Marcin Jurdziński
Dejan Ničković (Eds.)

LNCS 7595

Formal Modeling and Analysis of Timed Systems

10th International Conference, FORMATS 2012
London, UK, September 2012
Proceedings

Springer

Marcin Jurdziński
Dejan Ničković (Eds.)

Formal Modeling and Analysis of Timed Systems

10th International Conference, FORMATS 2012
London, UK, September 18-20, 2012
Proceedings

Springer

Volume Editors

Marcin Jurdziński
University of Warwick
Department of Computer Science
Coventry, CV4 7AL, UK
E-mail: marcin.jurdzinski@dcs.warwick.ac.uk

Dejan Ničković
AIT Austrian Institute of Technology
Business Unit Safe and Autonomous Systems
Department of Safety and Security
Donau-City-Str. 1
1220 Vienna, Austria
E-mail: dejan.nickovic@ait.ac.at

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-33364-4 e-ISBN 978-3-642-33365-1
DOI 10.1007/978-3-642-33365-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012946474

CR Subject Classification (1998): F.3, D.2, D.3, D.2.4, F.4.1, C.3

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

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at the 10th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2012), held on September 18–20, 2012 in London, UK.

Timing aspects of systems from a variety of computer science domains have been treated independently by different communities. Researchers interested in semantics, verification, and performance analysis study models such as timed automata and timed Petri nets, the digital design community focuses on propagation and switching delays, while designers of embedded controllers have to take account of the time taken by controllers to compute their responses after sampling the environment.

Timing-related questions in these separate disciplines do have their particularities. However, there is a growing awareness that there are basic problems that are common to all of them. In particular, all these sub-disciplines treat systems whose behavior depends upon combinations of logical and temporal constraints; namely, constraints on the temporal distances between occurrences of events.

The aim of FORMATS is to promote the study of fundamental and practical aspects of timed systems, and to bring together researchers from different disciplines that share interests in modeling and analysis of timed systems. Typical topics include (but are not limited to):

- Foundations and Semantics: theoretical foundations of timed systems and languages; comparison between different models (timed automata, timed Petri nets, hybrid automata, timed process algebra, max-plus algebra, probabilistic models).
- Methods and Tools: techniques, algorithms, data structures, and software tools for analyzing timed systems and resolving temporal constraints (scheduling, worst-case execution time analysis, optimization, model-checking, testing, constraint solving, etc.).
- Applications: adaptation and specialization of timing technology in application domains in which timing plays an important role (real-time software, hardware circuits, and problems of scheduling in manufacturing and telecommunications).

This year FORMATS received 34 submissions. Each submission was reviewed by at least 3, and on average 4, Program Committee members. The committee decided to accept 16 papers for publication and presentation at the conference. The program also included three invited talks:

- Moshe Vardi, Rice University, USA: *Compositional Temporal Synthesis*;
- Twan Basten, Eindhoven University of Technology, the Netherlands: *Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems*;
- Kim G. Larsen, Aalborg University, Denmark: *Statistical Model Checking, Refinement Checking, Optimization, ...for Stochastic Hybrid Systems.*

For the third time, FORMATS was co-located with the International Conference on Quantitative Evaluation of SysTems (QEST), and the two conferences shared an invited speaker and social events. The two conferences focus on complementary and tightly related themes. While FORMATS puts emphasis on fundamental and practical aspects of timed systems, QEST focuses on evaluation and verification of computer systems and networks, through stochastic models and measurements. We would like to thank QEST organizers, in particular William Knottenbelt, Anton Stefanek, Giuliano Casale, Lucy Cherkasova, and Holger Hermanns for their pleasant cooperation.

We would like to thank all the authors for submitting to FORMATS. We wish to thank the invited speakers for accepting our invitation and providing extended abstracts for the conference proceedings. We are particularly grateful to the Program Committee members and the other reviewers for their competent and timely reviews of the submissions and the subsequent discussions, which were greatly appreciated. Throughout the entire process of organizing the conference and preparing this volume, we used the EasyChair conference management system, which provided excellent support. Finally, we gratefully acknowledge the financial support provided by the Austrian Institute of Technology (AIT), Vienna, Austria, and by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, UK.

July 2012

Marcin Jurdziński
Dejan Ničković

Organization

Program Committee

Parosh Aziz Abdulla	Uppsala University
Marius Bozga	Verimag/CNRS
Thomas Brihaye	Université de Mons
Franck Cassez	National ICT Australia and CNRS
Krishnendu Chatterjee	IST Austria
Georgios Fainekos	Arizona State University
Holger Hermanns	Saarland University
Marcin Jurdziński	University of Warwick
Christoph Kirsch	University of Salzburg
Kai Lampka	Uppsala University
Kim G. Larsen	Aalborg University
Insup Lee	University of Pennsylvania
Axel Legay	IRISA/INRIA
Nicolas Markey	LSV, CNRS and ENS Cachan
Dejan Ničković	Austrian Institute of Technology
Nir Piterman	University of Leicester
Jean-François Raskin	Université Libre de Bruxelles
Olivier Roux	IRCCyN, École Centrale de Nantes
Jeremy Sproston	Università degli Studi di Torino
Jiří Srba	Aalborg University
P.S. Thiagarajan	National University of Singapore
Stavros Tripakis	University of California, Berkeley
Frits Vaandrager	Radboud University Nijmegen
James Worrell	Oxford University

Additional Reviewers

Abbas, Houssam
Abraham, Erika
André, Étienne
Bechennec, Jean-Luc
Bernardo, Marco
Berthomieu, Bernard
Bertrand, Nathalie
Bortolussi, Luca
Boucheneb, Hanifa
Bouyer-Decitre, Patricia
Boyer, Benoît
Braberman, Victor
Bruyère, Véronique
Bulychev, Peter
Chen, Sanjian
Chen, Taolue
Cimatti, Alessandro
David, Alexandre
Delahaye, Benoit
Doyen, Laurent
Ekberg, Pontus
Fahrenberg, Uli

Ferrer Fioriti, Luis María
Forejt, Vojtech
Geeraerts, Gilles
Giannopoulou, Georgia
Haas, Andreas
Haddad, Serge
Herbreteau, Frédéric
Hohm, Tim
Hoxha, Bardh
King, Andrew
Křetínský, Jan
Laroussinie, François
Lime, Didier
Lipari, Giuseppe
Lippautz, Michael
Møller, Mikael H.
Norman, Gethin
Nyman, Ulrik
Olesen, Mads Chr.
Pajic, Miroslav
Phan, Linh Thi Xuan
Poulsen, Danny Bøgsted
Prabhu, Vinayak
Ratschan, Stefan
Reynier, Pierre-Alain
Roederer, Alex
Ruemmer, Philipp
Saivasan, Prakash
Sangnier, Arnaud
Sankaranarayanan, Sriram
Sassolas, Mathieu
Steffen, Martin
Stigge, Martin
Traonouez, Louis-Marie
Wolf, Verena
Yang, Shaofa

Table of Contents

Invited Talks

Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems (Extended Abstract) 1
Twan Basten, Martijn Hendriks, Lou Somers, and Nikola Trčka

Statistical Model Checking, Refinement Checking, Optimization, ... for Stochastic Hybrid Systems 7
Kim G. Larsen

Regular Papers

Robustness of Time Petri Nets under Architectural Constraints 11
S. Akshay, Loïc Hélouët, Claude Jard, Didier Lime, and Olivier H. Roux

Toward a Timed Theory of Channel Coding 27
Eugene Asarin, Nicolas Basset, Marie-Pierre Béal, Aldric Degorre, and Dominique Perrin

Playing Optimally on Timed Automata with Random Delays 43
Nathalie Bertrand and Sven Schewe

Dynamically-Driven Timed Automaton Abstractions for Proving Liveness of Continuous Systems 59
Rebekah Carter and Eva M. Navarro-López

Revisiting Timed Specification Theories: A Linear-Time Perspective 75
Chris Chilton, Marta Kwiatkowska, and Xu Wang

Multi-core Reachability for Timed Automata 91
Andreas E. Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen, and Jaco van de Pol

Counterexample-Guided Synthesis of Observation Predicates 107
Rayna Dimitrova and Bernd Finkbeiner

Confidence Bounds for Statistical Model Checking of Probabilistic Hybrid Systems 123
Christian Ellen, Sebastian Gerwinn, and Martin Fränzle

Region-Based Analysis of Hybrid Petri Nets with a Single General One-Shot Transition 139
Hamed Ghasemieh, Anne Remke, Boudewijn Haverkort, and Marco Gribaudo

Reducing Quasi-Equal Clocks in Networks of Timed Automata 155
Christian Herrera, Bernd Westphal, Sergio Feo-Arenis, Marco Muñiz, and Andreas Podelski

SMT-Based Induction Methods for Timed Systems 171
Roland Kindermann, Tommi Junttila, and Ilkka Niemelä

Timed Automata with Disjoint Activity 188
Marco Muñiz, Bernd Westphal, and Andreas Podelski

The Complexity of Bounded Synthesis for Timed Control with Partial Observability 204
Hans-Jörg Peter and Bernd Finkbeiner

Static Detection of Zeno Runs in UPPAAL Networks Based on Synchronization Matrices and Two Data-Variable Heuristics 220
Jonas Rinast and Sibylle Schupp

Frequencies in Forgetful Timed Automata 236
Amélie Stainer

Mcta: Heuristics and Search for Timed Systems 252
Martin Wehrle and Sebastian Kupferschmid

Author Index 267

Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems[⋆]
(Extended Abstract)

Twan Basten[1,2], Martijn Hendriks[1], Lou Somers[2,3], and Nikola Trčka[4]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
[3] Océ Technologies B.V., Venlo, The Netherlands
[4] United Technologies Research Center, East Hartford, CT, USA
http://dse.esi.nl

Abstract. Software plays an increasingly important role in modern embedded systems, leading to a rapid increase in design complexity. Model-driven exploration of design alternatives leads to shorter, more predictable development times and better controlled product quality.

1 Motivation

Industries in the high-tech embedded domain (including for example professional printing, lithographic systems, medical imaging, automotive, etc.) are facing the challenge of rapidly increasing complexity of next generations of their systems. Ever more functionality is being added, user expectations regarding quality and reliability increase, an ever tighter integration between the physical processes being controlled and the embedded hardware and software is needed, and technological developments push towards networked, multi-processor and multi-core platforms. The added complexity materializes in the software and hardware embedded at the core of the systems. Important decisions need to be made early in the development trajectory. Which functionality should be realized in software and which in hardware? What is the number and type of processors to be integrated? How should storage (both working memory and persistent disk storage) be organized? Is dedicated hardware development beneficial? How to distribute functionality? How to parallelize software? How can we meet timing, reliability and robustness requirements? The decisions should take into account the application requirements, cost and time-to-market constraints, as well as aspects like the need to re-use earlier designs or to integrate third-party components.

⋆ This work was carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program. This extended abstract is based on earlier papers describing the Octopus tool set [2] and our philosophy behind model-driven design-space exploration [7]; the full version of this extended abstract will appear as [3]. Nikola Trčka was employed by Eindhoven University of Technology when this work was performed.

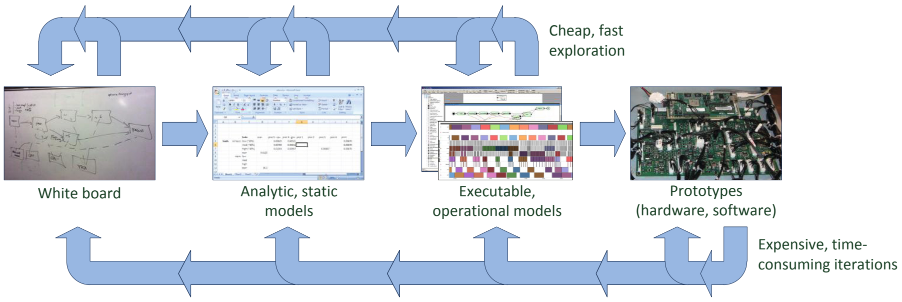

Cheap, fast exploration
White board
Analytic, static models
Executable, operational models
Prototypes (hardware, software)
Expensive, time-consuming iterations

Fig. 1. Embedded-systems development is an iterative process typically involving several costly iterations over physical prototypes. Model-driven DSE avoids these iterations through fast and efficient exploration using models, improving time-to-market and leading to better controlled product quality.

Industries typically adopt some form of model-based design for the software and hardware embedded in their systems. Fig. 1 illustrates such a process. Spreadsheets play an important role in early decision making about design alternatives. They provide a quick and easy method to explore alternatives at a high abstraction level. Executable operational models may then be developed for further exploration and/or synthesizing hardware and software. Current industrial practice uses such models mostly for the latter purpose, focussing on functional and structural aspects and not on extra-functional aspects such as timing and resource usage. Promising alternatives are realized in prototypes that include large parts of the software and hardware that are ultimately also used in the final system. Parts of the process may be iterated several times.

Design iterations through prototypes are time consuming and costly. Only a few alternatives can be explored. The number of design alternatives is extremely large though. The challenge is to more effectively exploit models for design-space exploration (DSE), avoiding design iterations over prototypes and extensive performance tuning and re-engineering at the implementation level. Spreadsheet analysis is suitable for a coarse pruning of options. However, it not well suited to capture system dynamics due to for example pipelined, parallel processing, data-dependent workload variations, resource scheduling and arbitration, variations in data granularity, etc. (see Fig. 2). High-level operational models can capture such dynamics. However, in industry, such models are not yet extensively used for DSE purposes. An important challenge is therefore to bridge the gap between spreadsheet type analysis and prototypes for DSE in industrial development practice. It is crucial to find the right abstractions, methods, and analysis techniques to support accurate and extensive DSE.

2 Model-Driven Design-Space Exploration

An important characteristic of DSE is that many different questions may need to be answered, related to system architecture and dimensioning, resource cost and performance of various design alternatives, identification of performance

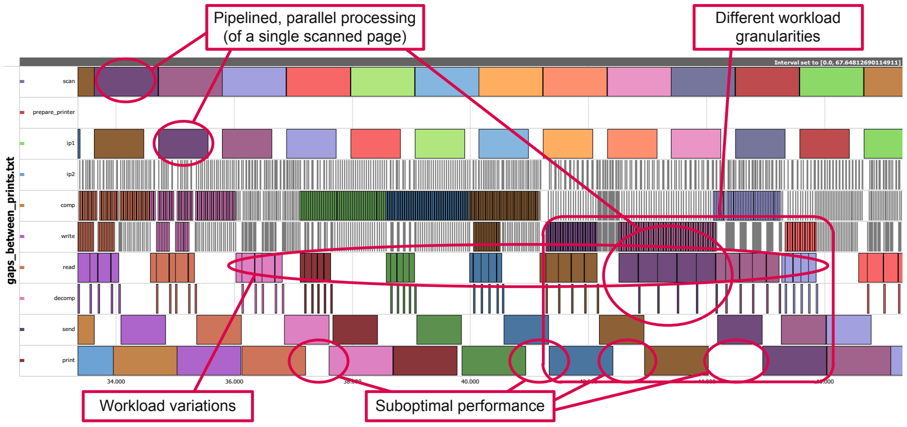
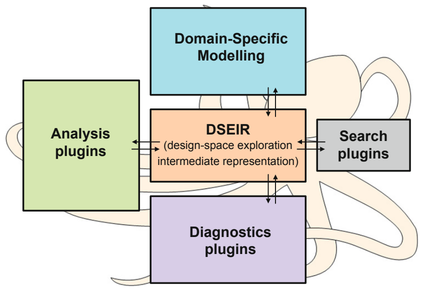


Fig. 2. An illustrative Gantt chart showing the execution of a printing pipeline. Dynamics in the processing pipeline cause hick-ups in print performance due to under-dimensioning of the embedded execution platform.

bottlenecks, sensitivity to workload variations or spec changes, energy efficiency, etc. Different models may be needed to address these questions. Models should be intuitive to develop for engineers from different disciplines (hardware, software, control), and they should be consistent with each other. Multiple tools may be needed to support the modelling and analysis. The Embedded Systems Institute (ESI) and its academic and industrial partners have picked up the challenge to develop systematic methods and tool support for DSE of software-intensive embedded systems. Given the characteristics of DSE, our approach is based on two important principles: separation of concerns and re-use and integration of existing techniques and tools (see Fig. 3).

ESI coordinates its efforts on model-driven DSE through the Octopus tool set (http://dse.esi.nl). The tool set architecture (Fig. 3, left) separates the modelling of design alternatives, their analysis, the interpretation and diagnostics of analysis results, and the exploration of the space of alternatives. This separation is obtained by introducing an intermediate representation, the DSE Intermediate Representation DSEIR, and automatic model transformations to and from this representation. This set-up allows the use of a flexible combination of models and tools. It supports domain-specific modelling in combination with generic analysis tools. Multiple analyses can be applied on the same model, guaranteeing model consistency among these analyses; different analysis types and analyses based on multiple models can be integrated in a single search of the design space. Results can be interpreted in a unified diagnostics framework.

The modelling in Octopus follows the Y-chart paradigm of [1,5] (Fig. 3, right) that separates the concerns of modelling the application functionality, the embedded platform, and the mapping of application functionality onto the platform. This separation allows to easily explore variations in some of these aspects, for example the platform configuration or the resource arbitration, while fixing other


Domain-Specific Modelling
Analysis plugins
DSEIR (design-space exploration intermediate representation)
Search plugins
Diagnostics plugins

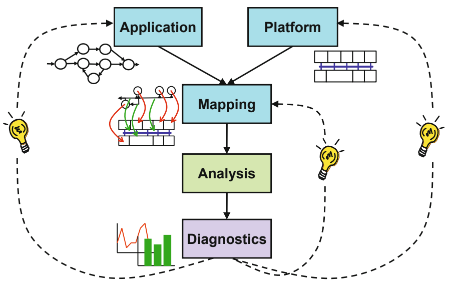


Fig. 3. Separation of concerns. The Octopus tool set architecture separates modeling, analysis, search and diagnostics through the intermediate representation DSEIR (left). Modeling and design-space exploration follow the Y-chart paradigm ([1,5]; figure from [2]), separating application, platform and mapping aspects (right).

aspects, such as the parallelized task structure of the application. It also facilitates re-use of aspect models over different designs.

Intermediate representation DSEIR [7] plays a crucial role in the Octopus approach. It follows the Y-chart paradigm and is specifically designed for the purpose of model-driven DSE. DSEIR is realized as a Java library. The current implementation supports four views: application, platform, mapping, and experiment. DSEIR can be used through a Java interface, an Eclipse-based XML interface, and an Eclipse-based prototype GUI.

Applications in DSEIR are modelled as task graphs. Tasks communicate via ports, and their work loads are specified in terms of required services (such as CPU computation, storage needs). The use of services avoids direct references to the platform definitions, thus realizing the Y-chart separation of concerns. A platform consists of a number of resource declarations. Each resource has a capacity and provides services at a certain speed. The combination of service speed of a resource and service work load of a task can be used to compute task execution times. The mapping connects an application to a platform. It consists of resource allocations and priority specifications. Resource allocations specify whether or not preemption is allowed. Execution follows the dynamic priority scheduling paradigm. DSEIR models are all parameterized, with for example processor speeds, memory sizes, priorities, etc. as parameters. DSEIR has a well-defined operational semantics. The abstraction level is such that it is possible to efficiently and effectively capture the system behaviour and dynamics that is essential for a fast but accurate assessment of design alternatives. The experiment view of DSEIR allows to define sets of DSE experiments for selected models and model parameter settings. The analyses to be performed and the way to handle the output of the experiments can be specified as well.

The current tool set implementation supports modelling directly in DSEIR [7] as well as modeling of printer data paths for professional printers through a Domain-Specific Language (DSL) called DPML, the Data-Path Modelling Language [6]. Several types of analysis are supported. Performance analysis through discrete-event simulation is supported via CPN Tools (http://cpntools.org), model checking is supported via Uppaal (http://www.uppaal.org), and dataflow analysis through SDF3 (http://www.es.ele.tue.nl/sdf3). Exploration support is available through JGAP (http://jgap.sourceforge.net) and the tool set has built-in support for dividing multiple analyses over the available processors of a multi-processor machine. Diagnostic support is provided through Excel, Gantt charts (see Fig. 2) and Pareto analysis [4].

3 Industrial Experiences

We have used Octopus in several case studies at Océ Technologies. These case studies involve design-space exploration of printer data paths of professional printers. Professional printing systems perform a variety of image processing functions on digital documents that support the standard scanning, copying and printing use cases, as well as many combinations and variations of these use cases. The printer data path encompasses the complete trajectory of the image data from source (for example the scanner or the network) to target (the imaging unit). The case studies show that the Octopus tool set can successfully deal with several modeling challenges, like various and mixed abstraction levels (from pages to pixels and everything in between), preemptive and non-preemptive scheduling, stochastic behavior, dynamic memory management, page caching policies, heterogeneous processing platforms with CPUs, GPUs, and FPGAs, realistic PCI and USB arbitration, etc. Our analyses identified performance bounds for printing pipelines and resource bottlenecks limiting performance, they gave designers a better understanding of the systems, confirmed design decisions (scheduling, arbitration, and caching), and suggested small design improvements (buffering, synchronization). Both DPML and DSEIR models can be made with little effort, very similar to the time investment needed for a spreadsheet model. An important advantage is that one model suffices to use different analysis tools. The analysis time in CPN Tools and Uppaal using models automatically generated from DSEIR models was always at least as good as with handcrafted models.

4 Challenges

The first experiences with the Octopus approach to model-driven DSE have been successful, but many challenges remain, both scientific challenges and challenges related to industrial adoption of model-driven DSE:

- How do we properly handle combinations of discrete, continuous, and probabilistic aspects? Such combinations materialize from combinations of timing aspects, user interactions, discrete objects being manipulated, physical processes being controlled, failures, wireless communication, etc.
- How can we effectively combine the strengths of different types of analysis, involving for example model checking, simulation, dataflow analysis, constraint programming, SAT/SMT solving, etc.? No single analysis technique is suitable for all purposes. Integration needs to be achieved without resorting to one big unified model.

- How do we achieve scalability? Can we support modular analysis and compositional reasoning across analysis techniques, across abstraction levels, and for combinations of discrete, continuous, and probabilistic aspects?
- How to cope with uncertain and incomplete information? Information is often unavailable early in the design process, environment parameters and user interactions may be uncontrollable and unpredictable. How do we guarantee robustness of the end result of DSE against (small) variations in parameter values? Can we develop appropriate sensitivity analysis techniques?
- How do we cope with the increasing dynamics in modern embedded systems? Today's systems are open, connected, and adaptive in order to enrich their functionality, enlarge their working range and extend their life time, to reduce cost, and to improve quality under uncertain and changing circumstances. System-level control loops play an increasingly important role. What is the best way to co-design control and embedded hardware and software?
- Can we develop systematic, semi-automatic DSE methods that can cope with the complexity of next generations of high-tech systems? How do we provide model consistency when combining multiple models and analysis techniques?
- How do we handle the many different use cases that a typical embedded platform needs to support? How to support trade-off analysis over the many objectives that play a role in DSE?
- How do we incorporate DSE in the system development processes? This involves aspects like model calibration, model validation, and model versioning, but also linking DSE to code generation, hardware synthesis, and possibly model-based testing.
- How do we achieve industrial acceptance? Industrially mature DSE tools are a prerequisite. DSL support, tool chain customization, integration with other development tools, and training all need to be taken care of.

References

1. Balarin, F., et al.: Hardware-Software Co-design of Embedded Systems: The POLIS Approach. Kluwer (1997)
2. Basten, T., van Benthum, E., Geilen, M., Hendriks, M., Houben, F., Igna, G., Reckers, F., de Smet, S., Somers, L., Teeselink, E., Trčka, N., Vaandrager, F., Verriet, J., Voorhoeve, M., Yang, Y.: Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 90–105. Springer, Heidelberg (2010)
3. Basten, T., et al.: Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems. In: Basten, T., et al. (eds.) Model-based Design of Adaptive Embedded Systems. Springer (to appear, 2013)
4. Hendriks, M., et al.: Pareto Analysis with Uncertainty. In: Proc. EUC 2011, pp. 189–196. IEEE CS Press (2011)
5. Kienhuis, B., et al.: An Approach for Quantitative Analysis of Application-specific Dataflow Architectures. In: Proc. ASAP 1997, pp. 338–349. IEEE (1997)
6. Teeselink, E., et al.: A Visual Language for Modeling and Analyzing Printer Data Path Architectures. In: Proc. ITSLE 2011, 20 p. (2011), http://planet-sl.org/itsle2011/
7. Trcka, N., et al.: Integrated Model-Driven Design-Space Exploration for Embedded Systems. In: Proc. IC-SAMOS 2011, pp. 339–346. IEEE CS Press (2011)

Statistical Model Checking, Refinement Checking, Optimization, ... for Stochastic Hybrid Systems*

Kim G. Larsen

Computer Science, Aalborg University, Denmark

Statistical Model Checking (SMC) [19,16,21,23,15] is an approach that has recently been proposed as new validation technique for large-scale, complex systems. The core idea of SMC is to conduct some simulations of the system, monitor them, and then use statistical methods (including sequential hypothesis testing or Monte Carlo simulation) in order to decide with some degree of confidence whether the system satisfies the property or not. By nature, SMC is a compromise between testing and classical formal method techniques. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, and are some times the only option.

In a series of recent works [14,13], we have investigated the problem of Statistical Model Checking for networks of Priced Timed Automata (PTAs), being timed automata, whose clocks can evolve with different rates, while[1] being used with no restrictions in guards and invariants. In [13], we have proposed a natural stochastic semantics for such automata, which allows to perform statistical model checking. Our work has been implemented in Uppaal-smc, providing a new statistical model checking engine for the tool Uppaal. Uppaal-smc relies on a series of extensions of the statistical model checking approach generalized to handle real-time systems and estimate undecidable problems. Uppaal-smc comes together with a rich modeling and specification language [8,7], as well as a friendly user interface that allows a user to specify complex problems in an efficient manner as well as to get feedback in the form of probability distributions and compare probabilities to analyze performance aspects of systems. Also, distributed implementations of the various statistical model checking algorithms has been given with demonstrated linear speed-up [9].

In this talk we will report on the most recent developments of Uppaal-smc and contemplate on how other branches of Uppaal may benefit from the new scalable simulation engine of Uppaal-smc in order to improve their performance as well as scope in terms of the models that are supported.

Stochastic Hybrid Automata. Most recently, we have extended Uppaal-smc to networks of stochastic hybrid automata, allowing clock rates to depend not

* Work partially supported by the VKR Centre of Excellence MT-LAB, the Sino-Danish Basic Research Center IDEA4CPS.

[1] In contrast to the usual restriction of priced timed automata [2,1].

only on values of discrete variables but also on the value of other clocks, effectively amounting to ordinary differental equations. In particular our original race-based stochastic semantics extends to this setting with the use of Dirac's delta-functions, to allow for the co-existence of (time-wise) stochastic and determinstic components. This extension of UPPAAL-SMC has already been applied to a wide range of hybrid systems example from real-time scheduling and mixed criticality systems [11], energy aware systems [10] and systems biology [12].

Wittness Counter Examples. Based on the real-time scheduling problem of [11], we will show how statistical model checking may serve as an indispensable tool for exhibiting concrete (rare) counter examples witnessing non-schedulability in the setting of stop-watch automata, where the UPPAAL verification engine is over-approximate.

Refinement Checking. Statistical model checking may be used as an alternative to the game-based engine of ECDAR for checking refinements between model. In particular, for early development stages, where a desired refinement does *not* hold, SMC can provide an efficient quick check, with useful debugging information pin terms of the graphical simulation plots provided by the tool. Also, SMC may estimate the distance between two models, in terms of the probability measure of the set of runs in their difference.

Controller Synthesis. SMC may be the only way of establishing - to some level of confidence - that a rich hybrid model is endeed abstracted by a timed (game) automaton, allowing for subsequent model checking and synthesis possibly in composition with other components. Using the framework for energy aware buildings [10], we will indicate how UPPAAL-SMC in combination with UPPAAL-TIGA may be used to synthesize and performance-evaluate control strategies from a natural hybrid model.

Optimizations. The UPPAAL-CORA branch [17,3,20] offers an efficient, agent-based and symbolic engine for solving a large range of optimization problems given their model as priced timed automata [2]. However, the tool is restricted to models with a single cost-variable (though extensions have been proposed [18]), with – for decidability – crucial assumption that the cost-variable is only used as an observer (thus cannot be used in guards or invariants). This assumption is lifted slightly in the a sequence of recent work on *energy timed automata* [5,4,6], where the cost-variable is required to be within given bounds. We want to investigate whether the new SMC engine may provide a competitive method opening the possibility for optimization to a wider range of models.

Meta Modeling. The modeling formalism of UPPAAL-SMC allows various statistical model checking algorithms themselves to be modelled and their performance on given problems to be analysed. This usage of UPPAAL-SMC as a meta-modeling and -analysis tool may save significant implementation effort, and has already been applied to analysis of the distributed implementation of SMC [9] and a rewrite-technique for statistical model checking of WMTL [7]. In addition, we will investigate the potential effect of applying methods for rare-event simulation such as the RESTART method [22].

References

1. Alur, R., La Torre, S., Pappas, G.J.: Optimal Paths in Weighted Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
2. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.M.T., Vaandrager, F.W.: Minimum-Cost Reachability for Priced Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
3. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review 32(4), 34–40 (2005)
4. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 61–70. ACM (2010)
5. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
6. Bouyer, P., Larsen, K.G., Markey, N.: Lower-bound constrained runs in weighted timed automata. In: Proceedings of the 9th International Conference on Quantitative Evaluation of Systems (QEST 2012). IEEE Computer Society Press, London (to appear, September 2012)
7. Bulychev, P., David, A., Larsen, K., Legay, A., Li, G., Poulsen, D.: Rewrite-based statistical model checking of wmtl (under Submission)
8. Bulychev, P., David, A., Larsen, K.G., Legay, A., Li, G., Bøgsted Poulsen, D., Stainer, A.: Monitor-Based Statistical Model Checking for Weighted Metric Temporal Logic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 168–182. Springer, Heidelberg (2012)
9. Bulychev, P., David, A., Larsen, K.G., Mikucionis, M., Legay, A.: Distributed parametric and statistical model checking. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 30–42 (2011)
10. David, A., Du, D., Larsen, K.G., Mikučionis, M., Skou, A.: An evaluation framework for energy aware buildings using statistical model checking. Science China. Information Sciences (submitted, 2012)
11. David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Schedulability of herschel-planck revisited using statistical model checking. In: Steffen, B., Margaria, T. (eds.) 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Heraclion, Crete (accepted, October 2012)
12. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Runtime verification of biological systems. In: Steffen, B., Margaria, T. (eds.) 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Heraclion, Crete (accepted, October 2012)
13. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical Model Checking for Networks of Priced Timed Automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)

14. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
15. 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)
16. Laplante, S., Lassaigne, R., Magniez, F., Peyronnet, S., de Rougemont, M.: Probabilistic abstraction for model checking: An approach based on property testing. ACM TCS 8(4) (2007)
17. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.M.T.: As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)
18. Larsen, K.G., Rasmussen, J.I.: Optimal Conditional Reachability for Multi-priced Timed Automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 234–249. Springer, Heidelberg (2005)
19. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010)
20. Rasmussen, J.I., Behrmann, G., Larsen, K.G.: Complexity in Simplicity: Flexible Agent-Based State Space Exploration. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 231–245. Springer, Heidelberg (2007)
21. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
22. Villén-Altamirano, M., Villén-Altamirano, J.: Restart: a straightforward method for fast simulation of rare events. In: Winter Simulation Conference, pp. 282–289 (1994)
23. Younes, H.L.S., Simmons, R.G.: Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)

Robustness of Time Petri Nets under Architectural Constraints[⋆]

S. Akshay[1,2], Loïc Hélouët[1], Claude Jard[1,2], Didier Lime[3], and Olivier H. Roux[3]

[1] INRIA/IRISA Rennes, France
[2] ENS Cachan Bretagne, Rennes, France
[3] LUNAM Université, École Centrale de Nantes, IRCCyN (CNRS UMR 6597), Nantes, France

Abstract. This paper addresses robustness issues in Time Petri Nets (TPN) under constraints imposed by an external architecture. The main objective is to check whether a timed specification, given as a TPN behaves as expected when subject to additional time and scheduling constraints. These constraints are given by another TPN that constrains the specification via read arcs. Our robustness property says that the constrained net does not exhibit new timed or untimed behaviors. We show that this property is not always guaranteed but that checking for it is always decidable in 1-safe TPNs. We further show that checking if the set of untimed behaviors of the constrained and specification nets are the same is also decidable. Next we turn to the more powerful case of labeled 1-safe TPNs with silent transitions. We show that checking for the robustness property is undecidable even when restricted to 1-safe TPNs with injective labeling, and exhibit a sub-class of 1-safe TPNs (with silent transitions) for which robustness is guaranteed by construction. We demonstrate the practical utility of this sub-class with a case-study and prove that it already lies close to the frontiers of intractability.

1 Introduction

Robustness is a key issue for the implementation of systems. Once a system is implemented on a given architecture, one may discover that it does not behave as expected: some specified behaviors are never met or unspecified behaviors appear. Thus, starting from a description of a system, one wants to ensure that the considered system can run as expected on a given architecture with resource constraints (e.g., processors, memory), scheduling schemes on machines implementing several components of the system, imprecision in clocks, possible failures and so on.

We address this issue of robust implementability for systems in which time and concurrency play a key role. We start with a Petri net model of a concurrent system, which is constrained by another Petri net defining some implementation details (for example, the use of resources). We then want to check that such implementation features do not create or introduce new behaviors, which were not present in the original model. If the implementation features can only restrict (but not enlarge) the set of original behaviors, we say the model is robust with respect to the implementation constraints. We lift these ideas to the timed setting by using the model of time Petri nets. Time Petri nets (TPNs)

[⋆] This work was funded by the project ANR ImpRo (ANR-2010-BLAN-0317).

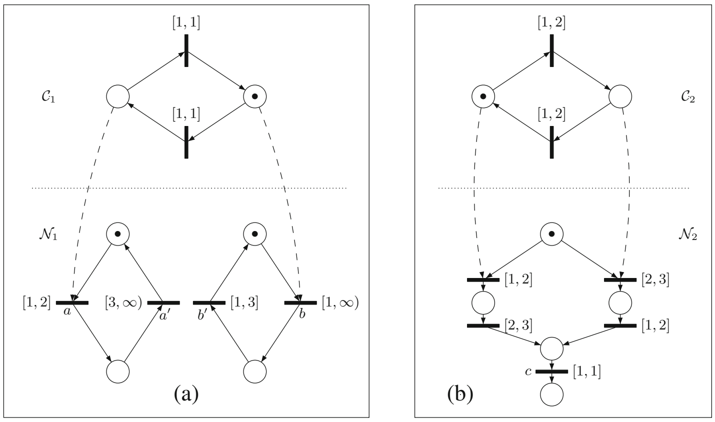


Fig. 1. Illustrative examples (a) and (b) of TPNs with read-arcs: transitions are represented as black rectangles, places as circles, arcs as thick lines, read arcs as dotted lines. Transitions can be labeled by letters (observable actions), or unlabeled (silent moves) and by intervals (constraints).

are Petri nets whose transitions are equipped with timing constraints, given as intervals. As soon as a transition is enabled, a clock attached to this transition is reset and starts measuring time. A transition is then allowed to fire if it is enabled and if its clock's value lies within the time interval of the transition. When a TPN contains read arcs, places that are read can enable/disable a transition, but tokens from read places are not consumed at firing time. In the literature of timed systems (for e.g., [15]), robustness in timed automata usually refers to invariance of behaviors under small time perturbations. We use the term "robustness" in a more general context: we consider preservation of specified behaviors when new architectural constraints (e.g., scheduling policies, resources) are imposed.

In this paper, we focus our study of robustness to TPNs whose underlying Petri nets are 1-safe (i.e., have at most one token in any place of a reachable marking). We consider bipartite architectures: a specification of a distributed system is given as a TPN, called the *ground net* and the architectural constraints are specified by another TPN, called the *controller net*. The controller net can read places of the ground net, but cannot consume tokens from the ground net, and vice versa. The net obtained by considering the ground net in the presence of the controller is called the *controlled net*. We first ask if the untimed language of the controlled net is contained in the untimed language of the ground net. This problem is called *untimed robustness*. Next, we ask if the untimed language is exactly the same in the presence of control, which we call the *untimed equivalence problem*. Finally the *timed robustness* problem asks if the timed language of the controlled net is contained in the timed language of the ground net.

Though our setting resembles supervisory control [12], there are some important differences. Supervisory control is used to restrict the behaviors of a system in order to meet some (safety) property P. The input of the problem is the property P, a description of the system, and the output a controller that restricts the system: the behavior of a system under control is a subset of the original specification satisfying P. In our setting, there is no property to ensure, but we want to preserve as much as possible the specified behaviors. We will show in the example below that architectural contraints may add behaviors to the specification. This situation can be particularly harmful, especially when the architecture changes for a system that has been running properly on a former architecture. New faults that were not expected may appear, even when the overall performance of the architecture improves. Detecting such situations is a difficult task that should be automated. The last difference with supervisory control is that we do not ask for synthesis of a controller. In our setting, the controller represents the architectural constraints, and is part of the input of the robustness problem. The question is then whether the ground net preserves its behaviors when controlled.

An example is shown in Figure 1-a, containing a ground net $\mathcal{N}_1$, with four transitions a, a', b, b', and a controller $\mathcal{C}_1$, that acts as a global scheduler allowing firing of a or b. In $\mathcal{N}_1$, transitions a, a' and b, b' are independant. The net $\mathcal{N}_1$ is not timed robust w.r.t. the scheduling imposed by $\mathcal{C}_1$: in the controlled net, a can be fired at date 3 which is impossible in $\mathcal{N}_1$ alone. However, if we consider the restriction of $\mathcal{N}_1$ to b, b', the resulting subnet is timed robust w.r.t $\mathcal{C}_1$. Figure 1-b shows a ground net $\mathcal{N}_2$ with four unobservable transitions, and one observable transition c. This transition can be fired at different dates, depending on whether the first transition to fire is the left transition (with constraint $[1, 2]$) or the right transition (with constraint $[2, 3]$) below the initially marked place. The net $\mathcal{C}_2$ imposes that left and right transitions are not enabled at the same time, and switches the enabled transition from time to time. With the constraints imposed by $\mathcal{C}_2$, c is firable at date 5 in the controlled net but not at date 6 while it is firable at both dates 5 and 6 in $\mathcal{N}_2$ alone. This example is timed robust w.r.t $\mathcal{C}_2$, as it allows a subset of its original behaviors.

Our results are the following. The problem of untimed robustness for 1-safe TPNs is decidable. The timed variant of this problem is decidable for 1-safe TPNs, under the assumption that there are no ϵ transitions and the labeling of the ground net is injective. However, with arbitrary labeling and silent transitions the timed robustness problem becomes undecidable. Further, even with injective labeling, timed robustness is undecidable as soon as the ground net contains silent transitions. We then show a natural relaxation on the way transitions are controlled and constrained, which ensures timed robustness. In the untimed setting we also consider the stronger notion of equivalence of untimed languages and show that checking this property is decidable with or without silent transitions. The paper is organized as follows: Section 2 introduces the TPN models and the problems considered. Section 3 shows decidability of robustness in the untimed setting, or when nets are unlabeled. Section 4 shows that this problem becomes undecidable in the timed setting as soon as silent transitions are introduced. Section 5 shows conditions on ground nets and control schemes ensuring timed robustness. Section 6 provides a case-study to show the relevance of these conditions, before concluding with Section 7. Missing proofs can be found in an extended version available at [1].

As further related work, we remark that several papers deal with control of Petri Nets where transitions are divided into untimed controllable and uncontrollable transitions. Among them, Holloway and Krogh [9] first proposed an efficient method to solve a control problem for a subclass of Petri Nets called *safe marked graphs*. Concerning TPNs, [7] propose a method inspired by the approach of Maler [11]. The controller is synthesized as a feedback function over the state space. However, in all these papers, the controller is given as a feedback law, and it is not possible to design a net model of the controlled system. To overcome this problem, [8] propose a solution using *monitors* to synthesize a Petri Net that models the closed-loop system. The method is extended to real time Supervisory Control in [13]. The supervisor uses enabling arcs (which are equivalent to read arcs) to enable or block a controllable transition. In [16], robustness is addressed in a weaker setting called *schedulability*: given an TPN $\mathcal{N}$, the question is whether the untimed language of $\mathcal{N}$, and the language of the underlying untimed net (i.e. without timing constraints) is the same. This problem is addressed for acyclic nets, or for nets with restricted cyclic behaviors.

2 The Model and the Questions

Let $\mathbb{Q}^+, \mathbb{R}^+$ denote the set of non-negative rationals and reals respectively. Then, $\mathcal{I}$ denotes the set of time intervals, i.e., intervals in $\mathbb{R}^+$ with end points in $\mathbb{Q}^+ \cup \{+\infty\}$. An interval $I \in \mathcal{I}$ can be open (I^-, I^+), closed $[I^-, I^+]$, semi-open $(I^-, I^+]$, $[I^-, I^+)$ or unbounded $[I^-, +\infty)$, $(I^-, +\infty)$, where I^- and $I^+ \in \mathbb{Q}^+$.

2.1 Time Petri Nets

Definition 1 (place/transition net with read arcs). *A time Petri net (TPN for short) with read arcs is a tuple $\mathcal{N} = (P, T, W, R, I)$ where P is a finite set of* places*, T is a finite set of* transitions*, with $P \cap T = \emptyset$, $W : (P \times T) \cup (T \times P) \to \{0, 1\}$ and $R : (P \times T) \to \{0, 1\}$ s.t., $W^{-1}(1) \cap R^{-1}(1) = \emptyset$ are* flow relations *and $I : T \to \mathcal{I}$ is a map from the transitions of $\mathcal{N}$ to time intervals $\mathcal{I}$.*

Every TPN can be seen as a union of an untimed Petri Net $N = (P, T, W, R)$ and of a timing function I. The untimed net N will be called the *underlying net* of $\mathcal{N}$.

Semantics. The net defines a bipartite directed graph with two kinds of edges: there exists a (consume) arc from x to y (drawn as a solid line) iff $W(x, y) = 1$ and there exists a (read) arc from x to y (drawn as a dashed line) iff $R(x, y) = 1$. For all $x \in P \cup T$, we define the following sets: ${}^\bullet x = \{y \in P \cup T \mid W(y, x) = 1\}$ and $x^\bullet = \{y \in P \cup T \mid W(x, y) = 1\}$. For all $x \in T$, we define ${}^\circ x = \{y \in P \mid R(y, x) = 1\}$. These definitions extend naturally to subsets by considering union of sets. A *marking* $m : P \to \mathbb{N}$ is a function such that (P, m) is a multiset. For all $p \in P$, $m(p)$ is the number of *tokens* in the place p. A transition $t \in T$ is said to be *enabled* by the marking m if $m(p) > 0$ for every place $p \in ({}^\bullet t \cup {}^\circ t)$. $\mathsf{en}(N, m)$ denotes the set of transitions of N enabled by m. The firing of an enabled transition t produces a new marking m' computed as $\forall p \in P, m'(p) = m(p) - W(t, p) + W(p, t)$. We fix a marking m^0 of N called its *initial marking*. We say that a transition t' is in conflict with a transition t if $({}^\bullet t \cup {}^\circ t) \cap ({}^\bullet t') \neq \emptyset$ (firing t' consumes tokens that enable t).

The semantics of a TPN is usually given as a timed transition system (TTS) [10]. This model contains two kinds of transitions: continuous transitions when time passes and discrete transitions when a transition of the net fires. A transition t_k is said to be *newly enabled* by the firing of the firable transition t_i from the marking m, and denoted

$\uparrow\mathsf{en}(t_k, m, t_i)$, if the transition t_k is enabled by the new marking $(m \setminus {}^\bullet t_i) \cup t_i^\bullet$ but was not by $m \setminus ({}^\bullet t_i)$. We will denote by $\uparrow\mathsf{en}(m, t_i)$ the set of transitions newly enabled by the firing of t_i from m. A valuation is a map $\nu : T \to \mathbb{R}^+$ such that $\forall t \in T, \nu(t)$ is the time elapsed since t was last newly enabled. For $\delta \in \mathbb{R}^+$, $\nu + \delta$ denotes the valuation that associates $\nu(t) + \delta$ to every transition $t \in T$. Note that $\nu(t)$ is meaningful only if t is an enabled transition. $\mathbf{0}$ is the null valuation such that $\forall t, \mathbf{0}(t) = 0$.

