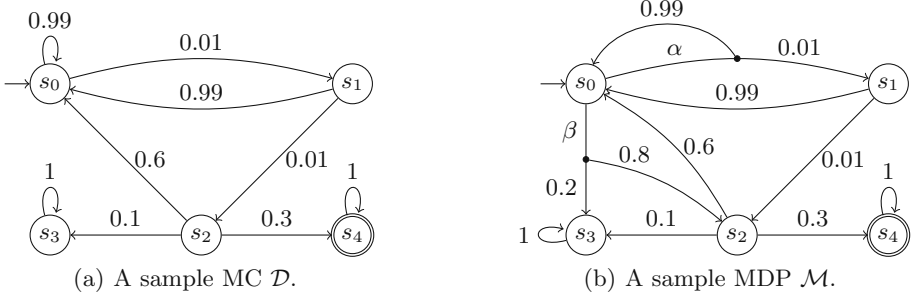


Fig. 1. Example models.

2 Preliminaries

For a finite set S and vector $x \in \mathbb{R}^{|S|}$, let $x[s] \in \mathbb{R}$ denote the entry of x that corresponds to $s \in S$. Let $S' \subseteq S$ and $a \in \mathbb{R}$. We write $x[S'] = a$ to denote that $x[s] = a$ for all $s \in S'$. Given $x, y \in \mathbb{R}^{|S|}$, $x \leq y$ holds iff $x[s] \leq y[s]$ holds for all $s \in S$. For a function $f: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ and $k \geq 0$ we write f^k for the function obtained by applying f k times, i.e., $f^0(x) = x$ and $f^k(x) = f(f^{k-1}(x))$ if $k > 0$.

2.1 Probabilistic Models and Measures

We briefly present probabilistic models and their properties. More details can be found in, e.g., [15].

Definition 1 (Probabilistic Models). A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (S, \text{Act}, \mathbf{P}, s_I, \rho)$, where

- S is a finite set of states, Act is a finite set of actions, s_I is the initial state,
- $\mathbf{P}: S \times \text{Act} \times S \rightarrow [0, 1]$ is a transition probability function satisfying $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\}$ for all $s \in S, \alpha \in \text{Act}$, and
- $\rho: S \times \text{Act} \rightarrow \mathbb{R}$ is a reward function.

\mathcal{M} is a Markov Chain (MC) if $|\text{Act}| = 1$.

Example 1. Figure 1 shows an example MC and an example MDP.

We often simplify notations for MCs by omitting the (unique) action. For an MDP $\mathcal{M} = (S, \text{Act}, \mathbf{P}, s_I, \rho)$, the set of *enabled actions* of state $s \in S$ is given by $\text{Act}(s) = \{\alpha \in \text{Act} \mid \sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1\}$. We assume that $\text{Act}(s) \neq \emptyset$ for each $s \in S$. Intuitively, upon performing action α at state s reward $\rho(s, \alpha)$ is collected and with probability $\mathbf{P}(s, \alpha, s')$ we move to $s' \in S$. Notice that rewards can be positive or negative.

A state $s \in S$ is called *absorbing* if $\mathbf{P}(s, \alpha, s) = 1$ for every $\alpha \in \text{Act}(s)$. A *path* of \mathcal{M} is an infinite alternating sequence $\pi = s_0 \alpha_0 s_1 \alpha_1 \dots$ where $s_i \in S, \alpha_i \in$

$Act(s_i)$, and $\mathbf{P}(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \geq 0$. The set of paths of \mathcal{M} is denoted by $Paths^{\mathcal{M}}$. The set of paths that start at $s \in S$ is given by $Paths^{\mathcal{M},s}$. A *finite path* $\hat{\pi} = s_0\alpha_0 \dots \alpha_{n-1}s_n$ is a finite prefix of a path ending with $last(\hat{\pi}) = s_n \in S$. $|\hat{\pi}| = n$ is the length of $\hat{\pi}$, $Paths_{fin}^{\mathcal{M}}$ is the set of finite paths of \mathcal{M} , and $Paths_{fin}^{\mathcal{M},s}$ is the set of finite paths that start at state $s \in S$. We consider LTL-like notations for sets of paths. For $k \in \mathbb{N} \cup \{\infty\}$ and $G, H \subseteq S$ let

$$HU^{\leq k} G = \{s_0\alpha_0s_1 \dots \in Paths^{\mathcal{M},s_I} \mid s_0, \dots, s_{j-1} \in H, s_j \in G \text{ for some } j \leq k\}$$

denote the set of paths that, starting from the initial state s_I , only visit states in H until after at most k steps a state in G is reached. Sets $HU^{>k} G$ and $HU^{=k} G$ are defined similarly. We use the shorthands $\Diamond^{\leq k} G := SU^{\leq k} G$, $\Diamond G := \Diamond^{\leq \infty} G$, and $\Box^{\leq k} G := Paths^{\mathcal{M},s_I} \setminus \Diamond^{\leq k}(S \setminus G)$.

A (*deterministic*) *scheduler* for \mathcal{M} is a function $\sigma: Paths_{fin}^{\mathcal{M}} \rightarrow Act$ such that $\sigma(\hat{\pi}) \in Act(last(\hat{\pi}))$ for all $\hat{\pi} \in Paths_{fin}^{\mathcal{M}}$. The set of (deterministic) schedulers for \mathcal{M} is $\mathfrak{S}^{\mathcal{M}}$. $\sigma \in \mathfrak{S}^{\mathcal{M}}$ is called *positional* if $\sigma(\hat{\pi})$ only depends on the last state of $\hat{\pi}$, i.e., for all $\hat{\pi}, \hat{\pi}' \in Paths_{fin}^{\mathcal{M}}$ we have $last(\hat{\pi}) = last(\hat{\pi}')$ implies $\sigma(\hat{\pi}) = \sigma(\hat{\pi}')$. For MDP \mathcal{M} and scheduler $\sigma \in \mathfrak{S}^{\mathcal{M}}$ the *probability measure* over finite paths is given by $\Pr_{fin}^{\mathcal{M},\sigma}: Paths_{fin}^{\mathcal{M},s_I} \rightarrow [0,1]$ with $\Pr_{fin}^{\mathcal{M},\sigma}(s_0 \dots s_n) = \prod_{i=0}^{n-1} \mathbf{P}(s_i, \sigma(s_0 \dots s_i), s_{i+1})$. The probability measure $\Pr^{\mathcal{M},\sigma}$ over measurable sets of infinite paths is obtained via a standard cylinder set construction [15].

Definition 2 (Reachability Probability). *The reachability probability of MDP $\mathcal{M} = (S, Act, \mathbf{P}, s_I, \rho)$, $G \subseteq S$, and $\sigma \in \mathfrak{S}^{\mathcal{M}}$ is given by $\Pr^{\mathcal{M},\sigma}(\Diamond G)$.*

For $k \in \mathbb{N} \cup \{\infty\}$, the function $\Diamond^{\leq k} G: \Diamond G \rightarrow \mathbb{R}$ yields the k -bounded reachability reward of a path $\pi = s_0\alpha_0s_1 \dots \in \Diamond G$. We set $\Diamond^{\leq k} G(\pi) = \sum_{i=0}^{j-1} \rho(s_i, \alpha_i)$, where $j = \min(\{i \geq 0 \mid s_i \in G\} \cup \{k\})$. We write $\Diamond G$ instead of $\Diamond^{\leq \infty} G$.

Definition 3 (Expected Reward). *The expected (reachability) reward of MDP $\mathcal{M} = (S, Act, \mathbf{P}, s_I, \rho)$, $G \subseteq S$, and $\sigma \in \mathfrak{S}^{\mathcal{M}}$ with $\Pr^{\mathcal{M},\sigma}(\Diamond G) = 1$ is given by the expectation $\mathbb{E}^{\mathcal{M},\sigma}(\Diamond G) = \int_{\pi \in \Diamond G} \Diamond G(\pi) d\Pr^{\mathcal{M},\sigma}(\pi)$.*

We write $\Pr_s^{\mathcal{M},\sigma}$ and $\mathbb{E}_s^{\mathcal{M},\sigma}$ for the probability measure and expectation obtained by changing the initial state of \mathcal{M} to $s \in S$. If \mathcal{M} is a Markov chain, there is only a single scheduler. In this case we may omit the superscript σ from $\Pr^{\mathcal{M},\sigma}$ and $\mathbb{E}^{\mathcal{M},\sigma}$. We also omit the superscript \mathcal{M} if it is clear from the context. The maximal reachability probability of \mathcal{M} and G is given by $\Pr^{\max}(\Diamond G) = \max_{\sigma \in \mathfrak{S}^{\mathcal{M}}} \Pr^{\sigma}(\Diamond G)$. There is a positional scheduler that attains this maximum [16]. The same holds for minimal reachability probabilities and maximal or minimal expected rewards.

Example 2. Consider the MDP \mathcal{M} from Fig. 1(b). We are interested in the maximal probability to reach state s_4 given by $\Pr^{\max}(\Diamond\{s_4\})$. Since s_4 is not reachable from s_3 we have $\Pr_{s_3}^{\max}(\Diamond\{s_4\}) = 0$. Intuitively, choosing action β at state s_0 makes reaching s_3 more likely, which should be avoided in order

to maximize the probability to reach s_4 . We therefore assume a scheduler σ that always chooses action α at state s_0 . Starting from the initial state s_0 , we then eventually take the transition from s_2 to s_3 or the transition from s_2 to s_4 with probability one. The resulting probability to reach s_4 is given by $\Pr^{\max}(\Diamond\{s_4\}) = \Pr^{\sigma}(\Diamond\{s_4\}) = 0.3/(0.1 + 0.3) = 0.75$.

2.2 Probabilistic Model Checking via Interval Iteration

In the following we present approaches to compute reachability probabilities and expected rewards. We consider approximative computations. Exact computations are handled in e.g. [17, 18]. For the sake of clarity, we focus on reachability probabilities and sketch how the techniques can be lifted to expected rewards.

Reachability Probabilities. We fix an MDP $\mathcal{M} = (S, Act, \mathbf{P}, s_I, \rho)$, a set of goal states $G \subseteq S$, and a precision parameter $\varepsilon > 0$.

Problem 1. Compute an ε -approximation of the maximal reachability probability $\Pr^{\max}(\Diamond G)$, i.e., compute a value $r \in [0, 1]$ with $|r - \Pr^{\max}(\Diamond G)| < \varepsilon$.

We briefly sketch how to compute such a value r via *interval iteration* [12, 13, 19]. The computation for minimal reachability probabilities is analogous.

W.l.o.g. it is assumed that the states in G are absorbing. Using graph algorithms, we compute $S_0 = \{s \in S \mid \Pr_s^{\max}(\Diamond G) = 0\}$ and partition the state space of \mathcal{M} into $S = S_0 \cup G \cup S_?$ with $S_? = S \setminus (G \cup S_0)$. If $s_I \in S_0$ or $s_I \in G$, the probability $\Pr^{\max}(\Diamond G)$ is 0 or 1, respectively. From now on we assume $s_I \in S_?$.

We say that \mathcal{M} is *contracting* with respect to $S' \subseteq S$ if $\Pr_s^{\sigma}(\Diamond S') = 1$ for all $s \in S$ and for all $\sigma \in \mathfrak{S}^{\mathcal{M}}$. We assume that \mathcal{M} is contracting with respect to $G \cup S_0$. Otherwise, we apply a transformation on the so-called *end components*¹ of \mathcal{M} , yielding a contracting MDP \mathcal{M}' with the same maximal reachability probability as \mathcal{M} . Roughly, this transformation replaces each end component of \mathcal{M} with a single state whose enabled actions coincide with the actions that previously lead outside of the end component. This step is detailed in [13, 19].

We have $x^*[s] = \Pr_s^{\max}(\Diamond G)$ for $s \in S$ and the unique fixpoint x^* of the function $f: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ with $f(x)[S_0] = 0$, $f(x)[G] = 1$, and

$$f(x)[s] = \max_{\alpha \in Act(s)} \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot x[s']$$

for $s \in S_?$. Hence, computing $\Pr^{\max}(\Diamond G)$ reduces to finding the fixpoint of f .

A popular technique for this purpose is the *value iteration* algorithm [1]. Given a starting vector $x \in \mathbb{R}^{|S|}$ with $x[S_0] = 0$ and $x[G] = 1$, standard value iteration computes $f^k(x)$ for increasing k until $\max_{s \in S} |f^k(x)[s] - f^{k-1}(x)[s]| < \varepsilon$ holds for a predefined precision $\varepsilon > 0$. As pointed out in, e.g., [13], there is no

¹ Intuitively, an end component is a set of states $S' \subseteq S$ such that there is a scheduler inducing that from any $s \in S'$ exactly the states in S' are visited infinitely often.

guarantee on the preciseness of the result $r = f^k(x)[s_I]$, i.e., standard value iteration does not give any evidence on the error $|r - \text{Pr}^{\max}(\Diamond G)|$. The intuitive reason is that value iteration only approximates the fixpoint x^* from one side, yielding no indication on the distance between the current result and x^* .

Example 3. Consider the MDP \mathcal{M} from Fig. 1(b). We invoked standard value iteration in **PRISM** [7] and **Storm** [8] to compute the reachability probability $\text{Pr}^{\max}(\Diamond\{s_4\})$. Recall from Example 2 that the correct solution is 0.75. With (absolute) precision $\varepsilon = 10^{-6}$ both model checkers returned 0.7248. Notice that the user can improve the precision by considering, e.g., $\varepsilon = 10^{-8}$ which yields 0.7497. However, there is no guarantee on the preciseness of a given result.

The *interval iteration* algorithm [12, 13, 19] addresses the impreciseness of value iteration. The idea is to approach the fixpoint x^* from below and from above. The first step is to find starting vectors $x_\ell, x_u \in \mathbb{R}^{|S|}$ satisfying $x_\ell[S_0] = x_u[S_0] = 0$, $x_\ell[G] = x_u[G] = 1$, and $x_\ell \leq x^* \leq x_u$. As the entries of x^* are probabilities, it is always valid to set $x_\ell[S_?] = 0$ and $x_u[S_?] = 1$. We have $f^k(x_\ell) \leq x^* \leq f^k(x_u)$ for any $k \geq 0$. Interval iteration computes $f^k(x_\ell)$ and $f^k(x_u)$ for increasing k until $\max_{s \in S} |f^k(x_\ell)[s] - f^k(x_u)[s]| < 2\varepsilon$. For the result $r = 1/2 \cdot (f^k(x_\ell)[s_I] + f^k(x_u)[s_I])$ we obtain that $|r - \text{Pr}^{\max}(\Diamond G)| < \varepsilon$, i.e., we get a sound approximation of the maximal reachability probability.

Example 4. We invoked interval iteration in **PRISM** and **Storm** to compute the reachability probability $\text{Pr}^{\max}(\Diamond\{s_4\})$ for the MDP \mathcal{M} from Fig. 1(b). Both implementations correctly yield an ε -approximation of $\text{Pr}^{\max}(\Diamond\{s_4\})$, where we considered $\varepsilon = 10^{-6}$. However, both **PRISM** and **Storm** required roughly 300,000 iterations for convergence.

Expected Rewards. Whereas [13, 19] only consider reachability probabilities, [12] extends interval iteration to compute expected rewards. Let \mathcal{M} be an MDP and G be a set of absorbing states such that \mathcal{M} is contracting with respect to G .

Problem 2. Compute an ε -approximation of the maximal expected reachability reward $\mathbb{E}^{\max}(\Diamond G)$, i.e., compute a value $r \in \mathbb{R}$ with $|r - \mathbb{E}^{\max}(\Diamond G)| < \varepsilon$.

We have $x^*[s] = \mathbb{E}_s^{\max}(\Diamond G)$ for the unique fixpoint x^* of $g: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ with

$$g(x)[G] = 0 \quad \text{and} \quad g(x)[s] = \max_{\alpha \in \text{Act}(s)} \rho(s, \alpha) + \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot x[s']$$

for $s \notin G$. As for reachability probabilities, interval iteration can be applied to approximate this fixpoint. The crux lies in finding appropriate starting vectors $x_\ell, x_u \in \mathbb{R}^{|S|}$ guaranteeing $x_\ell \leq x^* \leq x_u$. To this end, [12] describe graph based algorithms that give an upper bound on the expected number of times each individual state $s \in S \setminus G$ is visited. This then yields an approximation of the expected amount of reward collected at the various states.

3 Sound Value Iteration for MCs

We present an algorithm for computing reachability probabilities and expected rewards as in Problems 1 and 2. The algorithm is an alternative to the interval iteration approach [12, 20] but (i) does not require an a priori computation of starting vectors $x_\ell, x_u \in \mathbb{R}^{|S|}$ and (ii) converges faster on many practical benchmarks as shown in Sect. 5. For the sake of simplicity, we first restrict to computing reachability probabilities on MCs.

In the following, let $\mathcal{D} = (S, \mathbf{P}, s_I, \rho)$ be an MC, $G \subseteq S$ be a set of absorbing goal states and $\varepsilon > 0$ be a precision parameter. We consider the partition $S = S_0 \cup G \cup S_?$ as in Sect. 2.2. The following theorem captures the key insight of our algorithm.

Theorem 1. *For MC \mathcal{D} let G and $S_?$ be as above and $k \geq 0$ with $\Pr_s(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$. We have*

$$\begin{aligned} & \Pr(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot \min_{s \in S_?} \frac{\Pr_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?)} \\ & \leq \Pr(\Diamond G) \leq \Pr(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot \max_{s \in S_?} \frac{\Pr_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?)}. \end{aligned}$$

Theorem 1 allows us to approximate $\Pr(\Diamond G)$ by computing for increasing $k \in \mathbb{N}$

- $\Pr(\Diamond^{\leq k} G)$, the probability to reach a state in G within k steps, and
- $\Pr(\Box^{\leq k} S_?)$, the probability to stay in $S_?$ during the first k steps.

This can be realized via a value-iteration based procedure. The obtained bounds on $\Pr(\Diamond G)$ can be tightened arbitrarily since $\Pr(\Box^{\leq k} S_?)$ approaches 0 for increasing k . In the following, we address the correctness of Theorem 1, describe the details of our algorithm, and indicate how the results can be lifted to expected rewards.

3.1 Approximating Reachability Probabilities

To approximate the reachability probability $\Pr(\Diamond G)$, we consider the step bounded reachability probability $\Pr(\Diamond^{\leq k} G)$ for $k \geq 0$ and provide a lower and an upper bound for the ‘missing’ probability $\Pr(\Diamond G) - \Pr(\Diamond^{\leq k} G)$. Note that $\Diamond G$ is the disjoint union of the paths that reach G *within* k steps (given by $\Diamond^{\leq k} G$) and the paths that reach G only *after* k steps (given by $S_? \mathcal{U}^{>k} G$).

Lemma 1. *For any $k \geq 0$ we have $\Pr(\Diamond G) = \Pr(\Diamond^{\leq k} G) + \Pr(S_? \mathcal{U}^{>k} G)$.*

A path $\pi \in S_? \mathcal{U}^{>k} G$ reaches some state $s \in S_?$ after *exactly* k steps. This yields the partition $S_? \mathcal{U}^{>k} G = \bigcup_{s \in S_?} (S_? \mathcal{U}^{=k} \{s\} \cap \Diamond G)$. It follows that

$$\Pr(S_? \mathcal{U}^{>k} G) = \sum_{s \in S_?} \Pr(S_? \mathcal{U}^{=k} \{s\}) \cdot \Pr_s(\Diamond G).$$

Consider $\ell, u \in [0, 1]$ with $\ell \leq \Pr_s(\Diamond G) \leq u$ for all $s \in S_?$, i.e., ℓ and u are lower and upper bounds for the reachability probabilities within $S_?$. We have

$$\sum_{s \in S_?} \Pr(S_? \mathcal{U}^k \{s\}) \cdot \Pr_s(\Diamond G) \leq \sum_{s \in S_?} \Pr(S_? \mathcal{U}^k \{s\}) \cdot u = \Pr(\Box^{\leq k} S_?) \cdot u.$$

We can argue similar for the lower bound ℓ . With *Lemma 1* we get the following.

Proposition 1. *For MC \mathcal{D} with G , $S_?$, ℓ , u as above and any $k \geq 0$ we have*

$$\Pr(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot \ell \leq \Pr(\Diamond G) \leq \Pr(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot u.$$

Remark 1. The bounds for $\Pr(\Diamond G)$ given by Proposition 1 are similar to the bounds obtained after performing k iterations of interval iteration with starting vectors $x_\ell, x_u \in \mathbb{R}^{|S|}$, where $x_\ell[S_?] = \ell$ and $x_u[S_?] = u$.

We now discuss how the bounds ℓ and u can be obtained from the step bounded probabilities $\Pr_s(\Diamond^{\leq k} G)$ and $\Pr_s(\Box^{\leq k} S_?)$ for $s \in S_?$. We focus on the upper bound u . The reasoning for the lower bound ℓ is similar.

Let $s_{\max} \in S_?$ be a state with maximal reachability probability, that is $s_{\max} \in \arg \max_{s \in S_?} \Pr_s(\Diamond G)$. From Proposition 1 we get

$$\Pr_{s_{\max}}(\Diamond G) \leq \Pr_{s_{\max}}(\Diamond^{\leq k} G) + \Pr_{s_{\max}}(\Box^{\leq k} S_?) \cdot \Pr_{s_{\max}}(\Diamond G).$$

We solve the inequality for $\Pr_{s_{\max}}(\Diamond G)$ (assuming $\Pr_s(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$):

$$\Pr_{s_{\max}}(\Diamond G) \leq \frac{\Pr_{s_{\max}}(\Diamond^{\leq k} G)}{1 - \Pr_{s_{\max}}(\Box^{\leq k} S_?) \cdot \Pr_{s_{\max}}(\Diamond G)} \leq \max_{s \in S_?} \frac{\Pr_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?) \cdot \Pr_s(\Diamond G)}.$$

Proposition 2. *For MC \mathcal{D} let G and $S_?$ be as above and $k \geq 0$ such that $\Pr_s(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$. For every $\hat{s} \in S_?$ we have*

$$\min_{s \in S_?} \frac{\Pr_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?) \cdot \Pr_s(\Diamond G)} \leq \Pr_{\hat{s}}(\Diamond G) \leq \max_{s \in S_?} \frac{\Pr_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?) \cdot \Pr_s(\Diamond G)}.$$

Theorem 1 is a direct consequence of Propositions 1 and 2.

3.2 Extending the Value Iteration Approach

Recall the standard value iteration algorithm for approximating $\Pr(\Diamond G)$ as discussed in Sect. 2.2. The function $f: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ for MCs simplifies to $f(x)[S_0] = 0$, $f(x)[G] = 1$, and $f(x)[s] = \sum_{s' \in S} \mathbf{P}(s, s') \cdot x[s']$ for $s \in S_?$. We can compute the k -step bounded reachability probability at every state $s \in S$

Input : MC $\mathcal{D} = (S, \mathbf{P}, s_I, \rho)$, absorbing states $G \subseteq S$, precision $\varepsilon > 0$
Output : $r \in \mathbb{R}$ with $|r - \Pr(\diamond G)| < \varepsilon$

- 1 $S_? \leftarrow S \setminus (\{s \in S \mid \Pr_s(\diamond G) = 0\} \cup G)$
- 2 initialize $x_0, y_0 \in \mathbb{R}^{|S|}$ with $x_0[G] = 1, x_0[S \setminus G] = 0, y_0[S_?] = 1, y_0[S \setminus S_?] = 0$
- 3 $\ell_0 \leftarrow -\infty; u_0 \leftarrow +\infty$
- 4 $k \leftarrow 0$
- 5 **repeat**
- 6 $k \leftarrow k + 1$
- 7 $x_k \leftarrow f(x_{k-1}); y_k \leftarrow h(y_{k-1})$
- 8 **if** $y_k[s] < 1$ for all $s \in S_?$ **then**
- 9 $\ell_k \leftarrow \max(\ell_{k-1}, \min_{s \in S_?} \frac{x_k[s]}{1 - y_k[s]}); u_k \leftarrow \min(u_{k-1}, \max_{s \in S_?} \frac{x_k[s]}{1 - y_k[s]})$
- 10 **until** $y_k[S_I] \cdot (u_k - \ell_k) < 2 \cdot \varepsilon$
- 11 **return** $x_k[S_I] + y_k[S_I] \cdot \frac{\ell_k + u_k}{2}$

Algorithm 1: Sound value iteration for MCs.

by performing k iterations of value iteration [15, Remark 10.104]. More precisely, when applying f k times on starting vector $x \in \mathbb{R}^{|S|}$ with $x[G] = 1$ and $x[S \setminus G] = 0$ we get $\Pr_s(\diamond^{\leq k} G) = f^k(x)[s]$. The probabilities $\Pr_s(\square^{\leq k} S_?)$ for $s \in S$ can be computed similarly. Let $h: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ with $h(y)[S \setminus S_?] = 0$ and $h(y)[s] = \sum_{s' \in S} \mathbf{P}(s, s') \cdot y[s']$ for $s \in S_?$. For starting vector $y \in \mathbb{R}^{|S|}$ with $y[S_?] = 1$ and $y[S \setminus S_?] = 0$ we get $\Pr_s(\square^{\leq k} S_?) = h^k(y)[s]$.

Algorithm 1 depicts our approach. It maintains vectors $x_k, y_k \in \mathbb{R}^{|S|}$ which, after k iterations of the loop, store the k -step bounded probabilities $\Pr_s(\diamond^{\leq k} G)$ and $\Pr_s(\square^{\leq k} S_?)$, respectively. Additionally, the algorithm considers lower bounds ℓ_k and upper bounds u_k such that the following invariant holds.

Lemma 2. *After executing the loop of Algorithm 1 k times we have for all $s \in S_?$ that $x_k[s] = \Pr_s(\diamond^{\leq k} G)$, $y_k[s] = \Pr_s(\square^{\leq k} S_?)$, and $\ell_k \leq \Pr_s(\diamond G) \leq u_k$.*

The correctness of the algorithm follows from Theorem 1. Termination is guaranteed since $\Pr(\diamond(S_0 \cup G)) = 1$ and therefore $\lim_{k \rightarrow \infty} \Pr(\square^{\leq k} S_?) = \Pr(\square S_?) = 0$.

Theorem 2. *Algorithm 1 terminates for any MC \mathcal{D} , goal states G , and precision $\varepsilon > 0$. The returned value r satisfies $|r - \Pr(\diamond G)| < \varepsilon$.*

Example 5. We apply Algorithm 1 for the MC in Fig. 1(a) and the set of goal states $G = \{s_4\}$. We have $S_? = \{s_0, s_1, s_2\}$. After $k = 3$ iterations it holds that

$$\begin{array}{lll} x_3[s_0] = 0.00003 & x_3[s_1] = 0.003 & x_3[s_2] = 0.3 \\ y_3[s_0] = 0.99996 & y_3[s_1] = 0.996 & y_3[s_2] = 0.6 \end{array}$$

Hence, $\frac{x_3[s]}{1 - y_3[s]} = \frac{3}{4} = 0.75$ for all $s \in S_?$. We get $\ell_3 = u_3 = 0.75$. The algorithm converges for any $\varepsilon > 0$ and returns the correct solution $x_3[s_0] + y_3[s_0] \cdot 0.75 = 0.75$.

3.3 Sound Value Iteration for Expected Rewards

We lift our approach to expected rewards in a straightforward manner. Let $G \subseteq S$ be a set of absorbing goal states of MC \mathcal{D} such that $\Pr(\Diamond G) = 1$. Further let $S_? = S \setminus G$. For $k \geq 0$ we observe that the expected reward $\mathbb{E}(\Diamond G)$ can be split into the expected reward collected within k steps and the expected reward collected only after k steps, i.e., $\mathbb{E}(\Diamond G) = \mathbb{E}(\Diamond^{\leq k} G) + \sum_{s \in S_?} \Pr(S_? \mathcal{U}^k \{s\}) \cdot \mathbb{E}_s(\Diamond G)$. Following a similar reasoning as in Sect. 3.1 we can show the following.

Theorem 3. *For MC \mathcal{D} let G and $S_?$ be as before and $k \geq 0$ such that $\Pr_s(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$. We have*

$$\begin{aligned} & \mathbb{E}(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot \min_{s \in S_?} \frac{\mathbb{E}_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?)} \\ & \leq \mathbb{E}(\Diamond G) \leq \mathbb{E}(\Diamond^{\leq k} G) + \Pr(\Box^{\leq k} S_?) \cdot \max_{s \in S_?} \frac{\mathbb{E}_s(\Diamond^{\leq k} G)}{1 - \Pr_s(\Box^{\leq k} S_?)}. \end{aligned}$$

Recall the function $g: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ from Sect. 2.2, given by $g(x)[G] = 0$ and $g(x)[s] = \rho(s) + \sum_{s' \in S} \mathbf{P}(s, s') \cdot x[s']$ for $s \in S_?$. For $s \in S$ and $x \in \mathbb{R}^{|S|}$ with $x[S] = 0$ we have $\mathbb{E}_s(\Diamond^{\leq k} G) = g^k(x)[s]$. We modify Algorithm 1 such that it considers function g instead of function f . Then, the returned value r satisfies $|r - \mathbb{E}(\Diamond G)| < \varepsilon$.

3.4 Optimizations

Algorithm 1 can make use of *initial bounds* $\ell_0, u_0 \in \mathbb{R}$ with $\ell_0 \leq \Pr_s(\Diamond G) \leq u_0$ for all $s \in S_?$. Such bounds could be derived, e.g., from domain knowledge or during preprocessing [12]. The algorithm always chooses the largest available lower bound for ℓ_k and the lowest available upper bound for u_k , respectively. If Algorithm 1 and interval iteration are initialized with the same bounds, Algorithm 1 always requires as most as many iterations compared to interval iteration (cf. Remark 1).

Gauss-Seidel value iteration [1, 12] is an optimization for standard value iteration and interval iteration that potentially leads to faster convergence. When computing $f(x)[s]$ for $s \in S_?$, the idea is to consider already computed results $f(x)[s']$ from the current iteration. Formally, let $\prec \subseteq S \times S$ be some strict total ordering of the states. Gauss-Seidel value iteration considers instead of function f the function $f_{\prec}: \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$ with $f_{\prec}[S_0] = 0$, $f_{\prec}[G] = 1$, and

$$f_{\prec}(x)[s] = \sum_{s' \prec s} \mathbf{P}(s, s') \cdot f_{\prec}(x)[s'] + \sum_{s' \not\prec s} \mathbf{P}(s, s') \cdot x[s'].$$

Values $f_{\prec}(x)[s]$ for $s \in S$ are computed in the order defined by \prec . This idea can also be applied to our approach. To this end, we replace f by f_{\prec} and h by h_{\prec} , where h_{\prec} is defined similarly. More details are given in [21].

Topological value iteration [14] employs the graphical structure of the MC \mathcal{D} . The idea is to decompose the states S of \mathcal{D} into strongly connected components²

² $S' \subseteq S$ is a connected component if s can be reached from s' for all $s, s' \in S'$. S' is a strongly connected component if no superset of S' is a connected component.

(SCCs) that are analyzed individually. The procedure can improve the runtime of classical value iteration since for a single iteration only the values for the current SCC have to be updated. A topological variant of interval iteration is introduced in [12]. Given these results, sound value iteration can be extended similarly.

4 Sound Value Iteration for MDPs

We extend sound value iteration to compute reachability probabilities in MDPs. Assume an MDP $\mathcal{M} = (S, Act, \mathbf{P}, s_I, \rho)$ and a set of absorbing goal states G . For simplicity, we focus on maximal reachability probabilities, i.e., we compute $\text{Pr}^{\max}(\Diamond G)$. Minimal reachability probabilities and expected rewards are analogous. As in Sect. 2.2 we consider the partition $S = S_0 \cup G \cup S_?$ such that \mathcal{M} is contracting with respect to $G \cup S_0$.

4.1 Approximating Maximal Reachability Probabilities

We argue that our results for MCs also hold for MDPs under a given scheduler $\sigma \in \mathfrak{S}^{\mathcal{M}}$. Let $k \geq 0$ such that $\text{Pr}_s^\sigma(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$. Following the reasoning as in Sect. 3.1 we get

$$\text{Pr}^\sigma(\Diamond^{\leq k} G) + \text{Pr}^\sigma(\Box^{\leq k} S_?) \cdot \min_{s \in S_?} \frac{\text{Pr}_s^\sigma(\Diamond^{\leq k} G)}{1 - \text{Pr}_s^\sigma(\Box^{\leq k} S_?)} \leq \text{Pr}^\sigma(\Diamond G) \leq \text{Pr}^{\max}(\Diamond G).$$

Next, assume an upper bound $u \in \mathbb{R}$ with $\text{Pr}_s^{\max}(\Diamond G) \leq u$ for all $s \in S_?$. For a scheduler $\sigma_{\max} \in \mathfrak{S}^{\mathcal{M}}$ that attains the maximal reachability probability, i.e., $\sigma_{\max} \in \arg \max_{\sigma \in \mathfrak{S}^{\mathcal{M}}} \text{Pr}^\sigma(\Diamond G)$ it holds that

$$\begin{aligned} \text{Pr}^{\max}(\Diamond G) &= \text{Pr}^{\sigma_{\max}}(\Diamond G) \leq \text{Pr}^{\sigma_{\max}}(\Diamond^{\leq k} G) + \text{Pr}^{\sigma_{\max}}(\Box^{\leq k} S_?) \cdot u \\ &\leq \max_{\sigma \in \mathfrak{S}^{\mathcal{M}}} (\text{Pr}^\sigma(\Diamond^{\leq k} G) + \text{Pr}^\sigma(\Box^{\leq k} S_?) \cdot u). \end{aligned}$$

We obtain the following theorem which is the basis of our algorithm.

Theorem 4. *For MDP \mathcal{M} let G , $S_?$, and u be as above. Assume $\sigma \in \mathfrak{S}^{\mathcal{M}}$ and $k \geq 0$ such that $\sigma \in \arg \max_{\sigma' \in \mathfrak{S}^{\mathcal{M}}} \text{Pr}^{\sigma'}(\Diamond^{\leq k} G) + \text{Pr}^{\sigma'}(\Box^{\leq k} S_?) \cdot u$ and $\text{Pr}_s^\sigma(\Box^{\leq k} S_?) < 1$ for all $s \in S_?$. We have*

$$\begin{aligned} &\text{Pr}^\sigma(\Diamond^{\leq k} G) + \text{Pr}^\sigma(\Box^{\leq k} S_?) \cdot \min_{s \in S_?} \frac{\text{Pr}_s^\sigma(\Diamond^{\leq k} G)}{1 - \text{Pr}_s^\sigma(\Box^{\leq k} S_?)} \\ &\leq \text{Pr}^{\max}(\Diamond G) \leq \text{Pr}^\sigma(\Diamond^{\leq k} G) + \text{Pr}^\sigma(\Box^{\leq k} S_?) \cdot u. \end{aligned}$$

Similar to the results for MCs it also holds that $\text{Pr}^{\max}(\Diamond G) \leq \max_{\sigma \in \mathfrak{S}^{\mathcal{M}}} \hat{u}_k^\sigma$ with

$$\hat{u}_k^\sigma := \text{Pr}^\sigma(\Diamond^{\leq k} G) + \text{Pr}^\sigma(\Box^{\leq k} S_?) \cdot \max_{s \in S_?} \frac{\text{Pr}_s^\sigma(\Diamond^{\leq k} G)}{1 - \text{Pr}_s^\sigma(\Box^{\leq k} S_?)}.$$

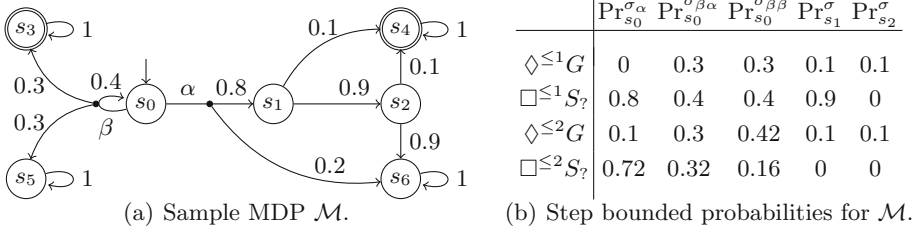


Fig. 2. Example MDP with corresponding step bounded probabilities.

However, this upper bound can not trivially be embedded in a value iteration based procedure. Intuitively, in order to compute the upper bound for iteration k , one can not necessarily build on the results for iteration $k - 1$.

Example 6. Consider the MDP \mathcal{M} given in Fig. 2(a). Let $G = \{s_3, s_4\}$ be the set of goal states. We therefore have $S_? = \{s_0, s_1, s_2\}$. In Fig. 2(b) we list step bounded probabilities with respect to the possible schedulers, where σ_α , $\sigma_{\beta\alpha}$, and $\sigma_{\beta\beta}$ refer to schedulers with $\sigma_\alpha(s_0) = \alpha$ and for $\gamma \in \{\alpha, \beta\}$, $\sigma_{\beta\gamma}(s_0) = \beta$ and $\sigma_{\beta\gamma}(s_0\beta s_0) = \gamma$. Notice that the probability measures $\Pr_{s_1}^{\sigma}$ and $\Pr_{s_2}^{\sigma}$ are independent of the considered scheduler σ . For step bounds $k \in \{1, 2\}$ we get

- $\max_{\sigma \in \mathfrak{S}(\mathcal{M})} \hat{u}_1^\sigma = \hat{u}_1^{\sigma_\alpha} = 0 + 0.8 \cdot \max(0, 1, 0) = 0.8$ and
- $\max_{\sigma \in \mathfrak{S}(\mathcal{M})} \hat{u}_2^\sigma = \hat{u}_2^{\sigma_{\beta\beta}} = 0.42 + 0.16 \cdot \max(0.5, 0.19, 1) = 0.5$.

4.2 Extending the Value Iteration Approach

The idea of our algorithm is to compute the bounds for $\Pr_s^{\max}(\Diamond^k G)$ as in Theorem 4 for increasing $k \geq 0$. Algorithm 2 outlines the procedure. Similar to Algorithm 1 for MCs, vectors $x_k, y_k \in \mathbb{R}^{|S|}$ store the step bounded probabilities $\Pr_s^{\sigma_k}(\Diamond^{\leq k} G)$ and $\Pr_s^{\sigma_k}(\Box^{\leq k} S_?)$ for any $s \in S$. In addition, schedulers σ_k and upper bounds $u_k \geq \max_{s \in S_?} \Pr_s^{\max}(\Diamond^k G)$ are computed in a way that Theorem 4 is applicable.

Lemma 3. *After executing k iterations of Algorithm 2 we have for all $s \in S_?$ that $x_k[s] = \Pr_s^{\sigma_k}(\Diamond^{\leq k} G)$, $y_k[s] = \Pr_s^{\sigma_k}(\Box^{\leq k} S_?)$, and $\ell_k \leq \Pr_s^{\max}(\Diamond^k G) \leq u_k$, where $\sigma_k \in \arg \max_{\sigma \in \mathfrak{S}(\mathcal{M})} \Pr_s^{\sigma}(\Diamond^{\leq k} G) + \Pr_s^{\sigma}(\Box^{\leq k} S_?) \cdot u_k$.*

The lemma holds for $k = 0$ as x_0, y_0 , and u_0 are initialized accordingly. For $k > 0$ we assume that the claim holds after $k - 1$ iterations, i.e., for $x_{k-1}, y_{k-1}, u_{k-1}$ and scheduler σ_{k-1} . The results of the k th iteration are obtained as follows.

The function *findAction* illustrated in Algorithm 3 determines the choices of a scheduler $\sigma_k \in \arg \max_{\sigma \in \mathfrak{S}(\mathcal{M})} \Pr_s^{\sigma}(\Diamond^{\leq k} G) + \Pr_s^{\sigma}(\Box^{\leq k} S_?) \cdot u_{k-1}$ for $s \in S_?$. The idea is to consider at state s an action $\sigma_k(s) = \alpha \in \text{Act}(s)$ that maximizes

$$\Pr_s^{\sigma_k}(\Diamond^{\leq k} G) + \Pr_s^{\sigma_k}(\Box^{\leq k} S_?) \cdot u_{k-1} = \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot (x_{k-1}[s'] + y_{k-1}[s'] \cdot u_{k-1}).$$

Input : MDP $\mathcal{M} = (S, Act, \mathbf{P}, s_I, \rho)$, absorbing states $G \subseteq S$, precision $\varepsilon > 0$
Output : $r \in \mathbb{R}$ with $|r - \Pr^{\max}(\Diamond G)| < \varepsilon$

- 1 $S_0 \leftarrow \{s \in S \mid \Pr_s^{\max}(\Diamond G) = 0\}$
- 2 assert that \mathcal{M} is contracting with respect to $G \cup S_0$
- 3 $S_? \leftarrow S \setminus (S_0 \cup G)$
- 4 initialize $x_0, y_0 \in \mathbb{R}^{|S|}$ with $x_0[G] = 1, x_0[S \setminus G] = 0, y_0[S_?] = 1, y_0[S \setminus S_?] = 0$
- 5 $\ell_0 \leftarrow -\infty; u_0 \leftarrow +\infty; d_0 \leftarrow -\infty$
- 6 $k \leftarrow 0$
- 7 **repeat**
- 8 $k \leftarrow k + 1$
- 9 initialize $x_k, y_k \in \mathbb{R}^{|S|}$ with $x_k[G] = 1, x_k[S_0] = 0, y_k[S \setminus S_?] = 0$
- 10 $d_k \leftarrow d_{k-1}$
- 11 **foreach** $s \in S_?$ **do**
- 12 $\alpha \leftarrow \text{findAction}(x_{k-1}, y_{k-1}, s, u_{k-1})$
- 13 $d_k \leftarrow \max(d_k, \text{decisionValue}(x_{k-1}, y_{k-1}, s, \alpha))$
- 14 $x_k[s] \leftarrow \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot x_{k-1}[s']$
- 15 $y_k[s] \leftarrow \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot y_{k-1}[s']$
- 16 **if** $y_k[s] < 1$ for all $s \in S_?$ **then**
- 17 $\ell_k \leftarrow \max(\ell_{k-1}, \min_{s \in S_?} \frac{x_k[s]}{1 - y_k[s]})$
- 18 $u_k \leftarrow \min(u_{k-1}, \max(d_k, \max_{s \in S_?} \frac{x_k[s]}{1 - y_k[s]}))$
- 19 **until** $y_k[s_I] \cdot (u_k - \ell_k) < 2 \cdot \varepsilon$
- 20 **return** $x_k[s_I] + y_k[s_I] \cdot \frac{\ell_k + u_k}{2}$

Algorithm 2: Sound value iteration for MDPs

For the case where no real upper bound is known (i.e., $u_{k-1} = \infty$) we implicitly assume a sufficiently large value for u_{k-1} such that $\Pr_s^\sigma(\Diamond^{\leq k} G)$ becomes negligible. Upon leaving state s , σ_k mimics σ_{k-1} , i.e., we set $\sigma_k(s\alpha s_1\alpha_1 \dots s_n) = \sigma_{k-1}(s_1\alpha_1 \dots s_n)$. After executing Line 15 of Algorithm 2 we have $x_k[s] = \Pr_s^{\sigma_k}(\Diamond^{\leq k} G)$ and $y_k[s] = \Pr_s^{\sigma_k}(\Box^{\leq k} S_?)$.