The semantics of TPN $\mathcal{N}$ is defined as the TTS $(Q, q_0, \to)$ where a state of Q is a couple (m, ν) of a marking and valuation of $\mathcal{N}$, $q_0 = (m_0, \mathbf{0})$ and $\to \in (Q \times (T \cup \mathbb{R}^+) \times Q)$ is the transition relation describing continuous and discrete transitions. The continuous transition relation is defined for all $\delta \in \mathbb{R}^+$ by:

$$(m, \nu) \xrightarrow{\delta} (m, \nu') \text{ if } \nu' = \nu + \delta \text{ and } \forall t_k \in \mathsf{en}(m), \text{ we have,}$$
$$\begin{cases} \nu'(t_k) \leq I(t_k)^+ \text{ if } I(t_k) \text{ is of the form } [a, b] \text{ or } (a, b] \\ \nu'(t_k) < I(t_k)^+ \text{ if } I(t_k) \text{ is of the form } [a, b) \text{ or } (a, b) \end{cases}$$

Intuitively, time can progress iff letting time elapse does not violate the upper constraint $I(t)^+$ of any transition t (recall that we write $I(t)^+$ for the right endpoint of the interval $I(t)$). Now, the discrete transition relation is defined for all $t_i \in T$ by:

$$(m, \nu) \xrightarrow{t_i} (m', \nu') \text{ if } \begin{cases} t_i \in \mathsf{en}(m), m' = (m \setminus {}^\bullet t_i) \cup t_i^\bullet \\ \nu(t_i) \in I(t_i), \\ \forall t_k, \nu'(t_k) = 0 \text{ if } \uparrow\mathsf{en}(t_k, m, t_i) \text{ and } \nu(t_k) \text{ otherwise.} \end{cases}$$

That is, transition t_i can fire if it was enabled for a duration included in the time constraint $I(t)$. Firing t_i from m resets the clocks of newly enabled transitions.

A *run* of a TTS is a sequence of the form $p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} p_n$ where $p_1 = q_0$, and for all $i \in \{2, \ldots, n\}$, $(p_{i-1}, \alpha_i, p_i) \in \to$ and $\alpha_i = t_i \in T$ or $\alpha_i = \delta_i \in \mathbb{R}^+$. Each finite run defines a sequence over $(T \cup \mathbb{R}^+)^*$ from which we can obtain a *timed word over* T of the form $w = (t_1, d_1)(t_2, d_2) \ldots (t_n, d_n)$ where each t_i is a transition and $d_i \in \mathbb{R}^+$ the time at which transition t_i is fired. More precisely, if the sequence of labels read by the run is of the form $\delta_0\delta_1 \ldots \delta_{k_1} t_1 \delta_{k_1+1}\delta_{k_1+2} \ldots \delta_{k_2} t_2 \ldots t_n$, then the timed word obtained is $(t_1, d_1) \ldots (t_n, d_n)$ where $d_i = \sum_{0 \leq j \leq k_i} \delta_j$. We define a *dated run* of a TPN $\mathcal{N}$ as the sequence of the form $q_1 \xrightarrow{(d_1,t_1)} q_2 \ldots \xrightarrow{(d_n,t_n)} q_n$, where d_i's are the dates as defined above and each q_i is the state reached after firing t_i at date d_i.

We denote by $\mathcal{L}_{tw}(\mathcal{N})$ the timed words over T generated by the above semantics. This will be called the timed (transition) language of $\mathcal{N}$. We denote by $\mathcal{L}_w(\mathcal{N})$ the untimed language of sequences of transitions obtained by projecting onto the first component. Furthermore, given a timed word w over T, if we consider a subset of transitions $X \subseteq T$, we can project w onto X to obtain a timed word over X. We will denote this projected language by $\mathcal{L}_{tw}(\mathcal{N})|_X$. For simplicity, we have not considered final states in our TTS and hence we define prefix-closed languages as is standard in Petri nets. However, our results continue to hold with an appropriate definition of final states.

In this paper, we limit the study of robustness to TPNs where the underlying PN is *1-safe* i.e., nets such that $\forall p \in P,\ m(p) \leq 1$, for all reachable markings m in the underlying PN. The reason for using a property of the underlying net is that deciding if an untimed PN is 1-safe is PSPACE-complete [6], whereas checking if a TPN is bounded is undecidable [14]. Reachability of a marking m in a 1-safe net is also PSPACE-complete [6]. For 1-safe Petri nets a place contains either 0 or 1 token, hence we identify a marking m with the set of places p such that $m(p) = 1$. In the sequel, as we always consider 1-safe nets, we will frequently omit saying this explictly.

2.2 The Control Relation

Let us consider two Time Petri nets $\mathcal{N} = (P_\mathcal{N}, T_\mathcal{N}, W_\mathcal{N}, R_\mathcal{N}, I_\mathcal{N}, m^0_\mathcal{N})$ and $\mathcal{C} = (P_\mathcal{C}, T_\mathcal{C}, W_\mathcal{C}, R_\mathcal{C}, I_\mathcal{N}, m^0_\mathcal{C})$. $\mathcal{C}$ models time constraints and resources of an architecture. One can expect these constraints to restrict the behaviors of the original net (we will show however that this is not always the case), that is $\mathcal{C}$ could be seen as a controller. Rather than synchronizing the two nets (as is often done in supervisory control), we define a relation $R \subseteq (P_\mathcal{C} \times T_\mathcal{N}) \cup (P_\mathcal{N} \times T_\mathcal{C})$, connecting some places of $\mathcal{C}$ to some transitions of $\mathcal{N}$ and vice versa. The resulting net $\mathcal{N}^{(\mathcal{C},R)}$ is still a place/transition net defined by $\mathcal{N}^{(\mathcal{C},R)} = (P_\mathcal{N} \cup P_\mathcal{C}, T_\mathcal{N} \cup T_\mathcal{C}, W_\mathcal{N} \cup W_\mathcal{C}, R_\mathcal{N} \cup R_\mathcal{C} \cup R, I_\mathcal{N} \cup I_\mathcal{C}, m^0_\mathcal{N} \cup m^0_\mathcal{C})$. We call $\mathcal{N}$ the *ground net*, $\mathcal{C}$ the *controller net* and $\mathcal{N}^{(\mathcal{C},R)}$ the *controlled net*.

The reason for choosing this relation is two-fold. Firstly, the definition of control above preserves the formalism as the resulting structure is a time Petri net as well. This allows us to deal with a single formalism throughout the paper. Secondly, one can define several types of controllers. By allowing read arcs from the controller to the ground net only, we model *blind* controllers i.e., controllers whose states evolve independently of the ground net's state. The nets in Figure 1 are examples of such controlled nets. In the reverse direction, if read arcs are allowed from the ground net to the controller, the controller's state changes depending on the current state of the ground net. For the sake of clarity, all examples in the paper have blind controllers but our results hold even with general controllers and bi-directional read arcs.

Our goal is to compare the behaviors of $\mathcal{N}$ with its behaviors when controlled by $\mathcal{C}$ under R, i.e., $\mathcal{N}^{(\mathcal{C},R)}$. Therefore, the language of (timed and untimed) transitions, i.e., $\mathcal{L}_{tw}(\mathcal{N}), \mathcal{L}_{tw}(\mathcal{C}), \mathcal{L}_w(\mathcal{N}), \mathcal{L}_w(\mathcal{C})$, are as usual but when talking about the language of the controlled net, we will always mean the language projected onto transitions of $\mathcal{N}$, i.e., $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)})|_{T_\mathcal{N}}$ or $\mathcal{L}_w(\mathcal{N}^{(\mathcal{C},R)})|_{T_\mathcal{N}}$. Abusing notation, we will write $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)})$ (similarly $\mathcal{L}_w(\mathcal{N}^{(\mathcal{C},R)})$) to denote their projections onto $T_\mathcal{N}$.

2.3 The Robustness Problem

We now formally define and motivate the problems that we consider in this paper.

Definition 2. *Given TPNs $\mathcal{N}$ and $\mathcal{C}$, and a set of read arcs $R \subseteq (P_\mathcal{C} \times T_\mathcal{N}) \cup (P_\mathcal{N} \times T_\mathcal{C})$, $\mathcal{N}$ is said to be* untimed robust *under $(\mathcal{C}, R)$ if $\mathcal{L}_w(\mathcal{N}^{\mathcal{C},R}) \subseteq \mathcal{L}_w(\mathcal{N})$.*

For time Petri nets, the first problem we consider is the *untimed robustness* problem, which asks whether a given TPN $\mathcal{N}$ is untimed robust under $(\mathcal{C}, R)$. This corresponds to checking whether the controlled net $\mathcal{N}^{(\mathcal{C},R)}$ only exhibits a subset of the (untimed) behaviors of the ground TPN $\mathcal{N}$. The second question addressed is the *untimed equivalence* problem, which asks if the untimed behaviors of the controlled net $\mathcal{N}^{(\mathcal{C},R)}$ and ground net $\mathcal{N}$ are the same, i.e., if $\mathcal{L}_w(\mathcal{N}^{\mathcal{C},R}) = \mathcal{L}_w(\mathcal{N})$. In fact these questions can already be asked for "untimed Petri nets", i.e., for Petri nets without the timing function I and we also provide results for this setting.

Note that untimed robustness only says that every *untimed* behavior of the controlled net $\mathcal{N}^{(\mathcal{C},R)}$ is also exhibited by the ground net $\mathcal{N}$. However some *timed* behaviors of the controlled net $\mathcal{N}^{(\mathcal{C},R)}$ may *not* be timed behaviors of the ground net $\mathcal{N}$. For obvious safety reasons, one may require that a controlled system does not allow new behaviors, timed or untimed. Thus, we would like to ensure or check that even when considering timed behaviors, the set of timed behaviors exhibited by the controlled net $\mathcal{N}^{(\mathcal{C},R)}$ is a subset of the set of timed behaviors exhibited by the ground net $\mathcal{N}$. We call this the *timed robustness* problem.

Definition 3. *Given TPNs $\mathcal{N}$ and $\mathcal{C}$, and a set of read arcs $R \subseteq (P_\mathcal{C} \times T_\mathcal{N}) \cup (P_\mathcal{N} \times T_\mathcal{C})$, $\mathcal{N}$ is said to be* timed robust *under $(\mathcal{C}, R)$ if $\mathcal{L}_{tw}(\mathcal{N}^{\mathcal{C},R}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$.*

One can further ask if the timed behaviors are exactly the same, which means that the controller is useless. In our setting, it means that the architectural constraints do not affect the executions of the system, nor their timings. While untimed equivalence of unconstrained and constrained systems seems a reasonable notion, timed equivalence is rarely met, and hence seems too restrictive a requirement. We will see in Section 4 that introducing silent transitions gives a new meaning to these notions.

3 Controlling (Time) Petri Nets

Let us first consider *untimed* 1-safe Petri nets. Let N be an untimed net, and C be an untimed controller. We can observe that C can only restrict the behaviors of N, under *any* choice of R. Hence N is always untimed robust under (C, R). Furthermore one can effectively check if the controlled net has the same untimed language as the ground net, by building their marking graphs, and then checking inclusion. Thus, the robustness and equivalence problems are decidable for untimed nets.

Proposition 1. *Let N, C be two* untimed *(1-safe) Petri nets. Then,*

1. *For any $R \subseteq (P_C \times T_N) \cup (P_N \times T_C)$, N is untimed robust under (C, R).*
2. *For a fixed set of read arcs $R \subseteq (P_C \times T_N) \cup (P_N \times T_C)$ checking if $\mathcal{L}_w(N) = \mathcal{L}_w(N^{(C,R)})$ is PSPACE-complete.*

This property of untimed Petri nets has a counterpart for time Petri nets: let us consider *unconstrained* nets $\mathcal{N}$ and $\mathcal{C}$, i.e., such that $I_\mathcal{N}(t) = [0, \infty)$ for every $t \in T_\mathcal{N}$, and $I_\mathcal{C}(t) = [0, \infty)$ for every $t \in T_\mathcal{C}$. Let N and C be the underlying nets of $\mathcal{N}$ and $\mathcal{C}$. One can easily show that for any R, $\mathcal{L}_w(\mathcal{N}^{\mathcal{C},R}) \subseteq \mathcal{L}_w(\mathcal{N})$. As any timed word $w = (a_1, d_1) \dots (a_n, d_n)$ in $\mathcal{L}_{tw}(\mathcal{N}^{\mathcal{C},R})$ (resp. in $\mathcal{L}_{tw}(\mathcal{N})$) is such that $a_1 \dots a_n \in \mathcal{L}_w(N^{C,R})$ (resp. $\mathcal{L}_w(N)$) where each $d_1, \dots, d_n$ can be arbitrary dates, we also have $\mathcal{L}_{tw}(\mathcal{N}^{\mathcal{C},R}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$. Thus, unconstrained time Petri nets are also untimed robust.

The question for Time Petri Nets is whether the controlled TPN only restricts the set of behaviors of the original TPN. Unlike in the untimed case, in the timed setting the controlled TPN may exhibit more (and even a different set of) behaviors than the ground TPN, because of the urgency requirement of TPNs. Consider the example in Figure 2.

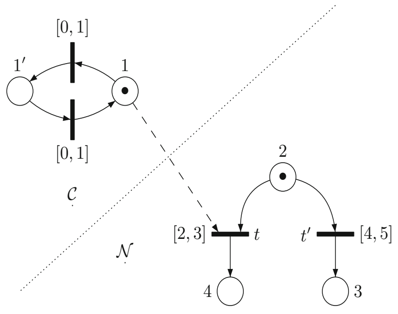


Fig. 2. An example of control of TPN through read-arcs leading to new behaviors

The ground net $\mathcal{N}$ always fires t in the absence of the controller $\mathcal{C}$ but in the presence of $\mathcal{C}$ with R as in the picture, transition t is never fired and t' is always fired. Thus set of (timed and untimed) behaviors of $\mathcal{N}$ and $\mathcal{N}^{(\mathcal{C},R)}$ are disjoint. Discrepancies between untimed languages can be checked using the state class graph construction [5,10], from which we obtain the following theorem.

Theorem 1. *For (1-safe) TPNs, the untimed robustness and untimed equivalence problems are both PSPACE-complete.*

Next we consider timed robustness properties for TPNs, for which we obtain the following result.

Theorem 2. *For (1-safe) TPNs, the timed robustness problem is decidable.*

Proof (sketch). Let $\mathcal{N}$ and $\mathcal{C}$ be 1-safe TPNs, and R be a set of read arcs. We can check if $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$ by using the state class timed automaton construction from [10]. It is shown that from the state class graph construction of a 1-safe TPN, $\mathcal{N}$, we can build a deterministic timed automaton $\mathcal{A}$ over the alphabet $T_{\mathcal{N}}$, called the state class timed automaton, such that $\mathcal{L}_{tw}(\mathcal{N}) = \mathcal{L}_{tw}(\mathcal{A})$. As a result, $\mathcal{L}_{tw}(\mathcal{N})$ can be complemented and its complement is accepted by some timed automaton $\mathcal{A}'$, which is computed from $\mathcal{A}$ (see [2] for complementation of deterministic timed automata). On the other hand, the state class timed automaton $\mathcal{B}$ constructed from $\mathcal{N}^{(\mathcal{C},R)}$ is over the language $T_{\mathcal{N}} \cup T_{\mathcal{C}}$. By projecting this language onto $T_{\mathcal{N}}$, we obtain the timed (transition) language $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)})$. We remark that the timed automaton corresponding to the projection, denoted $\mathcal{B}'$, can be easily obtained by replacing all transitions of $\mathcal{C}$ in the timed automaton $\mathcal{B}$ by ϵ-transitions [2,4]. Now we just check if $\mathcal{L}_{tw}(\mathcal{B}') \cap \mathcal{L}_{tw}(\mathcal{A}') = \emptyset$, which is decidable in PSPACE [2] (in the sizes of $\mathcal{A}'$ and $\mathcal{B}'$). □

4 Controlling TPNs with Silent Transitions

We now consider ground nets which may have silent or ϵ-transitions. The (timed and untimed) language of the ground net contains only sequences of observable (i.e., not ϵ) transitions and the robustness question asks if the controller introduces new timed behaviors with respect to this language of observable transitions. From a modeling perspective, robustness means that sequences of important actions remain unchanged in the presence of architectural constraints, which is a desirable property to have. Silent transitions can be used to model unimportant or unobservable transitions in the ground net. In this setting, it is natural to require that control does not add to the language of important/observable transitions, while it may allow new changes in other transitions.

An example of such a control is given in the introduction in Figure 1-(b). In that example, the ground net has a unique critical (visible) action c. All other transitions are left unlabeled and so we are not interested whether timed or untimed behaviors on those transitions are different in the ground and controlled nets. Then the timed robustness problem asks if c can occur in the controlled net at a date when it was not allowed to occur in the ground net. A more practical example will be studied in detail in Section 6.

With this as motivation, we introduce the class of *ϵ-TPN*, which are TPNs where some transitions may be silent, i.e. labeled by ϵ. The behavior of such nets is deterministic except on silent actions: from a configuration, if a discrete transition that is not labeled ϵ is fired, then the net reaches a unique successor marking.

Definition 4. *Let Σ be a finite set of labels containing a special label ϵ.*

1. *An LTPN over Σ is a structure $(\mathcal{N}, \lambda)$ where $\mathcal{N}$ is a TPN and $\lambda : T_{\mathcal{N}} \to \Sigma$ is the labeling function.*
2. *An ϵ-TPN is an LTPN $(\mathcal{N}, \lambda)$ over Σ such that, for all $t \in T_{\mathcal{N}}$, if $\lambda(t) \neq \epsilon$ then $\lambda(t) \neq \lambda(t')$ for any $t' \neq t \in T_{\mathcal{N}}$.*

For an ϵ-TPN or LTPN $\mathcal{N}$, its *timed* (resp. *untimed*) *language* denoted $\mathcal{L}_{tw}(\mathcal{N}, \lambda)$ (resp. $\mathcal{L}_{w}(\mathcal{N}, \lambda)$) is the set of timed (resp. untimed) words over $\Sigma \setminus \{\epsilon\}$ generated by the timed (resp. untimed) transition system, by ignoring the ϵ labels. A TPN $\mathcal{N}$ from Definition 1 can be seen as the LTPN $(\mathcal{N}, \lambda)$ over Σ such that for all $t \in T_{\mathcal{N}}$, $\lambda(t) = t$, that is, λ is the identity map. An ϵ-TPN can be seen as an LTPN $(\mathcal{N}, \lambda)$ over $\Sigma = T_{\mathcal{N}} \cup \{\epsilon\}$ such that $\lambda(t) = t$ or $\lambda(t) = \epsilon$ for all $t \in T_{\mathcal{N}}$. In [3] it was shown that LTPNs are expressively as powerful as timed automata. As a consequence, we have:

Proposition 2. *[3] The universality problem for timed automata reduces to the universality problem for LTPNs, and hence universality for LTPNs is undecidable.*

We are interested in the problem of *timed robustness*, i.e.,

Definition 5. *Given two ϵ-TPNs $(\mathcal{N}, \lambda)$ and $(\mathcal{C}, \lambda')$ over Σ and a set of read arcs R from $(P_{\mathcal{C}} \times T_{\mathcal{N}}) \cup (P_{\mathcal{N}} \times T_{\mathcal{C}})$,*

- *the controlled ϵ-TPN $(\mathcal{N}, \lambda)^{(\mathcal{C},R)}$ is defined as the ϵ-TPN $(\mathcal{N}^{(\mathcal{C},R)}, \lambda'')$ over Σ where $\lambda''(t) = \lambda(t)$ for $t \in T_{\mathcal{N}}$ and $\lambda''(t) = \epsilon$ for $t \in T_{\mathcal{C}}$.*
- *the timed robustness problem asks if $\mathcal{L}_{tw}((\mathcal{N}, \lambda)^{\mathcal{C},R}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$.*

Note that the labels in $\mathcal{C}$ are ignored (i.e., replaced by ϵ), since robustness only compares labels of the ground nets. We remark that untimed robustness and even untimed equivalence are decidable for ϵ-TPNs and LTPNs, since Theorem 1 can be easily adapted to deal with ϵ or labels. We now consider timed robustness and show that this problem is undecidable for ϵ-TPNs and LTPNs.

Theorem 3. *The timed robustness problem is undecidable for ϵ-TPNs (and LTPNs).*

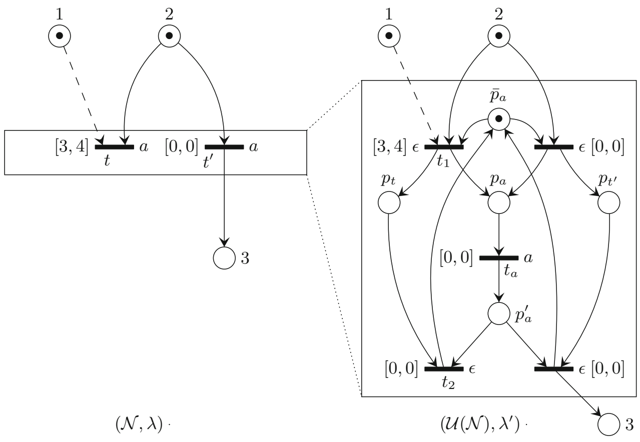


Fig. 3. Construction of a ϵ-TPN equivalent to a LTPN

The proof follows in three steps: First we show that LTPNs can be simulated by ϵ-TPNs. Thus, ϵ-TPNs are expressively as powerful as LTPNs. Then, we show that checking universality of a labeled net can be reduced to checking timed robustness of a related net. Finally, we use Proposition 2 above, which shows that checking universality of labeled nets is undecidable. We now prove the first step.

Lemma 1. *Given an LTPN $(\mathcal{N}, \lambda)$ over Σ, there exists a ϵ-TPN $(\mathcal{U}(\mathcal{N}), \lambda')$ over Σ such that $\mathcal{L}_{tw}(\mathcal{U}(\mathcal{N}), \lambda') = \mathcal{L}_{tw}(\mathcal{N}, \lambda)$.*

Proof. The construction is depicted in Figure 3. The idea is to have a unique transition t_a for each letter a which is urgent and will be fired for each transition labeled a in the original net, and use ϵ-transitions (and extra places) to capture the timing constraints on the different transitions (of the original net) labeled by a. Note that the place $\bar{p}_a$ is included in addition to ensure that the resulting net remains 1-safe.

Formally, given a LTPN $(\mathcal{N}, \lambda)$ over Σ, we construct the ϵ-TPN $(\mathcal{U}(\mathcal{N}), \lambda')$ as follows. We split each transition $t \in T_{\mathcal{N}}$ into two transitions t_1 and t_2 and also add a place p_t in $\mathcal{U}(\mathcal{N})$. Further for each action $a \in \Sigma$, such that $\lambda(\widehat{t}) = a$ for some transition $\widehat{t} \in T_{\mathcal{N}}$, we add three places $p_a, p'_a, \bar{p_a}$ and a transition t_a. Then

- we replace every incoming edge into t in $\mathcal{N}$, say (p, t) for some p, by the edge (p, t_1) in $\mathcal{U}(\mathcal{N})$.
- we replace every outgoing edge from t in $\mathcal{N}$, say (t, p') for some p', by the edge (t_2, p') in $\mathcal{U}(\mathcal{N})$.
- in $\mathcal{U}(\mathcal{N})$, we add edges from t_1 to p_t, from t_1 to p_a, from p_a to t_a,
- from t_a to p'_a and from p'_a to t_2. We also add an edge from $\bar{p_a}$ to each transition t_1 and from t_2 to $\bar{p_a}$ such that t is labeled by a. Note that this procedure is applied for each action a and every transition t labeled by a, so as a result, we can obtain a net with several outgoing edges from p'_a or incoming edges to p_a.
- Finally, for the timing constraints, we assign to each t_1 in $\mathcal{U}(\mathcal{N})$ the constraint $I(t)$ assigned to t in $\mathcal{N}$. All other transitions of $\mathcal{U}(\mathcal{N})$ are assigned the constraint $[0, 0]$, hence forcing them to be urgent.

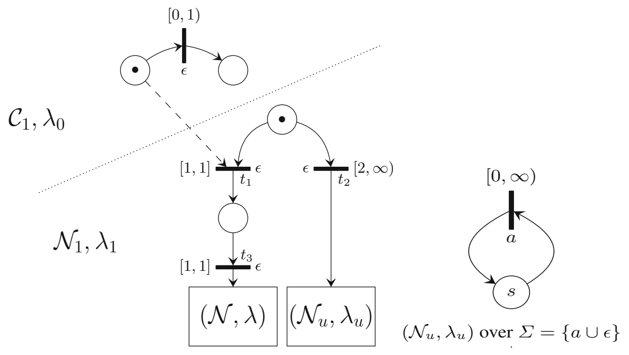

$[0,1)$
ϵ
$\mathcal{C}_1, \lambda_0$
$[1,1]$
t_1
ϵ
ϵ
$[2,\infty)$
t_2
$[0,\infty)$
a
$\mathcal{N}_1, \lambda_1$
t_3
$[1,1]$
ϵ
s
$(\mathcal{N}, \lambda)$
$(\mathcal{N}_u, \lambda_u)$
$(\mathcal{N}_u, \lambda_u)$ over $\Sigma = \{a \cup \epsilon\}$

Fig. 4. Reducing checking universality of LTPN to checking robustness of a new ϵ-TPN

Then, λ' is defined by $\lambda'(t_a) = a$ for each t_a, i.e., the transition of $\mathcal{U}(\mathcal{N})$ that was added above for each $a \in \Sigma$, and $\lambda'(\widetilde{t}) = \epsilon$ for all other transitions $\widetilde{t}$ of $\mathcal{U}(\mathcal{N})$. By construction, $(\mathcal{U}(\mathcal{N}), \lambda')$ is uniquely labeled. Each transition t of $\mathcal{N}$ is simulated by a sequence of transitions t_1, t_a, t_2 (place $\bar{p_a}$ ensures atomicity of this sequence). Then we can easily show that $\mathcal{L}_{tw}(\mathcal{U}(\mathcal{N}), \lambda') = \mathcal{L}_{tw}(\mathcal{N}, \lambda)$. □

Next we show a reduction from universality for LTPNs to timed robustness for ϵ-TPNs.

Lemma 2. *The universality problem for LTPNs can be reduced to the timed robustness problem for ϵ-TPNs.*

Proof. We use a gadget net $(\mathcal{N}_u, \lambda_u)$ which accepts the universal language of timed words over Σ. Such a net is shown in Figure 4 (right). The net depicted in the figure is only over the single discrete alphabet a, but we can by replicating it obtain the universal net over any finite alphabet. Now, as shown in Figure 4 (left), we construct a ground net $(\mathcal{N}_1, \lambda_1)$ which starts with a place and chooses between accepting the timed language of the LTPN $(\mathcal{N}, \lambda)$ and the universal language by using $(\mathcal{N}_u, \lambda_u)$.

Formally, this is defined by adding arcs from the last transition on the left (resp. right) side to the places in the initial marking of $\mathcal{N}$ (resp. $\mathcal{N}_u$). Now by adding disjoint time constraints $[1, 1]$ and $[2, \infty)$ on the transitions, we ensure that $(\mathcal{N}_1, \lambda_1)$ always chooses the left transition t_1 and hence, in the absence of controller, the language accepted is $L_1 = \{(w_1, d_1) \dots (w_n, d_n) \in (\Sigma \times \mathbb{R}^+)^* \mid (w_1, d_1 - 2) \cdots (w_n, d_n - 2) \in \mathcal{L}_{tw}(\mathcal{N}, \lambda)\}$, i.e., the timed language of $(\mathcal{N}, \lambda)$ delayed by 2. In the presence of the controller $(\mathcal{C}_1, \lambda_0)$, only transition t_2 can be fired (as t_1 is disabled by the controller) and hence, the language accepted is $L_2 = \{(w_1, d_1) \dots (w_n, d_n) \in (\Sigma \times \mathbb{R}^+)^* \mid (w_1, d_1 - 2) \cdots (w_n, d_n - 2) \in \mathcal{L}_{tw}(\mathcal{N}_u, \lambda_u)\}$ - the universal language delayed by 2.

Then checking timed robustness corresponds to checking if $L_2 \subseteq L_1$, and checking $L_2 \subseteq L_1$ reduces to checking that $\mathcal{L}_{tw}(\mathcal{N}, \lambda)$ contains the universal language, or equivalently if $\mathcal{L}_{tw}(\mathcal{N}, \lambda)$ is universal, which is undecidable. Note that $(\mathcal{N}_1, \lambda_1)$ is not uniquely labeled since every action a definitely occurs in $(\mathcal{N}_u, \lambda_u)$ and may also occur more than once in $(\mathcal{N}, \lambda)$. Thus the above proof only shows that checking timed robustness for LTPNs is undecidable. But now, using Lemma 1, we can build the ϵ-TPN $(\mathcal{U}(\mathcal{N}_1), \lambda'_1)$ over Σ, with the same timed language as $(\mathcal{N}_1, \lambda_1)$. Hence by the above argument checking timed robustness of ϵ-TPNs is also undecidable. □

Checking if $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)}) = \mathcal{L}_{tw}(\mathcal{N})$, i.e., timed language equivalence is a weaker notion in the context of ϵ-TPNs than in TPNs (as it only requires preserving the times of "important" observable actions). For instance in Figure 1(b), we may want to check if c can occur in the controlled net at every date at which it can occur in the ground net (even if the other ϵ-transitions are perturbed). Unfortunately, we easily obtain the undecidability of this problem as an immediate corollary of the above theorem, and even in restricted settings (see [1]).

5 Ensuring Robustness in TPNs with Silent Transitions

The situation for ϵ-TPNs is unsatisfactory since checking timed robustness is undecidable. Hence, we are interested in restrictions that make this problem decidable, or in ensuring that this property is met by construction. In this section, we will show that we can restrict the controlling set of read-arcs to ensure that a net is always timed robust. Indeed, it is natural to expect that a "good" controller never introduces new behaviors and we would like to ensure this.

We consider the restriction in which all transitions of the ground nets that have controller places in their preset are not urgent, i.e., the time constraint on the transition is $[\alpha, \infty)$ or (α, ∞) for some $\alpha \in \mathbb{Q}^+$. We call such controlled nets *R-restricted ϵ-TPNs*. We now show that R-restricted ϵ-TPNs are always timed robust (as in the case of untimed PNs shown in Proposition 1). That is,

Theorem 4. *Let $\mathcal{N}$ and $\mathcal{C}$ be two ϵ-TPNs, and R be a set of read arcs such that for every $(p,t) \in R \cap (P_\mathcal{C} \times T_\mathcal{N})$, $I(t)^+ = \infty$, then $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$.*

Proof. We start with some notations. Let $q^{(\mathcal{C},R)}$ be a state of $\mathcal{N}^{(\mathcal{C},R)}$ and $\rho^{(\mathcal{C},R)}$ be a dated run of $\mathcal{N}^{(\mathcal{C},R)}$. We denote by $\check{p}_\mathcal{N}(q^{(\mathcal{C},R)})$ the projection of $q^{(\mathcal{C},R)}$ obtained as follows: we keep in the state description, only places of the ground net and clocks associated with uncontrollable transitions of the ground net. Note that the obtained state is described by the same variables as a state of $\mathcal{N}$ but a priori, may not be reachable in the ground net $\mathcal{N}$. Similarly, we denote by $\check{p}_\mathcal{N}(\rho^{(\mathcal{C},R)})$ the projection of a dated run of $\mathcal{N}^{(\mathcal{C},R)}$ onto the variables of $\mathcal{N}$ i.e. onto transitions of the ground net and states as defined above. Finally, we denote the last state of the dated run ρ by $last(\rho)$.

We will now prove that for any dated run $\rho^{(\mathcal{C},R)}$ of $\mathcal{N}^{(\mathcal{C},R)}$, there exists a dated run ρ of $\mathcal{N}$ such that $\rho = \check{p}_\mathcal{N}(\rho^{(\mathcal{C},R)})$. The proof is done by induction on the number of transitions in the dated runs. The property obviously holds with no actions (same initial states: $q_0 = \check{p}_\mathcal{N}({q_0}^{(\mathcal{C},R)})$). Suppose it holds upto $n \geq 0$. Now, consider some run ${\rho'}^{(\mathcal{C},R)} = \rho^{(\mathcal{C},R)} \xrightarrow{(d,a)} {q_f}^{(\mathcal{C},R)}$ of $\mathcal{N}^{(\mathcal{C},R)}$ such that $\rho^{(\mathcal{C},R)}$ has n transitions.

By induction hypothesis, there exists run ρ of $\mathcal{N}$ such that $\rho = \check{p}_\mathcal{N}(\rho^{(\mathcal{C},R)})$. Now consider transition t (occuring at date d): either $t \in T_\mathcal{C}$ is a transition of the controller and hence is silent in the controlled net by definition, so we can discard it, and $\check{p}_\mathcal{N}({\rho'}^{(\mathcal{C},R)}) = \rho$; or $t \in T_\mathcal{N}$ is a transition of the ground net and two cases may arise:

- either t is not controlled, then, two new cases may arise:
  (1) either no controlled transitions are in conflict with t in $\mathcal{N}^{(\mathcal{C},R)}$ and since $last(\rho) = \check{p}_\mathcal{N}\big(last(\rho^{(\mathcal{C},R)})\big)$, it can occur in the ground net at the same date d;

(2) or a controlled transition t' is in conflict with t in $\mathcal{N}^{(\mathcal{C},R)}$ and the controller blocks t' and allows the firing of t. But then, by definition, we have $I(t')^+ = \infty$. Thus, t' is not urgent in the ground net, i.e., it is always possible to delay it and hence fire t' at a date greater than d in the ground net. As a result t can be fired in the ground net at date d leading to a state $q_f = \check{p}_{\mathcal{N}}({q_f}^{(\mathcal{C},R)})$;

– or it is controlled, and then we have $I(t)^+ = \infty$ and so $I(t) = [\alpha, \infty)$ (or (α, ∞), but this is handled similarly so we only consider the closed case) for some $\alpha \in \mathbb{Q}^+$. Then, by induction hypothesis the previous transition that newly enables t in the ground net and the controlled net were fired at the same date d'. Thus
  - if there is no transition in conflict with t in the ground net, then in the controlled net $\mathcal{N}^{(\mathcal{C},R)}$, for the run $last(\rho^{(\mathcal{C},R)}) \xrightarrow{(d,a)} {q_f}^{(\mathcal{C},R)}$, we are guaranteed that $d - d' \geq \alpha$. In ρ, several transitions of the controller may disable and then re-enable t, hence allowing d to be any date greater that $d' + \alpha$. However, as $I^+(t) = \infty$, for every choice of d in the controlled net, firing t at date d is allowed in the ground net, leading to a state $q_f = \check{p}_{\mathcal{N}}({q_f}^{(\mathcal{C},R)})$ as before.
  - Potential conflicts are handled by observing that any delay forced on the ground net will also be forced on the controlled net. More precisely, if there are transitions in conflict with t in the ground net $\mathcal{N}$, the problematic cases are when they either (i) disable t due to urgency or (ii) force t to be delayed by an arbitrary amount possibly greater than α (for instance, a conflicting transition may empty and refill the preset of t after α time units) in $\mathcal{N}$. But now any delay in firing of t forced on the ground net $\mathcal{N}$ will also be forced on the controlled net $\mathcal{N}^{(\mathcal{C},R)}$. Thus, if t is either disabled or forced to be delayed beyond d in $\mathcal{N}$, then in $\mathcal{N}^{(\mathcal{C},R)}$ too it will be disabled/ forced to delay beyond d which contradicts the assumption that t was firable in $\mathcal{N}^{(\mathcal{C},R)}$ at date d. Thus the delay forced in $\mathcal{N}^{(\mathcal{C},R)}$ can only be more than the delay forced in $\mathcal{N}$ and hence t is firable at date d in $\mathcal{N}^{(\mathcal{C},R)}$ implies (due to $I(t)^+ = \infty$) that t is firable at date d in $\mathcal{N}$.

Then there exists a run $\rho' = \rho \xrightarrow{(d,a)} q_f$ of $\mathcal{N}$ such that $q_f = \check{p}_{\mathcal{N}}({q_f}^{(\mathcal{C},R)})$ which concludes the induction. □

Note that while timed robustness is ensured for nets and control schemes that are R-restricted, timed equivalence remains undecidable for such nets (see [1] for details). The R-restricted condition in Theorem 4 is quite strong, but relaxing it rapidly leads to undecidability:

Proposition 3. *The timed robustness problem is undecidable for ϵ-TPNs with at least one read arc from a place of the controller to any transition t of the ground net such that $I(t)^+ \neq \infty$.*

Proof. The proof of Theorem 3 actually gives the result if t is a silent transition. Now, if t is a non-silent transition, then that proof does not work off-the-shelf anymore and we need to modify the construction of Fig. 4. The resulting net is shown in Fig. 5.

As before, $(\mathcal{N}, \lambda)$ is any LTPN on some alphabet Σ and $(\mathcal{N}_u, \lambda_u)$ is an ϵ-TPN universal on Σ. Apart from those components, $(\mathcal{N}_1, \lambda_1)$ contains only one non-silent transition ($a \notin \Sigma$). This transition furthermore has a controller place in its preset and its

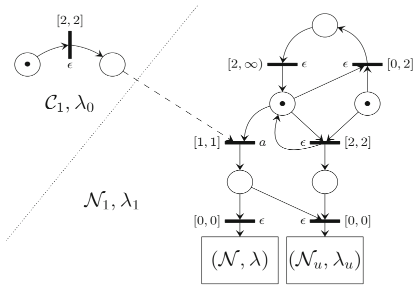


Fig. 5. Reducing universality to robustness in an ϵ-TPN

time interval has a finite upper bound. So, using Lemma 1, $\mathcal{N}_1$ and $\mathcal{N}_1^{(\mathcal{C},R)}$ can indeed be transformed into ϵ-TPNs satisfying our relaxed condition.

From the intial configuration, transition a can fire exactly at date 1. The two ϵ transitions at the top of the ground net simulate an arbitrary delay greater than 2, which can occur only once before firing a. Hence, the timed language of the ground net is the empty word plus the set of all the words of the form $(a, x)w$ with $x = 1$ or $x \geq 3$ and w is either the empty word or any timed word in $\mathcal{L}_{tw}(\mathcal{N}, \lambda)$ delayed by x time units.

Similarly, in the controlled net, a can only fire at a date greater than 2. So, the timed language of the controlled net is the empty word plus the set of all the words of the form $(a, x)w$ with $x \geq 3$ and w is either the empty word or any timed word in $\mathcal{L}_{tw}(\mathcal{N}, \lambda)$ delayed by x time units, or of the form $(a, 3)w'$ where w' is either the empty word or any timed word in $\mathcal{L}_{tw}(\mathcal{N}_u, \lambda_u)$ delayed by 3 time units. Thus, the net is *timed robust* iff $\mathcal{L}_{tw}(\mathcal{N}_u, \lambda_u) \subseteq \mathcal{L}_{tw}(\mathcal{N}, \lambda)$, i.e., iff $(\mathcal{N}, \lambda)$ is universal. □

6 A Small Case Study

We consider a heater-cooler system depicted in Figure 6, which improves the hardness of a particular material by first heating and then cooling it. The heater-cooler is equipped with two sensors: *Toohot* is raised when the heater reaches its maximal temperature. If it occurs, the heating stops automatically. *Cold* is raised when the temperature is cold enough in the cooling stage. If it occurs, the cooler stops automatically. The heater-cooler starts in the $heating$ state and the operator can push the $StartCooling$ button if the constraints of the system allow it.

We assume architectural constraints imposing that the $StartCooling$ action is not allowed after 20 t.u. in the heating stage, and if the $toohot$ sensor has been raised, then it cannot occur before 120 t.u. The constraints are encoded as a controller $\mathcal{C}$, and read arcs as shown in Figure 6.

We can show that $\mathcal{L}_w(\mathcal{N}^{\mathcal{C},R}) = \mathcal{L}_w(\mathcal{N})$. Hence, $\mathcal{N}$ is untimed robust and even untimed equivalent under $(\mathcal{C}, R)$. The net $\mathcal{N}$ is not an ϵ-TPN (the $StartCooling$ action label occurs twice), but can be converted to an ϵ-TPN (by Lemma 1). The resulting net is R-restricted, so according to Theorem 4, we have $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)}) \subseteq \mathcal{L}_{tw}(\mathcal{N})$ and therefore $\mathcal{N}$ is timed robust under $(\mathcal{C}, R)$.

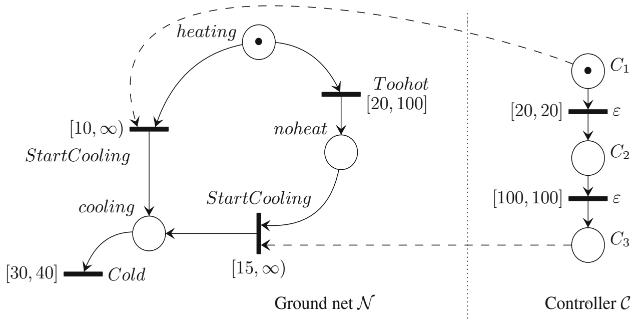


Fig. 6. Case Study

7 Conclusion and Discussion

We have defined and studied notions of timed and untimed robustness as well as untimed equivalence for time Petri nets. We are interested in checking or/and guaranteeing these properties. The results are summarized in the table below.

	TPN	R-restricted ϵ-TPN	ϵ-TPN	LTPN
Untimed robustness	Pc (thm 1)	G (thm 4)	Pc	Pc
Untimed equivalence	Pc (cor 1)	Pc	Pc	Pc
Timed robustness	D (thm 2)	G (thm 4)	U (thm 3)	U (thm 3)

U stands for undecidable, D for decidable, Pc for PSPACE-complete, and G for guaranteed.

Overall, with injective labels and no ϵ, robustness is decidable. We believe that the timed robustness problem is also PSPACE-complete (as is the case for the other decidable problems), but we leave the formal development of this complexity analysis for future work. From a modeling perspective it is important to allow silent transitions. With silent transitions, the untimed properties are still decidable, but timed properties become undecidable. To overcome this problem, we proposed a sufficient and practically relevant condition to guarantee timed robustness which we showed is already at the border of undecidability. We also showed that while untimed equivalence is easily decidable in all these cases, timed equivalence is undecidable in most cases. This is not really a surprise nor a limitation, as asking preservation of timed behaviors under architectural constraints is a rather strong requirement.

As further discussion, we remark that other criteria can be used for comparing the controlled and ground nets such as (timed) bisimulation or weak bisimulation. While these would be interesting avenues to explore, they seem to be more restrictive and hence less viable from a modeling perspective. Possible extensions could be to define tractable subclasses of nets, for instance by considering semantic properties of the net rather than syntactic conditions to ensure decidability. It would also be interesting to consider robustness of nets *up to some small delay*. Formally, we can fix a delay as a small positive number δ, and define $\mathcal{L}^{\delta}_{tw}(\mathcal{N}) = \{(w_1, t_1) \dots (w_n, t_n) \mid \exists (w_1, t'_1) \dots (w_n, t'_n) \in \mathcal{L}_{tw}(\mathcal{N}), \forall i \in 1 \dots n, |t'_i - t_i| \leq \delta\}$. Then a possible extension of the definitions would be to consider δ-robustness under $\mathcal{C}, R$ as the timed inclusion $\mathcal{L}_{tw}(\mathcal{N}^{(\mathcal{C},R)}) \subseteq \mathcal{L}^{\delta}_{tw}(\mathcal{N})$.