It remains to derive an upper bound u_k . To ensure that Lemma 3 holds we require (i) $u_k \geq \max_{s \in S_?} \Pr_s^{\max}(\Diamond G)$ and (ii) $u_k \in U_k$, where

$$U_k = \{u \in \mathbb{R} \mid \sigma_k \in \arg \max_{\sigma \in \mathfrak{S}^{\mathcal{M}}} \Pr_s^\sigma(\Diamond^{\leq k} G) + \Pr_s^\sigma(\Box^{\leq k} S_?) \cdot u \text{ for all } s \in S_?\}.$$

Intuitively, the set $U_k \subseteq \mathbb{R}$ consists of all possible upper bounds u for which σ_k is still optimal. $U_k \subseteq \mathbb{R}$ is convex as it can be represented as a conjunction of inequalities with $U_0 = \mathbb{R}$ and $u \in U_k$ if and only if $u \in U_{k-1}$ and for all $s \in S_?$ with $\sigma_k(s) = \alpha$ and for all $\beta \in Act(s) \setminus \{\alpha\}$

$$\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot (x_{k-1}[s'] + y_{k-1}[s'] \cdot u) \geq \sum_{s' \in S} \mathbf{P}(s, \beta, s') \cdot (x_{k-1}[s'] + y_{k-1}[s'] \cdot u).$$

The algorithm maintains the so-called *decision value* d_k which corresponds to the minimum of U_k (or $-\infty$ if the minimum does not exist). Algorithm 4 outlines the

```

1 function findAction( $x, y, s, u$ )
2   if  $u \neq \infty$  then
3     return  $\alpha \in \arg \max_{\alpha \in \text{Act}(s)} \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot (x[s'] + y[s'] \cdot u)$ 
4   else
5     return  $\alpha \in \arg \max_{\alpha \in \text{Act}(s)} \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot (y[s'])$ 

```

Algorithm 3: Computation of optimal action.

```

1 function decisionValue( $x, y, s, \alpha$ )
2    $d \leftarrow -\infty$ 
3   foreach  $\beta \in \text{Act}(s) \setminus \{\alpha\}$  do
4      $y_\Delta \leftarrow \sum_{s' \in S} (\mathbf{P}(s, \alpha, s') - \mathbf{P}(s, \beta, s')) \cdot y[s']$ 
5     if  $y_\Delta > 0$  then
6        $x_\Delta \leftarrow \sum_{s' \in S} (\mathbf{P}(s, \beta, s') - \mathbf{P}(s, \alpha, s')) \cdot x[s']$ 
7        $d \leftarrow \max(d, x_\Delta / y_\Delta)$ 
8   return  $d$ 

```

Algorithm 4: Computation of decision value.

procedure to obtain the decision value at a given state. Our algorithm ensures that u_k is only set to a value in $[d_k, u_{k-1}] \subseteq U_k$.

Lemma 4. *After executing Line 18 of Algorithm 2: $u_k \geq \max_{s \in S_\gamma} \text{Pr}_s^{\max}(\Diamond G)$.*

To show that u_k is a valid upper bound, let $s_{\max} \in \arg \max_{s \in S_\gamma} \text{Pr}_s^{\max}(\Diamond G)$ and $u^* = \text{Pr}_{s_{\max}}^{\max}(\Diamond G)$. From Theorem 4, $u_{k-1} \geq u^*$, and $u_{k-1} \in U_k$ we get

$$\begin{aligned}
 u^* &\leq \max_{\sigma \in \mathfrak{SM}} \text{Pr}_{s_{\max}}^\sigma(\Diamond^{\leq k} G) + \text{Pr}_{s_{\max}}^\sigma(\Box^{\leq k} S_\gamma) \cdot u_{k-1} \\
 &= \text{Pr}_{s_{\max}}^{\sigma_k}(\Diamond^{\leq k} G) + \text{Pr}_{s_{\max}}^{\sigma_k}(\Box^{\leq k} S_\gamma) \cdot u_{k-1} = x_k[s_{\max}] + y_k[s_{\max}] \cdot u_{k-1}
 \end{aligned}$$

which yields a new upper bound $x_k[s_{\max}] + y_k[s_{\max}] \cdot u_{k-1} \geq u^*$. We repeat this scheme as follows. Let $v_0 := u_{k-1}$ and for $i > 0$ let $v_i := x_k[s_{\max}] + y_k[s_{\max}] \cdot v_{i-1}$. We can show that $v_{i-1} \in U_k$ implies $v_i \geq u^*$. Assuming $y_k[s_{\max}] < 1$, the sequence v_0, v_1, v_2, \dots converges to $v_\infty := \lim_{i \rightarrow \infty} v_i = \frac{x_k[s_{\max}]}{1 - y_k[s_{\max}]}$. We distinguish three cases to show that $u_k = \min(u_{k-1}, \max(d_k, \max_{s \in S_\gamma} \frac{x_k[s]}{1 - y_k[s]})) \geq u^*$.

- If $v_\infty > u_{k-1}$, then also $\max_{s \in S_\gamma} \frac{x_k[s]}{1 - y_k[s]} > u_{k-1}$. Hence $u_k = u_{k-1} \geq u^*$.
- If $d_k \leq v_\infty \leq u_{k-1}$, we can show that $v_i \leq v_{i-1}$. It follows that for all $i > 0$, $v_{i-1} \in U_k$, implying $v_i \geq u^*$. Thus we get $u_k = \max_{s \in S_\gamma} \frac{x_k[s]}{1 - y_k[s]} \geq v_\infty \geq u^*$.
- If $v_\infty < d_k$ then there is an $i \geq 0$ with $v_i \geq d_k$ and $u^* \leq v_{i+1} < d_k$. It follows that $u_k = d_k \geq u^*$.

Example 7. Reconsider the MDP \mathcal{M} from Fig. 2(a) and goal states $G = \{s_3, s_4\}$. The maximal reachability probability is attained for a scheduler that always chooses β at state s_0 , which results in $\Pr^{\max}(\Diamond G) = 0.5$. We now illustrate how Algorithm 2 approximates this value by sketching the first two iterations. For the first iteration *findAction* yields action α at s_0 . We obtain:

$$x_1[s_0] = 0, x_1[s_1] = 0.1, x_1[s_2] = 0.1, y_1[s_0] = 0.8, y_1[s_1] = 0.9, y_1[s_2] = 0, \\ d_1 = 0.3/(0.8 - 0.4) = 0.75, \ell_1 = \min(0, 1, 0) = 0, u_1 = \max(0.75, 0, 1, 0) = 1.$$

In the second iteration *findAction* yields again α for s_0 and we get:

$$x_2[s_0] = 0.08, x_2[s_1] = 0.19, x_2[s_2] = 0.1, y_2[s_0] = 0.72, y_2[s_1] = 0, y_2[s_2] = 0, \\ d_2 = 0.75, \ell_2 = \min(0.29, 0.19, 0.1) = 0.1, u_2 = \max(0.75, 0.29, 0.19, 0.1) = 0.75.$$

Due to the decision value we do not set the upper bound u_2 to $0.29 < \Pr^{\max}(\Diamond G)$.

Theorem 5. *Algorithm 2 terminates for any MDP \mathcal{M} , goal states G and precision $\varepsilon > 0$. The returned value r satisfies $|r - \Pr^{\max}(\Diamond G)| \leq \varepsilon$.*

The correctness of the algorithm follows from Theorem 4 and Lemma 3. Termination follows since \mathcal{M} is contracting with respect to $S_0 \cup G$, implying $\lim_{k \rightarrow \infty} \Pr^\sigma(\Box^{\leq k} S_?) = 0$. The optimizations for Algorithm 1 mentioned in Sect. 3.4 can be applied to Algorithm 2 as well.

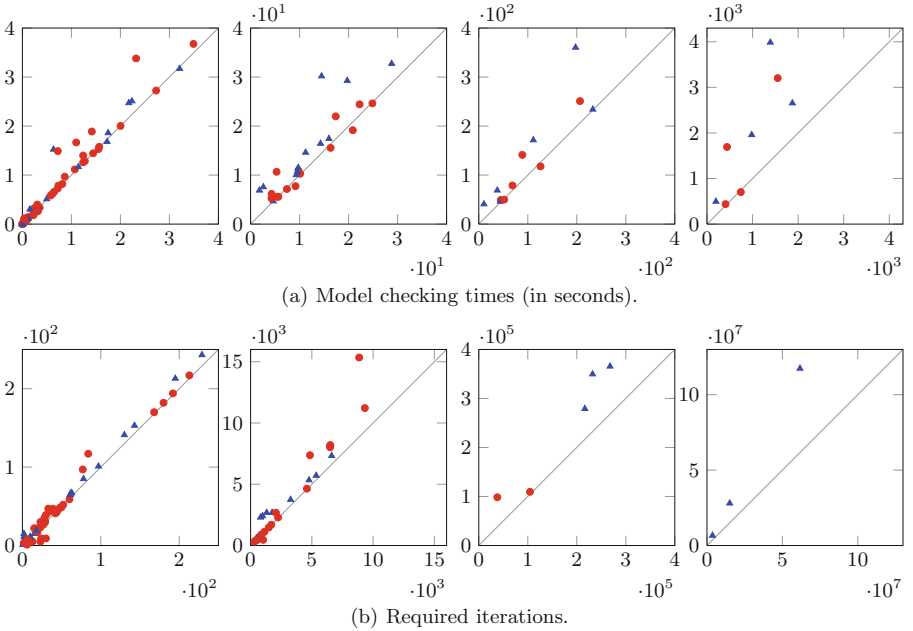


Fig. 3. Comparison of sound value iteration (x-axis) and interval iteration (y-axis).

5 Experimental Evaluation

Implementation. We implemented sound value iteration for MCs and MDPs into the model checker **Storm** [8]. The implementation computes reachability probabilities and expected rewards using explicit data structures such as sparse matrices and vectors. Moreover, Multi-objective model checking is supported, where we straightforwardly extend the value iteration-based approach of [22] to sound value iteration. We also implemented the optimizations given in Sect. 3.4.

The implementation is available at www.stormchecker.org.

Experimental Results. We considered a wide range of case studies including

- all MCs, MDPs, and CTMCs from the PRISM benchmark suite [23],
- several case studies from the PRISM website www.prismmodelchecker.org,
- Markov automata accompanying IMCA [24], and
- multi-objective MDPs considered in [22].

In total, 130 model and property instances were considered. For CTMCs and Markov automata we computed (untimed) reachability probabilities or expected rewards on the underlying MC and the underlying MDP, respectively. In all experiments the precision parameter was given by $\varepsilon = 10^{-6}$.

We compare sound value iteration (SVI) with interval iteration (II) as presented in [12, 13]. We consider the Gauss-Seidel variant of the approaches and compute initial bounds ℓ_0 and u_0 as in [12]. For a better comparison we consider the implementation of II in **Storm**. [21] gives a comparison with the implementation of II in PRISM. The experiments were run on a single core (2GHz) of an HP BL685C G7 with 192GB of available memory. However, almost all experiments required less than 4GB. We measured model checking times and required iterations. All logfiles and considered benchmarks are available at [25].

Figure 3(a) depicts the model checking times for SVI (x-axis) and II (y-axis). For better readability, the benchmarks are divided into four plots with different scales. Triangles (\blacktriangle) and circles (\bullet) indicate MC and MDP benchmarks, respectively. Similarly, Fig. 3(b) shows the required iterations of the approaches. We observe that SVI converged faster and required fewer iterations for almost all MCs and MDPs. SVI performed particularly well on the challenging instances where many iterations are required. Similar observations were made when comparing the topological variants of SVI and II. Both approaches were still competitive if no a priori bounds are given to SVI. More details are given in [21].

Figure 4 indicates the model checking times of SVI and II as well as their topological variants. For reference, we also consider standard (unsound) value iteration (VI). The x -axis depicts the number of instances that have been solved by the corresponding approach within the time limit indicated on the y -axis. Hence, a point (x, y) means that for x instances the model checking time was less or equal than y . We observe that the topological variant of SVI yielded the best run times among all sound approaches and even competes with (unsound) VI.

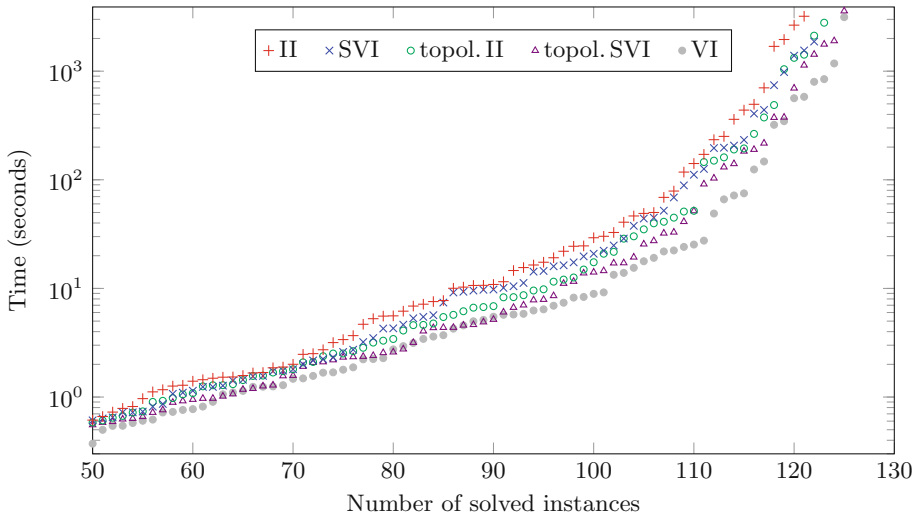


Fig. 4. Runtime comparison between different approaches.

6 Conclusion

In this paper we presented a sound variant of the value iteration algorithm which safely approximates reachability probabilities and expected rewards in MCs and MDPs. Experiments on a large set of benchmarks indicate that our approach is a reasonable alternative to the recently proposed interval iteration algorithm.

References

1. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (1994)
2. Feinberg, E.A., Shwartz, A.: Handbook of Markov Decision Processes: Methods and Applications. Kluwer, Dordrecht (2002)
3. Katoen, J.P.: The probabilistic model checking landscape. In: LICS, pp. 31–45. ACM (2016)
4. Baier, C.: Probabilistic model checking. In: Dependable Software Systems Engineering. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 45, pp. 1–23. IOS Press (2016)
5. Etessami, K.: Analysis of probabilistic processes and automata theory. In: Handbook of Automata Theory. European Mathematical Society (2016, to appear)
6. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Probabilistic model checking. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer, Cham (2018)
7. 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). https://doi.org/10.1007/978-3-642-22110-1_47

8. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A STORM is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
9. Katoen, J., Zapreev, I.S.: Safe on-the-fly steady-state detection for time-bounded reachability. In: QEST, pp. 301–310. IEEE Computer Society (2006)
10. Malhotra, M.: A computationally efficient technique for transient analysis of repairable markovian systems. *Perform. Eval.* **24**(4), 311–331 (1996)
11. Daca, P., Henzinger, T.A., Kretínský, J., Petrov, T.: Faster statistical model checking for unbounded temporal properties. *ACM Trans. Comput. Log.* **18**(2), 12:1–12:25 (2017)
12. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for markov decision processes. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
13. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. *Theor. Comput. Sci.* **735**, 111–131 (2017)
14. Dai, P., Weld, D.S., Goldsmith, J.: Topological value iteration algorithms. *J. Artif. Intell. Res.* **42**, 181–209 (2011)
15. Baier, C., Katoen, J.P.: *Principles of Model Checking*. The MIT Press, Cambridge (2008)
16. Bertsekas, D.P., Tsitsiklis, J.N.: An analysis of stochastic shortest path problems. *Math. Oper. Res.* **16**(3), 580–595 (1991)
17. Giro, S.: Efficient computation of exact solutions for quantitative model checking. In: QAPL. EPTCS, vol. 85, pp. 17–32 (2012)
18. Bauer, M.S., Mathur, U., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. In: FMCAD, pp. 92–99. IEEE (2017)
19. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
20. Haddad, S., Monmege, B.: Reachability in MDPs: refining convergence of value iteration. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP 2014. LNCS, vol. 8762, pp. 125–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11439-2_10
21. Quatmann, T., Katoen, J.P.: Sound value iteration. Technical report, CoRR abs/1804.05001 (2018)
22. Forejt, V., Kwiatkowska, M., Parker, D.: Pareto curves for probabilistic model checking. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 317–332. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_25
23. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: *Proceedings of QEST*, pp. 203–204. IEEE CS (2012)
24. Guck, D., Timmer, M., Hatefi, H., Ruijters, E., Stoelinga, M.: Modelling and analysis of markov reward automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 168–184. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_13
25. Quatmann, T., Katoen, J.P.: Experimental Results for Sound Value Iteration. figshare (2018). <https://doi.org/10.6084/m9.figshare.6139052>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Safety-Aware Apprenticeship Learning

Weichao Zhou and Wenchao Li(✉)

Department of Electrical and Computer Engineering, Boston University, Boston, USA
{zwc662,wenchao}@bu.edu

Abstract. Apprenticeship learning (AL) is a kind of Learning from Demonstration techniques where the reward function of a Markov Decision Process (MDP) is unknown to the learning agent and the agent has to derive a good policy by observing an expert's demonstrations. In this paper, we study the problem of how to make AL algorithms inherently safe while still meeting its learning objective. We consider a setting where the unknown reward function is assumed to be a linear combination of a set of state features, and the safety property is specified in Probabilistic Computation Tree Logic (PCTL). By embedding probabilistic model checking inside AL, we propose a novel *counterexample-guided* approach that can ensure safety while retaining performance of the learnt policy. We demonstrate the effectiveness of our approach on several challenging AL scenarios where safety is essential.

1 Introduction

The rapid progress of artificial intelligence (AI) comes with a growing concern over its safety when deployed in real-life systems and situations. As highlighted in [3], if the objective function of an AI agent is wrongly specified, then maximizing that objective function may lead to harmful results. In addition, the objective function or the training data may focus only on accomplishing a specific task and ignore other aspects, such as safety constraints, of the environment. In this paper, we propose a novel framework that combines explicit safety specification with learning from data. We consider safety specification expressed in Probabilistic Computation Tree Logic (PCTL) and show how probabilistic model checking can be used to ensure safety and retain performance of a learning algorithm known as *apprenticeship learning* (AL).

We consider the formulation of apprenticeship learning by Abbeel and Ng [1]. The concept of AL is closely related to *reinforcement learning* (RL) where an agent learns what actions to take in an environment (known as a policy) by maximizing some notion of long-term reward. In AL, however, the agent is not given the reward function, but instead has to first estimate it from a set of expert demonstrations via a technique called *inverse reinforcement learning* [18]. The formulation assumes that the reward function is expressible as a linear combination of *known state features*. An expert demonstrates the task by maximizing this reward function and the agent tries to derive a policy that can match the feature expectations of the expert's demonstrations. Apprenticeship learning can also be

viewed as an instance of the class of techniques known as Learning from Demonstration (LfD). One issue with LfD is that *the expert often can only demonstrate how the task works but not how the task may fail*. This is because failure may cause irrecoverable damages to the system such as crashing a vehicle. In general, the lack of “negative examples” can cause a heavy bias in how the learning agent constructs the reward estimate. In fact, *even if all the demonstrations are safe, the agent may still end up learning an unsafe policy*.

The key idea of this paper is to incorporate formal verification in apprenticeship learning. We are inspired by the line of work on formal inductive synthesis [10] and counterexample-guided inductive synthesis [22]. Our approach is also similar in spirit to the recent work on safety-constrained reinforcement learning [11]. However, our approach uses the results of model checking in a novel way. We consider safety specification expressed in probabilistic computation tree logic (PCTL). We employ a verification-in-the-loop approach by embedding PCTL model checking as a safety checking mechanism inside the learning phase of AL. In particular, when a learnt policy does not satisfy the PCTL formula, we leverage counterexamples generated by the model checker to steer the policy search in AL. In essence, counterexample generation can be viewed as supplementing negative examples for the learner. Thus, the learner will try to find a policy that not only imitates the expert’s demonstrations but also stays away from the failure scenarios as captured by the counterexamples.

In summary, we make the following contributions in this paper.

- We propose a novel framework for incorporating formal safety guarantees in Learning from Demonstration.
- We develop a novel algorithm called CounterExample Guided Apprenticeship Learning (CEGAL) that combines probabilistic model checking with the optimization-based approach of apprenticeship learning.
- We demonstrate that our approach can guarantee safety for a set of case studies and attain performance comparable to that of using apprenticeship learning alone.

The rest of the paper is organized as follows. Section 2 reviews background information on apprenticeship learning and PCTL model checking. Section 3 defines the safety-aware apprenticeship learning problem and gives an overview of our approach. Section 4 illustrates the counterexample-guided learning framework. Section 5 describes the proposed algorithm in detail. Section 6 presents a set of experimental results demonstrating the effectiveness of our approach. Section 7 discusses related work. Section 8 concludes and offers future directions.

2 Preliminaries

2.1 Markov Decision Process and Discrete-Time Markov Chain

Markov Decision Process (MDP) is a tuple $M = (S, A, T, \gamma, s_0, R)$, where S is a finite set of states; A is a set of actions; $T : S \times A \times S \rightarrow [0, 1]$ is a

transition function describing the probability of transitioning from one state $s \in S$ to another state by taking action $a \in A$ in state s ; $R : S \rightarrow \mathbb{R}$ is a reward function which maps each state $s \in S$ to a real number indicating the reward of being in state s ; $s_0 \in S$ is the initial state; $\gamma \in [0, 1)$ is a discount factor which describes how future rewards attenuate when a sequence of transitions is made. A deterministic and stationary (or memoryless) policy $\pi : S \rightarrow A$ for an MDP M is a mapping from states to actions, i.e. the policy deterministically selects what action to take solely based on the current state. In this paper, we restrict ourselves to deterministic and stationary policy. A policy π for an MDP M induces a Discrete-Time Markov Chain (DTMC) $M_\pi = (S, T_\pi, s_0)$, where $T_\pi : S \times S \rightarrow [0, 1]$ is the probability of transitioning from a state s to another state in one step. A trajectory $\tau = s_0 \xrightarrow{T_\pi(s_0, s_1) > 0} s_1 \xrightarrow{T_\pi(s_1, s_2) > 0} s_2, \dots$, is a sequence of states where $s_i \in S$. The accumulated reward of τ is $\sum_{i=0}^{\infty} \gamma^i R(s_i)$. The value function $V_\pi : S \rightarrow \mathbb{R}$ measures the expectation of accumulated reward $E[\sum_{i=0}^{\infty} \gamma^i R(s_i)]$ starting from a state s and following policy π . An *optimal policy* π for MDP M is a policy that maximizes the value function [4].

2.2 Apprenticeship Learning via Inverse Reinforcement Learning

Inverse reinforcement learning (IRL) aims at recovering the reward function R of $M \setminus R = (S, A, T, \gamma, s_0)$ from a set of m trajectories $\Gamma_E = \{\tau_0, \tau_1, \dots, \tau_{m-1}\}$ demonstrated by an expert. *Apprenticeship learning (AL)* [1] assumes that the reward function is a linear combination of state features, i.e. $R(s) = \omega^T f(s)$ where $f : S \rightarrow [0, 1]^k$ is a vector of known features over states S and $\omega \in \mathbb{R}^k$ is an unknown weight vector that satisfies $\|\omega\|_2 \leq 1$. The expected features of a policy π are the expected values of the cumulative discounted state features $f(s)$ by following π on M , i.e. $\mu_\pi = E[\sum_{t=0}^{\infty} \gamma^t f(s_t) | \pi]$. Let μ_E denote the expected features of the unknown expert's policy π_E . μ_E can be approximated by the expected features of expert's m demonstrated trajectories $\hat{\mu}_E = \frac{1}{m} \sum_{\tau \in \Gamma_E} \sum_{t=0}^{\infty} \gamma^t f(s_t)$ if m

is large enough. With a slight abuse of notations, we use μ_Γ to also denote the expected features of a set of paths Γ . Given an error bound ϵ , a policy π^* is defined to be ϵ -close to π_E if its expected features μ_{π^*} satisfies $\|\mu_E - \mu_{\pi^*}\|_2 \leq \epsilon$. The expected features of a policy can be calculated by using Monte Carlo method, value iteration or linear programming [1, 4].

The algorithm proposed by Abbeel and Ng [1] starts with a random policy π_0 and its expected features μ_{π_0} . Assuming that in iteration i , a set of i candidate policies $\Pi = \{\pi_0, \pi_1, \dots, \pi_{i-1}\}$ and their corresponding expected features $\{\mu_\pi | \pi \in \Pi\}$ have been found, the algorithm solves the following optimization problem.

$$\delta = \max_{\omega} \min_{\pi \in \Pi} \omega^T (\hat{\mu}_E - \mu_\pi) \quad s.t. \|\omega\|_2 \leq 1 \quad (1)$$

The optimal ω is used to find the corresponding optimal policy π_i and the expected features μ_{π_i} . If $\delta \leq \epsilon$, then the algorithm terminates and π_i is produced

as the output. Otherwise, μ_{π_i} is added to the set of features for the candidate policy set Π and the algorithm continues to the next iteration.

2.3 PCTL Model Checking

Probabilistic model checking can be used to verify properties of a stochastic system such as “is the probability that the agent reaches the unsafe area within 10 steps smaller than 5%?”. *Probabilistic Computation Tree Logic* (PCTL) [7] allows for probabilistic quantification of properties. The syntax of PCTL includes state formulas and path formulas [13]. A state formula ϕ asserts property of a single state $s \in S$ whereas a path formula ψ asserts property of a trajectory.

$$\phi ::= \text{true} \mid l_i \mid \neg\phi_i \mid \phi_i \wedge \phi_j \mid P_{\bowtie p^*}[\psi] \quad (2)$$

$$\psi ::= \mathbf{X}\phi \mid \phi_1 \mathbf{U}^{\leq k} \phi_2 \mid \phi_1 \mathbf{U} \phi_2 \quad (3)$$

where l_i is atomic proposition and ϕ_i, ϕ_j are state formulas; $\bowtie \in \{\leq, \geq, <, >\}$; $P_{\bowtie p^*}[\psi]$ means that the probability of generating a trajectory that satisfies formula ψ is $\bowtie p^*$. $\mathbf{X}\phi$ asserts that the next state after initial state in the trajectory satisfies ϕ ; $\phi_1 \mathbf{U}^{\leq k} \phi_2$ asserts that ϕ_2 is satisfied in at most k transitions and all preceding states satisfy ϕ_1 ; $\phi_1 \mathbf{U} \phi_2$ asserts that ϕ_2 will be eventually satisfied and all preceding states satisfy ϕ_1 . The semantics of PCTL is defined by a satisfaction relation \models as follows.

$$s \models \text{true} \quad \text{iff state } s \in S \quad (4)$$

$$s \models \phi \quad \text{iff state } s \text{ satisfies the state formula } \phi \quad (5)$$

$$\tau \models \psi \quad \text{iff trajectory } \tau \text{ satisfies the path formula } \psi. \quad (6)$$

Additionally, \models_{\min} denotes the minimal satisfaction relation [6] between τ and ψ . Defining $\text{pref}(\tau)$ as the set of all prefixes of trajectory τ including τ itself, then $\tau \models_{\min} \psi$ iff $(\tau \models \psi) \wedge (\forall \tau' \in \text{pref}(\tau) \setminus \tau, \tau' \not\models \psi)$. For instance, if $\psi = \phi_1 \mathbf{U}^{\leq k} \phi_2$, then for any finite trajectory $\tau \models_{\min} \phi_1 \mathbf{U}^{\leq k} \phi_2$, only the final state in τ satisfies ϕ_2 . Let $P(\tau)$ be the probability of transitioning along a trajectory τ and let Γ_ψ be the set of all finite trajectories that satisfy $\tau \models_{\min} \psi$, the value of PCTL property ψ is defined as $P_{=?|s_0}[\psi] = \sum_{\tau \in \Gamma_\psi} P(\tau)$. For a DTMC

M_π and a state formula $\phi = P_{\leq p^*}[\psi]$, $M_\pi \models \phi$ iff $P_{=?|s_0}[\psi] \leq p^*$.

A *counterexample* of ϕ is a set $\text{cex} \subseteq \Gamma_\psi$ that satisfies $\sum_{\tau \in \text{cex}} P(\tau) > p^*$.

Let $\mathbb{P}(\Gamma) = \sum_{\tau \in \Gamma} P(\tau)$ be the sum of probabilities of all trajectories in a set Γ .

Let $\text{CEX}_\phi \subseteq 2^{\Gamma_\psi}$ be the set of all counterexamples for a formula ϕ such that $(\forall \text{cex} \in \text{CEX}_\phi, \mathbb{P}(\text{cex}) > p^*)$ and $(\forall \Gamma \in 2^{\Gamma_\psi} \setminus \text{CEX}_\phi, \mathbb{P}(\Gamma) \leq p^*)$. A *minimal counterexample* is a set $\text{cex} \in \text{CEX}_\phi$ such that $\forall \text{cex}' \in \text{CEX}_\phi, |\text{cex}| \leq |\text{cex}'|$. By converting DTMC M_π into a weighted directed graph, counterexample can be found by solving a k-shortest paths (KSP) problem or a hop-constrained KSP (HKSP) problem [6]. Alternatively, counterexamples can be found by using Satisfiability Modulo Theory solving or mixed integer linear programming to

determine the minimal critical subsystems that capture the counterexamples in M_π [23].

A policy can also be synthesized by solving the objective $\min_\pi P_{=?}[\psi]$ for an MDP M . This problem can be solved by linear programming or policy iteration (and value iteration for step-bounded reachability) [14].

3 Problem Formulation and Overview

Suppose there are some unsafe states in an $MDP \setminus RM = (S, A, T, \gamma, s_0)$. A safety issue in apprenticeship learning means that an agent following the learnt policy would have a higher probability of entering those unsafe states than it should. There are multiple reasons that can give rise to this issue. First, it is possible that the expert policy π_E itself has a high probability of reaching the unsafe states. Second, human experts often tend to perform only successful demonstrations that do not highlight the unwanted situations [21]. This *lack of negative examples* in the training set can cause the learning agent to be unaware of the existence of those unsafe states.

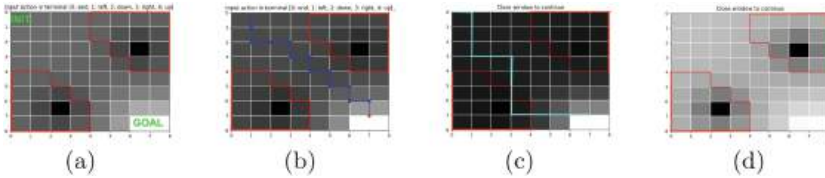


Fig. 1. The 8×8 grid-world. (a) Lighter grid cells have higher rewards than the darker ones. The two black grid cells have the lowest rewards, while the two white ones have the highest rewards. The grid cells enclosed by red lines are considered *unsafe*. (b) The blue line is an example trajectory demonstrated by the expert. (c) Only the goal states are assigned high rewards and there is little difference between the unsafe states and their nearby states. As a result, the learnt policy has a high probability of passing through the unsafe states as indicated by the cyan line. (d) $p^* = 20\%$. The learnt policy is optimal to a reward function that correctly assigns low rewards to the unsafe states. (Color figure online)

We use a 8×8 grid-world navigation example as shown in Fig. 1 to illustrate this problem. An agent starts from the upper-left corner and moves from cell to cell until it reaches the lower-right corner. The ‘unsafe’ cells are enclosed by the red lines. These represent regions that the agent should avoid. In each step, the agent can choose to stay in current cell or move to an adjacent cell but with 20% chance of moving randomly instead of following its decision. The goal area, the unsafe area and the reward mapping for all states are shown in Fig. 1(a). For each state $s \in S$, its feature vector consists of 4 radial basis feature functions with respect to the squared Euclidean distances between s and the 4 states with the highest or lowest rewards as shown in Fig. 1(a). In addition, a specification

Φ formalized in PCTL is used to capture the safety requirement. In (7), p^* is the required upper bound of the probability of reaching an unsafe state within $t = 64$ steps.

$$\Phi ::= P_{\leq p^*}[\text{true } \mathbf{U}^{\leq t} \text{ unsafe}] \quad (7)$$

Let π_E be the optimal policy under the reward map shown in Fig. 1(a). The probability of entering an unsafe region within 64 steps by following π_E is 24.6%. Now consider the scenario where the expert performs a number of demonstrations by following π_E . *All demonstrated trajectories in this case successfully reach the goal areas without ever passing through any of the unsafe regions.* Figure 1(b) shows a representative trajectory (in blue) among 10,000 such demonstrated trajectories. The resulting reward map by running the AL algorithm on these 10,000 demonstrations is shown in Fig. 1(c). Observe that only the goal area has been learnt whereas the agent is oblivious to the unsafe regions (treating them in the same way as other dark cells). In fact, the probability of reaching an unsafe state within 64 steps with this policy turns out to be 82.6% (thus violating the safety requirement by a large margin). To make matters worse, the value of p^* may be decided or revised after a policy has been learnt. In those cases, even the original expert policy π_E may be unsafe, e.g., when $p^* = 20\%$. Thus, we need to adapt the original AL algorithm so that it will take into account of such safety requirement. Figure 1(d) shows the resulting reward map learned using our proposed algorithm (to be described in detail later) for $p^* = 20\%$. It clearly matches well with the color differentiation in the original reward map and captures both the goal states and the unsafe regions. This policy has an unsafe probability of 19.0%. We are now ready to state our problem.

Definition 1. *The **safety-aware apprenticeship learning (SafeAL) problem** is, given an MDP $\langle R, \gamma \rangle$, a set of m trajectories $\{\tau_0, \tau_1, \dots, \tau_{m-1}\}$ demonstrated by an expert, and a specification Φ , to learn a policy π that satisfies Φ and is ϵ -close to the expert policy π_E .*

Remark 1. We note that a solution may not always exist for the SafeAL problem. While the decision problem of checking whether a solution exists is of theoretical interest, in this paper, we focus on tackling the problem of finding a policy π that satisfies a PCTL formula Φ (if Φ is satisfiable) and whose performance is as close to that of the expert's as possible, i.e. we relax the condition on μ_π being ϵ -close to μ_E .

4 A Framework for Safety-Aware Learning

In this section, we describe a general framework for safety-aware learning. This novel framework utilizes information from both the expert demonstrations and a verifier. The proposed framework is illustrated in Fig. 2. Similar to the *counterexample-guided inductive synthesis* (CEGIS) paradigm [22], our framework consists of a *verifier* and a *learner*. The verifier checks if a candidate policy satisfies the safety specification Φ . In case Φ is not satisfied, the verifier generates a

counterexample for Φ . The main difference from CEGIS is that our framework considers not only functional correctness, e.g., safety, but also performance (as captured by the learning objective). Starting from an initial policy π_0 , each time the learner learns a new policy, the verifier checks if the specification is satisfied. If true, then this policy is added to the candidate set, otherwise the verifier will generate a (minimal) counterexample and add it to the counterexample set. During the learning phase, the learner uses both the counterexample set and candidate set to find a policy that is close to the (unknown) expert policy and far away from the counterexamples. The goal is to find a policy that is ϵ -close to the expert policy and satisfies the specification. For the grid-world example introduced in Sect. 3, when $p^* = 5\%$ (thus presenting a stricter safety requirement compared to the expert policy π_E), our approach produces a policy with only 4.2% of reaching an unsafe state within 64 steps (with the correspondingly inferred reward mapping shown in Fig. 1(d)).

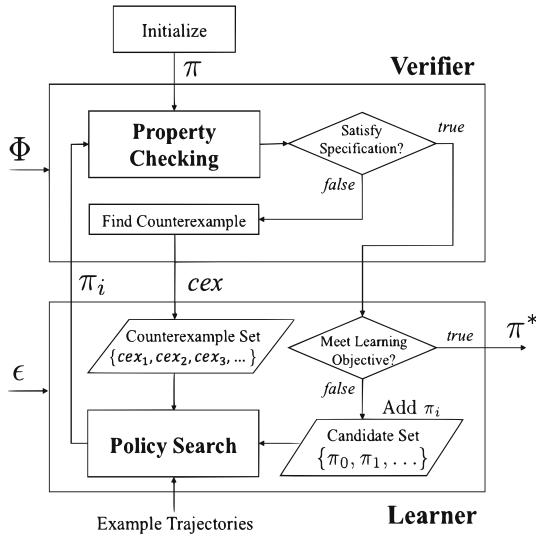


Fig. 2. Our safety-aware learning framework. Given an initial policy π_0 , a specification Φ and a learning objective (as captured by ϵ), the framework iterates between a *verifier* and a *learner* to search for a policy π^* that satisfies both Φ and ϵ . One invariant that this framework maintains is that all the π_i 's in the candidate policy set satisfy Φ .

Learning from a (minimal) counterexample cex_π of a policy π is similar to learning from expert demonstrations. The basic principle of the AL algorithm proposed in [1] is to find a weight vector ω under which the expected reward of π_E maximally outperforms any mixture of the policies in the candidate policy set $\Pi = \{\pi_0, \pi_1, \pi_2, \dots\}$. Thus, ω can be viewed as the normal vector of the hyperplane $\omega^T(\mu - \mu_E) = 0$ that has the maximal distance to the convex hull of the set $\{\mu_\pi \mid \pi \in \Pi\}$ as illustrated in the 2D feature space in Fig. 3(a). It can be shown

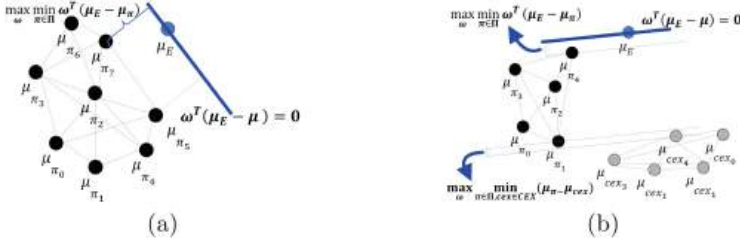


Fig. 3. (a) Learn from expert. (b) Learn from both expert demonstrations and counterexamples.

that $\omega^T \mu_{\pi} \geq \omega^T \mu_{\pi'}$ for all previously found π' s. Intuitively, this helps to move the candidate μ_{π} closer to μ_E . Similarly, we can apply the same max-margin separation principle to maximize the distance between the candidate policies and the counterexamples (in the μ space). Let $CEX = \{cex_0, cex_1, cex_2, \dots\}$ denote the set of counterexamples of the policies that do not satisfy the specification Φ . Maximizing the distance between the convex hulls of the sets $\{\mu_{cex} | cex \in CEX\}$ and $\{\mu_{\pi} | \pi \in \Pi\}$ is equivalent to maximizing the distance between the parallel supporting hyperplanes of the two convex hulls as shown in Fig. 3(b). The corresponding optimization function is given in Eq. (8).

$$\delta = \max_{\omega} \min_{\pi \in \Pi, cex \in CEX} \omega^T(\mu_{\pi} - \mu_{cex}) \quad s.t. \|\omega\|_2 \leq 1 \quad (8)$$

To attain good performance similar to that of the expert, we still want to learn from μ_E . Thus, the overall problem can be formulated as a multi-objective optimization problem that combines (1) and (8) into (9).

$$\max_{\omega} \min_{\pi \in \Pi, \tilde{\pi} \in \Pi, cex \in CEX} (\omega^T(\mu_E - \mu_{\pi}), \omega^T(\mu_{\tilde{\pi}} - \mu_{cex})) \quad s.t. \|\omega\|_2 \leq 1 \quad (9)$$

5 Counterexample-Guided Apprenticeship Learning

In this section, we introduce the CounterExample Guided Apprenticeship Learning (CEGAL) algorithm to solve the SafeAL problem. It can be viewed as a special case of the safety-aware learning framework described in the previous section. In addition to combining policy verification, counterexample generation and AL, our approach uses an adaptive weighting scheme to weight the separation from μ_E with the separation from μ_{cex} .

$$\begin{aligned} & \max_{\omega} \min_{\pi \in \Pi_S, \tilde{\pi} \in \Pi_S, cex \in CEX} \omega^T(k(\mu_E - \mu_{\pi}) + (1 - k)(\mu_{\tilde{\pi}} - \mu_{cex})) \quad (10) \\ & s.t. \|\omega\|_2 \leq 1, k \in [0, 1] \\ & \omega^T(\mu_E - \mu_{\pi}) \leq \omega^T(\mu_E - \mu_{\pi'}), \forall \pi' \in \Pi_S \\ & \omega^T(\mu_{\tilde{\pi}} - \mu_{cex}) \leq \omega^T(\mu_{\tilde{\pi}} - \mu_{cex'}), \forall \tilde{\pi}' \in \Pi_S, \forall cex' \in CEX \end{aligned}$$

In essence, we take a weighted-sum approach for solving the multi-objective optimization problem (9). Assuming that $\Pi_S = \{\pi_1, \pi_2, \pi_3, \dots\}$ is a set of candidate policies that all satisfy Φ , $CEX = \{cex_1, cex_2, cex_3, \dots\}$ is a set of counterexamples. We introduce a parameter k and change (9) into a weighted sum optimization problem (10). Note that π and $\tilde{\pi}$ can be different. The optimal ω solved from (10) can be used to generate a new policy π_ω by using algorithms such as policy iteration. We use a probabilistic model checker, such as PRISM [13], to check if π_ω satisfies Φ . If it does, then it will be added to Π_S . Otherwise, a counterexample generator, such as COMICS [9], is used to generate a (minimal) counterexample cex_{π_ω} , which will be added to CEX .