References

1. Akshay, S., Hélouët, L., Jard, C., Lime, D., Roux, O.H.: Robustness of time Petri nets under architectural constraints. Technical report (2012), http://people.rennes.inria.fr/Loic.Helouet/Papers/AHJLR12.pdf
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the Expressiveness of Timed Automata and Time Petri Nets. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 211–225. Springer, Heidelberg (2005)
4. Berard, B., Petit, A., Diekert, V., Gastin, P.: Characterization of the expressive power of silent transitions in timed automata. Fundam. Inform. 36(2-3), 145–182 (1998)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Transactions on Software Engineering 17(3), 259–273 (1991)
6. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. Theoretical Computer Science 147(1-2), 117–136 (1995)
7. Gardey, G., Roux, O.F., Roux, O.H.: Safety control synthesis for time Petri nets. In: 8th International Workshop on Discrete Event Systems (WODES 2006), pp. 222–228. IEEE Computer Society Press, Ann Arbor (2006)
8. Giua, A., DiCesare, F., Silva, M.: Petri net supervisors for generalized mutual exclusion constraints. In: Proc. 12th IFAC World Congress, Sidney, Australia, pp. 267–270 (July 1993)
9. Holloway, L.E., Krogh, B.H.: Synthesis of feedback control logic for a class of controlled Petri nets. IEEE Trans. on Automatic Control 35(5), 514–523 (1990)
10. Lime, D., Roux, O.H.: Model checking of time Petri nets using the state class timed automaton. Journal of Discrete Events Dynamic Systems - Theory and Applications (DEDS) 16(2), 179–205 (2006)
11. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
12. Ramadge, P., Wonham, W.: The control of discrete event systems. Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems 77(1), 81–98 (1989)
13. Uzam, M., Jones, A.H., Yucel, I.: Using a Petri-net-based approach for the real-time supervisory control of an experimental manufacturing system. Journal of Electrical Engineering and Computer Sciences 10(1), 85–110 (2002)
14. Valero, V., Frutos-Escrig, D., Cuartero, F.: On non-decidability of reachability for timed-arc Petri nets. In: Proc. 8th International Workshop on Petri Nets and Performance Models, PNPM 1999 (1999)
15. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Formal Methods in System Design 33(1-3), 45–84 (2008)
16. Xu, D., He, X.H., Deng, Y.: Compositional schedulability analysis of real-time systems using time Petri nets. IEEE Transactions on Software Engineering 28(10), 984–996 (2002)

Toward a Timed Theory of Channel Coding[⋆]

Eugene Asarin[1], Nicolas Basset[2], Marie-Pierre Béal[2], Aldric Degorre[1], and Dominique Perrin[2]

[1] LIAFA, University Paris Diderot and CNRS, France
[2] LIGM, University Paris-Est Marne-la-Vallée and CNRS, France

Abstract. The classical theory of constrained-channel coding deals with the following questions: given two languages representing a source and a channel, is it possible to encode source messages to channel messages, and how to realize encoding and decoding by simple algorithms, most often transducers. The answers to this kind of questions are based on the notion of entropy.

In the current paper, the questions and the results of the classical theory are lifted to timed languages. Using the notion of entropy of timed languages introduced by Asarin and Degorre, the question of timed coding is stated and solved in several settings.

1 Introduction

This paper is the first attempt to lift the classical theory of constrained-channel coding to timed languages.

Let a language S represent all the possible messages that can be generated by a *source*, and C all the messages that can transit over a *channel*. Typical problems addressed by coding theory are:

– Is it possible to transmit any source generated message via the channel?
– What would be the transmission speed?
– How to encode the message before and to decode it after transmission?

The answers given by the theory of channel coding are as follows: to each language L is associated a non-negative real number $h(L)$, called its entropy, which characterizes the quantity of information in bits per symbol. In order to transmit information in real-time (resp. with speed α) the entropy of the source should not exceed the one of the channel: $h(S) \le h(C)$ (resp. $\alpha h(S) \le h(C)$). For regular (or more precisely sofic) languages, whenever the information inequalities above are strict, the theory of channel coding provides a transmission protocol with a simple encoding and decoding (realized by a finite-state transducer). For the practically important case when $S = \Sigma^*$ and $h(S) < h(C)$, the decoding can be made even simpler (sliding-window). A typical example is EFMPlus code [12] allowing writing any binary file (i.e. the source $\{0,1\}^*$, with entropy 1) onto a

[⋆] The support of Agence Nationale de la Recherche under the project EQINOCS (ANR-11-BS02-004) is gratefully acknowledged.

DVD (the channel $C = (2, 10)$ – RLL admits all the words without factors 11, 101 and 0^{11}, its entropy is 0.5418, see [9]) with almost optimal rate $\alpha = 1/2$.

Classical theory of channel coding deals with discrete messages. It is, however, important to consider data words, i.e. discrete words augmented with data, e.g. real numbers. In this paper, we develop the theory of channel coding for the most studied class of data languages: timed languages. Several models of information transmission are possible for the latter:

– the source is a timed language; the channel is a discrete language. In this case, lossless encoding is impossible, and we will consider encoding with some precision ε;
– the source and the channel are timed languages, we are interested in exact (lossless) encoding;
– the source and the channel are timed languages, some scaling of time data is allowed.

Our solution will be based on the notion of entropy of timed languages [3]. For several models of transmission of timed data we will write *information inequalities* relating entropies of sources and channels with parameters of encodings (rate, precision, scaling, see below). Such an inequality is a necessary condition for existence of an encoding. On the other hand, whenever the information inequality holds (in its strict form) and the languages are regular (sofic), we give an explicit construction for simple timed encoding-decoding functions.

Related work. Constrained-channel coding theory for finite alphabets is a well-established domain; we refer the reader to monographs [13, 9, 8], handbook chapters [14, 7] and references therein. We started exploring information contents of timed languages in [2–4] where the notion of entropy was introduced and related to information measures such as Kolmogorov complexity and ε-entropy. Technically, we strongly build on discretization of timed languages, especially [3, 6]. In a less formal way, a vision of timed languages as a special kind of languages with data [11, 10] was another source of inspiration.

Paper structure. In Sect. 2 we recall basic notions of the discrete theory of constrained-channel coding. In Sect. 3 we briefly recall some results and constructions on volume, entropy and discretization of timed languages. In Sect. 4 we state our main results on timed theory of constrained-channel coding. In Sect. 5 we discuss the rationale, perspectives and applications of this work.

2 Theory of Channel Coding for Finite Alphabet Languages

In this section we give an elementary exposition[1] of some basic notions and results from the theory of constrained-channel coding, see [14, 13, 8, 7] for more details.

[1] We avoid here the terminology of symbolic dynamics, standard in the area of coding.

2.1 Terminology

Let Σ be a finite alphabet. A *factor* of a word $w \in \Sigma^*$ is a contiguous subsequence of its letters. A language L is *factorial* whenever for each word $w \in L$ every factor of it is in the language. A language L is *extensible* whenever for each word $w \in L$ there exists a letter $a \in \Sigma$ such that $wa \in L$.

These two conditions are usual in the context of coding and can be justified in practice as follows. If we can encode (decode) some long word w (e.g. a movie file), then we want also to encode (decode) its contiguous fragments (e.g. a short scene in the middle of the movie). On the other hand, some extension of w should correspond to a longer movie.

A language L, which is regular, factorial and extensible, is called *sofic*. Sofic languages can be recognized by finite automata whose states are all initial and final (this ensures factoriality) and all have outgoing transitions (this ensures extensibility).

Given a language L, we denote by L_n its sublanguage of words of length n. The *entropy* $h(L)$ of a language over a finite alphabet is the asymptotic growth rate of the cardinality of L_n, formally defined by (all the logarithms are base 2):

$$h(L) = \lim_{n\to\infty} \frac{1}{n} \log |L_n|.$$

The limit (finite or $-\infty$) always exists if L is factorial. For a sofic language L recognized by a given automaton, its entropy $h(L)$ can be effectively computed using linear algebra. In particular if $L = \Sigma^*$ for a k-letter alphabet Σ then $h(L) = \log k$. Finally, the intuitive meaning of the entropy is the amount of information (in bits per symbol) in typical words of the language.

Most of our coding functions have a special property defined below.

Definition 1 (almost injective). *A (partial) function $\phi : \Sigma^* \to \Gamma^*$ is called* almost injective *with delay $d \in \mathbb{N}$, if for any n and $w, w' \in \Sigma^n$, and $u, u' \in \Sigma^d$ it holds that*

$$\phi(wu) = \phi(w'u') \Rightarrow w = w'.$$

Intuitively, if such a function is used to encode messages, then knowing the code of some message wu one can decode w, i.e. the whole message except its last d symbols. Thus the decoding is possible with delay d. This can be formalized as follows:

Definition 2 (almost inverse). *For an almost injective function $\phi : \Sigma^* \to \Gamma^*$ with delay d its d-*almost inverse *family of functions $\psi_n : \Gamma^* \to \Sigma^n$ is characterized by the following property: for any $w \in \Sigma^n$ and $v \in \Gamma^*$,*

$$w = \psi_n(v) \Leftrightarrow \exists u \in \Sigma^d : \ \phi(wu) = v.$$

Lemma 1. *If the domain of an almost injective function ϕ is extensible and ψ_n is its almost inverse, then ψ_n is a surjection to this domain (constrained to words of length n).*

2.2 Coding: The Basic Case

Let A and A' be two alphabets (source and channel alphabets), $S \subset A^*$ and $C \subset A'^*$ factorial extensible languages and $d \in \mathbb{N}$. The aim is to encode any source message $w \in S$ to a channel message $\phi(w) \in C$. The latter message can be transmitted over the channel.

Definition 3. *An (S, C)-encoding with delay d is a function $\phi : S \to C$ (total on S but not necessarily onto C) such that*

- *it is length preserving: $\forall w \in S,\ |\phi(w)| = |w|$,*
- *it is almost injective with delay d.*

The first condition means that the information is transmitted in real-time (with the transmission rate 1). The second one permits decoding.

A natural question is to find necessary and sufficient conditions on S and C for an (S, C)-encoding (with some delay) to exist. This question can be addressed by comparing the entropy of the languages S and C. Roughly, the channel language should contain at least as much information per symbol as the source language. Formally, we define the *information inequality*:

$$h(S) \leq h(C). \tag{II1}$$

Proposition 1. *Let S and C be factorial and extensible languages. If an (S, C)-encoding exists then (II1) necessarily holds.*

Proof. Let $\phi : S \to C$ be an (S, C)-encoding with delay d. By Lemma 1 its almost inverse ψ maps C onto S. More precisely, for every n we have $\psi_n(C_{n+d}) = S_n$. Hence, the cardinalities should satisfy: $|S_n| \leq |C_{n+d}|$. Finally we have

$$\lim_{n\to\infty} \frac{1}{n} \log |S_n| \leq \lim_{n\to\infty} \frac{1}{n} \log |C_{n+d}|$$

and the expected inequality $h(S) \leq h(C)$. $\square$

Thus, (II1) is necessary for existence of the coding. For sofic languages (if the inequality is strict) it is also sufficient. Moreover, the encoding can be realized by a sort of finite-state machine. We present this fundamental result in the following form which is essentially the finite-state coding theorem from [14], Theorem 10.3.7 in [13].

Theorem 1. *Let S and C be sofic languages. If the strict version of (II1) holds, then there exists an (S, C)-encoding realized by a finite-state transducer which is right-resolving on input and right-closing on output*[2].

The reader is now motivated to get through a couple of definitions.

Definition 4 (transducer). *A transducer is a tuple $\tau = (Q, A, A', \Delta, \mathcal{I}, \mathcal{O})$ with a finite set Q of control states; finite input and output alphabets A and A'; a set of transitions Δ (each transition δ has a starting state* $\texttt{ori}(\delta)$ *and an ending state* $\texttt{ter}(\delta)$*); input and output labeling functions $\mathcal{I} : \Delta \to A$ and $\mathcal{O} : \Delta \to A'$.*

[2] Such a transducer is also called a finite-state (S, C)-encoder.

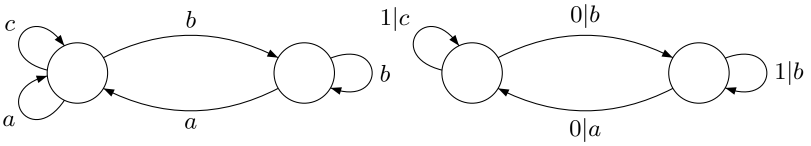


Fig. 1. A sofic automaton of a channel C and a $(\{0,1\}^*, C)$-encoder

The transducer is said to be *right-resolving* on input whenever for each state $q \in Q$ every two different edges starting from q have different input labels. For such a transducer, the input automaton with a fixed initial state i, i.e. $\mathcal{A}_i = (Q, A, \Delta, \mathcal{I}, i, Q)$ is deterministic, we denote by S_i the language of this automaton.

The transducer is *right-closing* on output with delay d whenever every two paths π and π' of length $d+1$ with the same output label and the same starting state q always have the same initial edge $\pi_1 = \pi'_1$.

A transducer τ satisfying both properties performs the encoding process in a natural way: an input word w of S is read from a state i along a path π_w, and this path determines the output word $\mathcal{O}(\pi_w)$. The function $\phi_i : S_i \to A'^*$ defined as $w \mapsto \mathcal{O}(\pi_w)$ is length preserving and almost injective with delay d.

Example 1. Consider the source language $S = \{0,1\}^*$ and the channel language C recognized by the sofic automaton on the left of Fig. 1. The language C is composed by all the words on $\{a,b,c\}$ that do not contain any block bc. The entropy of the source is $h(S) = 1$, and the one of the channel is $h(C) = 2\log\left[(1+\sqrt{5})/2\right] \approx 1.3885$. The information inequality $h(S) < h(C)$ holds and we can encode S in C using the transducer on the right of Fig. 1.

2.3 Other Coding Settings

Similarly to the previous section, other coding settings can be considered. For example, we can transmit information over a channel with some rate $\alpha = \frac{p}{q}$, when q letters of the channel message correspond to p letters of the source message (the previous section corresponds thus to the case $\alpha = 1$).

Definition 5. *An (S,C)-encoding with rate $\alpha \in \mathbb{Q}^+$ and delay d is a function $\phi : S \to C$ (total on S and not necessarily onto C) such that*

- *it is of rate α, i.e. $\forall w \in S$, $\lceil \alpha|\phi(w)|\rceil = |w|$;*
- *it is almost injective (with delay d).*

In this setting, the information inequality takes the form:

$$\alpha h(S) \le h(C), \tag{II2}$$

and it is a necessary and almost sufficient condition for the code to exist:

Proposition 2. *Let S and C be factorial and extensible languages. If an (S,C)-encoding with rate α exists, then ([112]) necessarily holds.*

Proposition 3. *Let S and C be sofic languages. If a strict inequality ([112]) holds, then an (S,C)-encoding of rate α exists. Moreover, it can be realized by a finite-state transducer of rate α.*

We skip here a natural definition of such a transducer.

3 Preliminaries on Timed Languages

3.1 Timed Alphabets, Words, Languages, and Automata

We call *k-M-alphabet* a set $A = [0,M] \times \Sigma$ where Σ is a k-letter alphabet and M a positive integer bound, so that every letter in A corresponds to a real-valued delay (seen as data) in $[0,M]$ and a discrete event in Σ. Timed words and languages are respectively words and languages over a k-M-alphabet. We define factor-closed and extensible timed language as in the untimed case.

Timed automata as described below can be used to define timed languages, which are called *regular*. First we note $G_{c,M}$ the set of all c-dimensional M-bounded rectangular integer guards, i.e. Cartesian products of c real intervals I_i, $i \in \{1..c\}$, having integer bounds in $\{0..M\}$. A *timed automaton* is thus a tuple $\mathcal{A} = (Q, c, M, \Sigma, \Delta, I, F)$ where Q is a finite set of locations; $c \in \mathbb{N}$ is the number of clocks; $M \in \mathbb{N}$ is an upper bound on clock values; Σ is a finite alphabet of events; $\Delta \subseteq Q \times Q \times \Sigma \times G_{c,M} \times 2^{\{1..c\}}$ is a set of transitions; $I : Q \to G_{c,M}$ maps each location to an initial constraint; $F : Q \to G_{c,M}$ maps each location to a final constraint. A transition $\delta = \langle \mathtt{ori}(\delta), \mathtt{ter}(\delta), \mathcal{L}(\delta), \mathfrak{g}(\delta), \mathfrak{r}(\delta) \rangle \in \Delta$ is such that $\mathtt{ori}(\delta)$ is its origin location, $\mathtt{ter}(\delta)$ is its ending location, $\mathcal{L}(\delta)$ is its label, $\mathfrak{g}(\delta)$ is the guard that clock values must satisfy for firing δ and $\mathfrak{r}(\delta)$ is the set of clocks reset by firing it.

A timed word $(t_1, a_1) \dots (t_k, a_k)$ is in the language of a timed automaton if there exists a run $(q_i, \mathbf{x}_i)_{i \in \{0..k\}}$ with $q_i \in Q$ and $\mathbf{x}_i \in [0,M]^c$, such that $\mathbf{x}_0 \in I(q_0)$, $\mathbf{x}_k \in F(q_k)$ and such that for all i, there exists $\delta_i \in \Delta$ satisfying $\mathtt{ori}(\delta_i) = q_{i-1}$, $\mathtt{ter}(\delta_i) = q_i$, $\mathcal{L}(\delta_i) = a_i$, $\mathbf{x}_{i-1} + t_i \in \mathfrak{g}(\delta_i)$ and $\mathbf{x}_i = \mathfrak{r}(\delta_i)(\mathbf{x}_{i-1} + t_i)$ (i.e. equal to $\mathbf{x}_{i-1} + t_i$ where coordinates in $\mathfrak{r}(\delta_i)$ are substituted by zeros).

A timed automaton is said to be right-resolving if outgoing transitions from the same location with the same label have pairwise incompatible guards. If we add the condition of having only one initial state we obtain the classical definition of determinism of [1]. Languages recognized by right-resolving timed automata are also said right-resolving. Right-resolving factor-closed and extensible timed regular languages are called *sofic*.

An example of timed automaton is given in Fig. 2.

3.2 Volume and Entropy

From now on A denotes a k-M-alphabet $[0,M] \times \Sigma$. Let $L \subseteq A^n$ be a timed language and $n \in \mathbb{N}$, we denote by L_n the set of timed words of length n

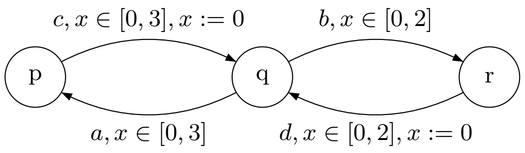


Fig. 2. A timed automaton (all the states are initial and final)

in L. For each $w = w_1 \dots w_n \in \Sigma^n$ we denote by L_w the (possibly empty) set of points $\boldsymbol{t} = (t_1, \dots, t_n) \in \mathbb{R}^n$ such that $(t_1, w_1) \dots (t_n, w_n) \in L_n$. Thus $L_n = \biguplus_{w \in \Sigma^n} L_w \times \{w\}$ (by a slight abuse of notation).

The language L is said to be measurable whenever for all $w \in \Sigma^*$, L_w is Lebesgue measurable, i.e. its (hyper-)volume $\texttt{Vol}(L_w)$ is well defined. The volume is just the n-dimensional generalization of interval length in $\mathbb{R}$, area in $\mathbb{R}^2$, volume in $\mathbb{R}^3$,... Examples of timed languages and their volume are given later, we also refer to the papers [3, 6] for more detailed examples. For a timed regular language L, languages L_w ($w \in \Sigma^*$) are just finite unions of convex polytopes (see for instance Fig. 4) and hence measurable [3]. In the sequel, all timed languages considered are measurable. The *volume* of L_n and the *volumetric entropy* of L are defined respectively as

$$\texttt{Vol}(L_n) = \sum_{w \in \Sigma^n} \texttt{Vol}(L_w); \quad \mathcal{H}(L) = \lim_{n \to \infty} \frac{1}{n} \log \texttt{Vol}(L_n),$$

and it can be shown that the limit exists for any factorial language.

For our running example on Fig. 2, $L_{ac} = \{(t_1, t_2) \mid t_1 + t_2 \leq 3\}$ and $L_{bd} = \{(t_1, t_2) \mid t_1 + t_2 \leq 2\}$; and their volumes are respectively 4.5 and 2; and thus $L_2(q)$, the sublanguage of L_2 of words accepted by runs starting from q, has volume 6.5. We have $L_{2n}(q) = (L_2(q))^n$ whose volume is 6.5^n, the entropy of the whole language $\mathcal{H}(L)$ is thus at least $0.5 \log 6.5$ (in fact it is exactly $0.5 \log 6.5$).

3.3 Discretization of Languages and Entropy

Here we adapt some definitions and results from [3, 6]. For a k-M-alphabet $A = [0, M] \times \Sigma$, we denote by A_ε its ε discretization: $A_\varepsilon = \{0, \varepsilon, 2\varepsilon, \dots, M\} \times \Sigma$. We remark that A_ε is a finite alphabet of size $k(\frac{M}{\varepsilon} + 1)$. An ε-discrete word is a timed word whose timed delays are multiples of ε. Given a timed language $L \subseteq A^*$, its ε-discretization L_ε is the discrete language on A_ε composed by ε-discrete words in L: $L_\varepsilon = L \cap {A_\varepsilon}^*$.

Given an ε-discrete word $w = (t_1, a_1) \dots (t_n, a_n)$, its ε-North-East-neighborhood is the timed set $\mathcal{B}_\varepsilon^{NE}(w) = \{(u_1, a_1) \dots (u_n, a_n) \mid u_i \in [t_i, t_i + \varepsilon],\ i = 1..n\}$. We extend this definition to associate timed languages with languages of ε-discrete words $\mathcal{B}_\varepsilon^{NE}(L_\varepsilon) = \cup_{w \in L_\varepsilon} \mathcal{B}_\varepsilon^{NE}(w)$.

We will use discretization in a three-step reduction scheme:

1. discretize the timed languages S, C with a sampling rate ε to obtain $S_\varepsilon, C_\varepsilon$;
2. use classical coding theorem 1 with $S_\varepsilon, C_\varepsilon$;
3. go back to timed languages by taking ε-NE-neighborhood of S_ε and C_ε.

The following lemma is the main tool for this reduction scheme, it is a variant of results of [3, 6]:

Lemma 2. *Let S be a timed sofic language. If $\mathcal{H}(S) > -\infty$ then for all positive small enough ε, one can compute ε-discrete sofic languages S_ε^- and S_ε^+ that verify $\mathcal{B}_\varepsilon^{NE}(S_\varepsilon^-) \subseteq S \subseteq \mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+)$, and $\mathcal{H}(S) + \log \frac{1}{\varepsilon} = h(S_\varepsilon^-) + o(1) = h(S_\varepsilon^+) + o(1)$.*

4 Timed Coding

Similarly to classical results presented in Sect. 2, we will consider several settings for transmission of timed words over a channel. For every setting we will formulate an information inequality, and show that it is necessary and, with some additional hypotheses, sufficient for a coding to exist.

4.1 Timed Source, Discrete Channel, Approximate Transmission

In practice, timed and data words are often transmitted via discrete (finite alphabet) channels. For example, a timed log of events in an operating system (a timed message) can be stored as a text file (ASCII message). The delays in the timed word cannot be stored with infinite precision, thus the coding is necessarily approximate. More precisely, the set of timed source messages w of a length n is infinite, while the set of discrete channel messages of the same length is finite. For this reason, the coding cannot be injective, and necessarily maps many timed words to a same discrete word. It is natural to require that all the timed words with the same code are ε-close to each other. This justifies Def. 6 below. We give first some notation. For two timed words of same length $w = (t_1, a_1) \dots (t_n, a_n)$ and $w = (t'_1, a'_1) \dots (t'_n, a'_n)$, the distance $\mathtt{dist}(w, w')$ between w and w' is equal to $+\infty$ if $a_1 \dots a_n \neq a'_1 \dots a'_n$, otherwise it is $\max_{1 \le i \le n} |t_i - t'_i|$. Let A be a k-M alphabet, Σ' be a finite alphabet, S be a factorial extensible measurable timed language on A, C be a factorial extensible language on Σ', and α, ε be positive reals and d be a non negative integer.

Definition 6. *Similarly to Def. 1 we say that a partial function $\phi : A^* \to \Sigma'^*$ is* almost approximately injective *with precision ε and delay d if*

$$\forall n \in \mathbb{N}, w, w' \in A^n \; \forall u, u' \in A^d : \; \phi(wu) = \phi(w'u') \Rightarrow \mathtt{dist}(w, w') < \varepsilon.$$

Its almost inverse *is a multi-valued function family $\psi_n : \Sigma'^* \to A^n$ characterized by the following property: for any $w \in A^n$ it holds that $w \in \psi_n(v)$ if and only if some $u \in A^d$ yields $\phi(wu) = v$.*

Lemma 3. *Given an almost approximately injective function ϕ with precision ε and delay d, let ψ_n be its almost inverse family. Then for every v the diameter of $\psi_n(v)$ is at most ε. If the domain of ϕ is extensible, then the image of ψ_n coincides with this domain (constrained to length n).*

Definition 7. *An (S, C)-encoding of a rational rate α, precision ε and delay d is a function $\phi : S \to C$ (total on S) such that*

- *it is of rate α: i.e. $\forall w \in S,\ \lceil \alpha |\phi(w)| \rceil = |w|$;*
- *it is almost approximately injective with precision ε and delay d.*

The information inequality for this setting has the form:

$$\alpha(\mathcal{H}(S) + \log(1/\varepsilon)) \leq h(C), \tag{II3}$$

which corresponds to information contents of S equal to $\mathcal{H}(S) + \log(1/\varepsilon)$, see the formula for Kolmogorov complexity of timed words in [3].

Proposition 4. *For a factorial extensible measurable timed language S and a factorial extensible discrete language C the following holds. If an (S, C)-encoding of rate α, precision ε, and some delay d exists then necessarily (II3) must be satisfied.*

Proof. Let ϕ be an (S, C)-encoding of rate α, precision ε and delay d, and ψ its almost inverse. By Lemma 3, for every n it holds that $S_n = \psi_n(C_{\lfloor (n+d)/\alpha \rfloor})$. This leads to an inequality on volumes

$$\texttt{Vol}(S_n) \leq \sum_{v \in C_{\lfloor (n+d)/\alpha \rfloor}} \texttt{Vol}(\psi_n(v)).$$

Any $\psi(v)$ has a diameter $\leq \varepsilon$ and thus is included in a cube of side ε and volume ε^n. We have:

$$\texttt{Vol}(S_n) \leq \varepsilon^n |C_{\lfloor (n+d)/\alpha \rfloor}|.$$

Thus

$$\frac{\alpha}{n} \log \texttt{Vol}(S_n) \leq \frac{\alpha}{n} \log \varepsilon^n |C_{\lfloor (n+d)/\alpha \rfloor}| = \alpha \log \varepsilon + \frac{\lfloor (n+d)/\alpha \rfloor}{n/\alpha} \frac{|C_{\lfloor (n+d)/\alpha \rfloor}|}{\lfloor (n+d)/\alpha \rfloor}.$$

Taking the limit as n tends to infinity we obtain $\alpha \mathcal{H}(S) \leq \alpha \log \varepsilon + h(C)$ and then (II3) holds. □

We strengthen a little bit (II3) to have a (partial) converse result for sofic timed languages.

Proposition 5. *For a sofic timed language S with $\mathcal{H}(S) > -\infty$, there exists a function R_S such that $\lim_{x \to 0} R_S(x) = 0$ and the following holds. Whenever the entropy of a sofic discrete language C verifies the inequality $\alpha(\mathcal{H}(S) + \log(1/\varepsilon) + R_S(\varepsilon)) < h(C)$, then there exists an (S, C)-encoding of rate α, precision ε and some delay d. Moreover it can be realized by a "real-time transducer" sketched below in the proof.*

Proof (Sketch). Let S be a sofic timed language. For $\varepsilon > 0$, let S_ε^+ be its ε-discretized over-approximation given by Lemma 2. We define

$$R_S(\varepsilon) = h(S_\varepsilon^+) - \mathcal{H}(S) - \log(1/\varepsilon),$$

it satisfies the required condition: $R_S(\varepsilon) = o(1)$ (see Lemma 2). Let C be a sofic discrete language such that

$$\alpha(\mathcal{H}(S) + \log(1/\varepsilon) + R_S(\varepsilon)) < h(C),$$

we prove that an (S, C)-encoding of rate α, precision ε and some delay d exists. Lemma 2 gives us

$$S \subseteq \mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+) \text{ and } \alpha h(S_\varepsilon^+) = \alpha(\mathcal{H}(S) + \log(1/\varepsilon) + R_S(\varepsilon)) < h(C).$$

Thus by Prop. 3 an (S_ε^+, C)-encoding of rate α and some delay d exists and can be realized by a finite-state transducer τ_ε of rate α and delay d. If we replace for each transition its input label $(a, k\varepsilon)$ by the label a and the guard $t \in [k\varepsilon, (k+1)\varepsilon]$, we obtain a real-time transducer τ with input $\mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+)$ whose output is in C. The injectivity of τ_ε ensures that τ realizes an approximately injective function with precision ε. $\square$

4.2 Timed Source, Timed Channel, Exact Transmission

Another natural setting is when a timed message is transmitted via a timed channel. In this case, the coding can be exact (injective). For the moment we consider length-preserving transmission (see Sect. 4.4 for faster and slower transmission).

Let A, A' be a k-M and a k'-M' alphabet, S and C factorial extensible measurable timed languages on these alphabets, and $d \in \mathbb{N}$.

Let ℓ and σ be positive rationals. A function $f : A'^n \to A^*$ is said to be ℓ-Lipshitz whenever for all x, y in its domain, $\mathtt{dist}(f(x), f(x')) \leq \ell\, \mathtt{dist}(x, x')$. We call a function σ-piecewise ℓ-Lipshitz if its restriction to each cube of the standard σ-grid on A'^n is ℓ-Lipshitz.

We can now state the definition of an (S, C)-encoding:

Definition 8. *An (S, C)-encoding with delay d (and step σ) is a function $\phi : S \to C$ such that*

- *it is length preserving: $|\phi(w)| = |w|$,*
- *it is almost injective (with delay d),*
- *no time scaling: the almost inverse ψ_n are σ-piecewise 1-Lipshitz.*

The last condition rules out a possible cheating when all the time delays are divided by 1000 before transmission over the channel. We will come back to this issue in Sect. 4.3.

The information inequality in this setting takes a very simple form:

$$\mathcal{H}(S) \leq \mathcal{H}(C). \tag{II4}$$

The necessary condition for existence of a coding has a standard form (for technical reasons we require the channel to be sofic):

Proposition 6. *If for a factorial extensible measurable timed language S and a sofic timed language C an (S, C)-encoding of delay d exists, then* (II4) *holds.*

Proof. We consider first the most interesting case when $\mathcal{H}(C) > -\infty$. We will prove that for any $\zeta > 0$ the inequality $\mathcal{H}(S) \leq \mathcal{H}(C) + \zeta$ holds. Suppose ϕ is an (S, C)-encoding of delay d (and step σ), and ψ its almost inverse. By Lemma 1, for any natural n we have $S_n \subset \psi_n(C_{n+d})$.

Since C is sofic, Lemma 2 applies, and for a fixed ϵ, each C_n can be covered by C_n^+, a union of $K_{n,\varepsilon}$ cubes of size ε with $\mathcal{H}(C) + \log\frac{1}{\varepsilon} = \lim_{n\to\infty} \frac{\log K_{n,\varepsilon}}{n} + o(1)$. We choose ε dividing σ and small enough such that

$$\mathcal{H}(C) + \log\frac{1}{\varepsilon} > \lim_{n\to\infty} \frac{\log K_{n,\varepsilon}}{n} - \zeta. \tag{1}$$

Thus we have $S_n \subset \psi_n(C_{n+d}^+)$, and, passing to volumes we get

$$\mathtt{Vol}(S_n) \leq \mathtt{Vol}(\psi_n(C_{n+d}^+)) \leq K_{n+d,\varepsilon}\varepsilon^n, \tag{2}$$

indeed, since ψ_n is 1-Lipshitz on each ε-cube, ψ_n-image of each such cube has a diameter $\leq \varepsilon$ and thus a volume $\leq \varepsilon^n$. Passing to logarithms, dividing by n and taking the limit as $n \to \infty$ in (2) we get

$$\mathcal{H}(S) \leq \lim_{n\to\infty} \frac{\log K_{n+d,\varepsilon}}{n+d} + \log\varepsilon,$$

and applying inequality (1) we obtain

$$\mathcal{H}(S) \leq \mathcal{H}(C) + \log\frac{1}{\varepsilon} + \zeta + \log\varepsilon = \mathcal{H}(C) + \zeta,$$

which concludes the proof for the case when $\mathcal{H}(C) > -\infty$. The remaining case $\mathcal{H}(C) = -\infty$ is a simple corollary of the previous one. □

As usual, when both timed languages S and C are sofic and the information inequality (II4) strict, the converse holds.

Proposition 7. *If for sofic timed languages S and C it holds that $H(S) < H(C)$, then there exists an (S, C)-encoding (with some delay d). Moreover it can be realized by a "real-time transducer" described below in the proof.*

Proof (Sketch). Let S and C be sofic timed languages whose entropies verify $-\infty < \mathcal{H}(S) < \mathcal{H}(C)$. We prove that an (S, C)-encoding with some delay d exists. Let C_ε^- and S_ε^+ be as in Lemma 2, i.e. such that

$$S \subseteq \mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+);\ \mathcal{B}_\varepsilon^{NE}(C_\varepsilon^-) \subseteq C;$$
$$\mathcal{H}(S) + \log\frac{1}{\varepsilon} = h(S_\varepsilon^+) + o(1);\ \mathcal{H}(C) + \log\frac{1}{\varepsilon} = h(C_\varepsilon^-) + o(1).$$

The discretization step ε can be chosen small enough such that $h(S_\varepsilon^+) < h(C_\varepsilon^-)$. Thus by Theorem 1 a finite-state $(S_\varepsilon^+, C_\varepsilon^-)$-encoder τ_ε exists.

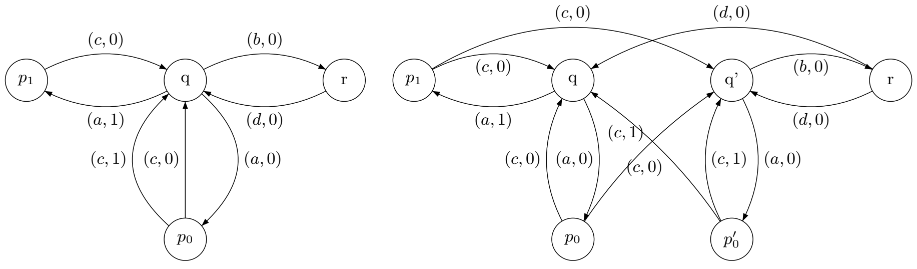


Fig. 3. Left: $\mathcal{A}_\varepsilon^-$: an automaton recognizing C_ε^- with $\varepsilon = 1$. Right: its split form.

We replace each transition δ_ϵ of τ_ϵ with input label $(a, k\varepsilon)$ and output label $(b, l\varepsilon)$ by a transition δ with input label a, guards $t \in [k\varepsilon, (k+1)\varepsilon]$, output label b and increment/decrement $c(\delta) = (l-k)\varepsilon$. We obtain what we call a *real-time transducer* τ. Its input language is $\mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+)$ and its output language is included in $\mathcal{B}_\varepsilon^{NE}(C_\varepsilon^-) \subseteq C$. The encoding is performed as follows: a timed word $(t_1, a_1) \dots (t_n, a_n)$ is in the input language if there is a path $\delta_1 \dots \delta_n$ such that $\mathcal{I}(\delta_i) = a_i$ and t_i satisfies the guard of δ_i: $t_i \in [k\varepsilon, (k+1)\varepsilon]$; the output timed word is in this case $(t_1', b_1) \dots (t_n', b_n)$ with $t_i' = t_i + c(\delta_i)$, $b_i = \mathcal{O}(\delta_i)$.

The collection of cubes $\mathcal{B}_\varepsilon^{NE}(w)$, $w \in S_\varepsilon^+$ (resp. $w \in C_\varepsilon^-$) forms a partition of timed languages $\mathcal{B}_\varepsilon^{NE}(S_\varepsilon^+)$ (resp. $\mathcal{B}_\varepsilon^{NE}(C_\varepsilon^-)$), they are cubes of the standard σ-grid with the step $\sigma = \epsilon$. Transducer τ only translates cubes. Translations are 1-Lipshitz and thus the last condition of an (S, C)-encoding holds.

The remaining degenerate case when $-\infty = \mathcal{H}(S) < \mathcal{H}(C)$ is an easy corollary of the non-degenerate one. $\square$

Table 1. The coding transducer

ori(δ)	ter(δ)	$\mathcal{I}(\delta)$	$\mathfrak{g}(\delta)$	$\mathcal{O}(\delta)$	$c(\delta)$
q	p_0	e	$[0,1]$	a	0
q	p_1	f	$[0,1]$	a	1
q'	p_0'	e	$[0,1]$	a	0
q'	r	f	$[0,1]$	b	1
p_0	q	e	$[0,1]$	c	0
p_0	q'	f	$[0,1]$	c	0
p_0'	q	e	$[0,1]$	c	1
p_0'	q'	f	$[0,1]$	c	1
p_1	q	e	$[0,1]$	c	0
p_1	q'	f	$[0,1]$	c	0
r	q	e	$[0,1]$	d	0
r	q'	f	$[0,1]$	d	0

The following example illustrates the construction of the transducer. Let the source timed language be $S = ([0,1] \times \{e, f\})^*$ and the channel timed language C be recognized by the automaton on Fig. 2. We have seen that the entropy $\mathcal{H}(C)$ is at least $0.5 \log 6.5$ and thus $\mathcal{H}(C) > \mathcal{H}(S) = \log 2$. By Prop. 7 an (S, C)-encoding exists. To realize this encoding we take $\varepsilon = 1$. There are four cubes included in the language $C_2(q)$: $([0,1] \times [0,1] \times \{ac\}$, $[0,1] \times [0,2] \times \{ac\}$, $[1,0] \times [0,1] \times \{ac\}$, $[0,1] \times [0,1] \times \{bd\})$ while the cubes to encode (language S_2) are $[0,1] \times \{ee\}$, $[0,1] \times \{ef\}$, $[0,1] \times \{fe\}$, $[0,1] \times \{ff\}$. The transducer will repeatedly map four "input cubes" to four "output cubes". We build an automaton for discrete words C_ε^- (as in Lemma 2, such words correspond to "output cubes") in Fig. 3, left. Then, as usual in coding, we first split the state p_0 and then the state q (each in two copies) to obtain an automaton with constant outdegree 2 (Fig. 3, right). This automaton accepts the same language C_ε^-, and can be transformed to the desired transducer just by adding input letters and increment/decrement to its transition. The transitions of the transducer are given in Table 1.

4.3 A Variant: Scaling Allowed

In some situations, timed data can be scaled for coding, which leads to a new term in the information inequality. Let $\lambda > 0$ be a rational bound on scaling factor. We modify Def. 8 by replacing 1-Lipshitz by $1/\lambda$-Lipshitz.

Definition 9. *An (S, C)-encoding with delay d, scaling λ and step σ is a function $\phi : S \to C$ such that*

- *it is length preserving: $|\phi(w)| = |w|$,*
- *it is almost injective (with delay d),*
- *it has scaling at most λ: the almost inverse ψ_n are σ-piecewise $1/\lambda$-Lipshitz.*

The information inequality for this case becomes:

$$\mathcal{H}(S) \le \mathcal{H}(C) + \log \lambda. \tag{II5}$$

The problem of coding with scaling can be easily reduced to the one considered in the previous section. Indeed, for a timed language C let λC be the same language with all times multiplied by λ (the entropy of this language is $\mathcal{H}(\lambda C) = \mathcal{H}(C) + \log \lambda$). A function ϕ is an (S, C)-encoding with scaling λ if and only if the "λ-scaled" function $\lambda\phi$ is an $(S, \lambda C)$-encoding without scaling. Using this reduction, the results below are corollaries of Prop. 6-7.

Proposition 8. *If for factorial extensible measurable timed language S and sofic timed C an (S, C)-encoding with scaling λ and delay d exists then (II5) holds.*

Proposition 9. *If for sofic timed languages S and C (with $\mathcal{H}(S) > -\infty$) the strict version of (II5) holds, then there exists an (S, C)-encoding with scaling λ (with some delay d). Moreover it can be realized by a "real-time transducer".*

4.4 A Speedup and a Slowdown Lead to a Collapse

For untimed channels, transmission with some rate $\alpha \neq 1$leads to the factor α in information inequalities (II2), (II3). Unfortunately, for timed channels this does not work: any rate $\alpha \neq 1$ leads to a collapse of the previous theory.

Definition 10. *An (S, C)-encoding with rational rate α, delay d and step σ is a function $\phi : S \to C$ such that*

- *its rate is α, i.e. $\forall w \in A^*$, $\lceil \alpha |\phi(w)| \rceil = |w|$;*
- *it is almost injective (with delay d);*
- *no time scaling: its almost inverse ψ is σ-piecewise 1-Lipschitz.*

For $\alpha > 1$ no coding is possible, and for $\alpha < 1$ it always exists. More precisely, the two following propositions hold.

Proposition 10. *For factorial measurable timed languages S and C if $\mathcal{H}(S) > -\infty$ and $\alpha > 1$, then no (S, C)-encoding with rate α exists.*

Proof (Sketch). The proof follows the same lines as that of Prop. 6. Suppose ϕ is such an (S, C)-encoding, and ψ its almost inverse. By Lemma 1, for any natural n we have $S_n \subset \psi_n(C_{\lfloor (n+d)/\alpha \rfloor})$. Each C_n can be covered by C_n^+, a union of $K_{n,\varepsilon}$ cubes of size ε satisfying inequality (1) We have $S_n \subset \psi_n(C^+_{\lfloor (n+d)/\alpha \rfloor})$, and, passing to volumes we get $\mathtt{Vol}(S_n) \leq \mathtt{Vol}(\psi_n(C^+_{\lfloor (n+d)/\alpha \rfloor})) \leq K_{\lfloor (n+d)/\alpha \rfloor, \varepsilon}\varepsilon^n$. Passing to logarithms, dividing by n and taking the limit as $n \to \infty$ we get

$$\mathcal{H}(S) \leq \alpha^{-1} \lim_{n\to\infty} \frac{\log K_{\lfloor (n+d)/\alpha \rfloor, \varepsilon}}{\lfloor (n+d)/\alpha \rfloor} + \log \varepsilon,$$

and applying inequality (1) we obtain

$$\alpha \mathcal{H}(S) \leq \mathcal{H}(C) + \log(1/\varepsilon) + \zeta + \alpha \log \varepsilon = \mathcal{H}(C) + \zeta - (\alpha - 1)\log(1/\varepsilon).$$

Choosing ε small enough makes the inequality wrong. This contradiction concludes the proof. □

Proposition 11. *For sofic timed languages S and C (with $\mathcal{H}(C) > -\infty$) and any $\alpha < 1$ there exists an (S, C)-encoding with rate α (and some delay d). Moreover it can be realized by a kind of timed transducer.*

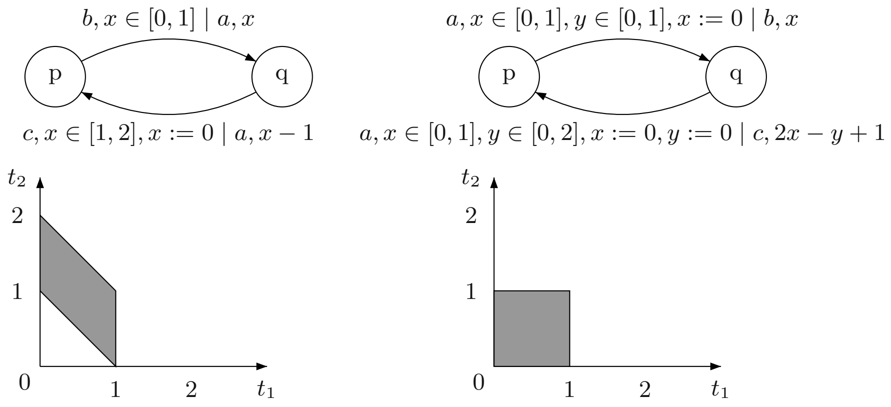
The construction is non-trivial and uses spare time durations in the channel message to transmit discrete information.

5 Discussion and Perspectives

In the previous section, we have established several results on timed channel coding following the standard scheme: a setting of information transmission – information inequality – coding existence theorem – synthesis of an encoder/decoder. We believe that this approach can be applied to various situations of data transmission (and compression). We also consider it as a justification of our previous research on entropy of timed languages [2–4]. In this concluding section, we explain some of our choices and immediate perspectives of this approach.

The Time Is not Preserved. In the central Def. 8 and Prop. 6,7, we consider codings of timed words that preserve the number of events, and not their duration. This choice is compatible to the general idea of dealing with data words (in our case, sequences of letters and real numbers), and less so with the standard timed paradigm. We use again the example of Fig. 2 to illustrate this feature. For $n \in \mathbb{N}$, the timed word $w = [(0.5, e)(0.5, f)]^n$ is encoded to the timed word $w' = [(0.5, a)(1.5, c)]^n$, both have 2 events. However, the duration of w is $(0.5 + 0.5)n = n$, while the duration of w' is $(0.5 + 1.5)n = 2n$.

Other Settings to Explore. It would be still interesting to explore coding functions preserving durations. On the theoretical side, a more detailed analysis for transmission speeds different from 1, as in Sect. 4.4 would probably lead to information inequalities instead of a collapse. Many other settings of transmission of



$b, x \in [0,1] \mid a, x$
p
q
$c, x \in [1,2], x := 0 \mid a, x-1$
$a, x \in [0,1], y \in [0,1], x := 0 \mid b, x$
p
q
$a, x \in [0,1], y \in [0,2], x := 0, y := 0 \mid c, 2x - y + 1$
t_2
2
1
0
1
2
t_1
t_2
2
1
0
1
2
t_1

Fig. 4. Top: transducers $\tau_1 : S \to C$ and $\tau_2 : C \to S$. Bottom: languages S_2 and C_2.

information could be practically relevant: approximated transmission of a timed source on a timed channel; coding of a discrete source on a timed channel; coding using transducers of a fixed precision; coding of other kinds of data languages. On the other hand, more physical models of timed and data channels would be interesting to study, one can think of a discrete channel with a fixed baud rate coupled with an analog channel with a bounded frequency bandwidth. Finally, some special codes, such as sliding-window, error-correcting etc., should be explored for timed and data languages.

What Is a Timed Transducer? In the classical theory of constrained channel coding, several kinds of transducers are used for encoding/decoding, such as those leading to a sliding-block window decoding. In this paper, we have realized our codes by using some very restricted ad hoc timed transducers. They behave like timed (in our case, real-time) automata on the input, and "print" letters and real numbers on the output; we believe that this is the correct approach. However, the right definition of a natural class of timed transducers adequate to coding remains an open question. As a preliminary definition, we suggest timed automata that output, on each transition, a letter and a real number (which is an affine combination of clock values). While reading a timed word, such a transducer would output another timed word. We illustrate this informal definition with the example of mutually inverse transducers τ_1 and τ_2 (encoder and decoder) on Fig. 4. Let us consider a run of the transducer τ_2 on the timed word $(t_1, b)(t_2, c) \ldots$ We start from q with $x = 0, y = 0$, after reading (t_1, b) the value of x and y is t_1, we fire the transition, the output is (t_1, a), we pass in q where we read (t_2, c), the value of x is t_2 and the value of y is $t_1 + t_2$, we fire the transition, the output is $(2t_2 - (t_1 + t_2) + 1, a) = (t_2 - t_1 + 1, a)$, we pass in p etc.

For τ_1, the input language of timed words of length 2 starting from p is $S_2(p) = \{(t_1, b)(t_2, c) \mid t_1 \in [0,1],\ t_1 + t_2 \in [1,2]\}$, while for the second transducer it is $C_2(p) = \{(t_1, a)(t_2, a) \mid t_1 \in [0,1],\ t_2 \in [0,1]\}$. These two languages are depicted in Fig. 4, they have the same volume. It would be impossible to realize this kind of encoding using "rectangular" transducers as in Sect. 4.

How to Improve Code Synthesis? The encoders in Sect. 4 are not completely satisfactory: they use non-integer guards even when the source and the target language are defined using integer timed automata. We believe that this issue can be avoided using a broader class of transducers as suggested just above.

What about Timed Symbolic Dynamics? The classical theory of constrained-channel coding uses as a convenient terminology, and as a toolset, a branch of the theory of dynamical systems called symbolic dynamics. One of us, in [5], started a formulation of timed languages and automata in terms of symbolic dynamics. Relating it to timed channel coding remains a future work.

Applications. In practice, when transmitting (or storing) information containing discrete events and real-valued data, all the information is first converted to the digital form and next encoded for transmission or storage. Our paradigm combines both steps and, in principle, provides better bounds and codes. However, more research is needed to come up with useful practical codes.

Acknowledgment. We thank the anonymous reviewers for valuable remarks that contributed to substantial improvement of this article.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Asarin, E., Degorre, A.: Volume and Entropy of Regular Timed Languages: Analytic Approach. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 13–27. Springer, Heidelberg (2009)
3. Asarin, E., Degorre, A.: Volume and Entropy of Regular Timed Languages: Discretization Approach. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 69–83. Springer, Heidelberg (2009)
4. Asarin, E., Degorre, A.: Two Size Measures for Timed Languages. In: FSTTCS. LIPIcs, vol. 8, pp. 376–387 (2010)
5. Basset, N.: Dynamique Symbolique et Langages Temporisés. Master's thesis, Master Parisien de la Recherche Informatique (2010)
6. Basset, N., Asarin, E.: Thin and Thick Timed Regular Languages. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 113–128. Springer, Heidelberg (2011)
7. Béal, M.P., Berstel, J., Marcus, B., Perrin, D., Reutenauer, C., Siegel, P.H.: Variable-length codes and finite automata. In: Selected Topics in Information and Coding Theory, ch. 14, pp. 505–584. World Scientific Publishing Company (2010)
8. Berstel, J., Perrin, D., Reutenauer, C.: Codes and Automata, Encyclopedia of Mathematics and its Applications, vol. 129. Cambridge University Press (2009)
9. Blahut, R.E.: Digital Transmission of Information. Addison Wesley (1990)
10. Bojanczyk, M.: Data monoids. In: STACS. LIPIcs, vol. 9, pp. 105–116 (2011)
11. Bouyer, P., Petit, A., Thérien, D.: An algebraic approach to data languages and timed languages. Information and Computation 182(2), 137–162 (2003)
12. Immink, K.: EFMPlus: The coding format of the multimedia compact disc. IEEE Transactions on Consumer Electronics 41(3), 491–497 (1995)
13. Lind, D., Marcus, B.: An Introduction to Symbolic Dynamics and Coding. Cambridge University Press (1995)
14. Marcus, B., Roth, R.M., Siegel, P.H.: Constrained systems and coding for recording channels. In: Handbook of Coding Theory, pp. 1635–1764. North-Holland (1998)

Playing Optimally on Timed Automata with Random Delays*

Nathalie Bertrand[1,2] and Sven Schewe[2]

[1] Inria Rennes Bretagne Atlantique, France
[2] University of Liverpool, UK

Abstract. We marry continuous time Markov decision processes (CTMDPs) with stochastic timed automata into a model with joint expressive power. This extension is very natural, as the two original models already share exponentially distributed sojourn times in locations. It enriches CTMDPs with timing constraints, or symmetrically, stochastic timed automata with one conscious player. Our model maintains the existence of optimal control known for CTMDPs. This also holds for a richer model with two players, which extends continuous time Markov games. But we have to sacrifice the existence of simple schedulers: polyhedral regions are insufficient to obtain optimal control even in the single-player case.

1 Introduction

Control problems have been widely investigated in the verification community as a generalisation of the original model-checking problem. Rather than checking whether a system satisfies a property, the goal is to control the system such that it fulfils a desired property. In a framework where timing constraints are essentials, the system can be modelled using timed automata [1], and timed games have been introduced to solve the control problem [2].

Another popular model for systems with nondeterministic choices and real-time aspects is the one of continuous time Markov decision processes (CTMDPs), where the real-time aspects are governed by probability distributions, while the nondeterministic choices are resolved by a scheduler. Time-bounded reachability requires that a goal region should be reached within some time-bound, and the objective is then to build a scheduler that maximises the probability of these executions. This problem has recently received a lot of attention for CTMDPs [5,8,15,18,16,13]. A more fundamental question than the quest for the construction or approximation of optimal schedulers is the question of their existence.

In this paper, we introduce a variant of timed games where delays are randomised rather than being nondeterministic. This model of *timed automata Markov decision processes* (TAMDPs) extends the probabilistic semantics for

* This work was supported by the Engineering and Physical Science Research Council grant EP/H046623/1 'Synthesis and Verification in Markov Game Structures' and a Leverhulme Trust Visiting Fellowship.

timed automata from [3,4] with nondeterminism. Our model also forms an extension of CTMDPs, roughly, by adding timing constraints to the firability of transitions. We consider the time-bounded reachability problem and provide a positive answer to the fundamental question of the existence of optimal schedulers. This result immediately extends to a more general setting with two players, where controlled and adversarial nondeterminism coexist.

The structure of the optimal schedulers is, however, involved. We show that it does not suffice to consider regions or, more generally, to divide the space defined by the relevant clock values into polyhedra, to obtain optimal control.

2 Preliminaries

2.1 Timed Automata

We recall here basics on timed automata, from [1], that will be useful for this paper. Timed automata are extension of finite automata with real-valued variables (called clocks) that all evolve at the same speed. Clocks can be tested and reset to 0.

For a given finite set of clocks X, a *valuation* $v : X \to \mathbb{R}_{\geq 0}$ maps every clock to a non-negative real. A *guard* over X is a finite conjunction of constraints $x \sim c$, with $\sim \in \{<, \leq, =, \geq, >\}$, for a clock $x \in X$ and an integer $c \in \mathbb{N}$. Given a guard g over X and a valuation $v \in \mathbb{R}_{\geq 0}^X$, we write $v \models g$ whenever v satisfies the constraints expressed by g, and define $[\![g]\!] = \{v \mid v \models g\}$. The set of all possible guards over X is denoted $\mathcal{G}(X)$. For $v \in \mathbb{R}_{\geq 0}^X$ a valuation and $t \in \mathbb{R}_{\geq 0}$, $v + t$ denotes the valuation defined by $v + t(x) = v(x) + t$ for every $x \in X$. Moreover, if $X' \subseteq X$ is a subset of clocks and v a valuation, $v_{[X' \leftarrow 0]}$ denotes the valuation that agrees with v on $X \setminus X'$ and is equal to 0 for all clocks in X'.

Definition 1 (Timed automaton). *A* timed automaton *is a tuple* $\mathcal{A} = (L, X, E)$ *where*

- L *is a finite set of* locations*,*
- X *is a finite set of* clocks*, and*
- $E \subseteq L \times \mathcal{G}(X) \times 2^X \times L$ *is a finite set of* edges*.*

The semantics of a timed automaton $\mathcal{A} = (L, X, E)$ is given in terms of an infinite-state transition system $\mathcal{T} = (L \times \mathbb{R}_{\geq 0}^X, \to, \mathbb{R}_{\geq 0} \times E)$, where the relation $\to$ is exactly composed of transitions $(\ell, v) \xrightarrow{t,e} (\ell', v')$ such that the edge $e = (\ell, g, X', \ell') \in E$ satisfies $v + t \models g$ and $v' = (v + t)_{[X' \leftarrow 0]}$. A *run* of $\mathcal{A}$ is a finite sequence of transitions $\rho = (\ell_0, v_0) \xrightarrow{t_0, e_0} (\ell_1, v_1) \xrightarrow{t_1, e_1} \cdots (\ell_n, v_n)$. We denote the last state (ℓ_n, v_n) of run ρ by $\mathsf{last}(\rho)$ and the value $\sum_{i=0}^{n-1} t_i$ is called the *total duration* of ρ. We write $\mathsf{Runs}(\mathcal{A})$ for the set of all runs of $\mathcal{A}$.

In order to encompass CTMDPs in our TAMDP model defined in the next subsection, we first extend timed automata with discrete probabilities. In probabilistic timed automata, introduced in [14], edges do not result in the reset of a fixed set of clocks and lead to a fixed location, but rather yield a distribution $\delta \in \mathsf{Dist}(2^X \times L)$ over resets and locations.

Definition 2 (Probabilistic timed automaton). *A* probabilistic timed automaton *is a tuple $\mathcal{A} = (L, X, E)$ where L and X are as for a timed automaton and $E \subseteq L \times \mathcal{G}(X) \times \mathsf{Dist}(2^X \times L)$ is a finite set of* probabilistic edges.

We write $(\ell, v) \xrightarrow{t,e,p} (\ell', v')$ if from state $(\ell, v+t)$ and assuming probabilistic edge e is selected, the next state is (ℓ', v') with probability p. The rest of the definitions is unchanged. In the sequel, we will consider *symbolic paths*, that is, special sets of runs in probabilistic timed automata. Given a prefix run $\rho \in \mathsf{Runs}(\mathcal{A})$, a sequence of edges $e_0, \cdots, e_n$, together with probabilisties $p_0, \cdots, p_n$, and a time-bound T, the finite symbolic path $\pi(\rho, e_0, p_0 \cdots, e_n, p_n, T)$ is defined by:

$$\pi(\rho, e_0, p_0 \cdots, e_n, p_n, T) = \{\rho \xrightarrow{t_0, e_0, p_0} (\ell_1, v_1) \xrightarrow{t_1, e_1, p_1} \cdots (\ell_n, v_n) \mid \sum_{i=0}^{n-1} t_i \leq T\}.$$

2.2 MDP Model for Timed Automata

A probabilistic semantics for timed automata [3,4], also referred to as stochastic timed automata, was introduced to address the problem of 'unrealistic' sets of paths, where unrealistic is identified with paths that have a very low probability (in particular 0-sets). Informally, the semantics of a stochastic timed automaton consists of an infinite-state infinitely-branching Markov chain (whose underlying graph is a timed transition system $\mathcal{T}$), where transitions between states are governed by the following: first, a delay is sampled randomly among possible delays, and second, an enabled transition is chosen randomly among enabled ones.

For technical convenience—and following [6]—we require our (probabilistic) timed automata to be reactive. A (probabilistic) timed automaton $\mathcal{A} = (L, X, E)$ is called *reactive* if, for each state $s = (\ell, v)$ of $\mathcal{A}$, there is an edge $e \in E$ such that $s \xrightarrow{0,e} s'$ for some state s' of $\mathcal{A}$. In words, we require that every state is the source of some edge, such that $\mathcal{A}$ never blocks. A (probabilistic) timed automaton can easily be made reactive by adding a self loop for all clock valuations where no guard of any transition leaving a state is satisfied.

A natural way of incorporating some control in stochastic timed automata is to have nondeterministic, rather than randomised, decisions among enabled actions. From a state $s = (\ell, v)$, first a delay t is sampled in $\mathbb{R}_{\geq 0}$, and then the controller chooses which probabilistic edge to fire from state $(\ell, v + t)$ among the possible ones.

We thus define the model of *timed automata Markov decision processes* (TAMDPs for short).

Definition 3 (Timed automaton MDP). *A* timed automaton Markov decision process *is a tuple $\mathcal{M} = (L, X, E, \Lambda)$, where $\mathcal{A} = (L, X, E)$ is a reactive probabilistic timed automaton and $\Lambda : L \to \mathbb{R}_{\geq 0}$ is a rate function.*

The semantics of a TAMDP is an infinite-state infinitely branching Markov decision process, whose states (resp. edges) are states (resp. edges) of the underlying probabilistic timed automaton $\mathcal{A}$. From a state $s = (\ell, v)$, the sojourn time in location ℓ follows an exponential distribution with rate $\Lambda(\ell)$ and some intermediary state $(\ell, v+t)$ is reached, where nondeterministically an edge $e \in E$ enabled in $(\ell, v+t)$ fires. The formal semantics of TAMDPs will be detailed further in the next subsection when defining probability measures associated with (a restricted class of well-behaved) schedulers. Note that runs of $\mathcal{M}$ coincide with runs of its underlying probabilistic timed automaton, thus we still write $\mathsf{Runs}(\mathcal{M})$ for the runs of $\mathcal{M}$. Also, in analogy to the common terminology of CTMDPs, we sometimes refer to edges (especially when an edge is selected) as *actions*.

2.3 Comparison with Existing Models

CTMDPs. Continuous-time Markov decision processes form a restricted class of TAMDPs where the underlying probabilistic timed automaton has no clock and is thus a finite automaton. For CTMDPs, it was proven in [16] that optimal control exists for time-bounded reachability, and that optimal schedulers can be taken from a restricted class of finitely representable schedulers.

Stochastic timed games. In stochastic timed games [7], locations are partitioned into locations owned by three players, a reachability player (who has a time-bounded reachability objective), a safety player (who has the opposite time-bounded safety objective), and an environment player (who makes random moves). In a location of the reachability or safety player, the respective player decides both the sojourn time and the edge to fire, whereas in the environment's locations, the delay as well as the edge are chosen randomly. For this model, it was shown that, assuming there is a single player and the underlying timed automaton has only one clock, the existence of a strategy for a reachability goal almost-surely (resp. with positive probability) is PTIME-complete (resp. NLOGSPACE-complete). Moreover, for two-player games, quantitative questions are undecidable.

Stochastic real-time games. In stochastic real-time games [9], states of the arena are partitionned into environment nodes —where the behaviour is similar to CTMDPs— and control nodes —where one player chooses a distribution over actions, which induces a probability distribution for the next state. For this game model, objectives are given by deterministic timed automata (DTA), and the goal for player 0 is to maximize the probability that a play satisfies the objective. The main result concerns qualitative properties, and states that if player 0 has an almost-sure winning strategy, then she has a simple one, that can be described by a DTA.

Markovian timed automata. The model closest to ours is the one of Markovian timed automata (MTA), that, similar to our TAMDPs, consist in an extension of timed automata with exponentially distributed sojourn time. MTA were first introduce as an intermediate model to model-check CTMCs or CTMDPs against deterministic timed automata specifications [10,11]. In the recent paper [12] approximations techniques are provided for the optimal time-(un)bounded reachability probabilities in MTA. In comparison, we focus on the existence of optimal schedulers/strategies for the same problems.

2.4 Schedulers for TAMDPs

Intuitively, a scheduler is responsible for choosing which edge to fire among enabled ones after a random delay has been sampled. To make its decision, the scheduler has access to the all history of the play so far. Formally:

Definition 4 (Scheduler). *Let $\mathcal{M} = (L, X, E, \Lambda)$ be a timed automaton Markov decision process. A* scheduler *for $\mathcal{M}$ is a function $\sigma : \mathsf{Runs}(\mathcal{M}) \times \mathbb{R}_{\geq 0} \to \mathsf{Dist}(E)$ such that for every e_n with $\sigma(s_0 \xrightarrow{t_0,e_0,p_0} s_1 \cdots s_n, t_n, p_n)(e_n) > 0$, there exists a state s_{n+1} with $s_n \xrightarrow{t_n,e_n,p_n} s_{n+1}$.*

Prior to defining a meaningful class of schedulers for TAMDPs, let us first recall the notions of deterministic and memoryless schedulers. A scheduler σ for $\mathcal{M}$ is *deterministic* if it only makes pure decisions: for all $\rho \in \mathsf{Runs}(\mathcal{M})$ and all $t \in \mathbb{R}_{\geq 0}$, $\sigma(\rho, t)$ is a Dirac distribution (*i.e.*, there exists an $e \in E$ such that $\sigma(\rho, t)(e) = 1$). A scheduler σ is *memoryless*[1] if, for all $\rho, \rho' \in \mathsf{Runs}(\mathcal{M})$ with equal total duration and such that $\mathsf{last}(\rho) = \mathsf{last}(\rho')$, $\sigma(\rho, t) = \sigma(\rho', t)$ whatever the delay $t \in \mathbb{R}_{\geq 0}$.

As pointed out in [17], not all schedulers are meaningful, even in the restricted case of continuous-time Markov decision processes (CTMDPs). In particular, under some schedulers, the set of runs reaching a given location can be non-measurable. We follow the approach from [16] and consider a class of schedulers obtained as the completion of the class of cylindrical schedulers. We only report what is necessary in our context, and refer to [16] for the details of this construction.

Definition 5 (Cylindrical scheduler). *A scheduler σ for $\mathcal{M}$ and time-bound T is* cylindrical *if there exists a finite partition $\mathcal{I}$ of $[0, T]$ into intervals $I_0 = [0, T_0]$ and $I_{i+1} = (T_i, T_{i+1}]$ such that, for every pair of runs $\rho = (\ell_0, v_0) \xrightarrow{t_0,e_0,p_0} (\ell_1, v_1) \cdots (\ell_n, v_n)$ and $\rho' = (\ell_0, v_0') \xrightarrow{t_0',e_0,p_0} (\ell_1, v_1') \cdots (\ell_n, v_n')$ and every pair of delays $(t_n, t_n') \in \mathbb{R}_{\geq 0}$, as soon as, for all $0 \leq j \leq n$, t_j and t_j' belong to the same interval I_j, then $\sigma(\rho, t_n) = \sigma(\rho', t_n')$.*

In plain English, a scheduler is cylindrical for a partition $\mathcal{I}$ of $[0, T]$ if it takes the same decision for runs and delays that are equivalent with respect to $\mathcal{I}$.

The set of cylindrical schedulers can then be extended to *measurable* schedulers by defining a metric on cylindrical schedulers and then taking the limits of

[1] Note that our notion of memoryless scheduler is looser than the usual one: the scheduler can also base its decision on the elapsed time so far. This particularity is due to the kind of properties we consider, namely time-bounded reachability.

Cauchy-sequences of cylindrical schedulers with respect to that metric (see [16] for details).

Given a TAMDP $\mathcal{M}$, any measurable scheduler σ yields a probability measure, denoted $\mathbb{P}_\sigma$, over $\mathsf{Runs}(\mathcal{M})$ with a fixed initial state, or more generally a fixed initial run. Let us define $\mathbb{P}_\sigma$ over $\mathsf{Runs}(\mathcal{M}, \rho)$ initiated by a finite prefix $\rho \in \mathsf{Runs}(\mathcal{M})$, by first associating a measure with every finite symbolic path $\pi = \pi(\rho, e_0, p_0 \cdots e_n, p_n, T)$. For every time-bound $T \geq 0$, $\mathbb{P}_\sigma(\rho, T) = 1$ and inductively for $\pi = \pi(\rho, e_0, p_0 \cdots e_n, p_n, T)$ with $\rho \in \mathsf{Runs}(\mathcal{M})$ ending in location ℓ_0, scheduler σ assigns the following probability

$$\mathbb{P}_\sigma(\pi) = \int_{t=0}^{T} \sigma(\rho, t)(e_0) \cdot p_0 \cdot \mathbb{P}_\sigma(\pi(\rho_1, e_1, p_1 \cdots e_n, p_n, T - t)) \cdot \Lambda(\ell_0) \cdot e^{-\Lambda(\ell_0)t} \, \mathrm{d}t,$$

where $\rho_1 = \rho \xrightarrow{t, e_0, p_0} s_1$. Mapping $\mathbb{P}_\sigma$ can then be extended in a unique way into a probability measure over $\mathsf{Runs}(\mathcal{M}, \rho)$ equipped with the σ-algebra generated by symbolic paths starting with ρ.

Time-Bounded Reachability Probability. In this paper, we are interested in time-bounded reachability probabilities. Let us introduce the time-bounded reachability probability problem. Given a TAMDP $\mathcal{M}$, an initial state (ℓ, v), a set of goal locations $G \subseteq L$, and a time-bound T, let $\mathsf{Reach}_\mathcal{M}(\ell, v, G, T)$ denote the set of runs of $\mathcal{M}$ that originate (ℓ, v) and reach the goal within T time-units:

$$\begin{aligned}\mathsf{Reach}_\mathcal{M}(\ell, v, G, T) = \{(\ell, v) = (\ell_0, v_0) \xrightarrow{t_0, e_0, p_0} (\ell_1, v_1) \cdots (\ell_n, v_n) \in \mathsf{Runs}(\mathcal{M}) \mid \\ \exists i \leq n,\ \ell_i \in G \text{ and } \sum_{j<i} t_j \leq T\}.\end{aligned}$$

Note that one can easily express $\mathsf{Reach}_\mathcal{M}(\ell, v, G, T)$ as a countable union of symbolic paths starting in (ℓ, v), and it is thus legal to consider its probability under measurable schedulers. The maximum time-bounded reachability probability problem consists in maximising the probability of $\mathsf{Reach}_\mathcal{M}(\ell, v, G, T)$ among measurable schedulers, and we write

$$\mathsf{Opt}_\mathcal{M}(\ell, v, G, T) = \sup_\sigma \ \mathbb{P}_\sigma(\mathsf{Reach}_\mathcal{M}(\ell, v, G, T)).$$

A natural question is whether optimal schedulers exist *at all*, that is, whether the supremum of the time-bounded reachability probability, $\mathsf{Opt}_\mathcal{M}(\ell, v, G, T)$, is taken for some scheduler. If this is the case, it is worth knowing if simple (e.g., cylindrical, region-based, or, more generally, polyhedral) optimal schedulers exist. In the remainder of the paper, we establish the existence of optimal schedulers for the time-bounded reachability probability problem for TAMDPs, and show that polyhedral schedulers are not sufficient.

3 Optimal Schedulers for TAMDPs

In this section, we establish the existence of optimal schedulers for the maximum time-bounded reachability probability problem in timed automata Markov decision processes. In order to do so, we start by providing lower bounds for $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$ by allowing, in addition to the time-bound, only a fixed number of steps to reach the goal, and then show that these lower bounds are sharp.

We consider the optimal probability to reach the goal G from (ℓ, v) within T time-units, with the additional constraint that it should be in no more than N discrete steps. This probability, which we denote $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)$, optimises the probability of the following set of runs:

$$\mathsf{Reach}^N(\ell, v, G, T) = \{(\ell, v) = (\ell_0, v_0) \xrightarrow{t_0, e_0, p_0} (\ell_1, v_1) \cdots (\ell_N, v_N) \in \mathsf{Runs}(\mathcal{M}) \mid \exists i \leq N,\ \ell_i \in G \text{ and } \sum_{j<i} t_j \leq T\}.$$

That is, we have $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) = \sup_\sigma\ \mathbb{P}_\sigma(\mathsf{Reach}^N_{\mathcal{M}}(\ell, v, G, T))$.

For all $N \in \mathbb{N}$, $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)$ is obviously a lower bound for $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$. Moreover, $\Big(\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)\Big)_{N\in\mathbb{N}}$ is non-decreasing, and, as we shall see, the sequence converges to the ordinary optimum $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$. In order to prove this, we start with the following simple lemma:

Lemma 1. *For all $\varepsilon > 0$ there exists an $M \in \mathbb{N}$ such that, for all $N \geq M$ and all measurable schedulers σ, $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T) \smallsetminus \mathsf{Reach}^N(\ell, v, G, T)\Big) < \varepsilon$ and $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big) - \mathbb{P}_\sigma\Big(\mathsf{Reach}^N(\ell, v, G, T)\Big) < \varepsilon$.*

Proof. Let $\lambda = \max_{\ell\in L} \Lambda(\ell)$ be the maximal transition rate in $\mathcal{M}$. Given $\varepsilon > 0$, we choose M such that $\sum_{k=M}^{\infty} \frac{\lambda^k}{k!} e^{-\lambda} < \varepsilon$. Under any measurable scheduler and assuming all locations have rate λ, the number of steps taken within T time-units is Poisson distributed, and the likelihood to perform M or more steps is bounded from above by $\sum_{k=M}^{\infty} \frac{\lambda^k}{k!} e^{-\lambda}$, and therefore smaller than ε. Of course, this upper bound also applies to the case where rates are smaller or equal λ in all locations. The set of all runs in $\mathcal{M}$ performing M or more steps obviously contains $\mathsf{Reach}(\ell, v, G, T) \smallsetminus \mathsf{Reach}^N(\ell, v, G, T)$ for every $N \geq M$, so we conclude that $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T) \smallsetminus \mathsf{Reach}^N(\ell, v, G, T)\Big) < \varepsilon$.

The second claim follows from $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big) - \mathbb{P}_\sigma\Big(\mathsf{Reach}^N(\ell, v, G, T)\Big) = \mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T) \smallsetminus \mathsf{Reach}^N(\ell, v, G, T)\Big)$. □

We can now establish that $\Big(\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)\Big)_{N\in\mathbb{N}}$ converges to the optimum:

Lemma 2. $\lim_{N\to\infty} \mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) = \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$.

Proof. The '$\leq$' direction is simple: for all schedulers σ and all $N \in \mathbb{N}$, $\mathbb{P}_\sigma\Big(\mathsf{Reach}^N(\ell, v, G, T)\Big) \leq \mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big)$ trivially holds, and consequently $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) \leq \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$.

For the '$\geq$' direction, we show that, for all $\varepsilon > 0$, $\lim_{N\to\infty} \mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) \geq \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T) - 2\varepsilon$. Let $\varepsilon > 0$. On one hand, we can always choose a scheduler σ such that $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big) > \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T) - \varepsilon$. On the other hand, applying Lemma 1, there exists $M \in \mathbb{N}$ such that, for every $N \geq M$, $\mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big) - \mathbb{P}_\sigma\Big(\mathsf{Reach}^N(\ell, v, G, T)\Big) < \varepsilon$. As a consequence, for N large enough, $\mathbb{P}_\sigma\Big(\mathsf{Reach}^N(\ell, v, G, T)\Big) > \mathbb{P}_\sigma\Big(\mathsf{Reach}(\ell, v, G, T)\Big) - \varepsilon > \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T) - 2\varepsilon$, and thus $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) > \mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T) - 2\varepsilon$. □

From now on, we focus on the under-approximation $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)$, in order to prove the existence of optimal schedulers for $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$.

Lemma 3. *For every state (ℓ, v) in $\mathcal{M}$, the sequence $\Big(\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T)\Big)_{N\in\mathbb{N}}$ is characterized inductively by:*

$$\mathsf{Opt}^0_{\mathcal{M}}(\ell, v, G, T) = 0 \text{ if } \ell \notin G, \tag{1}$$

$$\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) = 1 \text{ if } \ell \in G \text{ for all } N \in \mathbb{N}, \text{ and otherwise} \tag{2}$$

$$\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v, G, T) = \int_0^T \max_{e\in E} \sum_{(\ell,v)\xrightarrow{t,e,p}(\ell',v')} p \cdot \mathsf{Opt}^N_{\mathcal{M}}(\ell', v', G, T-t)\cdot \Lambda(\ell)\cdot e^{-\Lambda(\ell)t}\mathrm{d}t. \tag{3}$$

Equation (3), stating that optimality is memoryless, is the only non obvious one.

Proof. The correctness of Equation (3) can be shown by a simple inductive proof over N. The base case, for $N = 0$, clearly holds since all the schedulers return the same probability.

For the induction step, assume the equation holds up to N. Then, for $N + 1$ step-bounded reachability, the scheduler has to make a decision what action to choose from (ℓ, v) if a discrete action occurs after delay t. By induction hypothesis, this is to optimise the outcome in case of having $T - t$ time-units and N steps left. □

Lemma 4. $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) \in [0, 1]$*,* $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T) \in [0, 1]$*, and* $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v+t, G, T-t)$ *and* $\mathsf{Opt}_{\mathcal{M}}(\ell, v+t, G, T-t)$ *are uniformly continuous in t and v.*

Proof. First, it is easy to see that $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v, G, T) \in [0, 1]$, for all parameters. Taking the limit when N tends to infinity, this also holds for $\mathsf{Opt}_{\mathcal{M}}(\ell, v, G, T)$.

Let n be the number of clocks and $\|\cdot\|$ be any norm on valuations[2]. We now prove by induction on N that $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v+t, G, T-t)$ is uniformly continuous in t and v. Obviously, $\mathsf{Opt}^0_{\mathcal{M}}(\ell, v+t, G, T-t)$ is constant (for fixed ℓ and G) and thus uniformly continuous in v and t.

Let us show the uniform continuity of $\mathsf{Opt}^N_{\mathcal{M}}(\ell, v+t, G, T-t)$ in t for all $N \in \mathbb{N}$. Assume $|t - t'| < \varepsilon$, and, w.l.o.g., $t < t'$. Observe that

$$\begin{aligned}
&\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v+t, G, T-t)\\
&\quad= \int_0^{T-t} \max_{e\in E} \sum_{(\ell,v+t)\xrightarrow{\tau,e,p}(\ell',v')} p \cdot \mathsf{Opt}^N_{\mathcal{M}}(\ell', v', G, T-t-\tau)\cdot\Lambda(\ell)\cdot e^{-\Lambda(\ell)\tau}\mathrm{d}\tau\\
&\quad= \int_t^{T} \max_{e\in E} \sum_{(\ell,v)\xrightarrow{\tau,e,p}(\ell',v')} p \cdot \mathsf{Opt}^N_{\mathcal{M}}(\ell', v', G, T-\tau)\Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\mathrm{d}\tau\,.
\end{aligned}$$

[2] Recall that all norms over $\mathbb{R}^n$ are equivalent, so the choice of $\|\cdot\|$ is arbitrary.

Thus

$$\begin{aligned}
&\Big|\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v+t, G, T-t) - \mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v+t', G, T-t')\Big| \\
&= \Big| \int_t^T \max_{e\in E} \sum_{(\ell,v)\xrightarrow{\tau,e,p}(\ell',v')} p\cdot \mathsf{Opt}^{N}_{\mathcal{M}}(\ell', v', G, T-\tau)\Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\mathrm{d}\tau \\
&\qquad - \int_{t'}^T \max_{e\in E} \sum_{(\ell,v)\xrightarrow{\tau,e,p}(\ell',v')} p\cdot \mathsf{Opt}^{N}_{\mathcal{M}}(\ell', v', G, T-\tau)\Lambda(\ell)e^{-\Lambda(\ell)(\tau-t')}\mathrm{d}\tau\Big| \\
&\le \int_t^{t'} \max_{e\in E} \sum_{(\ell,v)\xrightarrow{\tau,e,p}(\ell',v')} p\cdot \mathsf{Opt}^{N}_{\mathcal{M}}(\ell', v', G, T-\tau)\Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\mathrm{d}\tau \\
&\qquad + \int_{t'}^T \max_{e\in E} \sum_{(\ell,v)\xrightarrow{\tau,e,p}(\ell',v')} p\cdot \mathsf{Opt}^{N}_{\mathcal{M}}(\ell', v', G, T-\tau)\Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\big|1-e^{-\Lambda(\ell)(t-t')}\big|\mathrm{d}\tau \\
&\le \int_t^{t'} \Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\mathrm{d}\tau + \big|1-e^{-\Lambda(\ell)(t-t')}\big| \int_{t'}^T \Lambda(\ell)e^{-\Lambda(\ell)(\tau-t)}\mathrm{d}\tau \\
&= (1-e^{-\Lambda(\ell)(t'-t)}) + \big|1-e^{-\Lambda(\ell)(t-t')}\big|(e^{-\Lambda(\ell)(t'-t)} - e^{-\Lambda(\ell)(T-t)}) \\
&\le (1-e^{-\Lambda(\ell)(t'-t)}) + \big|1-e^{-\Lambda(\ell)(t-t')}\big| = e^{-\Lambda(\ell)(t-t')} - e^{-\Lambda(\ell)(t'-t)} \\
&\le e^{\Lambda(\ell)\varepsilon} - e^{-\Lambda(\ell)\varepsilon},
\end{aligned}$$

and we can conclude the uniform continuity of $\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v+t, G, T-t)$ in t.

For the uniform continuity in v, we start with the induction hypothesis

$$\begin{aligned}
&\forall \ell\in L\ \forall G\subseteq L\ \forall\varepsilon>0\ \exists\delta>0\ \forall v,w\in\mathbb{R}^n_{\ge 0}.\ \|v-w\|<\delta \\
&\qquad\qquad \Rightarrow \Big|\mathsf{Opt}^{N}_{\mathcal{M}}(\ell, v, G, T) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell, w, G, T)\Big| < \varepsilon.
\end{aligned}$$

With such ε and δ, and letting $\lambda = \max\{\Lambda(\ell) \mid \ell\in L\}$, we will show that if $\|v-w\|<\varepsilon$ then $|\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, v, G, T) - \mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell, w, G, T)| < \lambda(T\varepsilon + n\lceil T\rceil\delta)$. This will be sufficient to establish the induction step by taking δ' for a given ε in the same way as choosing δ for $\lambda(n+1)\lceil T\rceil\varepsilon$. In order to do so, let us define

$$I^w_v = \Big\{\tau\in[0,T] \mid v+\tau \text{ and } w+\tau \text{ do } \textit{not} \text{ satisfy the same guards}\Big\}.$$

Observe that, provided $\|v-w\|<\delta$, the 'length' $\int_{\tau\in I^w_v}\mathrm{d}\tau$ of I^w_v is bounded by $n\lceil T\rceil\delta$. Indeed, $v+t$ and $w+t$ can only have different enabled transitions if for one of the clocks x the two valuations disagree on $x<c$ (for $c\in\mathbb{N}$), and the interval over which they differ is bounded in length by $\|v-w\|$. The number of such intervals is itself bounded by $\lceil T\rceil$ for clock x. Finally we obtain the bound $n\lceil T\rceil\delta$ when considering the n clocks.

Let us now turn to the induction step. First we assume without loss of generality that $\delta < \varepsilon$. (Obviously, δ can always be chosen this way.) We then get:

$$\begin{aligned}
&\big|\mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell,v,G,T) - \mathsf{Opt}^{N+1}_{\mathcal{M}}(\ell,w,G,T)\big| = \Big| \int_{t=0}^{T} \Lambda(\ell) e^{-\Lambda(\ell)t} \\
&\qquad \Big(\max_{e\in E} \sum_{(\ell,v)\xrightarrow{t,e,p}(\ell',v')} p\ \mathsf{Opt}^{N}_{\mathcal{M}}(\ell',v',G,T-t) - \max_{e\in E} \sum_{(\ell,w)\xrightarrow{t,e,p}(\ell',w')} p\ \mathsf{Opt}^{N}_{\mathcal{M}}(\ell',w',G,T-t)\Big)\mathrm{d}t\Big| \\
&\le \lambda \int_{t=0}^{T} \Big| \max_{e\in E} \sum_{(\ell,v)\xrightarrow{t,e,p}(\ell',v')} p\ \mathsf{Opt}^{N}_{\mathcal{M}}(\ell',v',G,T-t) \\
&\qquad\qquad - \max_{e\in E} \sum_{(\ell,w)\xrightarrow{t,e,p}(\ell',w')} p\ \mathsf{Opt}^{N}_{\mathcal{M}}(\ell',w',G,T-t)\Big|\mathrm{d}t \\
&\le \lambda\Big(\int_{[0,T]\setminus I_v^w} \max_{e\in E} \sum_{(\ell,v)\xrightarrow{t,e,p}(\ell',v')} p\ \big|\mathsf{Opt}^{N}_{\mathcal{M}}(\ell',v',G,T-t) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell',w',G,T-t)\big|\mathrm{d}t \\
&\qquad\qquad + \int_{I_v^w} \mathrm{d}t\Big) \qquad\qquad (\mathsf{Opt}^{N}_{\mathcal{M}} \text{ is in } [0,1]) \\
&\le \lambda\Big(\int_{[0,T]\setminus I_v^w} \varepsilon \mathrm{d}t + \int_{I_v^w} \mathrm{d}t\Big) \qquad\qquad \text{(induction hypothesis)} \\
&\le \lambda\Big(T\varepsilon + n\lceil T\rceil\delta\Big) \le \lambda(n+1)\lceil T\rceil\varepsilon
\end{aligned}$$

This ends the proof that $\mathsf{Opt}^{N}_{\mathcal{M}}$ is uniformly continuous in v —as the constant multiplicative factor $\lambda(n+1)\lceil T\rceil$ does not matter— for all $N \in \mathbb{N}$.

Finally, we exploit Lemma 2 to show that these properties of uniform continuity in t and v are inherited by the limit $\mathsf{Opt}_{\mathcal{M}}$. This can be shown using simple triangle inequalities. To establish

$$\begin{aligned}
&\forall \ell \in L\ \forall G \subseteq L\ \forall \varepsilon > 0\ \exists \delta > 0\ \forall v, w \in \mathbb{R}^n_{\ge 0}.\ \|v-w\| < \delta \\
&\qquad\qquad \Rightarrow \Big|\mathsf{Opt}_{\mathcal{M}}(\ell,v,G,T) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell,w,G,T)\Big| < 3\varepsilon,
\end{aligned}$$

we first fix an $N \in \mathbb{N}$ such that $\|\mathsf{Opt}_{\mathcal{M}} - \mathsf{Opt}^{N}_{\mathcal{M}}\| \le \varepsilon$. Then we spend one ε for $|\mathsf{Opt}^{N}_{\mathcal{M}}(\ell,v,G,T) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell,w,G,T)|$, because $\mathsf{Opt}^{N}_{\mathcal{M}}$ is uniformly continuous in the valuation, and one ε each for $|\mathsf{Opt}_{\mathcal{M}}(\ell,v,G,T) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell,v,G,T)|$ and $|\mathsf{Opt}_{\mathcal{M}}(\ell,w,G,T) - \mathsf{Opt}^{N}_{\mathcal{M}}(\ell,w,G,T)|$. □

We can now prove our main theorem using a topological argument that extends the argument from [16] to the more general case of TAMDPs.

Theorem 1. *For every TAMDP $\mathcal{M}$, with initial state $(\ell_0, 0^X)$, reachability objective G and time-bound T, there exists a measurable scheduler σ such that*

$$\mathbb{P}_\sigma(\mathsf{Reach}_{\mathcal{M}}(\ell_0, 0^X, G, T)) = \mathsf{Opt}_{\mathcal{M}}(\ell_0, 0^X, G, T).$$

Proof. As a consequence of the continuity of $\mathsf{Opt}_{\mathcal{M}}(\ell, v+t, G, T-t)$ in v and t, we describe in the following an abstract construction of a *measurable* scheduler σ that chooses, for all $v \in [0,T]^X$ and $t, T' \in [0,T]$, an action e that determines the transition $(\ell, v) \xrightarrow{t,e} (\ell', v')$ that maximises $\mathsf{Opt}_{\mathcal{M}}(\ell', v', G, T'-t)$.

For positions outside of $[0,T]^X \times [0,T] \times [0,T]$, the behaviour of the scheduler does not matter: σ can therefore be fixed to any constant decision for all of these clock valuations and times.

We fix a location ℓ for the rest of the proof, and write Range for $[0,T]^X \times [0,T] \times [0,T]$ the range of triples (v,t,T') we will consider. In order to determine optimal decisions, we start with fixing an arbitrary order $\succ$ on the actions available in ℓ. Next, we let, for each clock valuation $v \in [0,T]^X$, delay $t \in [0,T]$ and remaining time $T' \in [0,T]$ and action e, $\mathsf{val}(v,t,T',e) = \sum_{(\ell,v) \xrightarrow{t,e,p} (\ell',v')} p \cdot \mathsf{Opt}_{\mathcal{M}}(\ell', v', G, T'-t)$, provided e is enabled in $(\ell, v+t)$, otherwise we use $-\infty$. Last we introduce, for all (v,t,T'), an additional order $\sqsupset_{v,t,T'}$ on the actions, determined by $\mathsf{val}(\ell, v+t, T, e)$ and using $\succ$ as a tie-breaker.

We now define the following sets for every action e:

– $M_e = \{(v,t,T') \in \mathsf{Range} \mid e \text{ is maximal w.r.t. } \sqsupset_{v,t,T'}\}$ is the set of triplets (v,t,T') for which e is maximal with respect to the order $\sqsupset_{v,t,T'}$.
– $C_e = \{(v,t,T') \in \mathsf{Range} \mid \forall \delta > 0 \; \exists (v',t',T'') \in M_e.\; \|(v,t,T') - (v',t',T'')\| < \delta\}$ is the closure of M_e, and
– $D_e = C_e \smallsetminus \bigcup_{f \succ e} C_f$ is the set of triplets for which action e is $\sqsupset_{v,t,T'}$-better than all other actions and there is no $\succ$-better action with equal quality.

We define σ as the memoryless scheduler σ such that when the last state is (ℓ, v), the delay is t, and the remaining time is T', σ selects action e such that $(v,t,T') \in D_e$. To complete the proof, let us show that σ is (1) optimal and (2) measurable.

To show the first point, we observe that the decision e is optimal in M_e by definition. The fact that e is also optimal in the larger set C_e is a consequence of the continuity of $\mathsf{Opt}_{\mathcal{M}}$ in v and t. $D_e \subseteq C_e$ then implies that e is an optimal decision for all triplets contained in D_e. Note that optimality among the pure decisions entails optimality among mixed ones, as the value of mixed decisions is the convex combination of the values for the respective pure decisions.