Algorithm 1. Counterexample-Guided Apprenticeship Learning (CEGAL)

```

1: Input:
2:    $M \leftarrow$  A partially known  $MDP \setminus R$ ;  $f \leftarrow$  A vector of feature functions
3:    $\mu_E \leftarrow$  The expected features of expert trajectories  $\{\tau_0, \tau_1, \dots, \tau_m\}$ 
4:    $\Phi \leftarrow$  Specification;  $\epsilon \leftarrow$  Error bound for the expected features;
5:    $\sigma, \alpha \in (0, 1) \leftarrow$  Error bound  $\sigma$  and step length  $\alpha$  for the parameter  $k$ ;
6: Initialization:
7:   If  $\|\mu_E - \mu_{\pi_0}\|_2 \leq \epsilon$ , then return  $\pi_0$   $\triangleright \pi_0$  is the initial safe policy
8:    $\Pi_S \leftarrow \{\pi_0\}$ ,  $CEX \leftarrow \emptyset$   $\triangleright$  Initialize candidate and counterexample set
9:    $inf \leftarrow 0$ ,  $sup \leftarrow 1$ ,  $k \leftarrow sup$   $\triangleright$  Initialize multi-optimization parameter  $k$ 
10:   $\pi_1 \leftarrow$  Policy learnt from  $\mu_E$  via apprenticeship learning
11: Iteration  $i$  ( $i \geq 1$ ):
12:   Verifier:
13:      $status \leftarrow Model\_Checker(M, \pi_i, \Phi)$ 
14:     If  $status = SAT$ , then go to Learner
15:     If  $status = UNSAT$ 
16:        $cex_{\pi_i} \leftarrow Counterexample\_Generator(M, \pi_i, \Phi)$ 
17:       Add  $cex_{\pi_i}$  to  $CEX$  and solve  $\mu_{cex_{\pi_i}}$ , go to Learner
18:   Learner:
19:     If  $status = SAT$ 
20:       If  $\|\mu_E - \mu_{\pi_i}\|_2 \leq \epsilon$ , then return  $\pi^* \leftarrow \pi_i$   $\triangleright$  Terminate.  $\pi_i$  is  $\epsilon$ -close to  $\pi_E$ 
21:       Add  $\pi_i$  to  $\Pi_S$ ,  $inf \leftarrow k$ ,  $k \leftarrow sup$   $\triangleright$  Update  $\Pi_S$ ,  $inf$  and reset  $k$ 
22:     If  $status = UNSAT$ 
23:       If  $|k - inf| \leq \sigma$ , then return  $\pi^* \leftarrow \underset{\pi \in \Pi_S}{argmin} \|\mu_E - \mu_\pi\|_2$   $\triangleright$  Terminate.  $k$  is too close to its lower bound.
24:        $k \leftarrow \alpha \cdot inf + (1 - \alpha)k$   $\triangleright$  Decrease  $k$  to learn for safety
25:        $\omega_{i+1} \leftarrow \underset{\omega}{argmax} \min_{\pi \in \Pi_S, \tilde{\pi} \in \Pi_S, cex \in CEX} \omega^T(k(\mu_E - \mu_\pi) + (1 - k)(\mu_{\tilde{\pi}} - \mu_{cex}))$ 
26:        $\triangleright$  Note that the multi-objective optimization function recovers AL when  $k = 1$ 
27:        $\pi_{i+1}, \mu_{\pi_{i+1}} \leftarrow$  Compute the optimal policy  $\pi_{i+1}$  and its expected features  $\mu_{\pi_{i+1}}$  for the MDP  $M$  with reward  $R(s) = \omega_{i+1}^T f(s)$ 
28:       Go to next iteration

```

Algorithm 1 describes CEGAL in detail. With a constant $sup = 1$ and a variable $inf \in [0, sup]$ for the upper and lower bounds respectively, the learner

determines the value of k within $[inf, sup]$ in each iteration depending on the outcome of the verifier and uses k in solving (10) in line 27. Like most nonlinear optimization algorithms, this algorithm requires an initial guess, which is an initial safe policy π_0 to make Π_S nonempty. A good initial candidate would be the maximally safe policy for example obtained using PRISM-games [15]. Without loss of generality, we assume this policy satisfies Φ . Suppose in iteration i , an intermediate policy π_i learnt by the learner in iteration $i - 1$ is verified to satisfy Φ , then we increase inf to $inf = k$ and reset k to $k = sup$ as shown in line 22. If π_i does not satisfy Φ , then we reduce k to $k = \alpha \cdot inf + (1 - \alpha)k$ as shown in line 26 where $\alpha \in (0, 1)$ is a step length parameter. If $|k - inf| \leq \sigma$ and π_i still does not satisfy Φ , the algorithm chooses from Π_S a best safe policy π^* which has the smallest margin to π_E as shown in line 24. If π_i satisfies Φ and is ϵ -close to π_E , the algorithm outputs π_i as show in line 19. For the occasions when π_i satisfies Φ and $inf = sup = k = 1$, solving (10) is equivalent to solving (1) as in the original AL algorithm.

Remark 2. The initial policy π_0 does not have to be maximally safe, although such a policy can be used to verify if Φ is satisfiable at all. Naively safe policies often suffice for obtaining a safe and performant output at the end. Such a policy can be obtained easily in many settings, e.g., in the grid-world example one safe policy is simply staying in the initial cell. In both cases, π_0 typically has very low performance since satisfying Φ is the only requirement for it.

Theorem 1. *Given an initial policy π_0 that satisfies Φ , Algorithm 1 is guaranteed to output a policy π^* , such that (1) π^* satisfies Φ , and (2) the performance of π^* is at least as good as that of π_0 when compared to π_E , i.e. $\|\mu_E - \mu_{\pi^*}\|_2 \leq \|\mu_E - \mu_{\pi_0}\|_2$.*

Proof Sketch. The first part of the guarantee can be proven by case splitting. Algorithm 1 outputs π^* either when π^* satisfies Φ and is ϵ -close to π_E , or when $|k - inf| \leq \sigma$ in some iteration. In the first case, π^* clearly satisfies Φ . In the second case, π^* is selected from the set Π_S which contains all the policies that have been found to satisfy Φ so far, so π^* satisfies Φ . For the second part of the guarantee, the initial policy π_0 is the final output π^* if π_0 satisfies Φ and is ϵ -close to π_E . Otherwise, π_0 is added to Π_S if it satisfies Φ . During the iteration, if $|k - inf| \leq \sigma$ in some iteration, then the final output is $\pi^* = \underset{\pi \in \Pi_S}{argmin} \|\mu_E - \mu_\pi\|_2$,

so it must satisfy $\|\mu_E - \mu_{\pi^*}\|_2 \leq \|\mu_E - \mu_{\pi_0}\|_2$. If a learnt policy π^* satisfies Φ and is ϵ -close to π_E , then Algorithm 1 outputs π^* without adding it to Π_S . Obviously $\|\mu_E - \mu_\pi\|_2 > \epsilon, \forall \pi \in \Pi_S$, so $\|\mu_E - \mu_{\pi^*}\|_2 \leq \|\mu_E - \mu_{\pi_0}\|_2$.

Discussion. In the worst case, CEGAL will return the initial safe policy. However, this can be because a policy that simultaneously satisfies Φ and is ϵ -close to the expert's demonstrations does not exist. Comparing to AL which offers no safety guarantee and finding the maximally safe policy which has very poor performance, CEGAL provides a principled way of guaranteeing safety while retaining performance.

Convergence. Algorithm 1 is guaranteed to terminate. Let inf_t be the t^{th} assigned value of inf . After inf_t is given, k is decreased from $k_0 = sup$ iteratively by $k_i = \alpha \cdot inf_t + (1 - \alpha)k_{i-1}$ until either $|k_i - inf_t| \leq \sigma$ in line 24 or a new safe policy is found in line 18. The update of k satisfies the following equality.

$$\frac{|k_{i+1} - inf_t|}{|k_i - inf_t|} = \frac{\alpha \cdot inf_t + (1 - \alpha)k_i - inf_t}{k_i - inf_t} = 1 - \alpha \quad (11)$$

Thus, it takes no more than $1 + \log_{1-\alpha} \frac{\sigma}{sup - inf_t}$ iterations for either the algorithm to terminate in line 24 or a new safe policy to be found in line 18. If a new safe policy is found in line 18, inf will be assigned in line 22 by the current value of k as $inf_{t+1} = k$ which obviously satisfies $inf_{t+1} - inf_t \geq (1 - \alpha)\sigma$. After the assignment of inf_{t+1} , the iterative update of k resumes. Since $sup - inf_t \leq 1$, the following inequality holds.

$$\frac{|inf_{t+1} - sup|}{|inf_t - sup|} \leq \frac{sup - inf_t - (1 - \alpha)\sigma}{sup - inf_t} \leq 1 - (1 - \alpha)\sigma \quad (12)$$

Obviously, starting from an initial $inf = inf_0 < sup$, with the alternating update of inf and k , inf will keep getting close to sup unless the algorithm terminates as in line 24 or a safe policy ϵ -close to π_E is found as in line 19. The extreme case is that finally $inf = sup$ after no more than $\frac{sup - inf_0}{(1 - \alpha)\sigma}$ updates on inf . Then, the problem becomes AL. Therefore, the worst case of this algorithm can have two phases. In the first phase, inf increases from $inf = 0$ to $inf = sup$. Between each two consecutive updates $(t, t + 1)$ on inf , there are no more than $\log_{1-\alpha} \frac{(1 - \alpha)\sigma}{sup - inf_t}$ updates on k before inf is increased from inf_t to inf_{t+1} . Overall, this phase takes no more than

$$\sum_{0 \leq i < \frac{sup - inf_0}{(1 - \alpha)\sigma}} \log_{1-\alpha} \frac{(1 - \alpha)\sigma}{sup - inf_0 - i \cdot (1 - \alpha)\sigma} = \sum_{0 \leq i < \frac{1}{(1 - \alpha)\sigma}} \log_{1-\alpha} \frac{(1 - \alpha)\sigma}{1 - i \cdot (1 - \alpha)\sigma} \quad (13)$$

iterations to reduce the multi-objective optimization problem to original apprenticeship learning and then the second phase begins. Since $k = sup$, the iteration will stop immediately when an unsafe policy is learnt as in line 24. This phase will not take more iterations than original AL algorithm does to converge and the convergence result of AL is given in [1].

In each iteration, the algorithm first solves a second-order cone programming (SOCP) problem (10) to learn a policy. SOCP problems can be solved in polynomial time by interior-point (IP) methods [12]. PCTL model checking for DTMCs can be solved in time linear in the size of the formula and polynomial in the size of the state space [7]. Counterexample generation can be done either by enumerating paths using the k -shortest path algorithm or determining a critical subsystem using either a *SMT* formulation or mixed integer linear programming (MILP) [23]. For the k -shortest path-based algorithm, it can be computationally expensive sometimes to enumerate a large amount of paths (i.e. a large k) when p^* is large. This can be alleviated by using a smaller p^* during calculation, which is equivalent to considering only paths that have high probabilities.

6 Experiments

We evaluate our algorithm on three case studies: (1) grid-world, (2) cart-pole, and (3) mountain-car. The cart-pole environment¹ and the mountain-car environment² are obtained from OpenAI Gym. All experiments are carried out on a quad-core i7-7700K processor running at 3.6GHz with 16GB of memory. Our prototype tool was implemented in Python³. The parameters are $\gamma = 0.99, \epsilon = 10, \sigma = 10^{-5}, \alpha = 0.5$ and the maximum number of iterations is 50. For the OpenAI-gym experiments, in each step, the agent sends an action to the OpenAI environment and the environment returns an observation and a reward (0 or 1). We show that our algorithm can guarantee safety while retaining the performance of the learnt policy compared with using AL alone.

6.1 Grid World

We first evaluate the scalability of our tool using the grid-world example. Table 1 shows the average runtime (per iteration) for the individual components of our tool as the size of the grid-world increases. The first and second columns indicate the size of the grid world and the resulting state space. The third column shows the average runtime that policy iteration takes to compute an optimal policy π for a known reward function. The forth column indicates the average runtime that policy iteration takes to compute the expected features μ for a known policy. The fifth column indicates the average runtime of verifying the PCTL formula using PRISM. The last column indicates the average runtime that generating a counterexample using COMICS.

Table 1. Average runtime per iteration in seconds.

| Size | Num. of states | Compute π | Compute μ | MC | Cex |
|----------------|----------------|---------------|---------------|--------|-------|
| 8×8 | 64 | 0.02 | 0.02 | 1.39 | 0.014 |
| 16×16 | 256 | 0.05 | 0.05 | 1.43 | 0.014 |
| 32×32 | 1024 | 0.07 | 0.08 | 3.12 | 0.035 |
| 64×64 | 4096 | 6.52 | 25.88 | 22.877 | 1.59 |

6.2 Cart-Pole from OpenAI Gym

In the cart-pole environment as shown in Fig. 4(a), the goal is to keep the pole on a cart from falling over as long as possible by moving the cart either to the left or to the right in each time step. The maximum step length is $t = 200$. The

¹ <https://github.com/openai/gym/wiki/CartPole-v0>.

² <https://github.com/openai/gym/wiki/MountainCar-v0>.

³ <https://github.com/zwc662/CAV2018>.

position, velocity and angle of the cart and the pole are continuous values and observable, but the actual dynamics of the system are unknown⁴.

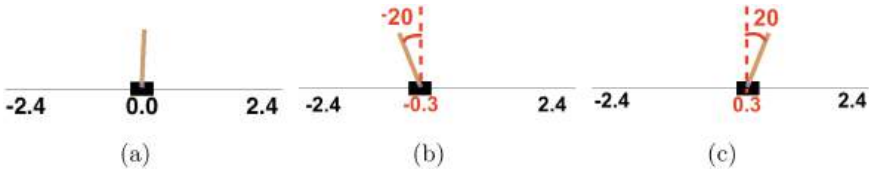


Fig. 4. (a) The cart-pole environment. (b) The cart is at -0.3 and pole angle is -20° . (c) The cart is at 0.3 and pole angle is 20° .

A maneuver is deemed *unsafe* if the pole angle is larger than $\pm 20^\circ$ while the cart’s horizontal position is more than ± 0.3 as shown in Fig. 4(b) and (c). We formalize the safety requirement in PCTL as (14).

$$\begin{aligned} \Phi ::= P_{\leq p^*} [true \text{ U}^{\leq t} (angle \leq -20^\circ \wedge position \leq -0.3) \\ \vee (angle \geq 20^\circ \wedge position \geq 0.3)] \end{aligned} \quad (14)$$

Table 2. In the cart-pole environment, *higher* average steps mean better performance. The safest policy is synthesized using PRISM-games.

| | MC Result | Avg. Steps | Num. of Iters |
|---------------|-----------|------------|---------------|
| AL | 49.1% | 165 | 2 |
| Safest Policy | 0.0% | 8 | N.A. |
| $p^* = 30\%$ | 17.2% | 121 | 10 |
| $p^* = 25\%$ | 9.3% | 136 | 14 |
| $p^* = 20\%$ | 17.2% | 122 | 10 |
| $p^* = 15\%$ | 6.9% | 118 | 22 |
| $p^* = 10\%$ | 7.2% | 136 | 22 |
| $p^* = 5\%$ | 0.04% | 83 | 50 |

We used 2000 demonstrations for which the pole is held upright without violating any of the safety conditions for all 200 steps in each demonstration. The safest policy synthesized by PRISM-games is used as the initial safe policy. We also compare the different policies learned by CEGAL for different safety threshold p^* s. In Table 2, the policies are compared in terms of model checking results

⁴ The MDP is built from sampled data. The feature vector in each state contains 30 radial basis functions which depend on the squared Euclidean distances between current state and other 30 states which are uniformly distributed in the state space.

(‘MC Result’) on the PCTL property in (14) using the constructed MDP, the average steps (‘Avg. Steps’) that a policy (executed in the OpenAI environment) can hold across 5000 rounds (the higher the better), and the number of iterations (‘Num. of Iters’) it takes for the algorithm to terminate (either converge to an ϵ -close policy, or terminate due to σ , or terminate after 50 iterations). The policy in the first row is the result of using AL alone, which has the best performance but also a 49.1% probability of violating the safety requirement. The safest policy as shown in the second row is always safe has almost no performance at all. This policy simply lets the pole fall and thus does not risk moving the cart out of the range $[-0.3, 0.3]$. On the other hand, it is clear that the policies learnt using CEGAL always satisfy the safety requirement. From $p^* = 30\%$ to 10% , the performance of the learnt policy is comparable to that of the AL policy. However, when the safety threshold becomes very low, e.g., $p^* = 5\%$, the performance of the learnt policy drops significantly. This reflects the phenomenon that the tighter the safety condition is the less room for the agent to maneuver to achieve a good performance.

6.3 Mountain-Car from OpenAI Gym

Our third experiment uses the mountain-car environment from OpenAI Gym. As shown in Fig. 5(a), a car starts from the bottom of the valley and tries to reach the mountaintop on the right as quickly as possible. In each time step the car can perform one of the three actions, accelerating to the left, coasting, and accelerating to the right. The agent fails if the step length reaches the maximum ($t = 66$). The velocity and position of the car are continuous values and observable while the exact dynamics are unknown⁵. In this game setting, the car cannot reach the right mountaintop by simply accelerating to the right. It has to accumulate momentum first by moving back and forth in the valley. The safety rules we enforce are shown in Fig. 5(b). They correspond to speed limits when the car is close to the left mountaintop or to the right mountaintop (in case it is a cliff on the other side of the mountaintop). Similar to the previous experiments, we considered 2000 expert demonstrations for which all of them successfully reach the right mountaintop without violating any of the safety conditions. The average number of steps for the expert to drive the car to the right mountaintop is 40. We formalize the safety requirement in PCTL as (15).

$$\begin{aligned} \Phi ::= P_{\leq p^*} [true \text{ } \mathbf{U}^{\leq t} (speed \leq -0.04 \wedge position \leq -1.1) \\ \vee (speed \geq 0.04 \wedge position \geq 0.5)] \end{aligned} \quad (15)$$

We compare the different policies using the same set of categories as in the cart-pole example. The numbers are averaged over 5000 runs. As shown in the

⁵ The MDP is built from sampled data. The feature vector for each state contains 2 exponential functions and 18 radial basis functions which respectively depend on the squared Euclidean distances between the current state and other 18 states which are uniformly distributed in the state space.

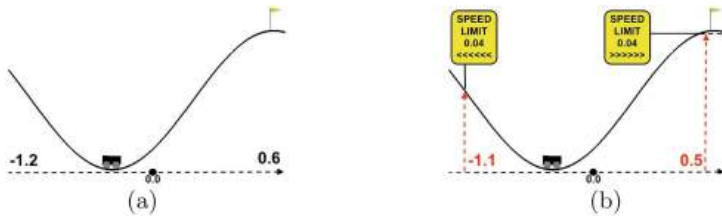


Fig. 5. (a) The original mountain-car environment. (b) The mountain-car environment with traffic rules: when the distance from the car to the left edge or the right edge is shorter than 0.1, the speed of the car should be lower than 0.04.

first row, the policy learnt via AL⁶ has the highest probability of going over the speed limits. We observed that this policy made the car speed up all the way to the left mountaintop to maximize its potential energy. The safest policy corresponds to simply staying in the bottom of the valley. The policies learnt via CEGAL for safety threshold p^* ranging from 60% to 50% not only have lower probability of violating the speed limits but also achieve comparable performance. As the safety threshold p^* decreases further, the agent becomes more conservative and it takes more time for the car to finish the task. For $p^* = 20\%$, the agent never succeeds in reaching the top within 66 steps (Table 3).

Table 3. In the mountain-car environment, *lower* average steps mean better performance. The safest policy is synthesized via PRISM-games.

| | MC Result | Avg. steps | Num. of Iters |
|----------------------|-----------|-------------|---------------|
| Policy Learnt via AL | 69.2% | 54 | 50 |
| Safest Policy | 0.0% | <i>Fail</i> | N.A. |
| $p^* = 60\%$ | 43.4% | 57 | 9 |
| $p^* = 50\%$ | 47.2% | 55 | 17 |
| $p^* = 40\%$ | 29.3% | 61 | 26 |
| $p^* = 30\%$ | 18.9% | 64 | 17 |
| $p^* = 20\%$ | 4.9% | <i>Fail</i> | 40 |

7 Related Work

A taxonomy of AI safety problems is given in [3] where the issues of misspecified objective or reward and insufficient or poorly curated training data are highlighted. There have been several attempts to address these issues from different angles. The problem of *safe exploration* is studied in [8, 17]. In particular, the latter work proposes to add a safety constraint, which is evaluated by amount

⁶ AL did not converge to an ϵ -close policy in 50 iterations in this case.

of damage, to the optimization problem so that the optimal policy can maximize the return without violating the limit on the expected damage. An obvious shortcoming of this approach is that actual failures will have to occur to properly assess damage.

Formal methods have been applied to the problem of AI safety. In [5], the authors propose to combine machine learning and reachability analysis for dynamical models to achieve high performance and guarantee safety. In this work, we focus on probabilistic models which are natural in many modern machine learning methods. In [20], the authors propose to use formal specification to synthesize a control policy for reinforcement learning. They consider formal specifications captured in Linear Temporal Logic, whereas we consider PCTL which matches better with the underlying probabilistic model. Recently, the problem of *safe reinforcement learning* was explored in [2] where a monitor (called shield) is used to enforce temporal logic properties either during the learning phase or execution phase of the reinforcement learning algorithm. The shield provides a list of safe actions each time the agent makes a decision so that the temporal property is preserved. In [11], the authors also propose an approach for controller synthesis in reinforcement learning. In this case, an SMT-solver is used to find a scheduler (policy) for the synchronous product of an MDP and a DTMC so that it satisfies both a probabilistic reachability property and an expected cost property. Another approach that leverages PCTL model checking is proposed in [16]. A so-called abstract Markov decision process (AMDP) model of the environment is first built and PCTL model checking is then used to check the satisfiability of safety specification. Our work is similar to these in spirit in the application of formal methods. However, while the concept of AL is closely related to reinforcement learning, an agent in the AL paradigm needs to learn a policy from demonstrations without knowing the reward function a priori.

A distinguishing characteristic of our method is the tight integration of formal verification with learning from data (apprenticeship learning in particular). Among imitation or apprenticeship learning methods, margin based algorithms [1, 18, 19] try to maximize the margin between the expert’s policy and all learnt policies until the one with the smallest margin is produced. The apprenticeship learning algorithm proposed by Abbeel and Ng [1] was largely motivated by the support vector machine (SVM) in that features of expert demonstration is maximally separated from all features of all other candidate policies. Our algorithm makes use of this observation when using counterexamples to steer the policy search process. Recently, the idea of learning from failed demonstrations started to emerge. In [21], the authors propose an IRL algorithm that can learn from both successful and failed demonstrations. It is done by reformulating maximum entropy algorithm in [24] to find a policy that maximally deviates from the failed demonstrations while approaching the successful ones as much as possible. However, this entropy-based method requires obtaining many failed demonstrations and can be very costly in practice.

Finally, our approach is inspired by the work on formal inductive synthesis [10] and counterexample-guided inductive synthesis (CEGIS) [22]. These

frameworks typically combine a constraint-based synthesizer with a verification oracle. In each iteration, the agent refines her hypothesis (i.e. generates a new candidate solution) based on counterexamples provided by the oracle. Our approach can be viewed as an extension of CEGIS where the objective is not just functional correctness but also meeting certain learning criteria.

8 Conclusion and Future Work

We propose a counterexample-guided approach for combining probabilistic model checking with apprenticeship learning to ensure safety of the apprenticeship learning outcome. Our approach makes novel use of counterexamples to steer the policy search process by reformulating the feature matching problem into a multi-objective optimization problem that additionally takes safety into account. Our experiments indicate that the proposed approach can guarantee safety and retain performance for a set of benchmarks including examples drawn from OpenAI Gym. In the future, we would like to explore other imitation or apprenticeship learning algorithms and extend our techniques to those settings.

Acknowledgement. This work is funded in part by the DARPA BRASS program under agreement number FA8750-16-C-0043 and NSF grant CCF-1646497.

References

1. Abbeel, P., Ng, A.Y.: Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the Twenty-First International Conference on Machine Learning, ICML 2004, p. 1. ACM, New York (2004)
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. CoRR, abs/1708.08611 (2017)
3. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in AI safety. CoRR, abs/1606.06565 (2016)
4. Bellman, R.: A Markovian decision process. *Indiana Univ. Math. J.* **6**, 15 (1957)
5. Gillulay, J.H., Tomlin, C.J.: Guaranteed safe online learning of a bounded system. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2979–2984. IEEE (2011)
6. Han, T., Katoen, J.P., Berteun, D.: Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.* **35**(2), 241–257 (2009)
7. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects Comput.* **6**(5), 512–535 (1994)
8. Held, D., McCarthy, Z., Zhang, M., Shentu, F., Abbeel, P.: Probabilistically safe policy transfer. CoRR, abs/1705.05394 (2017)
9. Jansen, N., Ábrahám, E., Scheffler, M., Volk, M., Vorpahl, A., Wimmer, R., Katoen, J., Becker, B.: The COMICS tool - computing minimal counterexamples for discrete-time Markov chains. CoRR, abs/1206.0603 (2012)
10. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. *Acta Informatica* **54**(7), 693–726 (2017)

11. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
12. Kuo, Y.-J., Mittelmann, H.D.: Interior point methods for second-order cone programming and or applications. *Comput. Optim. Appl.* **28**(3), 255–285 (2004)
13. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46029-2_13
14. Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 5–22. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_2
15. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. *Int. J. Softw. Tools Technol. Transfer* **20**, 195–210 (2017)
16. Mason, G.R., Calinescu, R.C., Kudenko, D., Banks, A.: Assured reinforcement learning for safety-critical applications. In: Doctoral Consortium at the 10th International Conference on Agents and Artificial Intelligence. SciTePress (2017)
17. Moldovan, T.M., Abbeel, P.: Safe exploration in Markov decision processes. *arXiv preprint arXiv:1205.4810* (2012)
18. Ng, A.Y., Russell, S.J.: Algorithms for inverse reinforcement learning. In: Proceedings of the Seventeenth International Conference on Machine Learning, ICML 2000, pp. 663–670. Morgan Kaufmann Publishers Inc., San Francisco (2000)
19. Ratliff, N.D., Bagnell, J.A., Zinkevich, M.A.: Maximum margin planning. In: Proceedings of the 23rd International Conference on Machine Learning, ICML 2006, pp. 729–736. ACM, New York (2006)
20. Sadigh, D., Kim, E.S., Coogan, S., Sastry, S.S., Seshia, S.A.: A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. *CoRR*, abs/1409.5486 (2014)
21. Shiarlis, K., Messias, J., Whiteson, S.: Inverse reinforcement learning from failure. In: Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems, pp. 1060–1068. International Foundation for Autonomous Agents and Multiagent Systems (2016)
22. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.* **40**(5), 404–415 (2006)
23. 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). https://doi.org/10.1007/978-3-642-28756-5_21
24. Ziebart, B.D., Maas, A., Bagnell, J.A., Dey, A.K.: Maximum entropy inverse reinforcement learning. In: Proceedings of the 23rd National Conference on Artificial Intelligence, AAAI 2008, vol. 3, pp. 1433–1438. AAAI Press (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Deciding Probabilistic Bisimilarity Distance One for Labelled Markov Chains

Qiyi Tang^(✉) and Franck van Breugel

DisCoVeri Group, York University, Toronto, Canada
`{qiyitang,franck}@eecs.yorku.ca`

Abstract. Probabilistic bisimilarity is an equivalence relation that captures which states of a labelled Markov chain behave the same. Since this behavioural equivalence only identifies states that transition to states that behave exactly the same with exactly the same probability, this notion of equivalence is not robust. Probabilistic bisimilarity distances provide a quantitative generalization of probabilistic bisimilarity. The distance of states captures the similarity of their behaviour. The smaller the distance, the more alike the states behave. In particular, states are probabilistic bisimilar if and only if their distance is zero. This quantitative notion is robust in that small changes in the transition probabilities result in small changes in the distances.

During the last decade, several algorithms have been proposed to approximate and compute the probabilistic bisimilarity distances. The main result of this paper is an algorithm that decides distance one in $O(n^2 + m^2)$, where n is the number of states and m is the number of transitions of the labelled Markov chain. The algorithm is the key new ingredient of our algorithm to compute the distances. The state of the art algorithm can compute distances for labelled Markov chains up to 150 states. For one such labelled Markov chain, that algorithm takes more than 49 h. In contrast, our new algorithm only takes 13 ms. Furthermore, our algorithm can compute distances for labelled Markov chains with more than 10,000 states in less than 50 min.

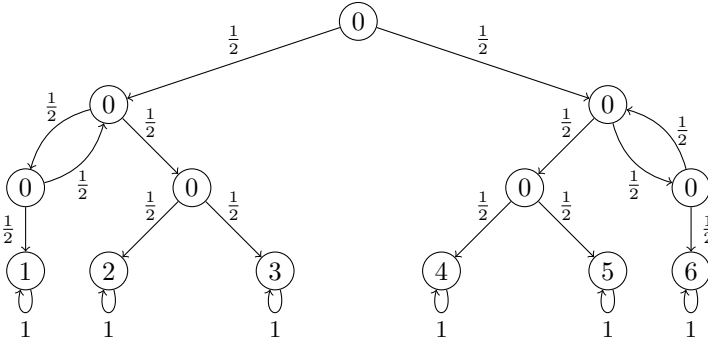
Keywords: Labelled Markov chain · Probabilistic bisimilarity
Probabilistic bisimilarity distance

1 Introduction

A *behavioural equivalence* captures which states of a model give rise to the same behaviour. Bisimilarity, due to Milner [22] and Park [25], is one of the best known behavioural equivalences. Verifying that an implementation satisfies a specification boils down to checking that the model of the implementation gives rise to the same behaviour as the model of the specification, that is, the models are behavioural equivalent (see [1, Chap. 3]).

In this paper, we focus on models of probabilistic systems. These models can capture randomized algorithms, probabilistic protocols, biological systems and

many other systems in which probabilities play a central role. In particular, we consider *labelled Markov chains*, that is, Markov chains the states of which are labelled.



The above example shows how the behaviour of rolling a die can be mimicked by flipping a coin, an example due to Knuth and Yao [19]. Six of the states are labelled with the values of a die and the other states are labelled zero. In this example, we are interested in the labels representing the value of a die. As the reader can easily verify, the states with these labels are each reached with probability $\frac{1}{6}$ from the initial, top most, state. In general, labels are used to identify particular states that have properties of interest. As a consequence, states with different labels are not behaviourally equivalent.

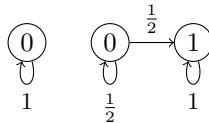
Probabilistic bisimilarity, due to Larsen and Skou [21], is a key behavioural equivalence for labelled Markov chains. As shown by Katoen et al. [16], minimizing a labelled Markov chain by identifying those states that are probabilistic bisimilar speeds up model checking. Probabilistic bisimilarity only identifies those states that behave exactly the same with exactly the same probability. If, for example, we replace the fair coin in the above example with a biased one, then none of the states labelled with zero in the original model with the fair coin are behaviourally equivalent to any of the states labelled with zero in the model with the biased coin. Behavioural equivalences like probabilistic bisimilarity rely on the transition probabilities and, as a result, are sensitive to minor changes of those probabilities. That is, such behavioural equivalences are not robust, as first observed by Giacalone et al. [12].

The *probabilistic bisimilarity distances* that we study in this paper were first defined by Desharnais et al. in [11]. Each pair of states of a labelled Markov chain is assigned a distance, a real number in the unit interval $[0, 1]$. This distance captures the similarity of the behaviour of the states. The smaller the distance, the more alike the states behave. In particular, states have distance zero if and only if they are probabilistic bisimilar. This provides a quantitative generalization of probabilistic bisimilarity that is robust in that small changes in the transition probabilities give rise to small changes in the distances. For example, we can model a biased die by using a biased coin instead of a fair coin in the above example. Let us assume that the odds of heads of the biased coin, that is, going to the left, is $\frac{51}{100}$. A state labelled zero in the model of the fair die

has a *non-trivial* distance, that is, a distance greater than zero and smaller than one, to the corresponding state in the model of the biased die. For example, the initial states have distance about 0.036. We refer the reader to [7] for a more detailed discussion of a similar example.

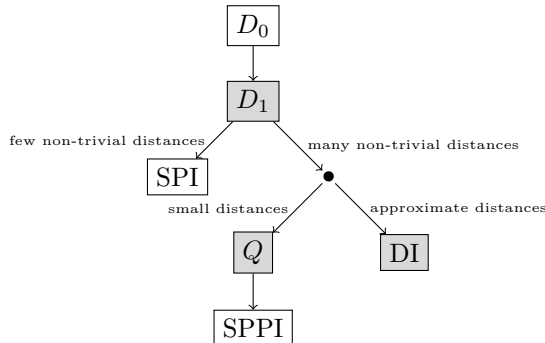
As we already mentioned earlier, behavioural equivalences can be used to verify that an implementation satisfies a specification. Similarly, the distances can be used to check how similar an implementation is to a specification. We also mentioned that probabilistic bisimilarity can be used to speed up model checking. The distances can be used in a similar way, by identifying those states that behave almost the same, that is, have a small distance (see [3, 23, 26]).

We focus in this paper on computing the probabilistic bisimilarity distances. In particular, we present a *decision procedure* for *distance one*. That is, we compute the set of pairs of states that have distance one. Recall that distance one is the maximal distance and, therefore, captures that states behave very differently. States with different labels have distance one. However, also states with the same label can have distance one, as the next example illustrates.



Instead of computing the set of state pairs that have distance one, we compute the complement, that is, the set of state pairs with distance smaller than one. Obviously, the set of state pairs with distance zero is included in this set. First, we decide distance zero. As we mentioned earlier, distance zero coincides with probabilistic bisimilarity. The first decision procedure for probabilistic bisimilarity was provided by Baier [4]. More efficient decision procedures were subsequently proposed by Derisavi et al. [10] and also by Valmari and Franceschinis [30]. The latter two both run in $O(m \log n)$, where n and m are the number of states and transitions of the labelled Markov chain. Subsequently, we use a traversal of a directed graph derived from the labelled Markov chain. This traversal takes $O(n^2 + m^2)$.

The decision procedures for distance zero and one can be used to compute or approximate probabilistic bisimilarity distances as indicated below.



Once we have computed the sets D_0 and D_1 of state pairs that have distance zero or one, we can easily compute the number of state pairs with non-trivial distances. If the number of non-trivial distances is small, then we can use the *simple policy iteration* (SPI) algorithm due to Bacci et al. [2] to compute those distances. Otherwise, we can either compute all distances smaller than a chosen $\varepsilon > 0$ or we can approximate the distances up to some chosen accuracy $\alpha > 0$. In the former case, we first compute a query set Q of state pairs that contains all state pairs the distances of which are at most ε . Subsequently, we apply the *simple partial policy iteration* (SPPI) algorithm due to Bacci et al. [2] to compute the distances for all state pairs in Q . In the latter case, we start with a pair of distance functions, one being a lower-bound and the other being an upper-bound of the probabilistic bisimilarity distances, and iteratively improve the accuracy of those until they are α close. We call this new approximation algorithm *distance iteration* (DI) as it is similar in spirit to Bellman's value iteration [5].

Chen et al. [8] presented an algorithm to compute the distances by means of Khachiyan's ellipsoid method [17]. Though the algorithm is polynomial time, in practice it is not as efficient as the policy iteration algorithms (see the examples in [28, Sect. 8]). The state of the art algorithm to compute the probabilistic bisimilarity distances consists of two components: D_0 and SPI. To compare this algorithm with our new algorithm consisting of the components D_0 , D_1 and SPI, we implemented all the components in Java and ran both implementations on several labelled Markov chains. These labelled Markov chains model randomized algorithms and probabilistic protocols that are part of the distribution of probabilistic model checkers such as PRISM [20]. Whereas the original state of the art algorithm can handle labelled Markov chains with up to 150 states, our new algorithm can handle more than 10,000 states. Furthermore, for one such labelled Markov chain with 150 states, the original algorithm takes more than 49 h, whereas our new algorithm takes only 13 ms. Also, the new algorithm consisting of the components D_0 , D_1 , Q and SPPI to compute only small distances along with the new algorithm consisting of the components D_0 , D_1 and DI to approximate the distances give rise to even less execution times for a number of the labelled Markov chains.

The main contributions of this paper are

- a polynomial decision procedure for distance one,
- an algorithm to compute the probabilistic bisimilarity distances,
- an algorithm to compute those probabilistic bisimilarity distances smaller than some given $\varepsilon > 0$, and
- an approximation algorithm to compute the probabilistic bisimilarity distances up to some given accuracy $\alpha > 0$.

Furthermore, by means of experiments we have shown that these three new algorithms are very effective, improving significantly on the state of the art.

2 Labelled Markov Chains and Probabilistic Bisimilarity Distances

We start by reviewing the model of interest, labelled Markov chains, its most well known behavioural equivalence, probabilistic bisimilarity due to Larsen and Skou [21], and the probabilistic bisimilarity pseudometric due to Desharnais et al. [11]. We denote the set of rational probability distributions on a set S by $\text{Distr}(S)$. For $\mu \in \text{Distr}(S)$, its support is defined by $\text{support}(\mu) = \{s \in S \mid \mu(s) > 0\}$. Instead of $S \times S$, we often write S^2 .

Definition 1. A labelled Markov chain is a tuple $\langle S, L, \tau, \ell \rangle$ consisting of

- a nonempty finite set S of states,
- a nonempty finite set L of labels,
- a transition function $\tau : S \rightarrow \text{Distr}(S)$, and
- a labelling function $\ell : S \rightarrow L$.

For the remainder of this section, we fix such a labelled Markov chain $\langle S, L, \tau, \ell \rangle$.

Definition 2. Let $\mu, \nu \in \text{Distr}(S)$. The set $\Omega(\mu, \nu)$ of couplings of μ and ν is defined by

$$\Omega(\mu, \nu) = \left\{ \omega \in \text{Distr}(S^2) \mid \begin{array}{l} \forall s \in S : \sum_{t \in S} \omega(s, t) = \mu(s) \wedge \\ \forall t \in S : \sum_{s \in S} \omega(s, t) = \nu(t) \end{array} \right\}.$$

Note that $\omega \in \Omega(\mu, \nu)$ is a joint probability distribution with marginals μ and ν . The following proposition will be used to prove Proposition 5.

Proposition 1. For all $\mu, \nu \in \text{Distr}(S)$ and $X \subseteq S^2$,

$$\forall \omega \in \Omega(\mu, \nu) : \text{support}(\omega) \subseteq X \text{ if and only if } \text{support}(\mu) \times \text{support}(\nu) \subseteq X.$$

Definition 3. An equivalence relation $R \subseteq S^2$ is a probabilistic bisimulation if for all $(s, t) \in R$, $\ell(s) = \ell(t)$ and there exists $\omega \in \Omega(\tau(s), \tau(t))$ such that $\text{support}(\omega) \subseteq R$. Probabilistic bisimilarity, denoted \sim , is the largest probabilistic bisimulation.

The probabilistic bisimilarity pseudometric of Desharnais et al. [11] maps each pair of states of a labelled Markov chain to a distance, an element of the unit interval $[0, 1]$. Hence, the pseudometric is a function from S^2 to $[0, 1]$, that is, an element of $[0, 1]^{S^2}$. As we will discuss below, it can be defined as a fixed point of the following function.

Definition 4. The function $\Delta : [0, 1]^{S^2} \rightarrow [0, 1]^{S^2}$ is defined by

$$\Delta(d)(s, t) = \begin{cases} 1 & \text{if } \ell(s) \neq \ell(t) \\ \min_{\omega \in \Omega(\tau(s), \tau(t))} \sum_{u, v \in S} \omega(u, v) d(u, v) & \text{otherwise} \end{cases}$$

Since a concave function on a convex polytope attains its minimum (see [18, p. 260]), the above minimum exists. We will use this fact in Proposition 4, one of the key technical results in this paper. We endow the set $[0, 1]^{S^2}$ of functions from S^2 to $[0, 1]$ with the following partial order: $d \sqsubseteq e$ if $d(s, t) \leq e(s, t)$ for all $s, t \in S$. The set $[0, 1]^{S^2}$ together with the order \sqsubseteq form a complete lattice (see [9, Chap. 2]). The function Δ is monotone (see [6, Sect. 3]). According to the Knaster-Tarski fixed point theorem [29, Theorem 1], a monotone function on a complete lattice has a least fixed point. Hence, Δ has a least fixed point, which we denote by $\mu(\Delta)$. This fixed point assigns to each pair of states their probabilistic bisimilarity distance.

Given that $\mu(\Delta)$ captures the probabilistic bisimilarity distances, we define the following sets.

$$\begin{aligned} D_0 &= \{(s, t) \in S^2 \mid \mu(\Delta)(s, t) = 0\} \\ D_1 &= \{(s, t) \in S^2 \mid \mu(\Delta)(s, t) = 1\} \end{aligned}$$

The probabilistic bisimilarity pseudometric $\mu(\Delta)$ provides a quantitative generalization of probabilistic bisimilarity as captured by the following result by Desharnais et al. [11, Theorem 1].

Theorem 1. $D_0 = \{(s, t) \in S^2 \mid s \sim t\}$.

3 Distance One

We concluded the previous section with the characterization of D_0 as the set of state pairs that are probabilistic bisimilar. In this section we present a characterization of D_1 as a fixed point of the function introduced in Definition 5.