To show the second point, we observe that the M_e's partition Range by their definition, because $\sqsupset_{v,t,T'}$ is a total order. Consequently, the C_e's cover Range, and the D_e's again partition Range. The C_e's are closed subsets of Range, and therefore measurable. By their definition, the D_e's inherit this measurability.

Our construction therefore provides us with a measurable scheduler, which is optimal, deterministic, and memoryless. □

4 Extensions

In this section we consider three potential extensions: the extension to games, the extension to time-unbounded reachability, and the strengthening of the results to schedulers with a simple finite structure. We show in Subsection 4.1 that the first of these extensions is possible: our results extend to a generalisation to two-player games. But the results from Theorem 1 do neither extend to time-unbounded reachability (Subsection 4.2), nor can they be strengthened to a simple class of schedulers, whose decisions are only based on polyhedral regions (Subsection 4.3).

4.1 Extension to Games

So far, we considered time-bounded reachability objectives for TAMDPs, which can be seen as a stochastic one-player game, that is, a game with a single player interacting with a randomised environment. Let us now discuss how to extend our results to stochastic two-player games by considering timed automata Markov games (TAMGs for short).

Definition 6 (Timed automaton Markov game). *A* timed automaton Markov game *is a tuple $\mathcal{G} = (\mathcal{A}, L_0, L_1, \Lambda)$ where $\mathcal{A} = (L, X, E)$ is a reactive probabilistic timed automaton $L = L_0 \sqcup L_1$ is a partition of the set of locations and $\Lambda : L \to \mathbb{R}_{\geq 0}$ is a rate function.*

Naturally, Player 0 owns states with location in L_0 and Player 1 is responsible for the decisions in states with location in L_1. The semantics of a TAMG is a stochastic two-player game. Informally, from a state (ℓ, v) with $\ell \in L_i$ (for $i \in \{0, 1\}$), the sojourn time in location ℓ follows an exponential distribution with rate $\Lambda(\ell)$ and Player i chooses in the intermediate state $\ell, v + t$ which enabled edge to fire. The resolution of nondeterministic choices by the players is governed by strategies. Similarly to TAMDPs, for which we introduced cylindrical and measurable schedulers, we consider here cylindrical and measurable strategies for each of the players. We write σ (resp. τ) for a measurable strategy of Player 0 (resp. Player 1). Any strategy profile (σ, τ) for $\mathcal{G}$ with σ and τ measurable strategies induces a probability measure $\mathbb{P}_{\sigma,\tau}$ over $\mathsf{Runs}(\mathcal{G}) = \mathsf{Runs}(\mathcal{A})$ (assuming an initial state is fixed).

The objective of Player 0 is to maximise the probability to reach a set of goal locations $G \subseteq L$ within time T. The optimum is thus defined as:

$$\mathsf{Opt}_{\mathcal{G}}(\ell, v, G, T) = \sup_{\sigma} \inf_{\tau} \mathbb{P}_{\sigma,\tau}(\mathsf{Reach}_{\mathcal{G}}(\ell, v, G, T)).$$

As announced earlier, the result established for TAMDPs carries over to TAMGs:

Theorem 2. *For every TAMG $\mathcal{G}$ with initial state $(\ell_0, 0^X)$, reachability objective G for Player 0 and time-bound T, there exists a measurable strategy profile (σ, τ) such that*

$$\mathbb{P}_{\sigma,\tau}(\mathsf{Reach}_{\mathcal{G}}(\ell_0, 0^X, G, T)) = \mathsf{Opt}_{\mathcal{G}}(\ell_0, 0^X, G, T) =$$
$$\sup_{\sigma'} \mathbb{P}_{\sigma',\tau}(\mathsf{Reach}_{\mathcal{G}}(\ell_0, 0^X, G, T)) = \inf_{\tau'} \mathbb{P}_{\sigma,\tau'}(\mathsf{Reach}_{\mathcal{G}}(\ell_0, 0^X, G, T)).$$

In order to extend the proof, we proceed in two steps. The first step is the extension of the lemmata from Section 3. This extension is simple: following the same structure, it suffices to replace, in the lemmata and their proofs, the max by min for all locations of Player 1.

Consequently, we obtain a set of equations that describe the value of the time-bounded reachability probability. We can then proceed with fixing optimal measurable strategies for σ only and τ only, respectively, such that their decisions is locally optimal. Note that the proof of Theorem 1 treats the different locations independently, so that a restriction to a subset of locations does not affect the proof at all. The proof is also not affected by swapping max for min for the locations owned by Player 1.

4.2 Time-Unbounded Reachability

We considered optimisation problem for time-bounded reachability, and justify now, a posteriori, why the time-bound is crucial for Theorem 1. Indeed, we show that optimal scheduling policies may not exist for *time-unbounded* reachability objectives. The situation thus ressembles the framework of stochastic real-time games [9] for which it was shown that optimal strategies do not always exist, using a similar example. To exemplify this, we consider the TAMDP $\mathcal{M}$ depicted on Figure 1 with constant transition rate $\Lambda = 1$ and the objective to reach the goal region G. We argue that this control objective does not admit an optimal scheduling policy.

It is easy to see that the chances of reaching G from ℓ_1 are 0 if the value of the clock x is greater or equal to 1, and $e^{-\varepsilon} - e^{-1}$ for a clock value $\varepsilon \in [0,1]$. This implies an upper bound on the time-unbounded reachability of $1 - e^{-1}$. This value can easily be approximated by choosing a scheduling policy that guarantees a time-unbounded reachability $> 1 - e^{-1} - \varepsilon$ by progressing to ℓ_1 iff the clock value of x is smaller than ε. (Almost surely such a value is eventually taken.)

While this determines the *value* of time-unbounded reachability, it does not provide a scheduling policy that realises this value. If we consider a scheduling policy that, for any $\varepsilon \in]0,1]$, provides a positive probability p_ε to progress to ℓ_1 with a clock valuation $\geq \varepsilon$, then the likelihood of reaching G is bounded by $1 - e^{-1} - p_\varepsilon(1 - e^{-\varepsilon})$. At the same time, this chance being 0 for all $\varepsilon > 0$ implies that we almost surely never progress to ℓ_1. (Progressing with clock valuation 0 can only happen on a 0 set.)

Consequently, no optimal scheduling policy exists.

4.3 Simple Schedulers

Beyond the existence of optimal schedulers, the simplicity of schedulers is also a concern. In the proof of Theorem 1 we show the existence of an optimal scheduler which is measurable, deterministic and memoryless. It is an interesting question whether the optimum can be reached by even simpler schedulers. For CTMDPs, e.g., timed positional schedulers (whose decisions only depend on the location and the time that remains) and even cylindrical schedulers (that have only finitely many intervals of constant decisions) are sufficient [16]. Timed positional schedulers are clearly not sufficient for TAMDPs. (Consider a scheduler that makes the decision of whether to progress to ℓ_1 in the example from Figure 1, with 0 to 0.5 time-units left and the clock valuation of x (a) less than 0.5 and (b) greater than 1. For (a), the optimal decision is clearly to progress, for (b), the optimal decision is clearly to stay in ℓ_0 and reset the clock. A scheduler that does not distinguish these cases cannot be optimal.) Still the question remains if considering simple regions of clock valuations suffices. A natural generalisation of finitely many intervals would be finitely many *polyhedra* that are distinguished by a scheduler.

Definition 7. *Let $\mathcal{M}$ be a TAMDP over $\mathcal{A}$ a timed automaton with N clocks. A scheduler σ for $\mathcal{M}$ is* polyhedral *if there exists a finite partition $\mathcal{P}$ of $(\mathbb{R}_{\geq 0})^{N+1}$ into polyhedra $P_1 \cdots P_k$ such that, for every pair of runs $\rho = (\ell_0, v_0) \xrightarrow{t_0, e_0, p_0} (\ell_1, v_1) \cdots (\ell_n, v_n)$ and $\rho' = (\ell_0, v'_0) \xrightarrow{t'_0, e'_0, p'_0} (\ell'_1, v'_1) \cdots (\ell'_m, v'_m)$ and every pair of delays $(t_n, t'_m) \in \mathbb{R}_{\geq 0}$, as soon as $\ell_n = \ell'_m$ and $(v_n + t_n, \sum_{i \leq n} t_i)$ and $(v'_m + t'_m, \sum_{i \leq m} t'_i)$ belong to the same polyhedron P_j, then $\sigma(\rho, t_n) = \sigma(\rho', t'_m)$.*

Note that polyhedral schedulers are in particular memoryless (and timed positional in the special case of CTMDPs). Polyhedral schedulers are natural in the context of timed automata since they extend region-based schedulers that are for example sufficient for timed games [2].

Proposition 1. *In TAMDPs, the optimal time-bounded reachability probability may not be taken by any polyhedral scheduler.*

Proof. To prove that polyhedral schedulers are not sufficient to obtain optimal control in TAMDPs, we consider again the example of Figure 1, where the rate is constant $\Lambda = 1$, the goal location is G and the time-bound is set to 1.

The only non-trivial decision the scheduler has to make in that example is in location ℓ_0, where it has to choose between looping back to ℓ_0 (action *loop* in the sequel) or moving right to ℓ_1 (action *progress* in the sequel). We are interested in determining a partition (D_l, D_p) of $(\mathbb{R}_{\geq 0_+})^2$ representing sets of valuation for x and remaining time t such that *loop* is optimal in D_l and *progress* is optimal in D_p. We focus on the sub-region $[0,1]^2$ to show that neither D_l nor D_p can be composed of finite unions of polyhedra, and consequently optimal schedulers cannot be polyhedral for this example. For this sub-region, we start with the following observations:

1. If $t \leq 1 - x$, then it is always advisable to select *progress*. The time-bounded reachability probability in this case is $1 - e^{-t}$.
2. If $t \geq 1 - x$ and the selected action is to *progress*, then the time-bounded reachability probability is $1 - e^{x-1}$.
3. If the selected action is to *loop*, then the time-bounded reachability probability is $1 - (t+1)e^{-t}$. (When looping, x is reset. After the reset of x, the guard of the edge from ℓ_1 to G is always satisfied in the remaining $t \leq 1$ time-units. The chance of reaching G is thus the probability of taking two or more steps in the remaining t time-units, and the number of such steps is Poisson distributed with parameter t.)

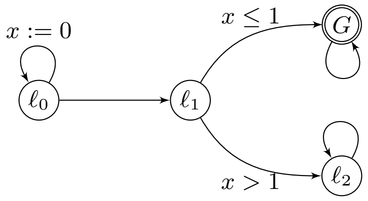


Fig. 1. A simple TAMDP example

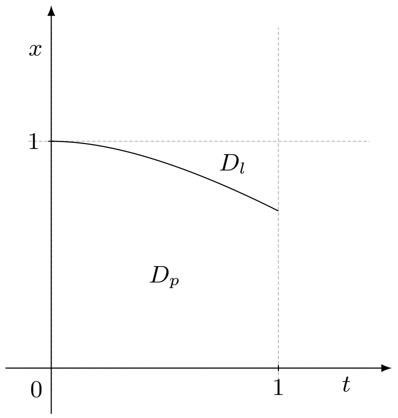


Fig. 2. Illustration of partition $(D_e)_{e \in E}$

Consequently, we loop in $[0,1]^2$ iff $(t+1)e^{-t} \leq e^{x-1}$ (modulo 0 sets). Obviously, this set is not representable by a finite union of polyhedra.

The construction of the partition (D_l, D_p) illustrates the proof of Theorem 1, where a measurable optimal scheduler is defined. The partition (D_p, D_l) intersected with $[0,1]^2$ is depicted on Figure 2. The area D_p, below the curve, represents pairs (t,x) of remaining time and clock valuation, for which progressing to ℓ_1 is the optimal decision. □

5 Conclusion

We have introduced the model of timed automata Markov games that synthesises stochastic timed automata and continuous time Markov games: TAMGs enhance stochastic timed automata with two conscious players and add timing constraints to the firing of actions in continuous time Markov games. We have proven the existence of measurable strategies that optimise the probability of time-bounded reachability properties. Different to CTMGs, optimal strategies are not necessarily simple: they cannot be represented by finite families of polyhedral regions. Also, in contrast to the positive result for time-bounded reachability, we have shown that optimal scheduling policies for time-unbounded reachability do not always exist.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller Synthesis for Timed Automata. In: Proc. of SCC 1998, pp. 469–474. Elsevier (1998)

3. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T., Größer, M.: Probabilistic and Topological Semantics for Timed Automata. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 179–191. Springer, Heidelberg (2007)
4. Baier, C., Bertrand, N., Bouyer, P., Brihaye, Th., Größer, M.: Almost-Sure Model Checking of Infinite Paths in One-Clock Timed Automata. In: Proc. of LICS 2008, pp. 217–226. IEEE (2008)
5. Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient Computation of Time-Bounded Reachability Probabilities in Uniform Continuous-Time Markov Decision Processes. Theoretical Computer Science 345(1), 2–26 (2005)
6. Bouyer, P., Brihaye, Th., Jurdziński, M., Menet, Q.: Almost-Sure Model-Checking of Reactive Timed Automata. In: Proc. of QEST 2012. IEEE (to appear, 2012)
7. Bouyer, P., Forejt, V.: Reachability in Stochastic Timed Games. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 103–114. Springer, Heidelberg (2009)
8. Brázdil, T., Forejt, V., Krcál, J., Kretínský, J., Kucera, A.: Continuous-Time Stochastic Games with Time-Bounded Reachability. In: Proc. of FSTTCS 2009. LIPIcs, vol. 4, pp. 61–72. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2009)
9. Brázdil, T., Krčál, J., Křetínský, J., Kučera, A., Řehák, V.: Stochastic Real-Time Games with Qualitative Timed Automata Objectives. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 207–221. Springer, Heidelberg (2010)
10. Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. Logical Methods in Computer Science 7(1:12), 1–34 (2011)
11. 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)
12. Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Reachability Probabilities in Markovian Timed Automata. In: Proc. of CDC-ECC 2011, pp. 7075–7080. IEEE (2011)
13. Fearnley, J., Rabe, M.N., Schewe, S., Zhang, L.: Efficient Approximation of Optimal Control for Continuous-Time Markov Games. In: Proc. of FSTTCS 2011. LIPIcs, vol. 13, pp. 399–410. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)
14. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic Verification of Real-Time Systems with Discrete Probability Distributions. Theoretical Computer Science 282(1), 101–150 (2002)
15. Neuhäußer, M.R., Zhang, L.: Time-Bounded Reachability Probabilities in Continuous-Time Markov Decision Processes. In: Proc. of QEST 2010, pp. 209–218. IEEE (2010)
16. Rabe, M.N., Schewe, S.: Finite Optimal Control for Time-Bounded Reachability in CTMDPs and Continuous-Time Markov Games. Acta Informatica 48(5-6), 291–315 (2011)
17. Wolovick, N., Johr, S.: A Characterization of Meaningful Schedulers for Continuous-Time Markov Decision Processes. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 352–367. Springer, Heidelberg (2006)
18. Zhang, L., Neuhäußer, M.R.: Model Checking Interactive Markov Chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)

Dynamically-Driven Timed Automaton Abstractions for Proving Liveness of Continuous Systems*

Rebekah Carter and Eva M. Navarro-López

School of Computer Science,
The University of Manchester, UK
{carterr,eva.navarro}@cs.man.ac.uk

Abstract. We look at the problem of proving inevitability of continuous dynamical systems. An inevitability property says that a region of the state space will eventually be reached: this is a type of liveness property from the computer science viewpoint, and is related to attractivity of sets in dynamical systems. We consider a method of Maler and Batt to make an abstraction of a continuous dynamical system to a timed automaton, and show that a potentially infinite number of splits will be made if the splitting of the state space is made arbitrarily. To solve this problem, we define a method which creates a finite-sized timed automaton abstraction for a class of linear dynamical systems, and show that this timed abstraction proves inevitability.

Keywords: Continuous-time systems, Abstraction, Automated verification, Liveness properties, Timed automata.

1 Introduction

Dynamical systems can have very complex behavior patterns, and over the years the mathematics and control communities have developed a lot of theory to analyze their behavior. In particular, Lyapunov stability theory [8] is important for proving properties about continuous dynamical systems. One of the elements of stability theory is the notion of *attractivity*, which says that trajectories of a system tend toward a set or a point as time goes to infinity.

From the computer science viewpoint, the related notion to attractivity ensures that a set is reached in finite time, and is expressed as the property "we eventually reach some set in the state space". This is an *inevitability* property, a type of liveness property. In this paper we address the problem of proving such liveness properties, as there are only a few methods currently defined [5,9,14].

In the computer science community, particularly the hybrid systems community, a lot of effort has gone into trying to verify dynamical systems, mostly

* This paper has been made under the framework of the EPSRC-funded project "DYVERSE: A New Kind of Control for Hybrid Systems" (EP/I001689/1). The second author is also grateful for the support of the RCUK (EP/E50048/1).

through using model checking methods [3,10,13,16]. One class of such methods involves *abstraction* of the system to a discrete system, usually a finite-state automaton, in order to be able to use discrete model checkers on the abstraction to prove properties about the original system (for example, see [3]). The properties that can be proved are typically safety properties, which say that something is always true. In fact, proving liveness properties in dynamical systems, where we want something to eventually be true, is not possible by means of a purely discrete abstraction as there is no guarantee of progress of time.

In order to prove inevitability properties, we need to transfer some information about the times at which events occur to the discrete system, and so we turn to using *timed automata* (TA) for our abstraction. Reachability of TA is decidable, and so they are a sensible candidate for an abstraction. There are various provers available for proving properties of TA, the most widely used one being Uppaal [2].

There are a few methods which propose how to abstract a dynamical system to a timed automaton, including [4,9,11,14,15]. We consider the method of Maler and Batt [9], who did not specify how the state space of a system should be split to create the TA, but stated that the accuracy of the model could be improved indefinitely by increasing the number of splits. However, in most systems arbitrary splitting will not be very good at capturing the dynamics of the system, and we may well be required to make a very large number of splits to verify the system.

In this work, we advocate *dynamically-driven* splitting of the state space, using known properties of the flow of the system to decide how to make the splits. We identify a terminating splitting method for a class of upper triangular linear systems which ensures that the resulting timed automaton will always prove the inevitability property. Our method is most closely related to that of [14], which abstracts continuous systems to timed automata using the idea of the method of [9], but the authors in that work use Lyapunov functions to define the slices considered, whereas we use the original idea of constant variable slices.

2 Overview of Systems and Method Being Considered

In this section, we consider all autonomous n-dimensional continuous dynamical systems, of the form

$$\dot{x} = f(x), \tag{1}$$

with $x = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$ and $f(x) = [f_1(x), \ldots, f_n(x)]^T$, with $f : \mathbb{R}^n \to \mathbb{R}^n$ smooth. The state space of the system is assumed to be a finite box, which is defined by the limits $x \in [s_1^-, s_1^+) \times \ldots \times [s_n^-, s_n^+) = S$.

The method we consider is from [9]. It is an approximation method defined to abstract a continuous system to a TA, and is based on minimum and maximum velocities of a system defining bounds on the time taken to cross a certain distance in the system. The resulting TA is an over-approximation of the system, in the sense that every trajectory in the system is matched in time and space by one in the abstraction, but additional trajectories may be allowed in the abstraction (see [9] for more discussion of this).

The basic idea of [9] is to split the state space into slices by making splits along lines of the form $x_i = C$, where C is constant. These slices also define hyper-rectangles (which we refer to as *boxes*) of the space by the intersection of a slice in each dimension (see Fig. 1 for the 2-dimensional (2-D) case with $x \in \mathbb{R}^2$).

A slice is a part of the state space, restricted only in one dimension. In this work we allow slices to be of differing widths, defined by a vector of split points $Verts_i$ in each dimension i.[1] Let v_i be an index which indicates which slice we are considering in dimension i, and then the slice is given by

$$X_{i,v_i} = \left[s_1^-, s_1^+\right) \times \ldots \times \left[Verts_i(v_i),\, Verts_i(v_i+1)\right) \times \ldots \times \left[s_n^-, s_n^+\right) \subseteq S. \quad (2)$$

Slices are right-open so that they do not intersect, and so the set of slices for each i will form a partition of the state space S (this is why S was defined as being right-open). Similarly, let the index for a box be defined by $v = [v_1, \ldots, v_n]$, then the box is defined as $X_v = \bigcap_{i=1}^n X_{i,v_i} \subseteq S$. The set of all boxes in the partitioning of S is denoted by V. To find possible crossings between these boxes we consider the sign of the velocity on each face between adjacent boxes.

Definition 1 (Automaton Abstraction [9]). *The automaton $A = (V, \delta)$ is an abstraction of the system if δ consists of the pairs of boxes with indices $v = [v_1, \ldots, v_i, \ldots, v_n]$ and $v' = [v_1, \ldots, v_i + 1, \ldots, v_n]$ where f_i can take a positive value on the face between them, or pairs of v with $v' = [v_1, \ldots, v_i - 1, \ldots, v_n]$ where f_i can take a negative value on the face between them.* ■

To make the timed automaton abstraction, we define clocks to keep track of the times at which crossings are made in each dimension. Within any box X_v, let d_i be the width of this box in dimension i, then the maximal time that it can take to leave this box is over-approximated by the *box time*, defined as

$$\overline{t}_v = \min_{1\le i\le n}\left(\frac{d_i}{\min(|f_i|) \text{ in box } X_v}\right). \quad (3)$$

If $\min(|f_i|) = 0$ in box X_v, then we define $\overline{t}_v = \infty$.

The times spent in slices of the space can also be limited, both above and below, in the positive and negative directions. Let d_i be the size of the dimension i slice we are considering, and let $\underline{f_i}$ be the minimum velocity in this slice, and $\overline{f_i}$ be the maximum velocity. Then the minimum ($\underline{t}$) and maximum ($\overline{t}$) times that can be spent in this slice in the positive (+) and negative (−) directions are given in Table 1 (*slice times*).

If we enter a slice from the lower face in dimension i, then the minimum time we can take to leave by the opposite face is $\underline{t}^+$, and the maximum time to leave by the opposite face is $\overline{t}^+$, and similarly for entering from the upper face with $\underline{t}^-$ and $\overline{t}^-$. We use these values to bound timed automaton clocks within slices of the space: there are two clocks per dimension, z_i^+ which bounds positive direction movements using $\underline{t}^+$ and $\overline{t}^+$, and z_i^- which bounds the negative

[1] Note that the first and last elements of $Verts_i$ are s_i^- and s_i^+ respectively.

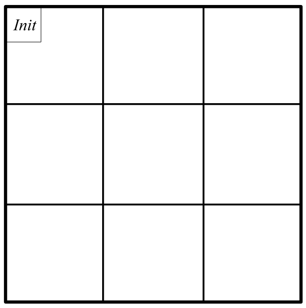


Fig. 1. Partition into boxes (rectangles in the 2-D case).

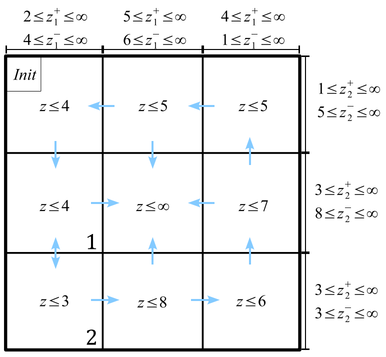


Fig. 2. Calculate (1) which transitions exist (denoted by arrows), and (2) the min/max times on the clocks $z, z_1^+, z_1^- \ldots$.

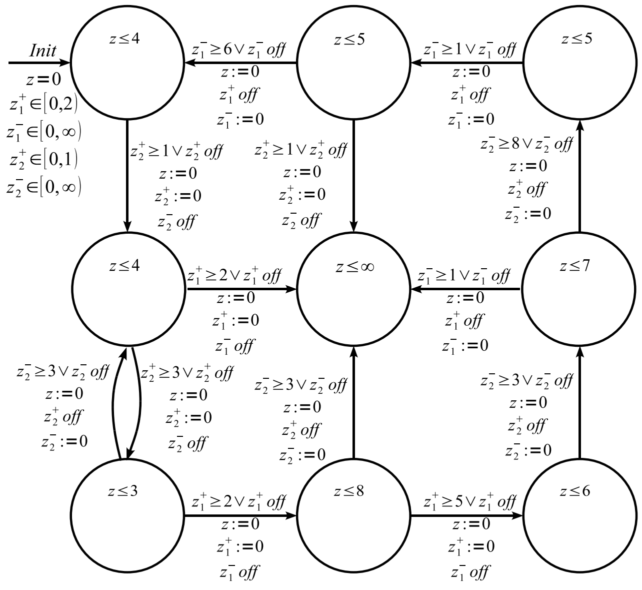


Fig. 3. The TA resulting from the method of [9] applied to the dynamical system. In this case, no slice clock has an upper limit, so they do not appear inside locations.

Table 1. Minimum and maximum times that can be spent in slice i in the positive and negative directions (respectively)

	$\underline{t}^+$	$\overline{t}^+$	$\underline{t}^-$	$\overline{t}^-$
$0 < \underline{f_i} < \overline{f_i}$	$d_i/\overline{f_i}$	$d_i/\underline{f_i}$	∞	∞
$\underline{f_i} < \overline{f_i} < 0$	∞	∞	$-d_i/\underline{f_i}$	$-d_i/\overline{f_i}$
$\underline{f_i} < 0 < \overline{f_i}$	$d_i/\overline{f_i}$	∞	$-d_i/\underline{f_i}$	∞

direction movements. We also use a box clock z in the TA to satisfy the conditions on how long we can stay in each box, using (3).

Figures 1–3 illustrate the abstraction process for an arbitrary 2-D system. This overview of the method involved should be enough to understand the content of this paper, but for further details see [9].

3 Problems for General Continuous Systems

There are various reasons why this method does not work well for general continuous systems, some of which we highlight in this section. One very fundamental reason can be that the trajectories of the system do not fit well with splitting based on crossing constant variable lines. A particular example of this is for 2-D linear systems with complex eigenvalues, where the trajectories of the system are spirals. Even when the real parts of the eigenvalues are negative and the trajectories go inward, the timed automaton can allow flow round the edge of the split space without forcing us to move closer to the center of the spiral (see for example Fig. 3, where the center location should be reached, but the TA allows flow through the eight locations round the edge indefinitely).

Part of the problem with this abstraction for general continuous dynamical systems is the fact that the discrete automaton abstraction can allow pairs of automaton locations where the trajectories can go both ways across the shared face (see locations marked 1 and 2 in Fig. 2). The timed automaton does not restrict when these transitions can be taken, as box times are not directional and slice times only limit the time to reach the *opposite* face, so these pairs of locations introduce Zeno behaviour into the abstraction [7]. This kind of behaviour prevents every trace of the abstraction from reaching the desired final location, so the inevitability property cannot be proved.

Another potential problem with this method is that it does not guarantee to calculate *finite* values of the box times $\bar{t}_v$, due to the fact that if a zero velocity occurs in the box v in every dimension, then $d_i/\min(|f_i|) = \infty$ in every dimension. If there is one box in the abstraction which has an infinite box time, then any trace which reaches this box will never be forced to leave this box even if the actual trajectories of the system would all leave it in finite time.

Some of the above problems can be remedied by choosing an appropriate method for splitting the state space. The method of [9] does not specify how we should choose the splitting, but just tells us the properties of the resulting TA when a choice has been made. As we see it, there are two ways to do such a splitting: either we make the splitting arbitrarily (systematically but not based on the system's dynamics) and rely on refinement to eventually capture enough information about a system, or we can use properties of the dynamics to choose where to split the system. The pros and cons of these are discussed below.

Arbitrary splitting. This approach does not rely on knowledge about the structure of the dynamical system, and so it can be used for complex systems no matter where the complexity comes from. The TA created by the abstraction method of Maler and Batt does approach the actual dynamics (theoretically) as more and more splits are made (see [9]). However, due to the two issues (1) that transitions both ways between pairs of automaton locations can exist and (2) that there can be an infinite over-approximation of the time it takes to get across a box, the number of splits required is often very large, if not infinite, and so we obtain a huge number of locations in the timed automaton.

Using system properties for splitting. Here we use the dynamics of the system we are looking at to automatically split the state space in a way which removes or reduces some of the problems associated with arbitrary splitting. In specific systems, it should be possible to make a splitting which can be proved to satisfy desirable properties, for instance that the liveness property is automatically satisfied. Even in systems where we cannot prove the liveness property immediately, it may be possible to at least have a much better starting point for refining the abstraction. The main problems with this idea are that a splitting method will only work for a certain class of systems, and that there are no automatic splitting methods in existence already; methods need to be designed for many different types of systems.

Given the considerations above, in this paper we wish to start the process of finding dynamically-driven automatic methods to create splittings. We will work from a theoretical basis to show that certain types of linear systems have a splitting which proves inevitability by the TA abstraction, with the idea that future work can extend these methods to be useful for more general systems (nonlinear, piecewise continuous, or hybrid).

4 Inevitability for Upper Triangular Linear Systems

In this section we consider the class of upper triangular linear dynamical systems, as a first step to using this method for general systems. We assume that a splitting has been created which removes some of the problems highlighted in the previous section. We then show that the timed automaton abstraction created from this splitting proves an inevitability property. In Sect. 5 we will identify a method for a sub-class of these systems which creates a splitting with the desired properties. These two parts together will prove that the sub-class considered in Sect. 5 can always have an inevitability-proving TA abstraction.

We consider upper triangular linear systems, with no input vector:

$$\dot{x} = Ax = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n,n} \end{pmatrix} x. \tag{4}$$

We assume that the eigenvalues (the diagonal entries of A) are negative. This type of system has a unique equilibrium point at zero,[2] and the negative eigenvalues mean that this point is stable and attractive, which is asymptotic stability.

[2] The equilibrium point x_e satisfies the equation $\dot{x} = 0$, that is $Ax_e = 0$. Invertibility of A means the only equilibrium point is at $x_e = 0$.

This means that some non-empty region L containing the equilibrium point must be reached within a finite time. From a dynamical systems perspective we know this property is true, but we want to prove the same property computationally — the long-term goal of this work is to prove inevitability properties of dynamical systems which cannot be proved by dynamical theory (due to complexity). In linear temporal logic (LTL), this inevitability property is written $\Diamond(x \in L)$, where $\Diamond$ is the temporal logic operator meaning "eventually".[3] In this paper, we will say the system is *live* if this LTL condition is satisfied for all trajectories.

We note that the only real restriction on the class of linear systems is the existence of real negative eigenvalues: for any linear system of the form $\dot{x} = Bx + b$ with B having real negative eigenvalues and b any real vector, it can be transformed to $\dot{x} = Bx$ without loss of generality, and can then be transformed to an equivalent upper triangular system by making the Schur decomposition.

4.1 The Assumptions

There are four assumptions we make about the splitting, each with various levels of difficulty in achieving them. In Sect. 5 we will define a method which can satisfy these assumptions for a subclass of the systems under study, which shows these are not unreasonable assumptions to make.

The first assumption is about where the equilibrium point occurs in relation to the splitting, and is here to give us one (and only one) box that we are interested in reaching for the inevitability property.

Assumption 1. There is exactly one box L in the splitting which contains the equilibrium point,[4] and the equilibrium is not on the boundary of L. We will call L the *live box.*

The second assumption specifies that there are a *finite* number of boxes in the abstraction, which is necessary for a useful abstraction.

Assumption 2. The automaton abstraction (Def. 1) of the system has a finite number of discrete states.

The next assumption has to do with how transitions are allowed in the abstracted system. We do not want it to be possible to keep transitioning between a pair of discrete states in the timed automaton, as discussed in Sect. 3.

Assumption 3. The continuous flow across any box face only occurs in one direction. That is, if the box face is $x_j = C$ with the other x_i's within the box limits, then the velocity $\dot{x}_j$ across this face will either be always $\dot{x}_j \geq 0$ or always $\dot{x}_j \leq 0$.

The fourth assumption is related to the timing constraints in the TA, ensuring we leave every box on a trace within a finite time.

Assumption 4. In each box v, except the live box L, the box time $\overline{t}_v$ is finite.

[3] See [12] for more information about LTL.

[4] This is always true for linear systems, where there is only one equilibrium point.

4.2 The Theorems

We wish to show that every system of form (4) is proved to satisfy the inevitability property by any splitting satisfying the assumptions. To do this we will prove that the discrete automaton abstraction (Def. 1) of the continuous system only has finite traces, and that the only location with no outgoing edges (hence the only possible final location) corresponds to the live box L. We then use the assumption of finite box time to ensure that L is reached in finite time.

Theorem 1. *Assume we have an n-dimensional system of the form of (4) with negative entries on the diagonal, and an automaton abstraction created by the method of [9] satisfying Assumptions 1–4. Then the automaton abstraction of the continuous system only has finite traces.*

Proof. Assume (for a contradiction) that we can find a trace of infinite length in the automaton. As the automaton abstraction must have a finite number of locations by Assumption 2, any infinite trace must go through at least one discrete location infinitely often. Hence, for any move in the infinite trace that is made in the k-th dimension in the positive direction, we must be able to find a corresponding move in the k-th dimension in the negative direction, and vice-versa. We now prove, by induction, that this requirement is not satisfied in the system.

Base Case. In the n-th dimension, the dynamics of the system is $\dot{x}_n = a_{n,n}x_n$ with $a_{n,n} < 0$. Hence, across any slice boundary in the n-th dimension, if $x_n > 0$ then $\dot{x}_n < 0$, and if $x_n < 0$ then $\dot{x}_n > 0$. Therefore crossing any n-th dimensional slice boundary can only be done in one direction, and so cannot be reversed as is necessary for this type of crossing to be present in the infinite trace. Hence n-th dimensional crossings are not involved in the infinite trace.

Inductive Part. Assume that $n-k+1, \dots, n$ dimensional crossings are not involved in the infinite trace, and so the $x_{n-k+1}, \dots, x_n$ variables are within one slice each for this trace. The $\dot{x}_{n-k}$ equation only depends on $x_{n-k}, \dots, x_n$. Assumption 3 does not allow the sign of $\dot{x}_{n-k}$ to change over the course of a box face, but since $x_{n-k+1}, \dots, x_n$ are within one slice each, any constant value of x_{n-k} makes the whole slice face the same sign as one of the component box face. Hence $\dot{x}_{n-k}$ is either completely non-negative or completely non-positive for a constant value of x_{n-k}. This means it is not possible to reverse a crossing of a constant x_{n-k} surface in the infinite trace, and so $(n-k)$-th dimensional crossings cannot be present in the infinite trace.

Hence, by base case and inductive part, the infinite trace through the automaton abstraction of the system cannot involve transitions in any dimension, which contradicts the existence of an infinite trace. So the automaton abstraction of the n-dimensional system under the assumptions only has finite traces. □

Theorem 2. *Assume we have a system of form (4) with negative diagonal entries, and an automaton abstraction created by the method of [9] satisfying Assumptions 1–4. Then the only location with no outgoing edges in the automaton abstraction corresponds to the box L containing the equilibrium point $x_e = 0$.*

Proof. Assume there is a location of the automaton abstraction which has no transitions out of it. Then all the existing transitions are inwards. It is not possible for whole box sides to have zero flow across them, due to not allowing splitting at $x_i = 0$ surfaces (by Ass. 1). Therefore the two opposite sides of a box in dimension i have opposite (inwards) flows, so by continuity of the flow, the zero surface $\dot{x}_i = 0$ must pass through this box. As this is the case for all dimensions, then the linearity of the system means the equilibrium point x_e must be in the corresponding box of the partition of the state space. By Assumption 1 the equilibrium point is wholly within one box L, and so the only location with no outgoing edges is box L. (Note that box L is an invariant set.) □

Theorem 3. *Assume we have a system of form (4) with negative diagonal entries, and an automaton abstraction created by the method of [9] satisfying Assumptions 1–4. Then all trajectories of the system from all initial boxes get to the live box L within finite time.*

Proof. Theorems 1 and 2 together imply that all traces of the TA lead to the box L containing the equilibrium point in a finite number of steps. Assumption 4 says that each cube on this route is left within a finite time, and so the total time for any TA trace to reach the box L is finite. By the over-approximation of number of trajectories, all trajectories of the original system reach the box L within finite time (and stay there as L is invariant). □

5 Dynamically-Driven Splitting Method

In this section we define a method to split the state space of a continuous system such that the TA abstraction created from it satisfies the four assumptions for a subset of the upper triangular linear systems. Together with the previous section this proves that we can automatically create a TA abstraction of such systems which proves inevitability. The method is shown to terminate for this particular class of systems, and the number of locations in the resulting abstraction is analyzed.

5.1 The Class of Systems Considered

The class we now consider are a subclass of upper triangular linear systems of form (4) with two conditions on them:

– All of the entries on the main diagonal are negative.
– In each row, a maximum of one other non-zero entry is allowed.

These conditions mean that x_i's differential equation is in one of two forms, for each $i = 1, \ldots, n$, either

$$\dot{x}_i = a_{i,i}x_i \quad \text{for } a_{i,i} \text{ negative, or} \tag{5}$$

$$\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j \quad \text{for } a_{i,i} \text{ negative}, a_{i,j} \neq 0 \text{ and } i < j \leq n. \tag{6}$$

This class of systems does have restrictions, but allows various special classes of systems. In particular, all 2-D systems with negative real eigenvalues can be transformed by the Schur decomposition to an equivalent dynamical system of this form. The verification results for such 2-D systems (with state space limits and live box limits suitably transformed) will prove the desired properties about the original system. Higher dimensional systems which can be of this form include systems modelling chains of behaviour, where each x_i's evolution only depends on itself and the element next in the chain. For example, simplified models of biological cascades can be expressed in this form [6]. There are also some piecewise linear models of such cascades, where each element depends only on itself and the element before it: these can be modelled in the form (5) or (6) for each x_i and each region of dynamics. The method proposed is easily extendible to piecewise systems of this form provided the divides between different dynamics in the system occur at constant values of x_i (this is the case in [1], for example).

If row i of the matrix has the form (5), this means that we always have the same value of $\dot{x}_i$ for any constant value of x_i. So Assumption 3 is always true on all box faces in dimension i. On the other hand, if row i has the second form (6), this means that when we try to separate $\dot{x}_i > 0$ from $\dot{x}_i < 0$ on a particular face $x_i = C$, we get a constant value of $x_j = -\frac{a_{i,i}C}{a_{i,j}}$ where the $\dot{x}_i = 0$ surface occurs through this face. Therefore, we can choose to split at this point in the j-th dimension to make sure of separating the positive and negative velocities in the i-th dimension. This ease of selecting where to split is not available to us for general upper triangular systems, and is what makes this special class better for automatic splitting.

5.2 The Splitting Method

The method we propose for splitting systems of this new form is described in Algorithm 1. The idea is to originally split based on the live box boundaries which creates a box satisfying Assumption 1 (Step 1), and then to split based on where a $\dot{x}_j = 0$ surface crosses any constant x_j surface (Step 2). After these two steps the resulting TA abstraction satisfies Assumptions 1 and 3.

Step 3 of the algorithm then finds and divides the boxes where infinite time has been found, whilst still keeping these other two assumptions intact, in order to satisfy Assumption 4. It works by finding the intervals of existence of $\dot{x}_i = 0$ surfaces, and splitting between two intervals calculated in dimension i if they do not intersect. This method is demonstrated on a 2-D example in Fig. 4.

We will now show that this algorithm terminates and quantify the size of the resulting abstraction, then we will give an overview of the proof of why the abstraction satisfies Assumptions 1–4.

Termination and Abstraction Size. Firstly we show that FollowSplits terminates. FollowSplits consists of two for loops, the first clearly has a finite number of executions ($n - 1$). The second iterates over a finite number of elements of the list *List* for $i = 1$, and each iteration removes an element from the current

Algorithm 1. Automatic splitting algorithm

Input: Linear dynamical systems with each x_i's dynamics of the form of (5) or (6), with state space $[s_1^-, s_1^+) \times \ldots \times [s_n^-, s_n^+)$, and live box $L = [l_1^-, l_1^+) \times \ldots \times [l_n^-, l_n^+)$.
Output: A splitting of the system such that the TA abstraction proves inevitability of the live box.

```
 1: add splits at $x_i = l_i^-$ and $x_i = l_i^+$ for each $i$          ▷ Step 1

 2: call FollowSplits                                                    ▷ Step 2

 3: Calculate box times                                                  ▷ Step 3
 4: $B \leftarrow$ list of boxes with infinite box time (except $L$)
 5: while $B$ is non-empty do
 6:     for $k = 1, \ldots, \text{length}(B)$ do
 7:         $V \leftarrow$ get vertices of box $B(k)$
 8:         $Z \leftarrow V$ (initialize the region where zero surfaces occur: let $Z_i^-$ be lower
    bound and $Z_i^+$ be upper bound in dimension $i$)
 9:         for $i = n, n-1, \ldots, 1$ do
10:             if $\dot{x}_i$ has an off-diagonal entry then
11:                 $j \leftarrow$ position of the off-diagonal entry in row $i$ of $A$
12:                 $\begin{bmatrix} c_i^- & c_i^+ \\ c_j^- & c_j^+ \end{bmatrix} \leftarrow$ limits of the surface $\dot{x}_i = 0$ in the box
13:                 if $c_j^- > Z_j^+$ then
14:                     add split at $x_j = (c_j^- + Z_j^+)/2$ (or nearby if this is $x_j = 0$)
15:                     break loop
16:                 else if $c_j^+ > Z_j^-$ then
17:                     add split at $x_j = (c_j^+ + Z_j^-)/2$ (or nearby if this is $x_j = 0$)
18:                     break loop
19:                 else if $c_i^- > Z_i^+$ then
20:                     add split at $x_i = (c_i^- + Z_i^+)/2$ (or nearby if this is $x_i = 0$)
21:                     break loop
22:                 else if $c_i^+ > Z_i^-$ then
23:                     add split at $x_i = (c_i^+ + Z_i^-)/2$ (or nearby if this is $x_i = 0$)
24:                     break loop
25:                 end if
26:             else(row $i$ does not have an off-diagonal entry)
27:                 if $Z_i^+ < 0$ then
28:                     add split at $Z_i^+/2$
29:                     break loop
30:                 else if $Z_i^- > 0$ then
31:                     add split at $Z_i^-/2$
32:                     break loop
33:                 end if
34:             end if
35:         end for
36:     end for
37:     call FollowSplits
38:     $B \leftarrow$ new list of boxes with infinite box time (except $L$)
39: end while
```

Algorithm 2. FollowSplits sub-algorithm

Input: Current splitting state of the system (including the newly made splits), with a finite list *List* of newly made splits that need to be followed down the dimensions.
Output: A new splitting with only one direction of flow across the faces of the boxes.

```
1: for $i = 1, \dots, n-1$ do
2:     $f_i$ = the sublist of $List$ of constant values for $x_i$ (splits in dimension $i$)
3:     for all $k \in f_i$ do
4:         $v \leftarrow$ find $\dot{x}_i$ velocity at vertices of the splitting surface $x_i = k$
5:         if all($v \geq 0$) or all($v \leq 0$) then
6:             remove $x_i = k$ from $List$
7:         else
8:             solve $\dot{x}_i = 0$ when $x_i = k$ giving $x_j = d$, for some $i < j \leq n$.
9:             add split at $x_j = d$
10:            add $x_j = d$ to $List$
11:            remove $x_i = k$ from $List$
12:        end if
13:    end for
14: end for
```

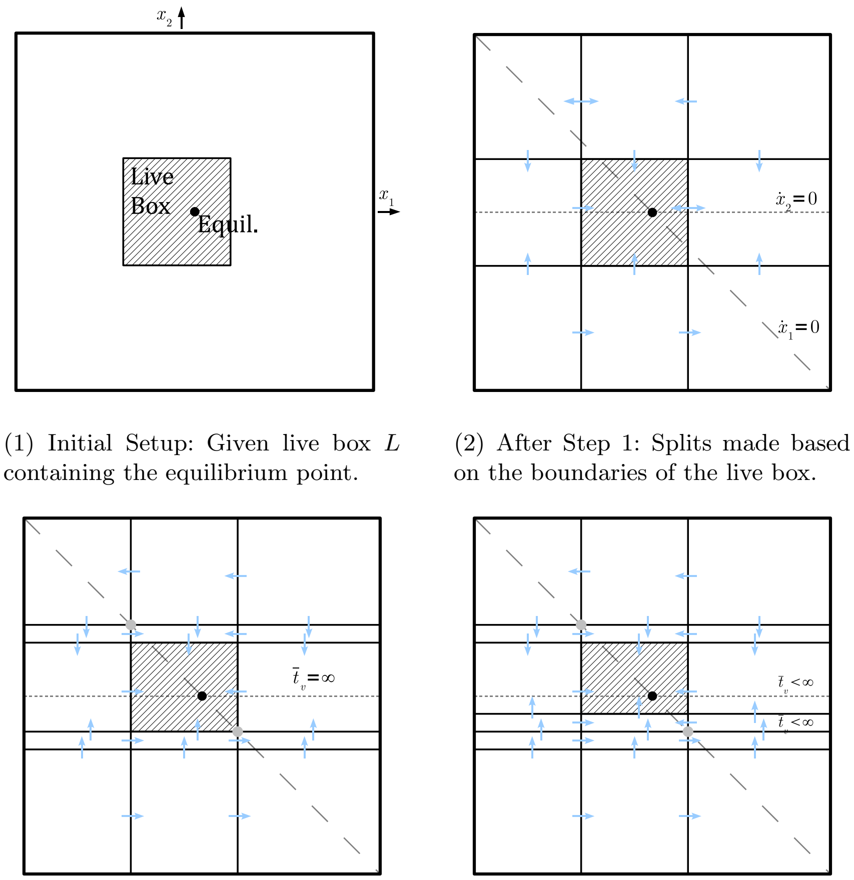
dimension list and possibly adds one to a lower dimension's list. Since this is done a finite number of times, each iteration of the inner loop is only done a finite number of times, and so FollowSplits terminates. The number of splits that can be added by this is dependent on the size of the current splitting in each direction (say this size is c_i for $i = 1 \dots, n$), and also dependent on the size of the newly made splits list (n_i for $i = 1, \dots, n$). The worst case is when each dimension causes a split in the dimension immediately after it, as the effect of these splits builds up, so the maximum number of splits (overall) after FollowSplits in each dimension $i = 1, \dots, n$ is

$$c_i + \sum_{j=1}^{i-1} n_j. \tag{7}$$

Now consider Algorithm 1 as a whole. Clearly line 1 of Algorithm 1 performs a finite number of splits (2 in each dimension), so terminates. Then, line 2 simply calls FollowSplits on these initial splits, which terminates, with maximum of $2, 4, \dots, 2n$ splits in each of the $1, 2, \dots, n$-th directions respectively.