Let us consider the case that the probabilistic bisimilarity distance of states s and t is one, that is, $\mu(\Delta)(s, t) = 1$. Then $\Delta(\mu(\Delta))(s, t) = 1$. From the definition of Δ , we can conclude that either $\ell(s) \neq \ell(t)$, or for all couplings $\omega \in \Omega(\tau(s), \tau(t))$ we have $\text{support}(\omega) \subseteq D_1$.

We partition the set S^2 of state pairs into

$$\begin{aligned} S_0^2 &= \{(s, t) \in S^2 \mid s \sim t\} \\ S_1^2 &= \{(s, t) \in S^2 \mid \ell(s) \neq \ell(t)\} \\ S_?^2 &= S^2 \setminus (S_0^2 \cup S_1^2) \end{aligned}$$

Hence, if $\mu(\Delta)(s, t) = 1$, then either $(s, t) \in S_1^2$, or $(s, t) \in S_?^2$ and for all couplings $\omega \in \Omega(\tau(s), \tau(t))$ we have $\text{support}(\omega) \subseteq D_1$. This leads us to the following function.

Definition 5. The function $\Gamma : 2^{S^2} \rightarrow 2^{S^2}$ is defined by

$$\Gamma(X) = S_1^2 \cup \{(s, t) \in S_?^2 \mid \forall \omega \in \Omega(\tau(s), \tau(t)) : \text{support}(\omega) \subseteq X\}.$$

Proposition 2. The function Γ is monotone.

Since the set 2^{S^2} of subsets of S^2 endowed with the order \subseteq is a complete lattice (see [9, Example 2.6(2)]) and the function Γ is monotone, we can conclude from the Knaster-Tarski fixed point theorem that Γ has a greatest fixed point, which we denote by $\nu(\Gamma)$. Next, we show that D_1 is a fixed point of Γ .

Proposition 3. $D_1 = \Gamma(D_1)$.

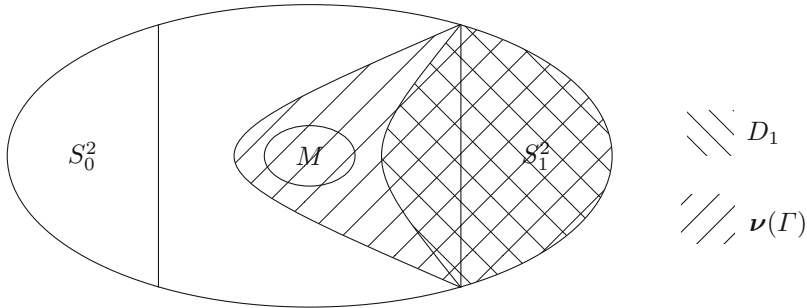
Since we have already seen that D_1 is a fixed point of Γ , we have that $D_1 \subseteq \nu(\Gamma)$. To conclude that D_1 is the greatest fixed point of Γ , it remains to show that $\nu(\Gamma) \subseteq D_1$, which is equivalent to the following.

Proposition 4. $\nu(\Gamma) \setminus D_1 = \emptyset$.

Proof. Towards a contradiction, assume that $\nu(\Gamma) \setminus D_1 \neq \emptyset$. Let

$$m = \min\{\mu(\Delta)(s, t) \mid (s, t) \in \nu(\Gamma) \setminus D_1\}$$

$$M = \{(s, t) \in \nu(\Gamma) \setminus D_1 \mid \mu(\Delta)(s, t) = m\}$$



Since $\nu(\Gamma) \setminus D_1 \neq \emptyset$, we have that $M \neq \emptyset$. Furthermore,

$$M \subseteq \nu(\Gamma) \setminus D_1. \quad (1)$$

Since $\nu(\Gamma) \setminus D_1 \subseteq \nu(\Gamma)$, we have

$$M \subseteq \nu(\Gamma) = \Gamma(\nu(\Gamma)) \subseteq S_1^2 \cup S_7^2. \quad (2)$$

For all $(s, t) \in M$,

$$\begin{aligned} (s, t) &\in \nu(\Gamma) \wedge (s, t) \notin D_1 \quad [(1)] \\ \Rightarrow (s, t) &\in \Gamma(\nu(\Gamma)) \wedge (s, t) \notin S_1^2 \\ \Rightarrow \forall \omega \in \Omega(\tau(s), \tau(t)) : \text{support}(\omega) &\subseteq \nu(\Gamma). \end{aligned} \quad (3)$$

For each $(s, t) \in M$, let

$$\omega_{s,t} = \underset{\omega \in \Omega(\tau(s), \tau(t))}{\operatorname{argmin}} \sum_{u,v \in S} \omega(u, v) \mu(\Delta)(u, v). \quad (4)$$

We distinguish the following two cases.

- Assume that there exists $(s, t) \in M$ such that $\text{support}(\omega_{s,t}) \cap D_1 \neq \emptyset$. Let

$$p = \sum_{(u,v) \in \nu(\Gamma) \cap D_1} \omega_{s,t}(u, v).$$

By (3), we have that $\text{support}(\omega_{s,t}) \subseteq \nu(\Gamma)$. Since $\text{support}(\omega_{s,t}) \cap D_1 \neq \emptyset$ by assumption, we can conclude that $p > 0$. Again using the fact that $\text{support}(\omega_{s,t}) \subseteq \nu(\Gamma)$, we have that

$$\sum_{(u,v) \in \nu(\Gamma) \setminus D_1} \omega_{s,t}(u, v) = 1 - p. \quad (5)$$

Furthermore,

$$\begin{aligned} m &= \mu(\Delta)(s, t) \\ &= \Delta(\mu(\Delta))(s, t) \\ &= \min_{\omega \in \Omega(\tau(s), \tau(t))} \sum_{u, v \in S} \omega(u, v) \mu(\Delta)(u, v) \\ &= \sum_{u, v \in S} \omega_{s,t}(u, v) \mu(\Delta)(u, v) \quad [(4)] \\ &= \sum_{(u,v) \in \nu(\Gamma)} \omega_{s,t}(u, v) \mu(\Delta)(u, v) \quad [(3)] \\ &= \sum_{(u,v) \in \nu(\Gamma) \cap D_1} \omega_{s,t}(u, v) \mu(\Delta)(u, v) + \sum_{(u,v) \in \nu(\Gamma) \setminus D_1} \omega_{s,t}(u, v) \mu(\Delta)(u, v) \\ &= p + \sum_{(u,v) \in \nu(\Gamma) \setminus D_1} \omega_{s,t}(u, v) \mu(\Delta)(u, v) \\ &\geq p + (1 - p)m. \end{aligned}$$

The last step follows from (5) and the fact that $\mu(\Delta)(u, v) \geq m$ for all $(u, v) \in \nu(\Gamma) \setminus D_1$. From the facts that $p > 0$ and $m \geq p + (1 - p)m$ we can conclude that $m \geq 1$. This contradicts (1).

- Otherwise, $\text{support}(\omega_{s,t}) \cap D_1 = \emptyset$ for all $(s, t) \in M$. Next, we will show that M is a probabilistic bisimulation under this assumption. From the fact that M is a probabilistic bisimulation, we can conclude from Theorem 1 that $\mu(\Delta)(s, t) = 0$ for all $(s, t) \in M$. Hence, since $M \neq \emptyset$ we have that $M \cap S_0^2 \neq \emptyset$ which contradicts (2).

Next, we prove that M is a probabilistic bisimulation. Let $(s, t) \in M$. Since $M \subseteq \nu(\Gamma) \setminus D_1$ by (1), we have that $(s, t) \notin D_1$ and, hence, $\Delta(\mu(\Delta))(s, t) = \mu(\Delta)(s, t) < 1$. From the definition of Δ , we can conclude that $\ell(s) = \ell(t)$. Since

$$\begin{aligned} m &= \mu(\Delta)(s, t) \\ &= \sum_{(u,v) \in \nu(\Gamma) \setminus D_1} \omega_{s,t}(u, v) \mu(\Delta)(u, v) \quad [\text{as above}] \end{aligned}$$

and $\mu(\Delta)(u, v) \geq m$ for all $(u, v) \in \nu(\Gamma) \setminus D_1$, we can conclude that $\mu(\Delta)(u, v) = m$ for all $(u, v) \in \text{support}(\omega_{s,t})$. Hence, $\text{support}(\omega_{s,t}) \subseteq M$. Therefore, M is a probabilistic bisimulation. \square

Theorem 2. $D_1 = \nu(\Gamma)$.

Proof. Immediate consequence of Proposition 3 and 4. \square

We have shown that D_1 can be characterized as the greatest fixed point of Γ . Next, we will show that D_1 can be decided in polynomial time.

Theorem 3. Distance one can be decided in $O(n^2 + m^2)$.

Proof. As we will show in Theorem 5, distance smaller than one can be decided in $O(n^2 + m^2)$. Hence, distance one can be decided in $O(n^2 + m^2)$ as well. \square

4 Distance Smaller Than One

To compute the set of state pairs which have distance one, we can first compute the set of state pairs which have distance less than one. The latter set we denote by $D_{<1}$. We can then obtain D_1 by taking the complement of $D_{<1}$. As we will discuss below, $D_{<1}$ can be characterized as the least fixed point of the following function.

Definition 6. The function $\Upsilon : 2^{S^2} \rightarrow 2^{S^2}$ is defined by

$$\Upsilon(X) = S^2 \setminus \Gamma(S^2 \setminus X).$$

The next theorem follows from Theorem 2.

Theorem 4. $D_{<1} = \mu(\Upsilon)$.

Next, we show that the computation of $D_{<1}$ can be formulated as a reachability problem on a directed graph which is induced by the labelled Markov chain. Thus, we can use standard search algorithms, for example, breadth-first search, on the induced graph.

Next, we present the graph induced by the labelled Markov chain.

Definition 7. The directed graph $G = (V, E)$ is defined by

$$\begin{aligned} V &= S_0^2 \cup S_1^2 \\ E &= \{ \langle (u, v), (s, t) \rangle \mid \tau(s)(u) > 0 \wedge \tau(t)(v) > 0 \} \end{aligned}$$

We are left to show that in the graph G defined above, a vertex (s, t) is reachable from some vertex in S_0^2 if and only if the state pair (s, t) in the labelled Markov chain has distance less than one.

As we have discussed earlier, if a state pair (s, t) has distance one, either s and t have different labels, or for all couplings $\omega \in \Omega(\tau(s), \tau(t))$ we have that $\text{support}(\omega) \subseteq D_1$. To avoid the universal quantification over couplings, we will use Proposition 1 in the proof of following proposition.

Proposition 5. $\mu(\top) = \{ (s, t) \mid (s, t) \text{ is reachable from some } (u, v) \in S_0^2 \}$.

Theorem 5. *Distance smaller than one can be decided in $O(n^2 + m^2)$.*

Proof. Distance smaller than one can be decided as follows.

1. Decide distance zero.
2. Breadth-first search of G , with the queue initially containing the pairs of states that have distance zero.

By Theorem 4 and Proposition 5, we have that s and t have distance smaller than one if and only if (s, t) is reachable in the directed graph G from some (u, v) such that u and v have distance zero. These reachable state pairs can be computed using breadth-first search, with the queue initially containing S_0^2 .

Distance zero, that is, probabilistic bisimilarity, can be decided in $O(m \log n)$ as shown by Derisavi et al. in [10]. The directed graph G has n^2 vertices and m^2 edges. Hence, breadth-first search takes $O(n^2 + m^2)$. \square

5 Number of Non-trivial Distances

As we have already discussed earlier, distance zero captures that states behave exactly the same, that is, they are probabilistic bisimilar, and distance one indicates that states behave very differently. The remaining distances, that is, those greater than zero and smaller than one, we call non-trivial. Being able to determine quickly the number of non-trivial distances of a labelled Markov chain allows us to decide whether computing all these non-trivial distances (using some policy iteration algorithm) is feasible.

To determine the number of non-trivial distances of a labelled Markov chain, we use the following algorithm.

1. Decide distance zero.
2. Decide distance one.

As first proved by Baier [4], distance zero, that is, probabilistic bisimilarity, can be decided in polynomial time. As we proved in Theorem 3, distance one can be decided in polynomial time as well. Hence, we can compute the number of non-trivial distances in polynomial time.

To decide distance zero, we implemented the algorithm to decide probabilistic bisimilarity due to Derisavi et al. [10] in Java. We also implemented our algorithm to decide distance one, described in the proof of Theorems 3 and 5.

We applied our implementation to labelled Markov chains that model randomized algorithms and probabilistic protocols. These labelled Markov chains have been obtained from the verification tool PRISM [20]. We compute the number of non-trivial distances for two models: the randomized self-stabilising algorithm due to Herman [14] and the bounded retransmission protocol by Helminck et al. [13].

For the randomized self-stabilising algorithm, the size of the labelled Markov chain grows exponentially in the numbers of processes, N . The results for the randomized self-stabilising algorithm are shown in the table below. As we can see from the table, for systems up to 128 states, the algorithm runs for less than a second. For the system with 512 states, the algorithm terminates within seven minutes. For the case $N = 3$, there are only 12 non-trivial distances. The size is so small that we can easily compute all the non-trivial distances. Section 6 will use the simple policy iteration algorithm as the next step to compute them. The same applies to the case $N = 5$. For $N = 7$ or 9, the number of non-trivial distances is around 11,000 and 200,000, respectively. This makes computing all of them infeasible. Thus, instead of computing all of them, we need to find alternative ways to handle systems with a large number of non-trivial distances. We will discuss two alternative ways in Sects. 7 and 8. Moreover, in this example, as $|D_1| = |S_1^2|$, we know that all the state pairs with distance one are those that have different labels.

| N | $ S $ | $D_0 + D_1$ | Non-trivial | $ D_0 $ | $ D_1 $ | $ S_1^2 $ |
|-----|-------|-------------|-------------|---------|---------|-----------|
| 3 | 8 | 1.00 ms | 12 | 38 | 14 | 14 |
| 5 | 32 | 6.06 ms | 280 | 304 | 440 | 440 |
| 7 | 128 | 0.77 s | 11,032 | 2,160 | 3,192 | 3,192 |
| 9 | 512 | 378.42 s | 230,712 | 13,648 | 17,784 | 17,784 |

In the bounded retransmission protocol, there are two parameters: N denotes the number of chunks and M the maximum allowed number of retransmissions of each chunk. The results are shown in the table below. The algorithm can handle systems up to 3,526 states within 11 min. In this example, there are no non-trivial distances. As a consequence, deciding distance zero and one suffices to compute all the distances in this case.

| N | M | S | $D_0 + D_1$ | $ D_0 $ | $ D_1 $ | $ S_1^2 $ |
|-----|-----|-------|-------------|------------|---------|-----------|
| 16 | 2 | 677 | 3.0 s | 456,977 | 1,352 | 1,352 |
| 16 | 3 | 886 | 8.6 s | 783,226 | 1,770 | 1,770 |
| 16 | 4 | 1,095 | 17.5 s | 1,196,837 | 2,188 | 2,188 |
| 16 | 5 | 1,304 | 22.8 s | 1,697,810 | 2,606 | 2,606 |
| 32 | 2 | 1,349 | 24.7 s | 1,817,105 | 2,696 | 2,696 |
| 32 | 3 | 1,766 | 69.7 s | 3,115,226 | 3,530 | 3,530 |
| 32 | 4 | 2,183 | 141.0 s | 4,761,125 | 4,364 | 4,364 |
| 32 | 5 | 2,600 | 208.6 s | 6,754,802 | 5,198 | 5,198 |
| 64 | 2 | 2,693 | 235.2 s | 7,246,865 | 5,384 | 5,384 |
| 64 | 3 | 3,526 | 616.4 s | 12,425,626 | 7,050 | 7,050 |

6 All Distances

To compute all distances of a labelled Markov chain, we augment the existing state of the art algorithm, which is based on algorithms due to Derisavi et al. [10] (step 1) and Bacci et al. [2] (step 3), by incorporating our decision procedure (step 2) as follows.

1. Decide distance zero.
2. Decide distance one.
3. Simple policy iteration.

Given that we not only decide distance zero, but also distance one, before running simple policy iteration, the correctness of the simple policy iteration algorithm in the augmented setting needs an adjusted proof.

As we already discussed in the previous section, step 1 and 2 are polynomial time. However, step 3 may take at least exponential time in the worst case, as we have shown in [27]. Hence, the overall algorithm is exponential time.

The first example we consider here is the synchronous leader election protocol of Itai and Rodeh [15] which is taken from PRISM. The protocol takes the number of processors, N , and a constant K as parameters. We compare the running time of our new algorithm with the state of the art algorithm, that combines algorithms due to Derisavi et al. and due to Bacci et al. The results are shown in the table below. In this protocol, the number of non-trivial distances is zero. Thus, our new algorithm terminates without running step 3 which is the simple policy iteration algorithm. On the other hand, the original simple policy iteration algorithm computes the distances of all the elements in the set $D_1 \setminus S_1^2$, the size of which is huge as can be seen from the last two columns of the table.

| N | K | $ S $ | $D_0 + \text{SPI}$ | $D_0 + D_1 + \text{SPI}$ | Speed-up | $ D_0 $ | $ D_1 $ | $ S_1^2 $ |
|-----|-----|--------|--------------------|--------------------------|------------|------------|-------------|-----------|
| 3 | 2 | 26 | 4 s | 1 ms | 4,281 | 122 | 554 | 50 |
| 3 | 4 | 147 | 49 h | 13 ms | 13,800,000 | 7,419 | 14,190 | 292 |
| 3 | 6 | 459 | - | 214 ms | - | 88,671 | 122,010 | 916 |
| 3 | 8 | 1,059 | - | 3 s | - | 508,851 | 612,630 | 2,116 |
| 4 | 2 | 61 | 812 s | 3 ms | 305,000 | 459 | 3,262 | 120 |
| 4 | 4 | 812 | - | 388 ms | - | 145,780 | 513,564 | 1,622 |
| 4 | 6 | 3,962 | - | 82 s | - | 4,350,292 | 11,347,152 | 7,922 |
| 4 | 8 | 12,400 | - | 2,971 s | - | 46,198,188 | 107,561,812 | 24,798 |
| 5 | 2 | 141 | - | 6 ms | - | 2,399 | 17,482 | 280 |
| 5 | 4 | 4,244 | - | 33 s | - | 3,318,662 | 14,692,874 | 8,486 |
| 6 | 2 | 335 | - | 25 ms | - | 14,327 | 97,898 | 668 |

The simple policy iteration algorithm can only handle a limited number of states. For the labelled Markov chain with 26 states ($N = 3$ and $K = 2$) the simple policy iteration algorithm takes four seconds, while our new algorithm

takes one millisecond. The speed-up is more than 4,000 times. For the labelled Markov chain with 61 states ($N = 4$ and $K = 2$), the simple policy iteration algorithm runs in 812 s, while our new algorithm takes three milliseconds. The speed-up of the new algorithm is 30,000 times. The biggest system the simple policy iteration algorithm can handle is the one with 147 states ($N = 3$ and $K = 4$) and it takes more than 49 h. In contrast, our new algorithm terminates within 13 ms. That makes the new algorithm seven orders of magnitude faster than the state of the art algorithm. This example also shows that the new algorithm can handle systems with at least 12,400 states.

In the second example, we model two dies, one using a fair coin and the other one using a biased coin. The goal is to compute the probabilistic bisimilarity distance between these two dies. An implementation of the die algorithm is part of PRISM. The resulting labelled Markov chain has 20 states.

As there are only 30 non-trivial distances, we run the simple policy iteration algorithm as step 3. The new algorithm is about 46 times faster than the original algorithm.

| $ S $ | $D_0 + \text{SPI}$ | $D_0 + D_1 + \text{SPI}$ | Speed-up | Non-trivial | $ D_0 $ | $ D_1 $ | $ S_1^2 $ |
|-------|--------------------|--------------------------|----------|-------------|---------|---------|-----------|
| 20 | 5.55 s | 0.12 s | 46.25 | 30 | 20 | 350 | 198 |

7 Small Distances

As we have discussed in Sect. 5, for systems of which the number of non-trivial distances is so large that computing all of them is infeasible, we have to find alternative ways. In practice, as we only identify the state pairs with small distances, we can cut down the number of non-trivial distances by only computing those with small distances.

To compute the non-trivial distances smaller than a positive number, ε , we use the following algorithm.

1. Decide distance zero.
2. Decide distance one.
3. Compute the query set

$$Q = \{ (s, t) \in S^2 \setminus (D_0 \cup D_1) \mid \Delta(d)(s, t) \leq \varepsilon \}$$

where

$$d(s, t) = \begin{cases} 1 & \text{if } (s, t) \in D_1 \\ 0 & \text{otherwise} \end{cases}$$

4. Simple partial policy iteration for Q .

The first two steps remain the same. In step 3, we compute a query set Q that contains all state pairs with distances no greater than ε , as shown in Proposition 6. In step 4, we use this set as the query set to run the simple partial policy iteration algorithm by Bacci et al. [2].

Proposition 6. *Let d be the distance function defined in step 3. For all $(s, t) \in S^2 \setminus (D_0 \cup D_1)$, if $\mu(\Delta)(s, t) \leq \varepsilon$, then $\Delta(d)(s, t) \leq \varepsilon$.*

Given that we not only decide distance zero, but also distance one, before running simple partial policy iteration, the correctness of the simple partial policy iteration algorithm in the augmented setting needs an adjusted proof.