For Step 3, lines 3–39, we must consider the while loop and the two for loops inside it. Both for loops iterate over only a finite number of values, so the combination of the two must terminate (given that all individual lines terminate). So we now need to show that the while loop terminates, which occurs when we have removed the infinite box time on all boxes except the live box.

The proof that all infinite-time boxes will be removed is a little more involved, and we need to understand how this algorithm splits the state space of the system. First we will show that all boxes with infinite time at the start of step 3 must touch the live box L (along an edge or at a corner). Assume not, then there



(1) Initial Setup: Given live box L containing the equilibrium point.

(2) After Step 1: Splits made based on the boundaries of the live box.

(3) After Step 2: Light grey dots indicate where $\dot{x}_1 = 0$ on x_1 constant lines. Splits made at these points.

(4) After Step 3: An extra split is added above the lower dot, removing infinite time in the box marked in (3).

Fig. 4. Applying the splitting algorithm to the example $\dot{x}_1 = -x_1 - x_2$ and $\dot{x}_2 = -x_2$, with the given live box defined slightly off centre around the equilibrium point (equil). Arrows indicate the allowed directions of flow across box boundaries, and the shaded region indicates the live box L as it changes size.

is a box B_1 which is $p > 1$ steps away from L in some dimension i. Then, letting j be the other dimension which occurs in the equation $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j = 0$, we take the projection of this $n-1$-dimensional surface to a line in the x_i-x_j plane. So if the box B_1 has infinite time, this line $\dot{x}_i = 0$ must pass through a corner of the edge nearest to L, by step 2 and linearity. But then, as this line also passes through the equilibrium point in the middle of box L, it must have passed through the middle of an edge $x_i = C$, contradicting that step 2 has been completed. So all infinite-time boxes must be a maximum of one step away from L in any dimension, which means that *every infinite-time box touches* L.

The rest of the proof is too complex to explain in detail, but is an inductive argument on the dimensions. Roughly, assume we have a box B_1 with infinite time which cannot be split on the first run through the while loop of Algorithm 1, and this box is offset from L's slices in the dimensions $i_1, i_2, \dots, i_k$ (where these are in order smallest to largest). Then, if equation $\dot{x}_{i_k}$ depends on x_{i_k} and x_j, then there must be a box B_2 with infinite box time which shares a dimension j edge with B_1. If B_2 is split by the algorithm, this makes B_1 splittable on the next run through the while loop (after FollowSplits). If B_2 is not split, then it too must have a neighboring box in dimension dependent on the $\dot{x}_j$ equation, and so on. Eventually we reach a splittable box (at dimension n if not before), which makes the previous box splittable after FollowSplits is completed, and so at most $n-1$ runs through the while loop are necessary to remove all infinite time boxes, hence Algorithm 1 terminates.

Algorithm 1 creates at most one split for each infinite-time box, with this split being, in dimension one greater than the largest dimension in which the box is offset from the live box L (in the worst case). Now, in one dimensional systems, there are clearly a maximum of two boxes in dimension 1 which are 'next to' the box L. In two dimensions, there are an extra 6, all with their maximal offsets in dimension 2, and 2 (as before) only offset in dimension 1. By induction we can find that there are $2 \times 3^{i-1}$ boxes with a highest dimensional offset in dimension i. Infinite time is not possible with any offset from the central slice in dimension n, so we only need consider the splits made in dimensions 1 to $n-1$.

As each of these possible infinite-time boxes can create a maximum of one split in the next dimension, this creates $2 \times 3^{i-2}$ splits in each of the $i = 2, \dots, n$ dimensions. When FollowSplits is done, using the formula in (7) and the initial number of splits $2i$ for each dimension $i = 1, \dots, n$, we can compute the number of splits as: for $i = 1$, 2 splits, and for $i = 2, \dots, n$, $2i + 3^{i-1} - 1$ splits. This makes the maximum number of slices 3 for $i = 1$ and $2i + 3^{i-1}$ for $i = 2, \dots, n$. The total number of boxes is the product of the slices in each dimension, so

$$NumBoxes = 3 \times \prod_{j=2}^{n} (2j + 3^{i-1}). \tag{8}$$

Satisfying the Assumptions. We will now give an outline of the proof of why this algorithm creates an abstraction which satisfies the assumptions.

Assumption 1. Step 1 creates one box containing the equilibrium $x_e = 0$, under the original specification that the live box should include x_e (not on the boundary). Step 2 can change the size of the box containing the equilibrium, but cannot add splits exactly at $x_i = 0$ for any i (because of linearity of dynamics), so the box containing x_e does not have it on the boundary. Step 3 similarly can make splits which affect the box L, but again they are chosen not to be at the equilibrium.

Assumption 2. There are a finite number of boxes in this splitting, which we have already quantified.

Assumption 3. After Step 2 has happened, Assumption 3 is satisfied due to splitting *at* the zero values, and then after each change in Step 3 the "FollowSplits" function is used, which again makes the TA satisfy this assumption.

Assumption 4. We showed that Algorithm 1 terminates, and in the process showed that it only terminates when all infinite times on boxes are removed (apart from L). Hence, this assumption is satisfied.

5.3 The Main Result

The above properties result in the statement that if a TA is created by the method of [9] using the splitting of Alg. 1, then the proof of inevitability of L on the TA abstraction will prove the inevitability of L for the original system. The proof follows from the method of Alg. 1 with the assumptions of Sect. 4.1 and the theorems of Sect. 4.2.

Corollary 1. *Given a continuous dynamical system with each x_i's dynamics of the form (5) or (6), then, by considering the TA abstraction, all possible trajectories of the linear system will reach box L containing $x_e = 0$.* ■

6 Conclusions and Future Work

In this work we have defined a method for a class of linear systems which creates a splitting of the state space. When using this splitting to create a timed automaton by the method of [9], we have shown that certain properties are true of the timed automaton. Together these properties mean that the timed automaton will prove inevitability of the original system reaching a set L around the equilibrium point x_e. The method is easily extendible to a related class of piecewise linear systems.

Our future goal is to extend this method to more general dynamical systems, be they linear, nonlinear, piecewise, or hybrid systems. There are various problems to be overcome with these systems, one of which will be the termination of the splitting method, as the current method only terminates because of the special dynamics involved. Hence, part of the future work will be to revise the splitting method to be more useful for more general systems. For piecewise and hybrid systems, we aim to develop dynamically-driven splitting methods for considering the guards/resets (changes between areas of different dynamics), so that general guards can be considered by the splitting.

References

1. Batt, G., Belta, C., Weiss, R.: Model Checking Liveness Properties of Genetic Regulatory Networks. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 323–338. Springer, Heidelberg (2007)
2. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

3. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. International Journal of Foundations of Computer Science 14(4), 583–604 (2003)
4. D'Innocenzo, A., Julius, A.A., Di Benedetto, M.D., Pappas, G.J.: Approximate timed abstractions of hybrid automata. In: 46th IEEE Conference on Decision and Control, pp. 4045–4050 (2007)
5. Duggirala, P.S., Mitra, S.: Lyapunov Abstractions for Inevitability of Hybrid Systems. In: Hybrid Systems: Computation and Control (HSCC), pp. 115–123 (2012)
6. Heinrich, R., Neel, B.G., Rapoport, T.A.: Mathematical Models of Protein Kinase Signal Transduction. Molecular Cell 9(5), 957–970 (2002)
7. Johansson, K.H., Egerstedt, M., Lygeros, J., Sastry, S.: On the regularization of Zeno hybrid automata. Systems & Control Letters 38(3), 141–150 (1999)
8. Lyapunov, A.M.: The general problem of the stability of motion. PhD thesis, Moscow University (1892); Reprinted in English in the International Journal of Control 55(3) (1992)
9. Maler, O., Batt, G.: Approximating Continuous Systems by Timed Automata. In: Fisher, J. (ed.) FMSB 2008. LNCS (LNBI), vol. 5054, pp. 77–89. Springer, Heidelberg (2008)
10. Mitchell, I., Tomlin, C.J.: Level Set Methods for Computation in Hybrid Systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 310–323. Springer, Heidelberg (2000)
11. Olivero, A., Sifakis, J., Yovine, S.: Using Abstractions for the Verification of Linear Hybrid Systems. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 81–94. Springer, Heidelberg (1994)
12. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th International Symposium on the Foundations of Computer Science, pp. 46–57 (1977)
13. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Transactions on Embedded Computing Systems 6(1), 573–589 (2007)
14. Sloth, C., Wisniewski, R.: Verification of continuous dynamical systems by timed automata. Formal Methods in System Design 39(1), 47–82 (2011)
15. Stursberg, O., Kowalewski, S., Engell, S.: On the Generation of Timed Discrete Approximations for Continuous Systems. Mathematical and Computer Modelling of Dynamical Systems: Methods, Tools and Applications in Engineering and Related Sciences 6(1), 51–70 (2000)
16. Tiwari, A.: Abstractions for hybrid systems. Formal Methods in System Design 32(1), 57–83 (2008)

Revisiting Timed Specification Theories: A Linear-Time Perspective

Chris Chilton, Marta Kwiatkowska, and Xu Wang

Department of Computer Science, University of Oxford, UK

Abstract. We consider the setting of component-based design for real-time systems with critical timing constraints. Based on our earlier work, we propose a compositional specification theory for timed automata with I/O distinction, which supports substitutive refinement. Our theory provides the operations of parallel composition for composing components at run-time, logical conjunction/disjunction for independent development, and quotient for incremental synthesis. The key novelty of our timed theory lies in a weakest congruence preserving safety as well as bounded liveness properties. We show that the congruence can be characterised by two linear-time semantics, *timed-traces* and *timed-strategies*, the latter of which is derived from a game-based interpretation of timed interaction.

1 Introduction

Component-based design methodologies can be encapsulated in the form of compositional specification theories, which allow the mixing of specifications and implementations, admit substitutive refinement to facilitate reuse, and provide a rich collection of operators. Several such theories have been introduced in the literature, but none simultaneously address the following requirements: support for asynchronous input/output (I/O) communication with non-blocking outputs and non-input receptiveness; linear-time refinement preorder, so as to interface with automata and learning techniques; substitutivity of refinement, to allow for component reuse at runtime without introducing errors; and strong algebraic and compositionality properties, to enable offline as well as runtime reasoning.

Previously [1], we developed a linear-time specification theory for reasoning about untimed components that interact by synchronisation of I/O actions. Models can be specified operationally by means of transition systems augmented by an inconsistency predicate on states, or declaratively using traces. The theory admits non-determinism, a substitutive refinement preorder based on traces, and the operations of parallel composition, conjunction and quotient. The refinement is strictly weaker than alternating simulation and is actually the weakest precongruence preserving freeness of inconsistent states.

In this paper we target component-based development for real-time systems with critical timing constraints, such as embedded system components, the middleware layer and asynchronous hardware. Amongst notable works in the literature, we surveyed the theory of timed interfaces [2] and the theory of timed specifications [3]. Though both support I/O distinctions, their refinement relations are not linear time: in [2], refinement (compatibility) is based on timed games, and in [3] it is a timed version of the alternating simulation originally defined for interface automata [4]. Consequently, it is too strong for determining when a component can be safely substituted for another. As an example, consider the transition systems P and Q in Figure 3: these should be equivalent in the sense of substitutivity under any environment, and are equivalent in our formulation (Definition 5), but they are not so according to timed alternating simulation.

Contributions. We formulate an elegant timed, asynchronous specification theory based on finite traces which supports substitutive refinement, as a timed extension of the linear-time specification theory of [1]. We allow for both operational descriptions of components, as well as declarative specifications based on traces. Our operational models are a variant of timed automata with I/O distinction (although we do not insist on input-enabledness, cf [5]), augmented by two special states: the *inconsistent* state $\bot$ represents safety and bounded-liveness errors, while the *timestop* state $\top$ is a novel addition representing either unrealisable output (if the component is not willing to produce that output) or unrealisable time-delay (if the delay would violate the invariant on that state).

Timestop models the ability to stop the clock and has been used before in embedded system and circuit design [6,7]. It is notationally convenient, accounting for simpler definitions and a cleaner formalism. By enhancing the automata with the notion of *co-invariant*, we can, for the first time, distinguish the roles of input/output guards and invariant/co-invariants as specifying safety and bounded-liveness timed assumptions/guarantees. We emphasise that this is achieved with finite traces only; note that in the untimed case it would be necessary to extend to infinite traces to model liveness. In addition to *timed-trace semantics*, we present *timed-strategy semantics*, which coincides with the former but relates our work closer to the timed-game frameworks used by [3] and [2], and could in future serve as a guide to implementation of the theory. Finally, the *substitutive refinement* of our framework gives rise to the weakest congruence preserving $\bot$-freeness, which is not the case in the formalism of [3].

Related work. Our work can be seen as an alternative to the timed theories of [2,3]. Being linear-time in spirit, it is also a generalisation of [8], an untimed theory inspired by asynchronous circuits, and Dill's trace theory [9]. The specification theory in [3] also introduces parallel, conjunction and quotient, but uses timed alternating simulation as refinement, which does not admit the weakest precongruence. An advantage of [3] is the algorithmic efficiency of branching-time simulation checking as well as the implementation reported in [10]. We briefly mention other related works, which include timed modal transition systems [11,12], the timed I/O model [5,13] and asynchronous circuits and embedded systems [14,15]. A more detailed comparison based on the technical details of our work is included in Section 5. A full version of this paper including an even greater comparison with related work, in addition to proofs, is available as [16].

2 Formal Framework

In this section we introduce timed I/O automata, timed I/O transition systems and a semantic mapping from the former to the latter. Timed I/O automata are compact representations of timed I/O transition systems. We also present an operational specification theory based on timed I/O transition systems, which are endowed with a richer repertoire of semantic machinery than the automata.

2.1 Timed I/O Automata

Clock constraints. Given a set X of real-valued clock variables, a *clock constraint* over X, $cc : CC(X)$, is a boolean combination of atomic constraints of the form $x \bowtie d$ and $x - y \bowtie d$ where $x, y \in X$, $\bowtie \in \{\leq, <, =, >, \geq\}$, and $d \in \mathbb{N}$.

A *clock valuation* over X is a map t that assigns to each clock variable x in X a real value from $\mathbb{R}^{\geq 0}$. We say t satisfies cc, written $t \in cc$, if cc evaluates to true under valuation t. $t + d$ denotes the valuation derived from t by increasing the assigned value on each clock variable by $d \in \mathbb{R}^{\geq 0}$ time units. $t[rs \mapsto 0]$ denotes the valuation obtained from t by resetting the clock variables in rs to 0. Sometimes we use 0 for the clock valuation that maps all clock variables to 0.

Definition 1. *A* timed I/O automaton *(TIOA) is a tuple* $(C, I, O, L, l^0, AT, Inv, coInv)$*, where:*

- $C \subseteq X$ *is a finite set of clock variables*
- A *(*$= I \uplus O$*) is a finite alphabet, consisting of inputs* I *and outputs* O
- L *is a finite set of* locations *and* $l^0 \in L$ *is the* initial location
- $AT \subseteq L \times CC(C) \times A \times 2^C \times L$ *is a set of* action transitions
- $Inv : L \to CC(C)$ *and* $coInv : L \to CC(C)$ *assign* invariants *and* co-invariants *to states, each of which is a downward-closed clock constraint.*

We use l, l', l_i to range over L and use $l \xrightarrow{g,a,rs} l'$ as a shorthand for $(l, g, a, rs, l') \in AT$. $g : CC(C)$ is the enabling guard of the transition, $a \in A$ the action, and rs the subset of clock variables to be reset.

Our TIOAs are timed automata that distinguish input from output and invariant from co-invariant. They are similar to existing variants of timed automata with input/output distinction, except for the introduction of co-invariants and non-insistence on input-enabledness. While invariants specify the bounds beyond which time may not progress, co-invariants specify the bounds beyond which the system will *time-out* and enter error states. It is designed for the assume/guarantee specification of timed components, in order to specify both the assumptions made by the component on the inputs and the guarantees provided by the component on the outputs, with respect to timing constraints.

Guards on output transitions express *safety timing guarantees*, while guards on input transitions express *safety timing assumptions*. On the other hand, invariants (urgency) express *liveness timing guarantees* on the outputs at the locations they decorate, while co-invariants (time-out) express *liveness timing assumptions* on the inputs at those locations.

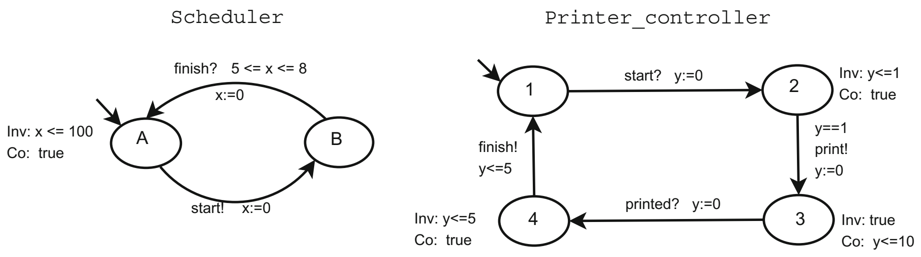


Fig. 1. Job scheduler and printer controller

When two components are composed, the parallel composition automatically checks whether the guarantees provided by one component meet the assumptions required by the other. For instance, the unexpected arrival of an input at a particular location and time (indicated by a non-enabled transition) leads to a *safety error* in the parallel composition. The non-arrival of an expected input at a location before its time-out (specified by the co-invariant) leads to a *bounded-liveness error* in the parallel composition.

Example. Figure 1 depicts TIOAs representing a job scheduler together with a printer controller. The invariant at location A of the scheduler forces a bounded-liveness guarantee on outputs in that location. As time must be allowed to progress beyond $t = 100$, the *start* action must be fired within the range $0 \leq t \leq 100$. After *start* has been fired, the clock x is reset to 0 and the scheduler waits (possibly indefinitely) for the job to *finish*. If the job does finish, the scheduler is only willing for this to take place between $5 \leq t \leq 8$ after the job started (safety assumption), otherwise an unexpected input error will be thrown.

The controller waits for the job to *start*, after which it will wait exactly 1 time unit before issuing *print* (forced by the invariant $y \leq 1$ on state 2 and the guard $y = 1$). The controller now requires the printer to indicate the job is *printed* within 10 time units of being sent to the printer, otherwise a time-out error on inputs will occur (co-invariant $y \leq 10$ in state 3 as liveness assumption). After the job has finished printing, the controller must indicate to the scheduler that the job has *finish*ed within 5 time units.

Notation. For a set of input actions I and a set of output actions O, define $tA = I \uplus O \uplus \mathbb{R}^{>0}$ to be the set of *timed actions*, $tI = I \uplus \mathbb{R}^{>0}$ to be the set of *timed inputs*, and $tO = O \uplus \mathbb{R}^{>0}$ to be the set of *timed outputs*. We use symbols like α, β, etc. to range over tA.

A *timed word* (ranged over by w, w', w_i etc.) is a finite mixed sequence of positive real numbers ($\mathbb{R}^{>0}$) and visible actions such that *no two numbers are adjacent to one another*. For instance, $\langle 0.33, a, 1.41, b, c, 3.1415 \rangle$ is a timed word denoting the observation that action a occurs at 0.33 time units, then another 1.41 time units lapse before the simultaneous occurrence of b and c, which is followed by 3.1415 time units of no event occurrence. ϵ denotes the empty word.

Concatenation of timed words w and w' is obtained by appending w' onto the end of w and coalescing adjacent reals (summing them). Prefix/extension are defined as usual by concatenation. We write $w \restriction tA_0$ for the projection of w onto timed alphabet tA_0, which is defined by removing from w all actions not inside tA_0 and coalescing adjacent reals.

2.2 Semantics as Timed I/O Transition Systems

The semantics of TIOAs are given as timed I/O transition systems, which are a special class of infinite labelled transition systems.

Definition 2. *A* timed I/O transition system *(TIOTS) is a tuple* $\mathcal{P} = \langle I, O, S, s^0, \rightarrow\rangle$*, where* I *and* O *are the input and output actions respectively,* $S = (L \times \mathbb{R}^C) \uplus \{\bot, \top\}$ *is a set of states,* $s^0 \in S$ *is the designated initial state, and* $\rightarrow \subseteq S \times (I \uplus O \uplus \mathbb{R}^{>0}) \times S$ *is the action and time-labelled transition relation.*

The states of the TIOTS for a TIOA capture the configuration of the automaton, i.e. its location and clock valuation. Therefore, each state of the TIOTS is a pair drawn from $L \times \mathbb{R}^C$, which we refer to as the set of *plain states*. In addition, we introduce two special states $\bot$ and $\top$, which are required for the semantic mapping of disabled inputs/outputs, invariants and co-invariants. In the rest of the paper, we use p, p', p_i to range over $P = L \times \mathbb{R}^C$ while s, s', s_i range over S.

$\bot$ is the so-called *inconsistent state*, arising through assumption/guarantee mismatches, i.e. safety and bounded-liveness errors. $\top$ is the so-called *timestop state*, representing the *magic moment* from which time stops elapsing and no error can occur. We assume that $\top$ refines plain states, which in turn refine $\bot$. For technical convenience (e.g. ease of defining time additivity and trace semantics), we require that $\top$ and $\bot$ are a *chaotic states*, i.e. states having self-loops for each $\alpha \in tA$.

On TIOTSs, a disabled input in a state p is equated to an input transition from p to $\bot$, while a disabled output/delay in p is equated to an output/delay from p to $\top$. The intuition here comes from the I/O game perspective. The component controls output and delay, while the environment controls input. $\bot$ is the losing state for the environment, so an input transition from p to $\bot$ is a transition that the environment tries to avoid at all cost (unless there is no choice). $\top$ is the losing state for the component, so an output/delay transition from p to $\top$ is a transition that the component tries to avoid at any cost. Thus we can have two semantic-preserving transformations on TIOTSs.

The $\bot$-*completion* of a TIOTS $\mathcal{P}$, denoted $\mathcal{P}^\bot$, adds an a-labelled transition from p to $\bot$ for every $p \in P$ $(= L \times \mathbb{R}^C)$ and $a \in I$ s.t. a is not enabled at p.[1] The $\top$-*completion*, denoted $\mathcal{P}^\top$, adds an α-labelled transition from p to $\top$ for every $p \in P$ and $\alpha \in tO$ s.t. α is not enabled at p.

Now, the transition relation $\rightarrow$ of the TIOTS is derived from the execution semantics of the TIOA.

Definition 3. *Let* $\mathcal{P}$ *be a TIOA. The execution semantics of* $\mathcal{P}$ *is a TIOTS* $\langle I, O, S, s^0, \rightarrow\rangle$*, where:*

[1] $\bot$-completion will make a TIOTS *input-receptive*, i.e. input-enabled in all states.

- $S = (L \times \mathbb{R}^C) \uplus \{\bot, \top\}$
- $s^0 = \top$ *providing* $0 \notin Inv(l^0)$, $s^0 = \bot$ *providing* $0 \in Inv(l^0) \wedge \neg coInv(l^0)$ *and* $s^0 = (l^0, 0)$ *providing* $0 \in Inv(l^0) \wedge coInv(l^0)$,
- $\rightarrow$ *is the smallest relation satisfying:*
  1. *If* $l \xrightarrow{g,a,rs} l'$, $t' = t[rs \mapsto 0]$, $t \in Inv(l) \wedge coInv(l) \wedge g$, *then:*
     (a) plain action: $(l, t) \xrightarrow{a} (l', t')$ *providing* $t' \in Inv(l') \wedge coInv(l')$
     (b) error action: $(l, t) \xrightarrow{a} \bot$ *providing* $t' \in Inv(l') \wedge \neg coInv(l')$
     (c) magic action: $(l, t) \xrightarrow{a} \top$ *providing* $t' \in \neg Inv(l')$ *and* $a \in I$.
  2. plain delay: $(l, t) \xrightarrow{d} (l, t + d)$ *if* $t, t + d \in Inv(l) \wedge coInv(l)$
  3. time-out delay: $(l, t) \xrightarrow{d} \bot$ *if* $t \in Inv(l) \wedge coInv(l)$, $t + d \notin coInv(l)$ *and* $\exists\, 0 < \delta \leq d : t + \delta \in Inv(l) \wedge \neg coInv(l)$.

Note that our semantics tries to minimise the use of transitions leading to $\top/\bot$ states. Thus there are no delay or output transitions leading to $\top$. However, there are *implicit timestops*, which we capture using the concept of *semi-timestop* (i.e. semi-$\top$). We say a plain state p is a *semi-*$\top$ iff 1) all output transitions enabled in p and all of its time-passing successors lead to the $\top$ state, and 2) there exists $d \in \mathbb{R}^{>0}$ s.t. $p \xrightarrow{d} \top$ or d is not enabled in p. Thus a semi-$\top$ is a state in which it is impossible for the component to avoid the timestop without suitable inputs from the environment.

The introduction of timestop ($\top$), which can model the operation of stopping the system clock, is an unconventional aspect of our semantics. Certain real-world systems have an inherent ability to stop the clock, e.g. [6,7], which are related to embedded systems and circuit design. When the suspension of clocks is not meaningful, it is necessary to remove timestop in order to leave the so-called realisable behaviour. Timestop is useful even for timestop free systems, as it can significantly simplify operations, such as quotient and conjunction.

TIOTS terminology. We say a TIOTS is *deterministic* iff $s \xrightarrow{\alpha} s' \wedge s \xrightarrow{\alpha} s''$ implies $s' = s''$, and is *time additive* providing $p \xrightarrow{d_1+d_2} s'$ iff $p \xrightarrow{d_1} s$ and $s \xrightarrow{d_2} s'$ for some s. In the sequel, we only consider time-additive TIOTSs.

Given a TIOTS $\mathcal{P}$, a timed word can be derived from a finite execution of $\mathcal{P}$ by extracting the labels in each transition and coalescing adjacent reals. The timed words derived from such executions are called *traces* of $\mathcal{P}$. We use tt, tt', tt_i to range over traces and write $s^0 \overset{tt}{\Rightarrow} s$ to denote a finite execution producing tt and leading to s.

2.3 Operational Specification Theory

In this section we develop a compositional specification theory for TIOTSs based on the operations of parallel composition $\|$, conjunction $\wedge$, disjunction $\vee$ and quotient %. The operators are defined via transition rules that are a variant on synchronised product.

Parallel composition yields a TIOTS that represents the combined effect of its operands interacting with one another. The remaining operations must

Table 1. State representations under composition operators

$\Vert$	$\top$	p_0	$\bot$
$\top$	$\top$	$\top$	$\top$
p_1	$\top$	$p_0{\times}p_1$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$

$\wedge$	$\top$	p_0	$\bot$
$\top$	$\top$	$\top$	$\top$
p_1	$\top$	$p_0{\times}p_1$	p_1
$\bot$	$\top$	p_0	$\bot$

$\vee$	$\top$	p_0	$\bot$
$\top$	$\top$	p_0	$\bot$
p_1	p_1	$p_0{\times}p_1$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\%$	$\top$	p_0	$\bot$
$\top$	$\bot$	$\bot$	$\bot$
p_1	$\top$	$p_0{\times}p_1$	$\bot$
$\bot$	$\top$	$\top$	$\bot$

be explained with respect to a refinement relation, which corresponds to safe-substitutivity in our theory. A TIOTS is a refinement of another if it will work in any environment that the original worked in without introducing safety or bounded-liveness errors. Conjunction yields the coarsest TIOTS that is a refinement of its operands, while disjunction yields the finest TIOTS that is refined by both of its operands. The operators are thus equivalent to the join and meet operations on TIOTSs[2]. Quotient is the adjoint of parallel composition, meaning that $\mathcal{P}_0\%\mathcal{P}_1$ is the coarsest TIOTS such that $(\mathcal{P}_0\%\mathcal{P}_1)\|\mathcal{P}_1$ is a refinement of $\mathcal{P}_0$.

Let $\mathcal{P}_i = \langle I_i, O_i, S_i, s_i^0, \rightarrow_i\rangle$ for $i \in \{0,1\}$ be two TIOTSs that are both $\bot$ and $\top$-completed, satisfying (wlog) $S_0 \cap S_1 = \{\bot, \top\}$. The composition of $\mathcal{P}_0$ and $\mathcal{P}_1$ under the operation $\otimes \in \{\|, \wedge, \vee, \%\}$, written $\mathcal{P}_0 \otimes \mathcal{P}_1$, is only defined when certain *composability* restrictions are imposed on the alphabets of the TIOTSs. $\mathcal{P}_0 \parallel \mathcal{P}_1$ is only defined when the output sets of $\mathcal{P}_0$ and $\mathcal{P}_1$ are disjoint, because an output should be controlled by at most one component. Conjunction and disjunction are only defined when the TIOTSs have *identical alphabets* (i.e. $O_0 = O_1$ and $I_0 = I_1$). This restriction can be relaxed at the expense of more cumbersome notation, which is why we focus on the simpler case in this paper. For the quotient, we require that the alphabet of $\mathcal{P}_0$ *dominates* that of $\mathcal{P}_1$ (i.e. $A_1 \subseteq A_0$ and $O_1 \subseteq O_0$), in addition to $\mathcal{P}_1$ being a deterministic TIOTS. As quotient is a synthesis operator, it is difficult to give a definition using just *state-local* transition rules, since quotient needs global information about the transition systems. This is why we insist on $\mathcal{P}_1$ being deterministic[3].

Definition 4. *Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be TIOTSs composable under $\otimes \in \{\|, \wedge, \vee, \%\}$. Then $\mathcal{P}_0 \otimes \mathcal{P}_1 = \langle I, O, S, s^0, \rightarrow\rangle$ is the TIOTS where:*

- *If $\otimes = \|$, then $I = (I_0 \cup I_1) \setminus O$ and $O = O_0 \cup O_1$*
- *If $\otimes \in \{\wedge, \vee\}$, then $I = I_0 = I_1$ and $O = O_0 = O_1$*
- *If $\otimes = \%$, then $I = I_0 \cup O_1$ and $O = O_0 \setminus O_1$*
- *$S = (P_0 \times P_1) \uplus P_0 \uplus P_1 \uplus \{\top, \bot\}$*
- *$s^0 = s_0^0 \otimes s_1^0$*
- *$\rightarrow$ is the smallest relation containing $\rightarrow_0 \cup \rightarrow_1$, and satisfying the rules:*

$$\frac{p_0 \xrightarrow{\alpha}_0 s_0' \quad p_1 \xrightarrow{\alpha}_1 s_1'}{p_0 \otimes p_1 \xrightarrow{\alpha} s_0' \otimes s_1'} \qquad \frac{p_0 \xrightarrow{a}_0 s_0' \quad a \notin A_1}{p_0 \otimes p_1 \xrightarrow{a} s_0' \otimes p_1} \qquad \frac{p_1 \xrightarrow{a}_0 s_1' \quad a \notin A_0}{p_0 \otimes p_1 \xrightarrow{a} p_0 \otimes s_1'}$$

[2] As we write $A \sqsubseteq B$ to mean A is refined by B, our operators $\wedge$ and $\vee$ are reversed in comparison to the standard symbols for meet and join.

[3] Technically speaking, the problem is a consequence of state quotient being right-distributive but not left-distributive over state disjunction (cf Table 1).

We adopt the notation of $s_0 \otimes s_1$ for states, where the associated interpretation is supplied in Table 1. Furthermore, given two plain states $p_i = (l_i, t_i)$ for $i \in \{0, 1\}$, we define $p_0 \times p_1 = ((l_0, l_1), t_0 \uplus t_1)$.

Table 1 tells us how states should be combined under the composition operators. For parallel, a state is magic if one component state is magic, and a state is error if one component is error while the other is not magic. For conjunction, encountering error in one component implies the component can be discarded and the rest of the composition behaves like the other component. The conjunction table follows the intuition of the join operation on the refinement preorder. Similarly for disjunction. Quotient is the adjoint of parallel composition. If the second component state does not refine the first, the quotient will try to rescue the refinement by producing $\top$ (so that its composition with the second will refine the first). If the second component state does refine the first, the quotient will produce the least refined value so that its composition with the second will not break the refinement.

An *environment* for a TIOTS $\mathcal{P}$ is any TIOTS $\mathcal{Q}$ such that the alphabet of $\mathcal{Q}$ is *complementary* to that of $\mathcal{P}$, meaning $I_{\mathcal{P}} = O_{\mathcal{Q}}$ and $O_{\mathcal{P}} = I_{\mathcal{Q}}$. Refinement in our framework corresponds to contextual substitutability, in which the context is an arbitrary environment.

Definition 5. *Let $\mathcal{P}_{imp}$ and $\mathcal{P}_{spec}$ be TIOTSs with identical alphabets. $\mathcal{P}_{imp}$* refines *$\mathcal{P}_{spec}$, denoted $\mathcal{P}_{spec} \sqsubseteq \mathcal{P}_{imp}$, iff for all environments $\mathcal{Q}$, $\mathcal{P}_{spec} \parallel \mathcal{Q}$ is $\bot$-free implies $\mathcal{P}_{imp} \parallel \mathcal{Q}$ is $\bot$-free. We say $\mathcal{P}_{imp}$ and $\mathcal{P}_{spec}$ are* substitutively equivalent*, i.e. $\mathcal{P}_{spec} \simeq \mathcal{P}_{imp}$, iff $\mathcal{P}_{imp} \sqsubseteq \mathcal{P}_{spec}$ and $\mathcal{P}_{spec} \sqsubseteq \mathcal{P}_{imp}$.*

It is obvious that $\simeq$ induces the weakest equivalence on TIOTSs that preserves $\bot$-freeness. In the sequel, we give two concrete characterisations of $\simeq$ and show it to be a congruence w.r.t. the operators of the specification theory.

The operational definition of quotient requires $\mathcal{P}_1$ to be deterministic. For any TIOTS $\mathcal{P}$, a semantically-equivalent deterministic component can be obtained, denoted $\mathcal{P}^D$, by means of a modified subset construction acting on $(\mathcal{P}^{\bot})^{\top}$. For any subset S_0 of states reachable by a given trace, we only keep those which are minimal w.r.t. the state refinement relation. So if the current state subset S_0 contains $\bot$, the procedure reduces S_0 to $\bot$; if $\bot \notin S_0 \neq \{\top\}$, it reduces S_0 by removing any potential $\top$ in S_0.[4]

Proposition 1. *For any TIOTS $\mathcal{P}$, it holds that $\mathcal{P} \simeq \mathcal{P}^D$.*

Equipped with determinisation, quotient is a fully defined operator on any pair of TIOTSs. Furthermore, we can give an alternative (although substitutively equivalent) formulation of quotient as the derived operator $(\mathcal{P}_0^{\neg} \parallel \mathcal{P}_1)^{\neg}$, where $\neg$ is a mirroring operation that first determinises its argument, then interchanges the input and output sets, as well as the $\top$ and $\bot$ states.

[4] A detailed definition of transforming untimed non-deterministic systems into substitutively-equivalent deterministic ones is contained in Definition 4.2 of [8].

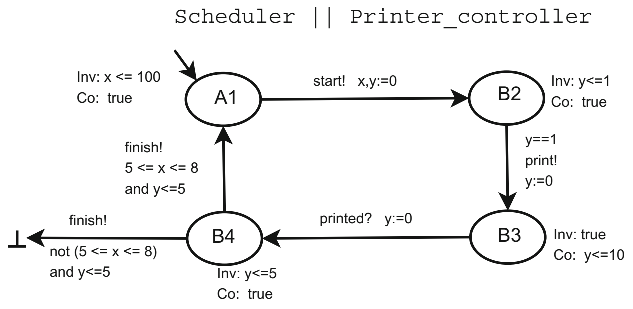


Fig. 2. Parallel composition of the job scheduler and printer controller

Example. Figure 2 shows the parallel composition of the job scheduler with the printer controller. In the transition from $B4$ to $A1$, the guard combines the effects of the constraints on the clocks x and y. As *finish* is an output of the controller, it can be fired at a time when the scheduler is not expecting it, meaning that a safety error will occur. This is indicated by the transition to $\perp$ when the guard constraint $5 \leq x \leq 8$ is not satisfied.

3 Timed I/O Game

Our specification theory can be seen as an I/O game between a *component* and an *environment* that uses a *coin* to break ties. The specification of a component (in the form of a TIOA or TIOTS) is built to encode the set of strategies possible for the component in the game (just like an NFA encodes a set of words).

- Given two TIOTSs $\mathcal{P}$ and $\mathcal{Q}$ with identical alphabets, we say $\mathcal{P}$ is a partial unfolding [17] of $\mathcal{Q}$ if there exists a function f from $S_\mathcal{P}$ to $S_\mathcal{Q}$ s.t. 1) f maps $\top$ to $\top$, $\perp$ to $\perp$, and plain states to plain states, 2) $f(s^0_\mathcal{P}) = s^0_\mathcal{Q}$, and 3) $p \xrightarrow{\alpha}_\mathcal{P} s \Rightarrow f(p) \xrightarrow{\alpha}_\mathcal{Q} f(s)$.
- We say an acyclic TIOTS is a *tree* if 1) there does not exist a pair of transitions in the form of $p \xrightarrow{a} p''$ and $p' \xrightarrow{d} p''$, 2) $p \xrightarrow{a} p'' \wedge p' \xrightarrow{b} p''$ implies $p = p'$ and $a = b$ and 3) $p \xrightarrow{d} p'' \wedge p' \xrightarrow{d} p''$ implies $p = p'$.
- We say an acyclic TIOTS is a *simple path* if 1) $p \xrightarrow{a} s' \wedge p \xrightarrow{\alpha} s''$ implies $s' = s''$ and $a = \alpha$ and 2) $p \xrightarrow{d} s' \wedge p \xrightarrow{d} s''$ implies $s' = s''$.
- We say a simple path $\mathcal{L}$ is a *run* of $\mathcal{P}$ if $\mathcal{L}$ is a partial unfolding of $\mathcal{P}$.

Strategies. A *strategy* $\mathcal{G}$ is a deterministic tree TIOTS s.t. each plain state in $\mathcal{G}$ is ready to accept all possible inputs by the environment, but allows a single move (delay or output) by the component, i.e. $eb_\mathcal{G}(p) = I \uplus mv_\mathcal{G}(p)$ s.t. $mv_\mathcal{G}(p) = \{a\}$ for some $a \in O$ or $mv_\mathcal{G}(p) \subseteq \mathbb{R}^{>0}$, where $eb_\mathcal{G}(p)$ denotes the set of enabled timed actions in state p of LTS $\mathcal{G}$, and $mv_\mathcal{G}(p)$ denotes the unique component move allowed by $\mathcal{G}$ at p.

A TIOTS $\mathcal{P}$ *contains* a strategy $\mathcal{G}$ if $\mathcal{G}$ is a partial unfolding of $(\mathcal{P}^\perp)^\top$. The set of strategies contained in $\mathcal{P}$ is denoted $stg(\mathcal{P})$. Since it makes little sense to

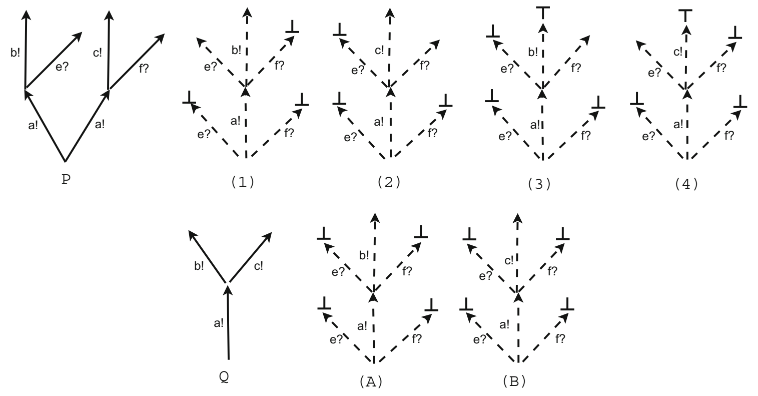


Fig. 3. Strategy example

distinguish strategies that are isomorphic, we will freely use strategies to refer to their isomorphism classes and write $\mathcal{G} = \mathcal{G}'$ to mean $\mathcal{G}$ and $\mathcal{G}'$ are isomorphic.

Figure 3 illustrates the idea of strategies. For simplicity, we use two untimed transition systems $\mathcal{P}$ and $\mathcal{Q}$ with identical alphabets $I = \{e, f\}$ and $O = \{a, b, c\}$. The transition systems use solid lines, while strategies use dotted lines. Plain states are unmarked, while the $\top$ and $\bot$ states are labelled as such[5]. A subset of the strategies for $\mathcal{P}$ and $\mathcal{Q}$ are shown on the right hand side of the respective components. Note that strategies 3 and 4 arise through $\top$-completion.

Comparing strategies. When the game is played, the component tries to avoid reaching $\top$, while the environment tries to avoid reaching $\bot$. Strategies in $stg(\mathcal{P})$ vary in their effectiveness to achieve this objective, which induces a hierarchy on strategies that closely resemble one another. We say $\mathcal{G}$ and $\mathcal{G}'$ are *affine* if $s^0_{\mathcal{G}} \overset{tt}{\Rightarrow} p$ and $s^0_{\mathcal{G}'} \overset{tt}{\Rightarrow} p'$ implies $mv_{\mathcal{G}}(p) = mv_{\mathcal{G}'}(p')$. Intuitively, it means $\mathcal{G}$ and $\mathcal{G}'$ propose the same move at the 'same' states. For instance, the strategies 1, 3 and A in Figure 3 are pairwise affine and so are the strategies 2, 4 and B.