As we have seen before, step 1 and 2 take polynomial time. In step 3, computing $\Delta(d)$ corresponds to solving a minimum cost network flow problem. Such a problem can be solved in polynomial time using, for example, Orlin’s network simplex algorithm [24]. As we have shown in [28], step 4 takes at least exponential time in the worst case. Therefore, the overall algorithm is exponential time.

We consider the randomized quicksort algorithm, an implementation of which is part of jpf-probabilistic [31]. The input of the algorithm is the list to be sorted. The list of size 6 gives rise to a labelled Markov chain with 82 states. We compare the running time of the new algorithm for small distances ($D_0 + D_1 + Q + \text{SPPI}$) to the original algorithm ($D_0 + \text{SPI}$) and the new algorithm presented in Sect. 6 ($D_0 + D_1 + \text{SPI}$). The original algorithm ($D_0 + \text{SPI}$) takes about 14 h, the new algorithm which incorporates the decision procedure of distance one takes less than 7 h. For $\varepsilon = 0.1$, the new algorithm for small distances takes 57 min. This makes it about 7 times faster than the algorithm presented in Sect. 6 and about 15 times faster than the original simple policy iteration algorithm. For $\varepsilon = 0.01$, the new algorithm for small distances takes even less time, namely 41 min. As can be seen in the table below, the total number of non-trivial distances is 2,300. The simple partial policy iteration algorithm starts with the query set Q but may have to compute the distances of other state pairs as well. The total number of state pairs considered by the simple partial policy iteration algorithm can be found in the column labelled Total.

| ε | $D_0 + D_1 + Q + \text{SPPI}$ | $ Q $ | Total | Non-trivial |
|---------------|-------------------------------|-------|-------|-------------|
| 0.1 | 57 min | 96 | 1,002 | 2,300 |
| 0.01 | 41 min | 84 | 842 | 2,300 |

8 Approximation Algorithm

We propose another solution to deal with a large number of non-trivial distances by approximating the distances rather than computing the exact values. To approximate the distances such that the approximate values differ from the exact ones by at most α , a positive number, we use the following algorithm.

1. Decide distance zero.
2. Decide distance one.

3. $l(s, t) = \begin{cases} 1 & \text{if } (s, t) \in D_1 \\ 0 & \text{otherwise} \end{cases}$
 $u(s, t) = \begin{cases} 0 & \text{if } (s, t) \in D_0 \\ 1 & \text{otherwise} \end{cases}$
 repeat
 for each $(s, t) \in S^2 \setminus (D_0 \cup D_1)$
 if $l(s, t) \neq u(s, t)$
 $l(s, t) = \Delta(l)(s, t)$
 $u(s, t) = \Delta(u)(s, t)$
 until $\|l - u\| \leq \alpha$

Again, the first two steps remain the same. Step 3 contains the new approximation algorithm called *distance iteration* (DI). In this step, we define two distance functions, a lower-bound l and an upper-bound u . We repeatedly apply Δ to these two functions until the difference of the non-trivial distances in these two functions is smaller than the threshold α . For each state pair we end up with an interval of at most size α in which their distance lies. To prove the algorithm correct, we modify the function Δ defining the probabilistic bisimilarity distances slightly as follows.

Definition 8. The function $\Delta_0 : [0, 1]^{S^2} \rightarrow [0, 1]^{S^2}$ is defined by

$$\Delta_0(d)(s, t) = \begin{cases} 0 & \text{if } (s, t) \in D_0 \\ \Delta(d)(s, t) & \text{otherwise} \end{cases}$$

Some properties of Δ_0 , which are key to the correctness proof of the above algorithm, are collected in the following theorem.

Theorem 6.

- (a) The function Δ_0 is monotone.
- (b) The function Δ_0 is nonexpansive.
- (c) $\mu(\Delta_0) = \mu(\Delta)$.
- (d) $\mu(\Delta_0) = \nu(\Delta_0)$.
- (e) $\mu(\Delta_0) = \sup_{m \in \mathbb{N}} \Delta_0^m(d_0)$, where $d_0(s, t) = 0$ for all $s, t \in S$.
- (f) $\nu(\Delta_0) = \inf_{n \in \mathbb{N}} \Delta_0^n(d_1)$, where $d_1(s, t) = 1$ for all $s, t \in S$.

Let us use randomized quicksort introduced in Sect. 7 and the randomized self-stabilising algorithm due to Herman [14] introduced in Sect. 5 as examples. Recall that for the randomized self-stabilising algorithm, when $N = 7$, the number of non-trivial distances is 11,032, which we are not able to handle using the simple policy iteration algorithm. We apply the approximation algorithm to this model and the randomized quicksort example with 82 states and present the results below. The accuracy α is set to be 0.01.

The approximation algorithm for randomized quicksort runs for about 14 min, which is about 3 to 4 times faster than the algorithm for small distances in Sect. 7. For the randomized self-stabilising algorithm with 128 states, the approximation algorithm terminates in about 54 h. Although the number of non-trivial

distances for the randomized self-stabilising algorithm is about 5 times of that of the randomized quicksort, the running time is more than 200 times slower. It is unknown whether this approximation algorithm has exponential running time.

| Model | $ S $ | Non-trivial | $D_0 + D_1 + DI$ |
|---------------------------------------|-------|-------------|------------------|
| Randomized quicksort | 82 | 2,300 | 14 min |
| Randomized self-stabilising algorithm | 128 | 11,032 | 54 h |

9 Conclusion

In this paper, we have presented a decision procedure for probabilistic bisimilarity distance one. This decision procedure provides the basis for three new algorithms to compute and approximate the probabilistic bisimilarity distances of a labelled Markov chain. The first algorithm decides distance zero, then decides distance one, and finally uses simple policy iteration to compute the remaining distances. As shown experimentally, this new algorithm significantly improves the state of the art algorithm that only decides distance zero and then uses simple policy iteration. The second algorithm computes all probabilistic bisimilarity distances that are smaller than some given upper bound, by deciding distance zero, deciding distance one, computing a query set, and running simple partial policy iteration for that query set. This second algorithm can handle labelled Markov chains that have considerably more non-trivial distances than our first algorithm. The third algorithm approximates the probabilistic bisimilarity distances up to a given accuracy, deciding distance zero, deciding distance one and running distance iteration. Also this third algorithm can handle labelled Markov chains that have considerably more non-trivial distances than our first algorithm. Whereas we know that the first two algorithms take at least exponential time in the worst case, the analysis of the running time of the third algorithm has not yet been determined. Moreover, if we are only interested in the probabilistic bisimilarity distances for a few state pairs, with pre-computation of distance zero and one we can exclude the state pairs with trivial distances. We can add the remaining state pairs to a query set and run simple partial policy iteration to get the distances. Alternatively, we can modify the distance iteration algorithm to approximate the distances for the predefined state pairs. The details of these new algorithms will be studied in the future.

Acknowledgements. The authors would like to thank Daniela Petrisan, Eric Rupert and Dana Scott for discussions related to this research. The authors are also grateful to the referees for their constructive feedback.

References

1. Aceto, L., Ingolfssdottir, A., Larsen, K., Srba, J.: *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, Cambridge (2003)
2. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: On-the-fly exact computation of bisimilarity distances. In: Piterman, N., Smolka, S.A. (eds.) *TACAS 2013*. LNCS, vol. 7795, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_1
3. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: On the metric-based approximate minimization of Markov chains. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming*, Warsaw, Poland, July 2017. *Leibniz International Proceedings in Informatics*, vol. 80, pp. 104:1–104:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
4. Baier, C.: Polynomial time algorithms for testing probabilistic bisimulation and simulation. In: Alur, R., Henzinger, T.A. (eds.) *CAV 1996*. LNCS, vol. 1102, pp. 50–61. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_57
5. Bellman, R.: A Markovian decision process. *J. Math. Mech.* **6**(5), 679–684 (1957)
6. van Breugel, F.: On behavioural pseudometrics and closure ordinals. *Inf. Process. Lett.* **112**(18), 715–718 (2012)
7. van Breugel, F.: Probabilistic bisimilarity distances. *ACM SIGLOG News* **4**(4), 33–51 (2017)
8. Chen, D., van Breugel, F., Worrell, J.: On the complexity of computing probabilistic bisimilarity. In: Birkedal, L. (ed.) *FoSSaCS 2012*. LNCS, vol. 7213, pp. 437–451. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28729-9_29
9. Davey, B., Priestley, H.: *Introduction to Lattices and Order*. Cambridge University Press, Cambridge (2002)
10. Derisavi, S., Hermanns, H., Sanders, W.: Optimal state-space lumping in Markov chains. In: *Process. Lett.* **87**(6), 309–315 (2003)
11. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labeled Markov systems. In: Baeten, J.C.M., Mauw, S. (eds.) *CONCUR 1999*. LNCS, vol. 1664, pp. 258–273. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_19
12. Giacalone, A., Jou, C.-C., Smolka, S.: Algebraic reasoning for probabilistic concurrent systems. In: *Proceedings of the IFIP WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, April 1990, pp. 443–458. North-Holland (1990)
13. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Barendregt, H., Nipkow, T. (eds.) *TYPES 1993*. LNCS, vol. 806, pp. 127–165. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58085-9_75
14. Herman, T.: Probabilistic self-stabilization. *Inf. Process. Lett.* **35**(2), 63–67 (1990)
15. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. *Inf. Comput.* **88**(1), 60–87 (1990)
16. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_9
17. Khachiyan, L.: A polynomial algorithm in linear programming. *Sov. Math. Dokl.* **20**(1), 191–194 (1979)

18. Klee, V., Witzgall, C.: Facets and vertices of transportation polytopes. In: Dantzig, G., Veinott, A. (eds.) *Proceedings of 5th Summer Seminar on the Mathematics of the Decision Sciences*, Stanford, CA, USA, July/August 1967. *Lectures in Applied Mathematics*, vol. 11, pp. 257–282. AMS (1967)
19. Knuth, D., Yao, A.: The complexity of nonuniform random number generation. In: Traub, J. (ed.) *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, Pittsburgh, PA, USA, April 1976, pp. 375–428. Academic Press (1976)
20. 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). https://doi.org/10.1007/978-3-642-22110-1_47
21. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. In: *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, USA, January 1989, pp. 344–352. ACM (1989)
22. Milner, R. (ed.): *A Calculus of Communicating Systems*. LNCS, vol. 92. Springer, Heidelberg (1980). <https://doi.org/10.1007/3-540-10235-3>
23. Murthy, A., et al.: Approximate bisimulations for sodium channel dynamics. In: Gilbert, D., Heiner, M. (eds.) *CMSB 2012*. LNCS, pp. 267–287. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33636-2_16
24. Orlin, J.: A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Program.* **78**(2), 109–129 (1997)
25. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) *GI-TCS 1981*. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). <https://doi.org/10.1007/BFb0017309>
26. Sen, P., Deshpande, A., Getoor, L.: Bisimulation-based approximate lifted inference. In: Bilmes, J., Ng, A. (eds.) *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, Montreal, QC, Canada, pp. 496–505. AUAI Press (2009)
27. Tang, Q., van Breugel, F.: Computing probabilistic bisimilarity distances via policy iteration. In: Desharnais, J., Jagadeesan, R. (eds.) *Proceedings of the 27th International Conference on Concurrency Theory*, Quebec City, QC, Canada, August 2016. *Leibniz International Proceedings in Informatics*, vol. 59, pp. 22:1–22:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
28. Tang, Q., van Breugel, F.: Algorithms to compute probabilistic bisimilarity distances for labelled Markov chains. In: Meyer, R., Nestmann, U. (eds.) *Proceedings of the 28th International Conference on Concurrency Theory*, Berlin, Germany, September 2017. *Leibniz International Proceedings in Informatics*, vol. 85, pp. 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
29. Tarski, A.: A lattice-theoretic fixed point theorem and its applications. *Pac. J. Math.* **5**(2), 285–309 (1955)
30. Valmari, A., Franceschinis, G.: Simple $O(m \log n)$ time Markov chain lumping. In: Esparza, J., Majumdar, R. (eds.) *TACAS 2010*. LNCS, vol. 6015, pp. 38–52. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_4
31. Zhang, X., van Breugel, F.: Model checking randomized algorithms with Java PathFinder. In: *Proceedings of the 7th International Conference on the Quantitative Evaluation of Systems*, Williamsburg, VA, USA, September 2010, pp. 157–158. IEEE (2010)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

- Abate, Alessandro [I-270](#)
Akshay, S. [I-251](#)
Albarghouthi, Aws [I-327](#)
Albert, Elvira [II-392](#)
Anderson, Greg [I-407](#)
Argyros, George [I-427](#)
Arndt, Hannah [II-3](#)
- Backes, John [II-20](#)
Bansal, Suguman [I-367](#), [II-99](#)
Bardin, Sébastien [II-294](#)
Barrett, Clark [II-236](#)
Bartocci, Ezio [I-449](#), [I-547](#)
Bauer, Matthew S. [II-117](#)
Becchi, Anna [I-230](#)
Berzish, Murphy [II-45](#)
Biere, Armin [I-587](#)
Bloem, Roderick [I-547](#)
Blondin, Michael [I-604](#)
Blotsky, Dmitry [II-45](#)
Bonichon, Richard [II-294](#)
Bønneland, Frederik M. [I-527](#)
Bouajjani, Ahmed [II-336](#), [II-372](#)
Büning, Julian [II-447](#)
- Češka, Milan [I-612](#)
Chadha, Rohit [II-117](#)
Chakraborty, Supratik [I-251](#)
Chatterjee, Krishnendu [II-178](#)
Chaudhuri, Swarat [II-99](#)
Chen, Taolue [II-487](#)
Cheval, Vincent [II-28](#)
Chudnov, Andrey [II-430](#)
Collins, Nathan [II-413](#), [II-430](#)
Cook, Byron [I-38](#), [II-430](#), [II-467](#)
Cordeiro, Lucas [I-183](#)
Coti, Camille [II-354](#)
Cousot, Patrick [II-75](#)
- D’Antoni, Loris [I-386](#), [I-427](#)
David, Cristina [I-270](#)
Dillig, Isil [I-407](#)
Dodds, Joey [II-430](#)
Dohrau, Jérôme [II-55](#)
- Dreossi, Tommaso [I-3](#)
Dureja, Rohit [II-37](#)
- Eilers, Marco [I-596](#), [II-12](#)
Emmi, Michael [I-487](#)
Enea, Constantin [I-487](#), [II-336](#), [II-372](#)
Esparza, Javier [I-604](#)
- Fan, Chuchu [I-347](#)
Farinier, Benjamin [II-294](#)
Fedyukovich, Grigory [I-124](#), [I-164](#)
Feng, Yijun [I-507](#)
Finkbeiner, Bernd [I-144](#), [I-289](#)
Frehse, Goran [I-468](#)
Fremont, Daniel J. [I-307](#)
- Gacek, Andrew [II-20](#)
Ganesh, Vijay [II-45](#), [II-275](#)
Gao, Pengfei [II-157](#)
Gao, Sicun [II-219](#)
Ghassabani, Elaheh [II-20](#)
Giacobazzi, Roberto [II-75](#)
Giacobbe, Mirco [I-468](#)
Goel, Shubham [I-251](#)
Gómez-Zamalloa, Miguel [II-392](#)
Goubault, Eric [II-523](#)
Grishchenko, Ilya [I-51](#)
Gu, Ronghui [II-317](#)
Gupta, Aarti [I-124](#), [I-164](#), [II-136](#)
- Hahn, Christopher [I-144](#), [I-289](#)
Hassan, Mostafa [II-12](#)
He, Jinlong [II-487](#)
Henzinger, Monika [II-178](#)
Henzinger, Thomas A. [I-449](#), [I-468](#)
Hsu, Justin [I-327](#)
Hu, Qinheping [I-386](#)
Huffman, Brian [II-430](#)
- Isabel, Miguel [II-392](#)
- Jaax, Stefan [I-604](#)
Jansen, Christina [II-3](#)
Jensen, Peter Gjøøl [I-527](#)

Jha, Somesh I-3
 Ji, Kailiang II-372

Kabir, Ifaz II-45
 Katoen, Joost-Pieter I-507, I-643, II-3
 Kelmendi, Edon I-623
 Kesseli, Pascal I-183, I-270
 Khazem, Kareem II-467
 Kolokolova, Antonina II-275
 Kong, Hui I-449
 Kong, Soonho II-219
 Kragl, Bernhard I-79
 Krämer, Julia I-623
 Kremer, Steve II-28
 Křetínský, Jan I-567, I-623
 Kroening, Daniel I-183, I-270, II-467
 Kulal, Sumith I-251

Larsen, Kim Guldstrand I-527
 Li, Haokun I-507
 Li, Jianwen II-37
 Li, Wenchao I-662
 Loitzenbauer, Veronika II-178
 Lukert, Philip I-289
 Luttenberger, Michael I-578

MacCárthaigh, Colm II-430
 Maffei, Matteo I-51
 Magill, Stephen II-430
 Malik, Sharad II-136
 Matheja, Christoph II-3
 Mathur, Umang I-347
 Matyáš, Jiří I-612
 McMillan, Kenneth L. I-191, I-407
 Meggendorfer, Tobias I-567
 Mertens, Eric II-430
 Meyer, Philipp J. I-578
 Mitra, Sayan I-347
 Mora, Federico II-45
 Mrazek, Vojtech I-612
 Mullen, Eric II-430
 Müller, Peter I-596, II-12, II-55
 Münger, Severin II-55
 Muñoz, Marco I-527
 Mutluergil, Suha Orhun II-336

Namjoshi, Kedar S. I-367
 Nguyen, Huyen T. T. II-354
 Nickovic, Dejan I-547

Niemetz, Aina I-587, II-236
 Noll, Thomas II-3, II-447

Oraee, Simin II-178

Petrucchi, Laure II-354
 Pick, Lauren I-164
 Pike, Lee II-413
 Polgreen, Elizabeth I-270
 Potet, Marie-Laure II-294
 Prasad Sistla, A. II-117
 Preiner, Mathias I-587, II-236
 Pu, Geguang II-37
 Püschel, Markus I-211
 Putot, Sylvie II-523

Qadeer, Shaz I-79, II-372
 Quatmann, Tim I-643

Rabe, Markus N. II-256
 Rakotonirina, Itsaka II-28
 Ranzato, Francesco II-75
 Rasmussen, Cameron II-256
 Reynolds, Andrew II-236
 Robere, Robert II-275
 Rodríguez, César II-354
 Roeck, Franz I-547
 Rozier, Kristin Yvonne II-37
 Rubio, Albert II-392

Sa'ar, Yaniv I-367
 Sahlmann, Lorenz II-523
 Satake, Yuki I-105
 Schemmel, Daniel II-447
 Schneidewind, Clara I-51
 Schrammel, Peter I-183
 Sekanina, Lukas I-612
 Seshia, Sanjit A. I-3, I-307, II-256
 Shah, Shetal I-251
 Sickert, Salomon I-567, I-578
 Singh, Gagandeep I-211
 Solar-Lezama, Armando II-219
 Song, Fu II-157, II-487
 Soria Dustmann, Oscar II-447
 Sousa, Marcelo II-354
 Srba, Jiří I-527
 Stenger, Marvin I-289
 Subramanyan, Pramod II-136
 Summers, Alexander J. II-55

Tang, Qiyi [I-681](#)
Tasiran, Serdar [II-336](#), [II-430](#), [II-467](#)
Tautschnig, Michael [II-467](#)
Tentrup, Leander [I-289](#), [II-256](#)
Tinelli, Cesare [II-236](#)
Toman, Viktor [II-178](#)
Tomb, Aaron [II-413](#), [II-430](#)
Torfah, Hazem [I-144](#)
Trtik, Marek [I-183](#)
Tullsen, Mark [II-413](#)
Tuttle, Mark R. [II-467](#)

Unno, Hiroshi [I-105](#)
Urban, Caterina [II-12](#), [II-55](#)

van Breugel, Franck [I-681](#)
van Dijk, Tom [II-198](#)
Vardi, Moshe Y. [II-37](#), [II-99](#)
Vasicek, Zdenek [I-612](#)
Vechev, Martin [I-211](#)
Viswanathan, Mahesh [I-347](#), [II-117](#)
Vizel, Yakir [II-136](#)
Vojnar, Tomáš [I-612](#)

Wagner, Lucas [II-20](#)
Walther, Christoph [II-505](#)

Wang, Chao [II-157](#)
Wang, Guozhen [II-487](#)
Wang, Xinyu [I-407](#)
Wehrle, Klaus [II-447](#)
Weininger, Maximilian [I-623](#)
Westbrook, Eddy [II-430](#)
Whalen, Mike [II-20](#)
Wolf, Clifford [I-587](#)
Wu, Zhilin [II-487](#)

Xia, Bican [I-507](#)

Yahav, Eran [I-27](#)
Yan, Jun [II-487](#)
Yang, Junfeng [II-317](#)
Yang, Weikun [II-136](#)
Yuan, Xinhao [II-317](#)

Zaffanella, Enea [I-230](#)
Zhan, Naijun [I-507](#)
Zhang, Jun [II-157](#)
Zhang, Yueling [I-124](#)
Zheng, Yunhui [II-45](#)
Zhou, Weichao [I-662](#)
Ziegler, Christopher [I-567](#)