Given two affine strategies $\mathcal{G}$ and $\mathcal{G}'$, we say $\mathcal{G}$ is *more aggressive* than $\mathcal{G}'$, denoted $\mathcal{G} \preceq \mathcal{G}'$, if 1) $s^0_{\mathcal{G}'} \overset{tt}{\Rightarrow} \bot$ implies there is a prefix tt_0 of tt s.t. $s^0_{\mathcal{G}} \overset{tt_0}{\Longrightarrow} \bot$ and 2) $s^0_{\mathcal{G}} \overset{tt}{\Rightarrow} \top$ implies there is a prefix tt_0 of tt s.t. $s^0_{\mathcal{G}'} \overset{tt_0}{\Longrightarrow} \top$. Intuitively, it means $\mathcal{G}$ can reach $\bot$ faster but $\top$ slower than $\mathcal{G}'$. $\preceq$ forms a partial order over $stg(\mathcal{P})$, or more generally, over any set of strategies with identical alphabets. For instance, strategy A is more aggressive than 1 and 3, while strategy B is more aggressive than 2 and 4.

When the game is played, the component $\mathcal{P}$ prefers to use the maximally aggressive strategies in $stg(\mathcal{P})$[6]. Thus two components that differ only in non-maximally aggressive strategies should be equated. We define the *strategy semantics* of component $\mathcal{P}$ to be $[\mathcal{P}]_s = \{\mathcal{G}' \mid \exists \mathcal{G} \in stg(\mathcal{P}) : \mathcal{G} \preceq \mathcal{G}'\}$, i.e. the upward-closure of $stg(\mathcal{P})$ w.r.t. $\preceq$.

[5] For simplicity, we allow multiple copies of $\top$ and $\bot$, which are assumed to be chaotic.
[6] This is because our semantics is designed to preserve $\bot$ rather than $\top$.

Game rules. When a component strategy $\mathcal{G}$ is played against an environment strategy $\mathcal{G}'$, at each game state (i.e. a product state $p_{\mathcal{G}} \times p_{\mathcal{G}'}$) $\mathcal{G}$ and $\mathcal{G}'$ each propose a move (i.e. $mv_{\mathcal{G}}(p_{\mathcal{G}})$ and $mv_{\mathcal{G}'}(p_{\mathcal{G}'})$). If one of them is a delay and the other is an action, the action will prevail. If both propose delay moves (i.e. $mv_{\mathcal{G}}(p_{\mathcal{G}}), mv_{\mathcal{G}'}(p_{\mathcal{G}'}) \subseteq \mathbb{R}^{>0}$), the smaller one (w.r.t. set containment) will prevail.[7]

Since a delay move proposed at a strategy state is the maximal set of possible delays enabled at that state, the next move proposed at the new state after firing the set must be an action move (due to time additivity). Thus a play cannot have two consecutive delay moves.

If, however, both propose action moves, there will be a tie, which will be resolved by tossing the coin. For uniformity's sake, the coin can be treated as a special component. A strategy of the coin is a function h from tA^* to $\{0, 1\}$. We denote the set of all possible coin strategies as H.

A play of the game can be formalised as a composition of three strategies, one each from the component, environment and coin, denoted $\mathcal{G}_{\mathcal{P}} \parallel_h \mathcal{G}_{\mathcal{Q}}$. At a current game state $p_{\mathcal{P}} \times p_{\mathcal{Q}}$, if the prevailing action is α and we have $p_{\mathcal{P}} \xrightarrow{\alpha} s'_{\mathcal{P}}$ and $p_{\mathcal{Q}} \xrightarrow{\alpha} s'_{\mathcal{Q}}$, then the next game state is $s_{\mathcal{P}} \parallel s_{\mathcal{Q}}$. The play will stop when it reaches either $\top$ or $\bot$. The composition will produce a simple path $\mathcal{L}$ that is a run of $\mathcal{P} \parallel \mathcal{Q}$. Since $\mathcal{P} \parallel \mathcal{Q}$ gives rise to a *closed system* (i.e. the input alphabet is empty), a run of $\mathcal{P} \parallel \mathcal{Q}$ is a strategy of $\mathcal{P} \parallel \mathcal{Q}$.

Thus, strategy composition of $\mathcal{P}$ and $\mathcal{Q}$ is closely related to their parallel composition: $stg(\mathcal{P} \parallel \mathcal{Q}) = \{\mathcal{G}_{\mathcal{P}} \parallel_h \mathcal{G}_{\mathcal{Q}} \mid \mathcal{G}_{\mathcal{P}} \in stg(\mathcal{P}), \mathcal{G}_{\mathcal{Q}} \in stg(\mathcal{Q}) \text{ and } h \in H\}$.

Parallel composition. Strategy composition, like component parallel composition, can be generalised to any pair of components $\mathcal{P}$ and $\mathcal{Q}$ with *composable alphabets*. That is, $O_{\mathcal{P}} \cap O_{\mathcal{Q}} = \{\}$. For such $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{G}_{\mathcal{P}} \parallel_h \mathcal{G}_{\mathcal{Q}}$ gives rise to a tree rather than a simple path TIOTS. That is, at each game state $p_{\mathcal{P}} \times p_{\mathcal{Q}}$, besides firing the prevailing $\alpha \in tO_{\mathcal{P}} \cup tO_{\mathcal{Q}}$, we need also to fire 1) all the synchronised inputs, i.e. $e \in I_{\mathcal{P}} \cap I_{\mathcal{Q}}$, and reach the new game state $s_{\mathcal{P}} \parallel s_{\mathcal{Q}}$ (assuming $p_{\mathcal{P}} \xrightarrow{e} s_{\mathcal{P}}$ and $p_{\mathcal{Q}} \xrightarrow{e} s_{\mathcal{Q}}$) and 2) all the independent inputs, i.e. $e \in (I_{\mathcal{P}} \cup I_{\mathcal{Q}}) \setminus (A_{\mathcal{P}} \cap A_{\mathcal{Q}})$, and reach the new game state $s_{\mathcal{P}} \times p_{\mathcal{Q}}$ or $p_{\mathcal{P}} \times s_{\mathcal{Q}}$. It is easy to verify that $\mathcal{G}_{\mathcal{P}} \parallel_h \mathcal{G}_{\mathcal{Q}}$ is a strategy of $\mathcal{P} \parallel \mathcal{Q}$.

Conjunction/disjunction. Strategy conjunction (&) and *strategy disjunction* (+) are binary operators defined only on pairs of affine strategies, by $\mathcal{G} \& \mathcal{G}' = \mathcal{G} \wedge \mathcal{G}'$ and $\mathcal{G} + \mathcal{G}' = \mathcal{G} \vee \mathcal{G}'$. If $\mathcal{G}$ and $\mathcal{G}'$ are not affine, $\mathcal{G} \wedge \mathcal{G}'$ and $\mathcal{G} \vee \mathcal{G}'$ may not produce a strategy. From Figure 3, the disjunction of strategies 1 and 2 will produce a transition system that stops to output after the a transition.

Refinement. Equality of strategies induces an equivalence on TIOTSs: $\mathcal{P}$ and $\mathcal{Q}$ are *strategy equivalent* iff $[\mathcal{P}]_s = [\mathcal{Q}]_s$. However, strategy equivalence is too fine for the purpose of *substitutive refinement* (cf Definition 5). For instance,

[7] Note that all invariants and co-invariants are downward-closed. Thus a delay move can be respresented as a time interval from 0 to some $d \in \mathbb{R}^{\geq 0}$.

transition systems $\mathcal{P}$ and $\mathcal{Q}$ in Figure 3 are substitutively equivalent, but are not strategy equivalent, because 1, 2, 3 and 4 are strategies of $\mathcal{Q}$ (due to upward-closure w.r.t. $\preceq$), while A and B are not strategies of $\mathcal{P}$.

However, we demonstrate that *substitutive equivalence is reducible to strategy equivalence* providing we perform *disjunction closure* on strategies.

Lemma 1. *Given a pair of affine component strategies $\mathcal{G}_0$ and $\mathcal{G}_1$, $\mathcal{G}_0 \parallel_h \mathcal{G}$ and $\mathcal{G}_1 \parallel_h \mathcal{G}$ are $\perp$-free for a pair of environment and coin strategies $\mathcal{G}$ and h iff $\mathcal{G}_0 + \mathcal{G}_1 \parallel_h \mathcal{G}$ is $\perp$-free.*

We say Π^+ is a *disjunction closure* of set of strategies Π iff it is the least superset of Π s.t. $\mathcal{G} + \mathcal{G}' \in \Pi^+$ for all pairs of affine strategies $\mathcal{G}, \mathcal{G}' \in \Pi^+$. It is easy to see disjunction closure preserves upward-closedness of strategy sets.

Proposition 2. *Disjunction closure is determinisation: $[\mathcal{P}^D]_s = [\mathcal{P}^D]_s^+ = [\mathcal{P}]_s^+$.*

Lemma 2. *For any TIOTS $\mathcal{P}$, $[\mathcal{P}^\neg]_s^+ = \{\mathcal{G}_{\mathcal{P}^\neg} \mid \forall\, \mathcal{G}_\mathcal{P} \in [\mathcal{P}]_s^+, h \in H : \mathcal{G}_{\mathcal{P}^\neg} \parallel_h \mathcal{G}_\mathcal{P}$ is $\perp$-free$\}$.*

Theorem 1. *Given TIOTSs $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{P} \sqsubseteq \mathcal{Q}$ iff $[\mathcal{Q}]_s^+ \subseteq [\mathcal{P}]_s^+$.*

Looking at Figure 3, the disjunction of strategies 1 and 3 produces A, while the disjunction of strategies 2 and 4 produces B. Thus $[\mathcal{P}]_s^+ = [\mathcal{Q}]_s^+$.

Relating operational composition to strategies. The operations of parallel composition, conjunction, disjunction and quotient defined on the operational models of TIOTSs (Section 2.3) can be characterised by simple operations on strategies in the game-based setting.

Lemma 3. *For $\parallel$-composable TIOTSs $\mathcal{P}$ and $\mathcal{Q}$, $[\mathcal{P} \parallel \mathcal{Q}]_s^+ = \{\mathcal{G}_{\mathcal{P}\parallel\mathcal{Q}} \mid \exists\, \mathcal{G}_\mathcal{P} \in [\mathcal{P}]_s^+, \mathcal{G}_\mathcal{Q} \in [\mathcal{Q}]_s^+, h \in H : \mathcal{G}_\mathcal{P} \parallel_h \mathcal{G}_\mathcal{Q} \preceq \mathcal{G}_{\mathcal{P}\parallel\mathcal{Q}}\}$.*

Lemma 4. *For $\vee$-composable TIOTSs $\mathcal{P}$ and $\mathcal{Q}$, $[\mathcal{P} \vee \mathcal{Q}]_s^+ = ([\mathcal{P}]_s^+ \cup [\mathcal{Q}]_s^+)^+$.*

Lemma 5. *For $\wedge$-composable TIOTSs $\mathcal{P}$ and $\mathcal{Q}$, $[\mathcal{P} \wedge \mathcal{Q}]_s^+ = [\mathcal{P}]_s^+ \cap [\mathcal{Q}]_s^+$.*

Lemma 6. *For $\%$-composable TIOTSs $\mathcal{P}$ and $\mathcal{Q}$, $[\mathcal{P}\%\mathcal{Q}]_s^+ = \{\mathcal{G}_{\mathcal{P}\%\mathcal{Q}} \mid \forall\, \mathcal{G}_\mathcal{Q} \in [\mathcal{Q}]_s^+, h \in H : \mathcal{G}_{\mathcal{P}\%\mathcal{Q}} \parallel_h \mathcal{G}_\mathcal{Q} \in [\mathcal{P}]_s^+\}$.*

Thus, conjunction and disjunction are the join and meet operations, and quotient produces the coarsest TIOTS s.t. $(\mathcal{P}_0\%\mathcal{P}_1)\|\mathcal{P}_1$ is a refinement of $\mathcal{P}_0$.

Theorem 2. *$\simeq$ is a congruence w.r.t. $\parallel$, $\vee$, $\wedge$ and $\%$ subject to composability.*

Summary. Strategy semantics has given us a weakest $\perp$-preserving congruence (i.e. $[\mathcal{P}]_s^+$) for timed specification theories based on operators for (parallel) composition, conjunction, disjunction and quotient. Strategy semantics captures nicely the game-theoretical nature as well as the operational intuition of the specification theory. In the next section, we give a more declarative characterisation of the equivalence by means of timed traces.

4 Declarative Specification Theory

In this section, we develop a compositional specification theory based on timed traces. We introduce the concept of a timed-trace structure, which is an abstract representation for a timed component. The timed-trace structure contains essential information about the component, for checking whether it can be substituted with another in a safety and liveness preserving manner.

Given any TIOTS $\mathcal{P} = \langle I, O, S, s^0, \rightarrow \rangle$, we can extract three sets of traces from $(\mathcal{P}^{\perp})^{\top}$: TP a set of timed traces leading to plain states; TE a set of timed traces leading to the error state $\perp$; and TM a set of timed traces leading to the magic state $\top$. TE and TM are extension-closed as $\top$ and $\perp$ are chaotic, while TP is prefix-closed. Due to $\top/\perp$-completion, it is easy to verify $TE \cup TP \cup TM$ gives rise to the full set of timed traces tA^*; thus TP and TE are sufficient.

However, TP and TE contain more information than necessary for our substitutive refinement, which is designed to preserve $\perp$-freeness. For instance, adding any trace $tt \in TE$ to TP should not change the semantics of the component. Based on a slight abstraction of the two sets, we can thus define a *trace structure* $\mathcal{TT}(\mathcal{P})$ as the semantics of $\mathcal{P}$.

Definition 6 (Trace structure). *$\mathcal{TT}(\mathcal{P}) := (I, O, TR, TE)$, where $TR := TE \cup TP$ the set of realisable traces. Obviously, TR is prefix-closed.*

From hereon let $\mathcal{P}_0$ and $\mathcal{P}_1$ be two TIOTSs with trace structures $\mathcal{TT}(\mathcal{P}_i) := (I_i, O_i, TR_i, TE_i)$ for $i \in \{0, 1\}$. Define $\bar{i} = 1 - i$.

The substitutive refinement relation $\sqsubseteq$ in Section 2.3 can equally be characterised by means of trace containment. Consequently, $\mathcal{TT}(\mathcal{P}_0)$ can be regarded as providing an alternative encoding of the set $[\mathcal{P}_0]_s^+$ of strategies.

Theorem 3. *$\mathcal{P}_0 \sqsubseteq \mathcal{P}_1$ iff $TR_1 \subseteq TR_0$ and $TE_1 \subseteq TE_0$.*

We are now ready to define the timed-trace semantics for the operators of our specification theory. Intuitively, the timed-trace semantics mimic the synchronised product of the operational definitions in Section 2.3.

Parallel composition. The idea behind parallel composition is that the projection of any trace in the composition onto the alphabet of one of the components should be a trace of that component.

Proposition 3. *If $\mathcal{P}_0$ and $\mathcal{P}_1$ are $\|$-composable, then $\mathcal{TT}(\mathcal{P}_0 \parallel \mathcal{P}_1) = (I, O, TR, TE)$ where $I = (I_0 \cup I_1) \setminus O$, $O = O_0 \cup O_1$ and the trace sets are given by:*

- $TE = \{tt \mid tt \restriction tA_i \in TE_i \wedge tt \restriction tA_{\bar{i}} \in TR_{\bar{i}}\} \cdot tA^*$
- $TR = TE \uplus \{tt \mid tt \restriction tA_i \in (TR_i \setminus TE_i) \wedge tt \restriction tA_{\bar{i}} \in (TR_{\bar{i}} \setminus TE_{\bar{i}})\}$

The above says tt is an error trace if the projection of tt on one component is an error trace, while the projection of tt on the other component is a realisable trace. tt is a realisable trace if tt is either an error trace or a (strictly) plain trace. tt is a (strictly) plain trace if the projections of tt on to $\mathcal{P}_0$ and $\mathcal{P}_1$ are (strictly) plain traces.

Disjunction. From any composite state in the disjunction of two components, the composition should only be willing to accept inputs that are accepted by both components, but should accept the union of outputs. After witnessing an output enabled by only one of the components, the disjunction should behave like that component. Because of the way that $\bot$ and $\top$ work in Table 1, this loosely corresponds to taking the union of the traces from the respective components.

Proposition 4. *If $\mathcal{P}_0$ and $\mathcal{P}_1$ are $\vee$-composable, then $\mathcal{TT}(\mathcal{P}_0 \vee \mathcal{P}_1) = (I, O, TR_0 \cup TR_1, TE_0 \cup TE_1)$, where $I = I_0 = I_1$ and $O = O_0 = O_1$.*

Conjunction. Similarly to disjunction, from any composite state in the conjunction of two components, the composition should only be willing to accept outputs that are accepted by both components, and should accept the union of inputs, until a stage when one of the component's input assumptions has been violated, after which it should behave like the other component. Because of the way that both $\bot$ and $\top$ work in Table 1, this essentially corresponds to taking the intersection of the traces from the respective components.

Proposition 5. *If $\mathcal{P}_0$ and $\mathcal{P}_1$ are $\wedge$-composable, then $\mathcal{TT}(\mathcal{P}_0 \wedge \mathcal{P}_1) = (I, O, TR_0 \cap TR_1, TE_0 \cap TE_1)$, where $I = I_0 = I_1$ and $O = O_0 = O_1$.*

Quotient. Quotient ensures its composition with the second component is a refinement of the first. Given the synchronised running of $\mathcal{P}_0$ and $\mathcal{P}_1$, if $\mathcal{P}_0$ is in a more refined state than $\mathcal{P}_1$, the quotient will try to rescue the refinement by taking $\top$ as its state (so that its composition with $\mathcal{P}_1$'s state will refine $\mathcal{P}_0$'s). If $\mathcal{P}_0$ is in a less or equally refined state than $\mathcal{P}_1$, the quotient will take the worst possible state without breaking the refinement.

Proposition 6. *If $\mathcal{P}_0$ dominates $\mathcal{P}_1$, then $\mathcal{TT}(\mathcal{P}_0 \% \mathcal{P}_1) = (I, O, TR, TE)$, where $I = I_0 \cup O_1$, $O = O_0 \setminus O_1$, and the trace sets satisfy:*

- $TE = TE_0 \cup \{tt \mid tt \restriction tA_1 \notin TR_1\} \cdot tA^*$
- $TR = TE \uplus \{tt \mid tt \in (TR_0 \setminus TE_0) \wedge tt \restriction tA_1 \in (TR_1 \setminus TE_1)\}$.

The above says tt is an error trace if either tt is an error trace in $\mathcal{P}_0$ or the projection of tt on $\mathcal{P}_1$ is not a realisable trace. A strictly plain trace must have strictly plain projections onto $\mathcal{P}_0$ and $\mathcal{P}_1$.

Mirroring of trace structures is equally straightforward: $\mathcal{TT}(\mathcal{P}_0)^\neg = (O_0, I_0, tA^* \setminus TE_0, tA^* \setminus TR_0)$. Consequently, quotient can also be defined as the derived operator $(\mathcal{TT}(\mathcal{P}_0)^\neg \parallel \mathcal{TT}(\mathcal{P}_1))^\neg$.

5 Comparison with Related Works

Our framework can be seen as a linear-time alternative to the timed specification theories of [2] and [3], albeit with significant differences. The specification theory in [3] also introduces parallel, conjunction and quotient, but uses timed alternating simulation as refinement, which does not admit the weakest precongruence.

An advantage of [3] is the algorithmic efficiency of branching-time simulation checking and the implementation reported in [10].

The work of [2] on timed games also bears conceptual similarities, although they do not define conjunction and quotient. We adopt most of the game rules in [2], except that, due to our requirement that proposed delay moves are maximal delays allowed by a strategy, a play cannot have consecutive delay moves. This enables us to avoid the complexity of time-blocking strategies and blame assignment, but does not ensure non-Zenoness[8]. Secondly, we do not use timestop/semi-timestop to model time errors (i.e. bounded-liveness errors). Rather, we introduce the explicit inconsistent state $\bot$ to model both time and immediate (i.e. safety) errors. This enables us to avoid the complexity of having two transition relations and well-formedness of timed interfaces.

Based on linear time, our timed theory owes much to the pioneering work of trace theories in asynchronous circuit verification, such as Dill's trace theory [9]. Our mirror operator is essentially a timed extension of the mirror operator from asynchronous circuit verification [15]. The definition of quotient based on mirroring (for the untimed case) was first presented by Verhoeff as his Factorisation Theorem [14].

In comparison with our untimed theory [1], our timed extension requires new techniques (e.g. those related to timestop) to handle delay transitions since time can be modelled neither as input nor as output. In the timed theory, the set of realisable traces (TR) is not required to be input-enabled, which is necessary for the set of untimed traces in [1]. Thus, the domain of trace structures is significantly enlarged. Furthermore, the timed theory supports the modelling of liveness assumptions/guarantees, with the checking of such violations reducing to $\bot$-reachability. Therefore, finite traces suffice to model and verify liveness properties, whereas in contrast, the untimed theory must employ infinite traces to treat liveness in a proper way.

We briefly mention other related works, which include timed modal transition systems [11,12], the timed I/O model [5,13] and embedded systems [18,19].

6 Conclusions

We have formulated a rich compositional specification theory for components with real-time constraints, based on a linear-time notion of substitutive refinement. The operators of hiding and renaming can also be defined, based on our previous work [8]. We believe that our theory can be reformulated as a timed extension of Dill's trace theory [9]. Future work will include an investigation of realisability and assume-guarantee reasoning.

Acknowledgments. The authors are supported by EU FP7 project CONNECT, ERC Advanced Grant VERIWARE and EPSRC project EP/F001096.

[8] Zeno behaviours (infinite action moves within finite time) in a play are not regarded as abnormal behaviours in our semantics.

References

1. Chen, T., Chilton, C., Jonsson, B., Kwiatkowska, M.: A Compositional Specification Theory for Component Behaviours. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 148–168. Springer, Heidelberg (2012)
2. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
3. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC 2010, pp. 91–100. ACM (2010)
4. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes 26, 109–120 (2001)
5. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: Timed I/O Automata: A mathematical framework for modeling and analyzing real-time systems. In: RTSS (2003)
6. Lim, W.: Design methodology for stoppable clock systems. Computers and Digital Techniques, IEE Proceedings E 133, 65–72 (1986)
7. Moore, S.W., Taylor, G.S., Cunningham, P.A., Mullins, R.D., Robinson, P.: Using stoppable clocks to safely interface asynchronous and synchronous subsystems. In: AINT (Asynchronous INTerfaces) Workshop, Delft, Netherlands (2000)
8. Wang, X., Kwiatkowska, M.Z.: On process-algebraic verification of asynchronous circuits. Fundam. Inform. 80, 283–310 (2007)
9. Dill, D.L.: Trace theory for automatic hierarchical verification of speed-independent circuits. ACM distinguished dissertations. MIT Press (1989)
10. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 365–370. Springer, Heidelberg (2010)
11. Bertrand, N., Legay, A., Pinchinat, S., Raclet, J.-B.: A Compositional Approach on Modal Specifications for Timed Systems. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 679–697. Springer, Heidelberg (2009)
12. Cerans, K., Godskesen, J.C., Larsen, K.G.: Timed Modal Specification - Theory and Tools. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 253–267. Springer, Heidelberg (1993)
13. Berendsen, J., Vaandrager, F.W.: Compositional Abstraction in Real-Time Model Checking. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 233–249. Springer, Heidelberg (2008)
14. Verhoeff, T.: A Theory of Delay-Insensitive Systems. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology (1994)
15. Zhou, B., Yoneda, T., Myers, C.: Framework of timed trace theoretic verification revisited. IEICE Trans. on Information and Systems 85, 1595–1604 (2002)
16. Chilton, C., Kwiatkowska, M., Wang, X.: Revisiting timed specification theories: A linear-time perspective. Technical Report RR-12-04, Department of Computer Science, University of Oxford (2012)
17. Wang, X.: Maximal Confluent Processes. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 188–207. Springer, Heidelberg (2012)
18. Thiele, L., Wandeler, E., Stoimenov, N.: Real-time interfaces for composing real-time systems. In: EMSOFT (2006)
19. Lee, I., Leung, J., Song, S.: Handbook of Real-Time and Embedded Systems. Chapman (2007)

Multi-core Reachability for Timed Automata

Andreas E. Dalsgaard[2], Alfons Laarman[1], Kim G. Larsen[2], Mads Chr. Olesen[2], and Jaco van de Pol[1]

[1] Formal Methods and Tools, University of Twente
{a.w.laarman,vdpol}@cs.utwente.nl
[2] Department of Computer Science, Aalborg University
{andrease,kgl,mchro}@cs.aau.dk

Abstract. Model checking of timed automata is a widely used technique. But in order to take advantage of modern hardware, the algorithms need to be parallelized. We present a multi-core reachability algorithm for the more general class of well-structured transition systems, and an implementation for timed automata.

Our implementation extends the opaal tool to generate a timed automaton successor generator in C++, that is efficient enough to compete with the UPPAAL model checker, and can be used by the discrete model checker LTSmin, whose parallel reachability algorithms are now extended to handle subsumption of semi-symbolic states. The reuse of efficient lockless data structures guarantees high scalability and efficient memory use.

With experiments we show that opaal+LTSmin can outperform the current state-of-the-art, UPPAAL. The added parallelism is shown to reduce verification times from minutes to mere seconds with speedups of up to 40 on a 48-core machine. Finally, strict BFS and (surprisingly) parallel DFS search order are shown to reduce the state count, and improve speedups.

1 Introduction

In industries developing safety-critical real-time systems, a number of safety requirements must be fulfilled. Model checking is a well-known method to achieve this and is critical for ensuring correct behaviour along all paths of execution of a system. One popular formalism for real-time systems is timed automata [3], where the time is modelled as a number of resettable clocks. Good tool support for timed automata exists [9].

However, as the desire to model check ever larger and more complex models arises, there is a need for more effective techniques. One option for handling large models has always been to buy a bigger machine. This provided great improvements; while early model checkers handled thousands of states, now we can handle billions. However, in recent years processor speed has stopped increasing, and instead more cores are added. These cores cannot be taken advantage of by the normal sequential algorithms for model checking.

The goal of this work is to develop scaling multi-core reachability for timed automata [3] as a first step towards full multi-core LTL model checking. A review of the history of discrete model checkers shows that indeed multi-core reachability is a crucial ingredient for efficient parallel LTL model checking (see Sec. 2). To attain our goal, we extended and combined several existing software tools:

LTSmin is a language-independent model checking framework, comprising, inter alia, an explicit-state multi-core backend [23,13].

opaal is a model checker designed for rapid prototype implementation of new model checking concepts. It supports a generalised form of timed automata [17], and uses the UPPAAL input format.

The UPPAAL DBM library is an efficient library for representing timed automata zones and operations thereon, used in the UPPAAL model checker [9].

Contributions: We describe a multi-core reachability algorithm for timed automata, which is generalizable to all models where a well-quasi-ordering on the behaviour of states exist [19]. The algorithm has been implemented for timed automata, and we report on the structure and performance of this prototype.

Before we move on to a description of our solution and its evaluation, we first review related work, and then briefly introduce the modelling formalism.

2 Related Work

One efficient model checker for timed automata is the UPPAAL tool [9,7]. Our work is closely related to UPPAAL in that we share the same input format and reuse its editor to create input models. In addition, we reused the open source UPPAAL DBM library for the internal symbolic representation of time zones.

Distributed model checking algorithms for timed automata were introduced in [11,6]. These algorithms exhibited almost linear scalability (50–90% efficiency) on a 14-node cluster of that time. However, analysis also shows that static partitioning used for distribution has some inherent limitations [15]. Furthermore, in the field of explicit-state model checking, the DiVinE tool showed that static partitioning can be reused in a shared-memory setting [5]. While the problem of parallelisation is considerably simpler in this setting, this tool nonetheless featured suboptimal performance with less than 40% efficiency on 16-core machines [22]. It was soon demonstrated that shared-memory systems are exploited better by combining local search stacks with a lockless hash table as shared passed set and an off-the-shelf load balancing algorithm for workload distribution [22]. Especially in recent experiments on newer 48-core machines [18, Sec. 5], the latter solution was clearly shown to have the edge with 50–90% efficiency.

Linear-time, on-the-fly liveness verification algorithms are based on depth-first search (DFS) order [20]. Next to the additional scalability, the shared hash table solution also provides more freedom for the search algorithm, which can be pseudo DFS and pseudo breadth-first search (BFS) order [22], but also strict BFS (see Sec. 6.2). This freedom has already been exploited by parallel NDFS algorithms for LTL model checking [20,18] that are linear in the size of the input graph (unlike their BFS-based counterparts). While these algorithms are heuristic in nature, their scalability has been shown to be superior to their BFS-based counterparts.

3 Preliminaries

We will now define the general formalism of well-structured transition systems [19,1], and specifically networks of timed automata under the zone abstraction [16].

Definition 1 (Well-quasi-ordering). *A well-quasi-ordering $\sqsubseteq$ is a reflexive and transitive relation over a set X, s.t. for any infinite sequence $x_0, x_1, \ldots$ eventually for some $i < j$ it will hold that $x_i \sqsubseteq x_j$.*

In other words, in any infinite sequence eventually an element exists which is "larger" than some earlier element.

Definition 2 (Well-structured transition system). *A well-structured transition system is a 3-tuple $(S, \rightarrow, \sqsubseteq)$, where S is the set of states, $\rightarrow : S \times S$ is the (computable) transition relation and $\sqsubseteq$ is a well-quasi-ordering over S, s.t. if $s \rightarrow t$ then $\forall s'.s \sqsubseteq s'$ there $\exists t'.s' \rightarrow t' \wedge t \sqsubseteq t'$.*[1]

We thus require $\sqsubseteq$ to be a monotonic ordering on the behaviour of states, i.e., if $s \sqsubseteq t$ then t has at least the behaviour of s (and possibly more), and we say that t *subsumes* or *covers* s.

One instance of well-structured transition systems arise from the symbolic semantics of timed automata. Timed automata are finite state machines with a finite set of real-valued, resettable clocks. Transitions between states can be guarded by constraints on clocks, denoted $G(C)$.

Definition 3 (Timed automaton). *An extended timed automaton is a 7-tuple $\mathcal{A} = (L, C, Act, s_0, \rightarrow, I_C)$ where*

- *L is a finite set of locations, typically denoted by ℓ*
- *C is a finite set of clocks, typically denoted by c*
- *Act is a finite set of actions*
- *$s_0 \in L$ is the initial location*
- *$\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ is the (non-deterministic) transition relation. We normally write $\ell \xrightarrow{g,a,r} \ell'$ for a transition, where ℓ is the source location, g is the guard over the clocks, a is the action, and r is the set of clocks reset.*
- *$I_C : L \rightarrow G(C)$ is a function mapping locations to downwards closed clock invariants.*

Using the definition of extended timed automata we can now define networks of timed automata, as modelled by UPPAAL, see [9] for details. A network of timed automata is a parallel composition of extended timed automata that enables synchronisation over a finite set of channel names $Chan$. We let $ch!$ and $ch?$ denote the output and input action on a channel $ch \in Chan$.

[1] With strong compatibility, see [19].

Definition 4 (Network of timed automata). *Let $Act = \{ch!, ch?|ch \in Chan\} \cup \{\tau\}$ be a finite set of actions, and let C be a finite set of clocks. Then the parallel composition of extended timed automata $\mathcal{A}_i = (L_i, C, Act, s_0^i, \rightarrow_i, I_C^i)$ for all $1 \leq i \leq n$, where $n \in \mathbb{N}$, is a network of timed automata, denoted $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2||\ldots||\mathcal{A}_n$.*

The concrete semantics of timed automata [9] gives rise to a possibly uncountable state space. To model check it a finite abstraction of the state space is needed; the abstraction used by most model checkers is the zone abstraction [14]. Zones are sets of clock constraints that can be efficiently represented by Difference Bounded Matrices (DBMs) [12]. The fundamental operations of DBMs are:

- $D\uparrow$ modifying the constraints such that the DBM represents all the clock valuations that can result from delay from the current constraint set
- $D \cap D'$ adding additional constraints to the DBM, e.g. because a transition is taken that imposes a clock constraint (guard clock constraints can also be represented as a DBM, and we will do so)[2]. The additional constraints might also make the DBM empty, meaning that no clock valuations can satisfy the constraints.
- $D[r]$ where $r \subseteq C$ is a clock reset of the clocks in r.
- D/B doing maximal bounds extrapolation, where $B : C \rightarrow \mathbb{N}_0$ is the maximal bounds needed to be tracked for each clock. Extrapolation with respect to maximal bounds [8] is needed to make the number of DBMs finite. Basically, it is a mapping for each clock indicating the maximal possible constant the clock can be compared to in the future. It is used in such a way that if the value of a clock has passed its maximal constant, the clock's value is indistinguishable for the model.
- $D \subseteq D'$ for checking if the constraints of D' imply the constraints of D, i.e. D' is a more relaxed DBM. D' has the behaviour of D and possibly more.

Lemma 1. *Timed automata under the zone abstraction are well-structured transition systems: $(S, \Rightarrow_{DBM}, Act, \sqsubseteq)$ s.t.*

1. *S consists of pairs (ℓ, D) where $\ell \in L$, and D is a DBM.*
2. *$\Rightarrow_{DBM}$ is the symbolic transition function using DBMs, and Act is as before*
3. *$\sqsubseteq$: $S \rightarrow S$ is defined as $(\ell, D) \sqsubseteq (\ell', D')$ iff $\ell = \ell'$, and $D \subseteq D'$.*

Remark that part of the ordering $\sqsubseteq$ is compared using discrete equality (the location vector), while only a subpart is compared using a well-quasi-ordering. Without loss of generality, and as done in [17], we can split the state into an explicit part $\mathcal{S}$, and a symbolic part Σ, s.t. the well-structured transition system is defined over $\mathcal{S} \times \Sigma$. We denote the explicit part as $s, t, r \in \mathcal{S}$ and the symbolic part of states by $\sigma, \tau, \rho, \pi, \upsilon \in \Sigma$, and a state as a pair (s, σ).

Model checking of safety properties is done by proving or disproving the reachability of a certain concrete goal location s_g.

[2] The DBM might need to be put into normal form after more constraints have been added [14].

Definition 5 ((Safety) Model checking of a well-structured transition system). *Given a well-structured transition system* $(\mathcal{S} \times \Sigma, \rightarrow, \sqsubseteq)$*, an initial state* $(s_0, \sigma_0) \in \mathcal{S} \times \Sigma$*, and a goal location* s_g *does a path exist* $(s_0, \sigma_0) \rightarrow \cdots \rightarrow (s_g, \sigma'_g)$*.*

In practice, the transition system is constructed *on-the-fly* starting from (s_0, σ_0) and recursively applying $\rightarrow$ to discover new states. To facilitate this, we extend the next-state interface of PINS with subsumption:

Definition 6. *A next-state interface with subsumption has three functions:*
INITIAL-STATE() $= (s_0, \sigma_0)$*,*
NEXT-STATE$((s, \sigma)) = \{(s_1, \sigma_1), \ldots, (s_n, \sigma_n)\}$ *returning all successors of* (s, σ)*,* $(s, \sigma) \rightarrow (s_i, \sigma_i)$*, and*
COVERS$(\sigma', \sigma) = \sigma \sqsubseteq \sigma'$ *returning whether the symbolic part* σ' *subsumes* σ*.*

4 A Multi-core Timed Reachability Tool

For the construction of our real-time multi-core model checker, we made an effort to reuse and combine existing components, while extending their functionality where necessary. For the specification models, we use the UPPAAL XML format. This enables the use of its extensive real-time modelling language through an excellent user interface. To implement the model's semantics (in the form of a next-state interface) we rely on opaal and the UPPAAL DBM library.[3] Finally, LTSMIN is used as a model checking backend, because of its language-independent design.

Fig. 1 gives an overview of the new toolchain. It shows how the XML input file is read by opaal which generates C++ code. The C++ file implements the PINS interface with subsumption specifically for the input model. Hence, after compilation (C++ compiler), LTSMIN can load the object file to perform the model checking.

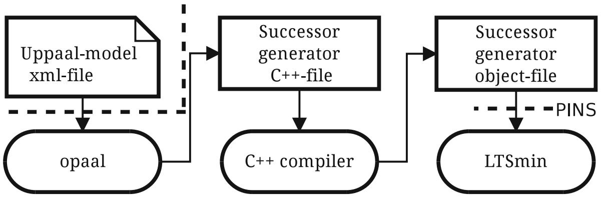


Fig. 1. Reachability with subsumption [17]

Previously, the opaal tool was used to generate Python code [17], but important parts of its infrastructure, e.g., analysing the model to find max clock constants [8], can be reused. In Sec. 5, we describe how opaal implements the semantics of timed automata, and the structure of the generated C++ code.

The PINS interface of the LTSMIN tool [13] has been shown to enable efficient, yet language-independent, model checking algorithms of different flavours, inter alia: distributed [13], symbolic [13] and multi-core reachability [22,24], and LTL model checking [20,21]. We extended the PINS interface to distinguish the new symbolic states of the opaal successor generator according to Def. 6. In Sec. 6, we describe our new multi-core reachability algorithms with subsumption.

[3] http://people.cs.aau.dk/~adavid/UDBM/

5 Successor Generation Using Opaal

The opaal tool was designed to rapidly prototype new model checking features and as such was designed to be extended with other successor generators. It already implements a substantial part of the UPPAAL features. For an explanation of the UPPAAL features see [9, p. 4-7]. The new C++ opaal successor generator supports the following features: templates, constants, bounded integer variables, arrays, selects, guards, updates, invariants on both variables and clocks, committed and urgent locations, binary synchronisation, broadcast channels, urgent synchronisation, selects, and much of the C-like language that UPPAAL uses to express guards and variable updates.

A state in the symbolic transition system using DBMs, is a location vector and a DBM. To represent a state in the C++ code we use a struct with a number of components: one integer for each location, and a pointer to a DBM object from the UPPAAL DBM library. Therefore a state is a tuple: $(\ell_1, \ldots, \ell_n, D)$.

The INITIAL-STATE function is rather straightforward: it returns a state struct initialised to the initial location vector, and a DBM representing the initial zone (delayed, and with invariants applied as necessary). The structure of the NEXT-STATE function is more involved, because it needs to consider the syntactic structure of the model, as can be seen in Alg. 1.

Alg. 1. Overall structure of the successor generator

```
proc NEXT-STATE($s_{in} = (\ell_1, \ldots, \ell_n, D)$)
out_states := $\emptyset$
for $\ell_i \in \ell_1, \ldots, \ell_n$
  for all $\ell_i \xrightarrow{g,a,r} \ell_i'$
   $D' := D \cap g$
   if $D' \neq \emptyset$                                         ▷is the guard satisfied?
    if $a = \tau$                                    ▷this is not a synchronising transition
       $D' := D'[r]\uparrow$                                            ▷clock reset, delay
       $D' := D' \cap I_C^i(\ell_i') \cap \bigcap_{k \neq i} I_C^k(\ell_k)$                     ▷apply clock invariants
       if $D' \neq \emptyset$
          $D' := D'/B(\ell_1, \ldots, \ell_i' \ldots, \ell_n)$
          out_states := out_states $\cup \{(\ell_1, \ldots, \ell_i', \ldots, \ell_n, D')\}$
    else if $a = ch!$                                              ▷binary sync. sender
     for $\ell_j \in \ell_1, \ldots, \ell_n, j \neq i$
     for all $\ell_j \xrightarrow{g_j, ch?, r_j} \ell_j'$                                      ▷find receivers
      if $D'' := D' \cap g_j \neq \emptyset$                             ▷receiver guard satisfied?
         $D'' := D''[r][r_j]\uparrow$                                         ▷clock resets, delay
         $D'' := D'' \cap I_C^i(\ell_i') \cap I_C^j(\ell_j') \cap \bigcap_{k \notin \{i,j\}} I_C^k(\ell_k)$       ▷apply clock invariants
      if $D'' \neq \emptyset$
         $D'' := D''/B(\ell_1, \ldots, \ell_i', \ldots, \ell_j' \ldots, \ell_n)$
         out_states := out_states $\cup \{(l_1, \ldots, l_i', \ldots, l_j', \ldots, l_n, D'')\}$
return out_states
```

At l. 4, we consider all outgoing transitions for the current location of each process (l. 3). If the transition is internal, we can evaluate it right away, and possibly generate a successor at l. 12. If it is a sending synchronisation (`ch!`), we need to find possible synchronisation partners (l. 15). So again we iterate over all processes and the transitions of their current locations (l. 14–21).

In the generated C++ code a few optimisations have been made, compared to Alg. 1: The loops on line l. 3 and l. 14 have been unrolled, since the number of processes they iterate over is known beforehand. In that manner the transitions to consider can be efficiently found. As an optimisation, before starting the code generation, we compute the set of all possible receivers for all channels, for the unrolling of l. 14. In practice there are usually many receivers but few senders for each channel, resulting in the unrolling being an acceptable trade-off.

When doing the max bounds extrapolation (/) in Alg. 1, we obtain the bounds from a location-dependent function $B : L_1 \times \cdots \times L_n \to (C \to \mathbb{N}_0)$. This function is pre-computed in opaal using the method described in [8].

Some features are not formalised in this work, but have been implemented for ease of modelling. We support integer variables, urgency that can be modelled using urgent/committed locations and urgent channels, but also channel arrays with dynamically computed senders, broadcast channels, and process priorities. These are all implemented as simple extensions of Alg. 1. Other features are supported in the form of a syntactic expansion, namely: selects, and templates.

To make the NEXT-STATE function thread-safe, we had to make the UPPAAL DBM library thread-safe. Therefore, we replaced its internal allocator with a concurrent memory allocator (see Sec. 7). We also replaced the internal hash table, used to filter duplicate DBM allocations, with a concurrent hash table.

6 Well-Structured Transition Systems in LTSmin

The current section presents the parallel reachability algorithm that was implemented in LTSMIN to handle well-structured transition systems. According to Def. 6, we can split up states into a discrete part, which is always compared using equality (for timed automata this consists of the locations and variables), and a part that is compared using a well-quasi-ordering (for timed automata this is the DBM).

Alg. 2. Reachability with subsumption [17]

```
1 proc reachability($s_g$)
2   $W$ := { INITIAL-STATE() }; $P := \emptyset$
3   while $W \neq \emptyset$
4     $W := W \setminus (s, \sigma)$ for some $(s, \sigma) \in W$
5     $P := P \cup \{(s, \sigma)\}$
6     for $(t, \tau) \in$ NEXT-STATE$((s, \sigma))$ do
7       if $t = s_g$ then report & exit
8       if $\nexists \rho\colon (t, \rho) \in W \cup P \wedge$ COVERS$(\rho, \tau)$
9         $W := W \setminus \{(t, \rho) \mid$ COVERS$(\tau, \rho)\} \cup (t, \tau)$
```

We recall the sequential algorithm from [17] (Alg. 2) and adapt it to use the next-state interface with subsumption. At its basis, this algorithm is a search with a waiting set (W), containing the states to be explored, and a passed set (P), containing the states that are already explored.

New successors (t, τ) are added to W (l. 9), but only if they are not subsumed by previous states (l. 8). Additionally, states in the waiting set W that are subsumed by the new state are discarded (l. 9), avoiding redundant explorations.

6.1 A Parallel Reachability Algorithm with Subsumption

In the parallel setting, we localize all work sets (Q_p, for each worker p) and create a shared data structure L storing both W and P. We attach a status flag passed or waiting to each state in L to create a global view of the passed and waiting set and avoid unnecessary reexplorations. L can be represented as a multimap, saving multiple symbolic state parts with each explicit state part $L : \mathcal{S} \to \Sigma^*$. To make L thread-safe, we protect its operations with a fine-grained locking mechanism that locks only the part of the map associated with an explicit state part s: **lock**$(L(s))$, similar to the spinlocks in [22]. An off-the-shelf load balancer takes care of distributing work at the startup and when some Q_p runs empty prematurely. This design corresponds to the shared hash table approach discussed in Sec. 2 and avoids a *static partitioning* of the state space.

Alg. 3 presents the discussed design. The algorithm is initialised by calling reachability with the desired number of threads P and a discrete goal location s_g. This method initialises the shared data structure L and gets the initial state using the INITIAL-STATE function from the next-state interface with subsumption. The initial state is then added to L and the worker threads are initialised at l. 6. Worker thread 1 explores the initial state; work load is propagated later.

The **while** loop on l. 20 corresponds closely to the sequential algorithm, in a quick overview: a state (s, σ) is taken from the work set at l. 21, its flag is set to passed by grab if it were not already, and then the successors (t, τ) of (s, σ) are checked against the passed and the waiting set by update. We now discuss the operations on L (update, grab) and the load balancing in more detail.

To implement the subsumption check (line l. 8–9 in Alg. 2) for successors (t, τ) and to update the waiting set concurrently, update is called. It first locks

Alg. 3. Reachability with cover update of the waiting set

```
1  global $L : \mathcal{S} \to (\Sigma \times \{\text{waiting}, \text{passed}\})^*$
2  proc reachability($P, s_g$)
3      $L := \mathcal{S} \to \emptyset$
4      $(s_0, \sigma_0) := s :=$ INITIAL-STATE()
5      $L(s_0) := (\sigma_0, \text{waiting})$
6      search($s, s_g, 1$)|| ... ||search($s, s_g, P$)
7  proc update($t, \tau$)
8      lock($L(t)$)
9      for $(\rho, f) \in L(t)$ do
10         if COVERS($\rho, \tau$)
11             unlock($L(t)$)
12             return true
13         else if $f = \text{waiting} \wedge$ COVERS($\tau, \rho$)
14             $L(t) := L(t) \setminus (\rho, \text{waiting})$
15     $L(t) := L(t) \cup (\tau, \text{waiting})$
16     unlock($L(t)$)
17     return false
18 proc search($(s_0, \sigma_0), s_g, p$)
19     $Q_p :=$ if $p = 1$ then $\{(s_0, \sigma_0)\}$ else $\emptyset$
20     while $Q_p \neq \emptyset \vee$ balance($Q_p$)
21         $Q_p := Q_p \setminus (s, \sigma)$ for some $(s, \sigma) \in Q_p$
22         if $\neg$grab($s, \sigma$) then continue
23         for $(t, \tau) \in$ NEXT-STATE($(s, \sigma)$) do
24             if $t = s_g$ then report & exit
25             if $\neg$update($t, \tau$)
26                 $Q_p := Q_p \cup (t, \tau)$
27 proc grab($s, \sigma$)
28     lock($L(s)$)
29     if $\sigma \notin L(s) \vee \text{passed} = L(s, \sigma)$
30         unlock($L(s)$)
31         return false
32     $L(s, \sigma) := \text{passed}$
33     unlock($L(s)$)
34     return true
```

L on t. Now, for all symbolic parts and status flag ρ, f associated with t, the method checks if τ is already covered by ρ. In that case (t, τ) will not be explored. Alternatively, all ρ with status flag waiting that are covered by τ are removed from $L(t)$ and τ is added. The update algorithm maintains the invariant that a state in the waiting set is never subsumed by any other state in L: $\forall s\, \forall(\rho, f), (\rho', f') \in L(s)\colon f = \mathsf{waiting} \wedge \rho \neq \rho' \Rightarrow \rho \not\sqsubseteq \rho'$ (**Inv. 1**). Hence, similar to Alg. 2 l. 8–9, it can never happen that (t, τ) first discards some (t, ρ) from $L(s)$ (l. 14) and is discarded itself in turn by some (t, ρ') in $L(s)$ (l. 10), since then we would have $\rho \sqsubseteq \tau \sqsubseteq \rho'$; by transitivity of $\sqsubseteq$ and the invariant, ρ and ρ' cannot be both in $L(t)$. Finally, notice that update unlocks $L(t)$ on all paths.

The task of the method grab is to check if a state (s, σ) still needs to be explored, as it might have been explored by another thread in the meantime. It first locks $L(s)$. If σ is no longer in $L(s)$ or it is no longer globally flagged waiting (l. 29), it is discarded (l. 22). Otherwise, it is "grabbed" by setting its status flag to passed. Notice again that on all paths through grab, $L(s)$ is unlocked.

Finally, the method balance handles termination detection and load balancing. It has the side-effect of adding work to Q_p. We use a standard solution [25].

6.2 Exploration Order

The shared hash table approach gives us the freedom to allow for a DFS or BFS exploration order depending on the implementation of Q_p. Note, however, that only pseudo-DFS/BFS is obtained, due to randomness introduced by parallelism.

It has been shown for timed automata that the number of generated states is quite sensitive to the exploration order and that in most cases strict BFS shows the best results [11]. Fortunately, we can obtain strict BFS by synchronising workers between the different BFS levels. To this end, we first split Q_p into two separate sets that hold the current BFS level (C_p) and the next BFS level (N_p) [2]. The order within these sets does not matter, as long as the current is explored before the next set. Load balancing will only be performed on C_p, hence a level terminates once $C_p = \emptyset$ for all p. At this point, if $N_p = \emptyset$ for all p, the algorithm can terminate because the next BFS level is empty. The synchronising reduce method counts $\sum_{i=1}^{P} |N_i|$ (similar to `mpi_reduce`).

Alg. 4. Strict parallel BFS

```
1  proc search($s_0, \sigma_0, p$)
2    $C_p$ := if $p = 1$ then $\{(s_0, \sigma_0)\}$ else $\emptyset$
3    do
4      while $C_p \neq \emptyset \vee$ balance($C_p$)
5        $C_p := C_p \setminus (s, \sigma)$ for some $(s, \sigma) \in C_p$
6          ...
7            $N_p := N_p \cup (t, \tau)$
8      load := reduce($sum, |N_p|, P$)
9      $C_p$, $N_p$ := $N_p$, $\emptyset$
10   while $load \neq 0$
```

Alg. 4 shows a parallel strict-BFS implementation. An extra outer loop iterates over the levels, while the inner loop (l. 4–7) is the same as in Alg. 3. Except for the lines that add and remove states to and from the work set, which now operate on N_p and C_p. The (pointers to) the work sets are swapped, after the reduce call at l. 8 calculates the load of the next level.

6.3 A Data Structure for Semi-symbolic States

In [22], we introduced a lockless hash table, which we reuse here to design a data structure for L that supports the operations used in Alg. 3. To allow for massive parallelism on modern multi-core machines with steep memory hierarchies, it is crucial to keep a low memory footprint [22, Sec. II]. To this end, lookups in the large table of state data are filtered through a separate smaller table of hashes. The table assigns a unique number (the hash location) to each explicit state stored in it: $D\colon \mathcal{S} \to \mathbb{N}$. In finite reality, we have: $D\colon \mathcal{S} \to \{1, \ldots, N\}$.

We now reuse the state numbering of D to create a multimap structure for L. The first component of the new data structure is an array $I[N]$ used for indexing on the explicit state parts. To associate a set of symbolic states (pointers to DBMs) with our explicit state stored in $D[x]$, we are going to attach a linked list structure to $I[x]$. Creating a standard linked list would cause a single *cache line* access per element, increasing the memory footprint, and would introduce costly synchronisations for each modification. Therefore, we allocate multi-buckets, i.e., an array of pointers as one linked list element. To save memory, we store lists of just one element directly in I and completely fill the last multi-bucket.

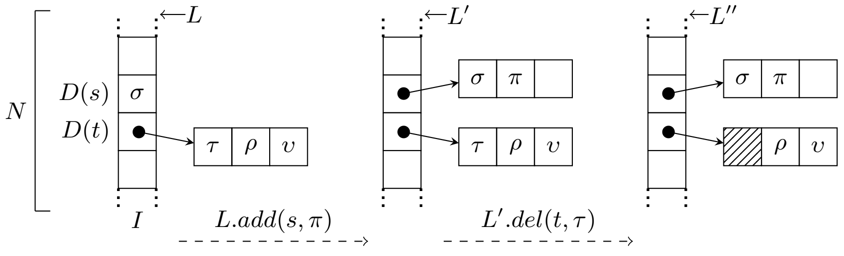


Fig. 2. Data structure for L, and operations

Fig. 2 shows three instances of the discussed data structure: L, L' and L''. Each multimap is a pointer (arrow) to an array I shown as a vertical bucket array. L contains $\{(s, \sigma), (t, \tau), (t, \rho), (t, \upsilon)\}$. We see how a multi-bucket with (fixed) length 3 is created for t, while the single symbolic state attached to s is kept directly in I. The figure shows how σ is moved when (s, π) is added by the add operation (dashed arrow), yielding L'. Adding π to t would have moved υ to a new linked multi-bucket together with π.

Removing elements from the waiting list is implemented by marking bucket entries as tombstone, so they can later be reused (see L''). This avoids memory fragmentation and expensive communication to reuse multi-buckets. For highest scalability, we allocate multi-buckets of size 8, equal to a cache line. Other values can reduce memory usage, but we found this sufficiently efficient (see Sec. 7).

We still need to deal with locking of explicit states, and storing of the various flags for symbolic states (waiting/passed). Internally, the algorithms also need to distinguish between the different buckets: empty, tomb stone, linked list pointers and symbolic state pointers. To this end, we can bitcram additional bits into the pointers in the buckets, as is shown in Fig. 3. Now **lock**$(L(s))$ can be implemented as a spinlock using the atomic *compare-and-swap* (CAS) instruction on $I[s]$ [22]. Since all operations on $L(s)$ are done after **lock**$(L(s))$, the corresponding bits of the buckets can be updated and read with normal load and store instructions.

```
struct link_or_dbm {
  bit pointer[60]
  bit flag ∈ {waiting, passed}
  bit lock ∈ {locked, unlocked}
  bit status[2] ∈ {empty, tomb,
                  dbm_ptr, list_ptr}
}
```

Fig. 3. Bit layout of word-sized bucket

6.4 Improving Scalability through a Non-blocking Implementation

The size of the critical regions in Alg. 3 depends crucially on the $|\Sigma|/|\mathcal{S}|$ ratio; a higher ratio means that more states in $L(t)$ have to be considered in the method update(t, τ), affecting scalability negatively. A similar limitation is reported for distributed reachability [15]. Therefore, we implemented a *non-blocking* version: instead of first deleting all subsumed symbolic states with a waiting flag, we atomically replace them with the larger state using CAS. For a failed CAS, we retry the subsumption check after a reread. L can be atomically extended using the well-known *read-copy-update* technique. However, workers might miss updates by others, as Inv. 1 no longer holds. This could cause $|\Sigma|$ to increase again.

7 Experiments

To investigate the performance of the generated code, we compare full reachability in opaal+LTSmin with the current state-of-the-art (uppaal).[4] To investigate scalability, we benchmarked on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168) with a varying number of threads. Statistics on memory usage were gathered and compared against uppaal. Experiments were repeated 5 times.

We consider three models from the uppaal demos: `viking` (one discrete variable, but many synchronisations), `train-gate` (relatively large amount of code, several variables), and `fischer` (very small discrete part). Additionally, we experiment with a generated model, `train-crossing`, which has a different structure from most hand-made models. For some models, we created multiple numbered instances, the numbers represent the number of processes in the model.

For uppaal, we ran the experiments with BFS and disabled space optimisation. The `opaal_ltsmin` script in opaal was used to generate and compile models. In LTSmin we used a fixed hash table (`--state=table`) size of 2^{26} states (`-s26`), waiting set updates as in Alg. 3 (`-u1`) and multi-buckets of size 8 (`-l8`).

Performance & Scalability. Table 1 shows the reachability runtimes of the different models in uppaal and opaal+LTSmin with strict BFS (`--strategy=sbfs`). Except for `fischer6`, we see that both tools compete with each other on the

[4] opaal is available at `https://code.launchpad.net/~opaal-developers/opaal/opaal-ltsmin-succgen`, LTSmin at `http://fmt.cs.utwente.nl/tools/ltsmin/`

Table 1. $\mathcal{S}$, $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) in UPPAAL and opaal+LTSMIN (strict BFS)

	$\|\mathcal{S}\|$	UPPAAL		opaal+LTSMIN (cores)							
		T	$\|\Sigma\|$	$\|\Sigma_1\|$	$\|\Sigma_{48}\|$	T_1	T_2	T_8	T_{16}	T_{32}	T_{48}
train-gate-N10	7e+07	837.4	1.0	1.0	1.0	573.3	297.8	76.7	39.4	21.1	14.4
viking17	1e+07	207.8	1.0	1.5	1.5	331.5	172.5	44.2	22.7	11.9	8.6
train-gate-N9	7e+06	76.8	1.0	1.0	1.0	52.4	28.5	7.7	4.1	2.4	2.0
viking15	3e+06	38.0	1.0	1.5	1.5	67.0	34.8	9.7	5.1	3.0	2.3
train-crossing	3e+04	48.3	20.8	16.1	17.3	24.5	37.2	5.8	2.7	2.0	2.1
fischer6	1e+04	0.1	0.3	50.1	50.1	219.2	129.2	46.4	36.1	32.9	31.8

sequential runtimes, with 2 threads however opaal+LTSMIN is faster than UPPAAL. With the massive parallelism of 48 cores, we see how verification tasks of minutes are reduced to mere seconds. The outlier, `fischer6`, is likely due to the use of more efficient clock extrapolations in UPPAAL, and other optimisations, as witnessed by the evolution of the runtime of this model in [10,4].

We noticed that the 48-core runtimes of the smaller models were dominated by the small BFS levels at the beginning and the end of the exploration due to synchronisation in the load balancer and the reduce function. This overhead takes consistently 0.5–1 second, while it handles less than thousand states. Hence to obtain useful scalability measurements for small models, we excluded this time in the speedup calculations (Fig. 4–7). The runtimes in Table 1–2 still include this overhead. Fig. 4 plots the speedups of strict BFS with the standard deviation drawn as vertical lines (mostly negligible, hence invisible). Most models show almost linear scalability with a speedup of up to 40, e.g. `train-gate-N10`. As expected, we see that a high $|\Sigma|/|\mathcal{S}|$ ratio causes low scalability (see `fischer` and `train-crossing` and Table 1). Therefore, we tried the non-blocking variant (Sec. 6.3) of our algorithm (`-n`). As expected, the speedups in Fig. 5 improve and the runtimes even show a threefold improvement for `fischer.6` (Table 2). The efficiency on 48 cores remains closely dependent to the $|\Sigma|/|\mathcal{S}|$ ratio of the model (or the average length of the lists in the multimap), but the scalability is now at least sub-linear and not stagnant anymore.

We further investigated different search orders. Fig. 6 shows results with pseudo BFS order (`--strategy=bfs`). While speedups become higher due to the lacking level synchronisations, the loose search order tends to reach "large" states later and therefore generates more states for two of the models ($|\Sigma_1|$ vs $|\Sigma_{48}|$ in Table 2). This demonstrates that our strict BFS implementation indeed pays off.

Finally, we also experimented with randomized DFS search order (`-prr --strategy=dfs`). Table 2 shows that DFS causes again more states to be generated. But, surprisingly, the number of states actually reduces with the parallelism for the `fischer6` model, even below the state count of strict BFS from Table 1! This causes a super-linear speedup in Fig. 7 and threefold runtime improvement over strict BFS. We do not consider this behaviour as an exception (even though `train-crossing` does not show it), since it is compatible with our observation that parallel DFS finds shorter counter examples than parallel BFS [18, Sec. 4.3].

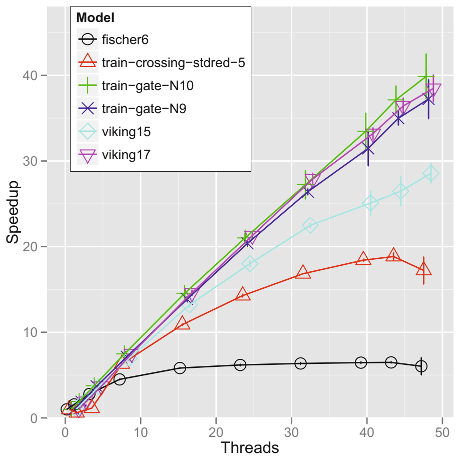


Fig. 4. Speedup strict BFS

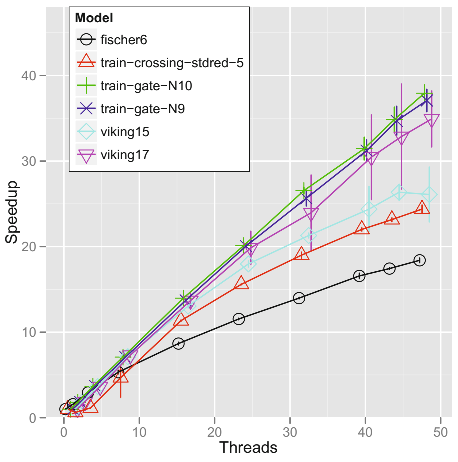


Fig. 5. Speedup non-blocking strict BFS

Design decisions. Some design decisions presented here were motivated by earlier work that has proven successful for multi-core model checking [22,18]. In particular, we reused the shared hash table and a *synchronous* load balancer [25]. Even though we observed load distributions close to ideal, a modern work stealing solution might still improve our results, since the work granularity for timed reachability is higher than for untimed reachability. The main bottlenecks, however, have proven to be the increase in state count by parallelism and the cost of the spinlocks due to a high $|\Sigma|/|\mathcal{S}|$ ratio. The latter we partly solved with a non-blocking algorithm. Strict BFS orders have proven to aid the former problem and randomized DFS orders could aid both problems.

Memory usage. Table 3 shows the memory consumption of UPPAAL (U-S0) and sequential opaal+LTSmin (O+L_1) with strict BFS. From it, we conclude that

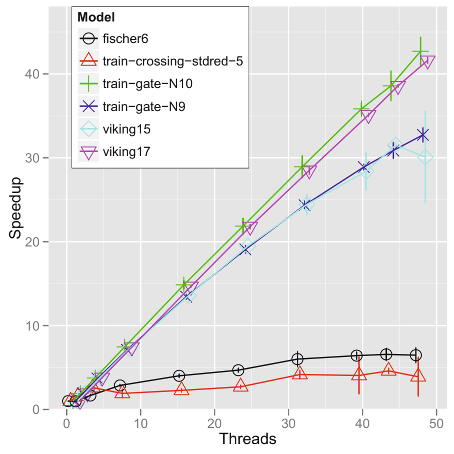


Fig. 6. Speedup pseudo BFS

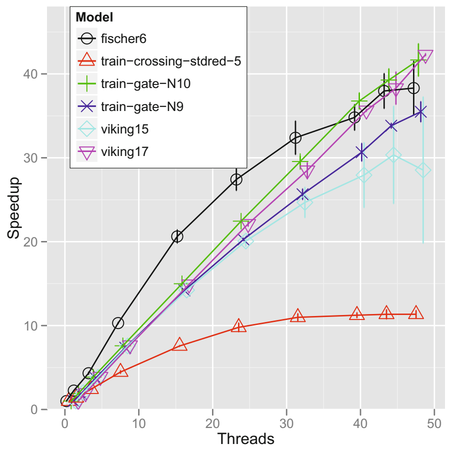


Fig. 7. Speedup randomized pseudo DFS

Table 2. $|\Sigma|$ ($\frac{|\Sigma|}{|S|}$) and runtimes (sec) with non-blocking SBFS, DFS and BFS

	NB SBFS				DFS				BFS			
	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}
train-gate-N10	1.0	1.0	547.9	14.5	1.0	1.0	647.8	15.6	1.0	1.0	559.3	13.1
viking17	1.5	1.5	320.1	9.2	1.6	1.6	386.5	9.1	1.5	1.5	325.6	7.8
train-gate-N9	1.0	1.0	52.1	2.1	1.0	1.0	61.7	1.7	1.0	1.0	51.9	1.6
viking15	1.5	1.5	64.8	2.5	1.6	1.6	80.2	3.1	1.5	1.5	66.0	2.3
train-crossing	16.1	16.1	24.1	1.8	169.8	179.0	3371.0	297.4	16.1	37.1	24.5	157.5
fischer6	50.1	50.1	201.3	12.0	54.4	39.4	405.1	10.6	50.1	58.1	206.0	32.3

Table 3. Memory usage (MB) of both UPPAAL (U-S0 and U-S2) and opaal+LTSMIN

	T	D	L	L2	Q	dbm	O+L$_1$	O+L$_{48}$	O+L$^T_1$	O+L$^T_{48}$	U-S0	U-S2
train-gate-N10	777	5989	499	499	249	1363	8101	8241	2790	3028	6091	3348
viking17	156	1040	536	214	40	87	1704	1931	828	1047	1579	722
train-gate-N9	81	549	50	50	24	61	684	815	214	347	607	332
viking15	32	190	112	44	8	55	364	581	203	423	333	162
train-crossing	0	2	5	7	0	419	426	623	425	622	48	64
fischer6	0	0	5	9	1	176	429	512	290	429	0	4

our memory usage is within 25% of UPPAAL's for the larger models (where these measurements are precise enough). Furthermore, we extensively experimented with different concurrent allocators and found that TBB malloc (used in this paper) yields the best performance for our algorithms.[5] Its overhead (O+L$_1$ vs O+L$_{48}$ in Table 3) appears to be limited to a moderate fixed amount of 250MB more than the sequential runs, for which we used the normal `glibc` allocator.

We also counted the memory usage inside the different data structures: the multimap L (including partly-filled multi-buckets), the hash table D, the combined local work sets (Q), and the DBM duplicate table (dbm). As we expected the overhead of the 8-sized multi-buckets is little compared to the size of D and the DBMs. We may however replace D with the compressed, parallel tree table (T) from [24]. The resulting total memory usage (O+L^T), can now be dominated by L, .i.e., for `viking17`. But if we reduce L to a linked list (`-l2`), its size shrinks by 60% to 214MB for this model (L2). Just a modest gain compared to the total.

For completeness, we included the results of UPPAAL's state space optimisation (U-S2). As expected, it also yields great reductions, which is the more interesting since the two techniques are orthogonal and could be combined.

8 Conclusions

We presented novel algorithms and data structures for multi-core reachability on well-structured transition systems and an efficient implementation for timed automata in particular. Experiments show good speedups, up to 40 times on a 48-core machine and also identify current bottlenecks. In particular, we see speedups

[5] cf. `http://fmt.cs.utwente.nl/tools/ltsmin/formats-2012/` for additional data.

of 58 times compared to UPPAAL. Memory usage is limited to an acceptable maximum of 25% more than UPPAAL.

Our experiments demonstrate the flexibility of the search order that our parallel approach allows for. BFS-like order is shown to be occasionally slightly faster than strict BFS but is substantially slower on other models, as previously observed in the distributed setting. A new surprising result is that parallel randomized (pseudo) DFS order sometimes reduces the state count below that of strict BFS, yielding a substantial speedup in those cases.

Previous work has shown that better parallel reachability [22,24] crucially enables new and better solutions to parallel model checking of liveness properties [20,18]. Therefore, our natural next step is to port multi-core nested depth-first search solutions to the timed automata setting.

Because of our use of generic toolsets, more possibilities are open to be explored. The opaal support for the UPPAAL language can be extended and support for optimisations like symmetry reduction and partial order reduction could be added, enabling easier modeling and better scalability. Additionally, lattice-based languages [17] can be included in the C++ code generator. On the backend side, the distributed [13] and symbolic [13] algorithms in LTSMIN can be extended to support subsumption, enabling other powerful means of verification. We also plan to add a join operator to the PINS interface, to enable abstraction/refinement-based approaches [17].

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General Decidability Theorems for Infinite-State Systems. In: Proceedings of Eleventh Annual IEEE Symposium on Logic in Computer Science, LICS 1996, pp. 313–321 (July 1996)
2. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable Graph Exploration on Multicore Processors. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Amnell, T., Behrmann, G., Bengtsson, J.E., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Pettersson, P., Weise, C., Yi, W.: UPPAAL - Now, Next, and Future. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer, Heidelberg (2001)
5. Barnat, J., Ročkai, P.: Shared Hash Tables in Parallel Model Checking. Electronic Notes in Theoretical Computer Science 198(1), 79–91 (2007); Proceedings of PDMC 2007
6. Behrmann, G.: Distributed Reachability Analysis in Timed Automata. International Journal on Software Tools for Technology Transfer 7(1), 19–30 (2005)
7. Behrmann, G., Bengtsson, J.E., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL Implementation Secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
8. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static Guard Analysis in Timed Automata Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)

9. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
10. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing Uppaal over 15 years. Software: Practice and Experience 41(2), 133–142 (2011)
11. Behrmann, G., Hune, T., Vaandrager, F.: Distributing Timed Model Checking - How the Search Order Matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
12. Bengtsson, J.: Clocks, DBMs and states in timed systems. PhD thesis, Uppsala University (2002)
13. Blom, S., van de Pol, J., Weber, M.: LTSMIN: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
14. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
15. Braberman, V., Olivero, A., Schapachnik, F.: Dealing with practical limitations of distributed timed model checking for timed automata. Formal Methods in System Design 29, 197–214 (2006), doi:10.1007/s10703-006-0012-3
16. Comon, H., Jurski, Y.: Timed Automata and the Theory of Real Numbers. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 242–257. Springer, Heidelberg (1999)
17. Dalsgaard, A.E., Hansen, R.R., Jørgensen, K.Y., Larsen, K.G., Olesen, M.C., Olsen, P., Srba, J.: `opaal`: A Lattice Model Checker. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 487–493. Springer, Heidelberg (2011)
18. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved Multi-Core Nested Depth-First Search. In: Mukund, M., Chakraborty, S. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
19. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)
20. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
21. Laarman, A.W., van de Pol, J.: Variations on Multi-Core Nested Depth-First Search. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 13–28 (2011)
22. Laarman, A.W., van de Pol, J., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Sharygina, N., Bloem, R. (eds.) Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss. IEEE Computer Society (October 2010)
23. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSMIN: Marrying Modularity and Scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
24. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011)
25. Sanders, P.: Lastverteilungsalgorithmen fur Parallele Tiefensuche. number 463. In: Fortschrittsberichte, Reihe 10. VDI. Verlag (1997)

Counterexample-Guided Synthesis of Observation Predicates

Rayna Dimitrova and Bernd Finkbeiner

Saarland University, Germany

Abstract. We present a novel approach to the safety controller synthesis problem with partial observability for real-time systems. This in general undecidable problem can be reduced to a decidable one by fixing the granularity of the controller: finite sets of clocks and constants in the guards. Current state-of-the-art methods are limited to brute-force enumeration of possible granularities or manual choice of a finite set of observations that a controller can track. We address this limitation by proposing a counterexample-guided method to successively refine a set of observations until a sufficiently precise abstraction is obtained. The size of the abstract games and strategies generated by our approach depends on the number of observation predicates and not on the size of the constants in the plant. Our experiments demonstrate that this results in better performance than the approach based on fixed granularity when fine granularity is necessary.

1 Introduction

Controller synthesis, both in the discrete and in the timed setting, has been an active field of research in the last decades. The timed controller synthesis problem asks to automatically find a controller for an open plant such that the controlled closed loop system satisfies a given property. It naturally reduces to the problem of finding a winning strategy for the controller player in a two-player *timed game* between a controller and its environment (the plant). This problem is well-understood for the case that the controller can fully observe the state and evolution of the plant. In reality, however, this assumption is usually violated due to limited sensors or the inability to observe the internal behavior of the plant. The controller must therefore win the game under *partial observability*.

The timed controller synthesis problem is undecidable under partial observability [2]. All known synthesis algorithms therefore rely on some *a-priori* restriction of the problem, such as fixing the *granularity* [2] of the controller by restricting the constants to which clocks may be compared to in the controller to integral multiples of $\frac{1}{m}$, where m is a predefined constant, or fixing a template for the controller [9]. Alternatively, one can predefine the observations of the controller [4,3], which amounts to providing a *finite set of predicates* over the locations and clocks on which the strategy of the controller may be based. How to efficiently find these observation predicates is an important research question, the only known approach being the brute-force enumeration of *all* possible granularities $(1, \frac{1}{2}, \frac{1}{4}, \ldots)$ until a sufficiently precise one is found.

In this paper, we present the first systematic method for the *automatic synthesis of observation predicates*. Before we describe the approach in more detail, let us clarify the role of the observation predicates. Figure 1 shows, as a toy example, the model of

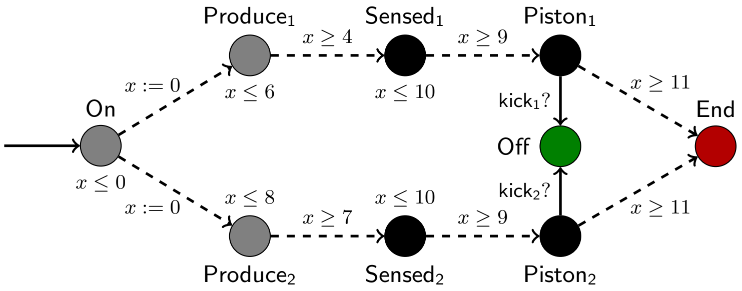


Fig. 1. Example of a partially observable plant for a production system. For readability we have omitted the kick_1? and kick_2? transitions from all other locations leading to location End.

a production system. The goal is to kick a box from a conveyor belt using a piston, before the box reaches the end of the belt. The locations On, $\mathsf{Producing}_1$, $\mathsf{Producing}_2$, Sensed_1, Sensed_2, Piston_1, Piston_2, End and Off of the plant indicate the position of the box on the belt. The plant produces two types of boxes, where producing a box of type 1 takes between 4 and 6 seconds and producing a box of type 2 takes between 7 and 8 seconds. However, regardless of its type, the box arrives at the respective location Piston_1 or Piston_2 between 9 and 10 seconds after the start. The goal of the controller is to avoid location End. For that, it has to execute the correct kick_1! or kick_2! action at the right time, namely when the box is in the respective location Piston_1 or Piston_2.

The challenge is that locations On, $\mathsf{Produce}_1$ and $\mathsf{Produce}_2$ are indistinguishable by the controller, and so are locations Sensed_1, Sensed_2, Piston_1 and Piston_2. The controller can only detect the presence of a box via a sensor (i.e, it observes the box entering locations Sensed_1 and Sensed_2) and use timing information to determine the time-frame in which the box is in location Piston_1 (or Piston_2). It cannot observe the clock x of the plant, but has its own clock y that it can test and reset. A solution to the synthesis problem is to use a clock y in the controller and activate the piston when $y = 21/2$, thus ensuring that the End is never reached as the box is guaranteed to reach location Piston_1 or Piston_2 in 9 to 10 seconds after it is sensed and remains there at least until $y = 11$. Additionally, in order to activate the correct piston, the controller needs to distinguish the type of the box. This can be done, again using timing information, by checking whether or not the box has been sensed by time $y = 7$. In order to find a correct controller, we thus need two observation predicates: $y = 21/2$ and $y >= 7$. Clearly, both predicates are necessary: if the controller only observes one of them or only some other predicate, say, only $y = 30$, then it is impossible to enforce the specification. Note also that two predicates play different roles in the control strategy. Predicate $y = 21/2$ identifies a particular point in time (out of the infinitely many) in which the controller may choose to *take an action*, predicate $y >= 7$ identifies an observation that is needed in order to be able to *decide* on the right action. In the following, we distinguish these two types of observation predicates as *action points* and *decision predicates*.

Our method works by successively *refining a finite set of observation predicates* based on the *analysis of spurious counterexamples*. The key is to use timed games with fixed observations as sound abstractions of the original timed game under incomplete information. Our method builds on the classic CEGAR loop, where one successively

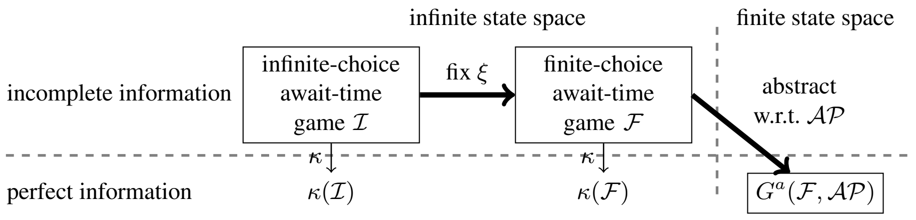

infinite state space
finite state space
incomplete information
infinite-choice await-time game $\mathcal{I}$
fix ξ
finite-choice await-time game $\mathcal{F}$
abstract w.r.t. $\mathcal{AP}$
κ
κ
perfect information
$\kappa(\mathcal{I})$
$\kappa(\mathcal{F})$
$G^a(\mathcal{F}, \mathcal{AP})$

Fig. 2. Overview of the abstraction process. An await-time game $\mathcal{I}$, representing the controller synthesis problem under partial observability, is first abstracted into a finite-choice await-time game $\mathcal{F}$, and then into a finite-state game $G^a(\mathcal{F}, \mathcal{AP})$. Games $\mathcal{I}$ and $\mathcal{F}$ have a possibly infinite state space and are played under incomplete information, game $G^a(\mathcal{F}, \mathcal{AP})$ is finite-state and is played under perfect information.

refines the abstraction until either no more abstract counterexamples exist or a concrete counterexample is found. As usual in CEGAR approaches for games [10,8], a spurious counterexample is a strategy tree for the environment player that is winning in the abstract game but not in the concrete game. For timed games, the main difficulty in the characterization of spurious counterexamples is caused by the fact that the number of moves available to the controller is infinite, corresponding to the infinite number of points in time when an action can be taken. As a result, the logical characterization of spurious counterexamples is a *quantified* formula in the theory of linear arithmetic. In the paper, we present a novel *refinement technique* for generating new observation predicates based on quantifier witnesses that eliminate the given strategy.

Figure 2 gives an overview on the abstraction process. We start with a symbolic representation of the timed safety controller synthesis problem with partial observability, which we call *await-time games*. Await-time games allow us to represent the infinite number of choices available to the controller using controllable variables. Corresponding to the two types of observation predicates, action points and decision predicates, we abstract the initial await-time game in two steps into a finite-state game with perfect information. First, we use the action points to eliminate (abstract away) the controllable variables that range over infinite domains. In the resulting *finite-choice* await-time game $\mathcal{F}$, the number of moves available to the controller is finite, but the number of states may still be infinite. In the second step we abstract $\mathcal{F}$ w.r.t. a finite set of predicates $\mathcal{AP}$ to obtain a finite-state perfect-information game $G^a(\mathcal{F}, \mathcal{AP})$. This abstraction completely fixes a finite set of observation predicates the controller can track. Since $G^a(\mathcal{F}, \mathcal{AP})$ is a finite game, we can apply standard algorithms to solve the game and find a winning strategies for the winning player. For a winning strategy for the environment player, we check if the strategy also wins in $\mathcal{F}$ and in $\mathcal{I}$. Strategies that are spurious in $\mathcal{F}$ can be eliminated with additional decision predicates, strategies that are concretizable in $\mathcal{F}$ but spurious in $\mathcal{I}$ need additional action points. We refine both sets until we find either a strategy for the controller in $G^a(\mathcal{F}, \mathcal{AP})$ or a counterexample concretizable in $\mathcal{I}$.

Related Work. The classic solution for *finite-state discrete games* under incomplete information is due to Reif [11] and is based on a determinization-like translation to perfect-information games with a *knowledge-based subset construction*.

Symbolic fixed-point algorithms based on antichains that avoid this determinization procedure were proposed in [7,5]. While these algorithms are applicable to infinite game graphs with a given finite region algebra, they require an *a priori fixed finite set of observations* that the controller is allowed to track.

Abstraction refinement methods were previously applied to games with *perfect information* [10,6] and to safety games with incomplete information [8]. Games under incomplete information are out of the scope of the first two works. The refinement procedure from [8] is based on the assumption that the controller can *choose a move from a finite set*. Unless a finite set of observations is fixed, this is not the case in real-time systems where the controller can let an arbitrary amount of time elapse.

To the best of our knowledge, prior to this work there was no approach to controller synthesis that can handle partial observability for systems that allow for infinitely many choices of the controller, without fixing a priori a finite set of available observations.

2 Timed Controller Synthesis with Partial Observability

Production system controllers are typically required to satisfy timing requirements formulated as safety properties. In a realistic setting, the information that such controllers have at their disposal is limited by their interface and sensor capabilities. Thus, all decisions in the controller's implementation are made based on (possibly) partial observations about the state and the evolution of the controller's external environment.

In this section we recall standard notions and notation and give a definition of the timed safety control problem with partial observability.

Timed Automata and Transition Systems. Given a set X of real-valued variables, $\mathcal{G}(X)$ is the set of constraints generated by: $\varphi := x \sim c \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$, where $x \in X$, $c \in \mathbb{Q}$ and $\sim \in \{<, \leq, >, \geq\}$. $\mathcal{C}(X)$ is the subset of $\mathcal{G}(X)$ that consists of *true* and conjunctions of constraints of the form $x \sim c$. We denote with $\mathbb{R}_{\geq 0}$ ($\mathbb{R}_{>0}$) the sets of non-negative (positive) reals and with $\mathbb{R}^X_{\geq 0}$ the set of total functions from X to $\mathbb{R}_{\geq 0}$. For $v \in \mathbb{R}^X_{\geq 0}$, $Z \subseteq X$ and $t \in \mathbb{R}_{>0}$ we denote with $v[0/Z]$ and with $v + t$ the valuations obtained from v by setting the values of the variables in Z to 0 or adding t to every value in v, respectively. For $v \in \mathbb{R}^X_{\geq 0}$ and $g \in \mathcal{G}(X)$ we write $v \models g$ iff v satisfies g.

For a finite or infinite sequence π of elements of some set A we denote with $|\pi|$ its length, i.e., the number of elements of π and write $|\pi| = \infty$ when π is infinite. For $n \in \mathbb{N}$ with $n < |\pi|$, we denote with $\pi[n]$ and $\pi[0, n]$ the $n + 1$-th element of π and the prefix of length $n + 1$, respectively. For $\pi \in A^+$, $\mathsf{last}(\pi)$ is the last element of π.

A *timed automaton* [1] is a tuple $\mathcal{A} = (\mathsf{Loc}, X, \Sigma, \mathsf{Inv}, R, l_0)$, where Loc is a finite set of locations, X is a finite set of real-valued clocks, Σ is a finite set of actions, $\mathsf{Inv} : \mathsf{Loc} \to \mathcal{C}(X)$ is a function mapping each location to an invariant and $R \subseteq \mathsf{Loc} \times \Sigma \times \mathcal{G}(X) \times 2^X \times \mathsf{Loc}$ is a finite set of transitions.

The semantics of a timed automaton $\mathcal{A}$ is defined by a *timed transition system* $\mathcal{T} = (S, s_0, \Sigma, \to)$, where $S = \mathsf{Loc} \times \mathbb{R}^X_{\geq 0}$ is the set of states, $s_0 = (l_0, 0)$ is the initial state, and the transition relation $\to \subseteq S \times (\Sigma \cup \mathbb{R}_{>0}) \times S$ is such that $((l, v), \sigma, (l', v')) \in \to$ iff $v \models \mathsf{Inv}(l)$, $v' \models \mathsf{Inv}(l')$ and either $\sigma \in \Sigma$ and there exists $(l, \sigma, g, Z, l') \in R$ with $v \models g$ and $v' = v[0/Z]$, or $\sigma \in \mathbb{R}_{>0}$, $v' = v + \sigma$ and $l' = l$. We write $(l, v) \xrightarrow{\sigma} (l', v')$ as a shortcut for $((l, v), \sigma, (l', v')) \in \to$.

Timed Safety Control with Partial Observability. A *partially observable plant* is a tuple $\mathcal{P} = (\mathcal{A}, \Sigma_\mathsf{c}, \Sigma_\mathsf{u}, X_\mathsf{o}, X_\mathsf{u}, =^\mathsf{L}_\mathsf{o})$, where $\mathcal{A}$ is a timed automaton, Σ_c and Σ_u partition the set Σ of actions into a set Σ_c of *controllable* and a set Σ_u of *uncontrollable* actions, X_o and X_u partition the set X of clocks into a set X_o of *observable* and a set X_u of *unobservable* clocks, and $=^\mathsf{L}_\mathsf{o}$ is an *observation equivalence* relation on Loc. We require that $\mathcal{P}$ is *input-enabled*: each $\sigma \in \Sigma_\mathsf{c}$ is enabled in every state s in the transition system. We also assume that at every state a transition from Σ_u is enabled or time can elapse.

A controller for a partially observable plant operates under incomplete information about the location the plant is in and about the values of the plant's clocks. It knows the equivalence class of Loc w.r.t. $=^\mathsf{L}_\mathsf{o}$ in which the plant currently is. The equivalence class of a location $l \in \mathsf{Loc}$ is denoted $[l]_{=^\mathsf{L}_\mathsf{o}}$. The controller observes the values of the observable clocks X_o and doesn't observe those of the clocks in X_u.

Let us consider a partially observable plant $\mathcal{P} = (\mathcal{A}, \Sigma_\mathsf{c}, \Sigma_\mathsf{u}, X_\mathsf{o}, X_\mathsf{u}, =^\mathsf{L}_\mathsf{o})$ with $\mathcal{A} = (\mathsf{Loc}, X, \Sigma, \mathsf{Inv}, R, l_0)$ and let X_c be a finite set of clocks with $X_\mathsf{c} \cap X = \emptyset$.

We note with $X_{\mathsf{o+c}} = X_\mathsf{o} \dot\cup X_\mathsf{c}$ the union of the observable clocks of the plant and the clocks X_c, which are the clocks that belong to the controller. Let $\Sigma_\mathsf{r} = \{reset_Z \mid Z \in 2^{X_\mathsf{c}} \setminus \{\emptyset\}\}$ be a set of actions disjoint from Σ used to model resets of clocks in X_c. Let us define $\widetilde{S} = S \times \mathbb{R}^{X_\mathsf{c}}_{\geq 0}$ and $\widetilde{\Sigma} = \Sigma \cup \mathbb{R}_{>0} \cup \Sigma_\mathsf{r}$.

A X_c*-control strategy for the partially observable plant* $\mathcal{P}$ is a total function $f : \widetilde{S} \times (\widetilde{\Sigma} \times \widetilde{S})^* \to (\Sigma_\mathsf{c} \cup (\Sigma_\mathsf{c} \times X_{\mathsf{o+c}} \times \mathbb{Q}_{>0}) \cup \{\bot\} \cup \Sigma_\mathsf{r})$, which maps finite execution histories to decisions of the controller, which can either be to execute a controllable action σ immediately ($f(\pi) = \sigma$) or when an observable clock x reaches the future time c ($f(\pi) = (\sigma, x, c)$), to remain idle ($f(\pi) = \bot$), or to immediately reset a set of controllable clocks Z ($f(\pi) = reset_Z$). We require the following:

- if the controller decides to execute $\sigma \in \Sigma_\mathsf{c}$ when x reaches c: $f(\pi) = (\sigma, x, c)$, then c is greater than the current value of x, i.e., $c > v(x)$, where $\mathsf{last}(\pi) = (l, v, v_c)$;
- if the controller decides to reset clocks $Z \subseteq X_\mathsf{c}$: $f(\pi) = reset_Z$, then these clocks have positive values, i.e., $v_c(x) > 0$ for every $x \in Z$, where $\mathsf{last}(\pi) = (l, v, v_c)$;
- the value of f changes only when the observation changes or an action in $\Sigma_\mathsf{c} \cup \Sigma_\mathsf{r}$ is executed, i.e., $f(\pi(l, v, v_c)\sigma(l', v', v_c')) = f(\pi(l, v, v_c))$ whenever $\sigma \in \mathbb{R}_{>0}$ or $\sigma \in \Sigma_\mathsf{u}$ and $l =^\mathsf{L}_\mathsf{o} l'$ and for every $x \in X_\mathsf{o}$, $v'(x) = 0$ only if $v(x) = 0$;
- f is *consistent with the observations of the controller*, that is $f(\pi_1) = f(\pi_2)$ for every $\pi_1 \equiv \pi_2$, where the equivalence relation $\equiv$ is defined below.

We first define a function $\mathsf{obs} : \widetilde{S} \times (\widetilde{\Sigma} \times \widetilde{S})^* \to \widetilde{S} \times (\widetilde{S} \times \widetilde{S})^*$ that maps a sequence π to the sequence $\mathsf{obs}(\pi)$ that consists of exactly those transitions of π where a controllable action is taken or a discrete change of the state-based observation occurs. If $\pi = \widetilde{s}$, then $\mathsf{obs}(\pi) = \widetilde{s}$. Otherwise let $\pi = \pi'(l, v, v_c)\sigma(l', v', v_c')$. If either $\sigma \in \Sigma_\mathsf{c} \cup \Sigma_\mathsf{r}$, or $\sigma \in \Sigma_\mathsf{u}$ and $l \neq^\mathsf{L}_\mathsf{o} l'$ or for some $x \in X_\mathsf{o}$, $v(x) > 0$ and $v'(x) = 0$, then $\mathsf{obs}(\pi) = \mathsf{obs}(\pi'(l, v, v_c))((l, v, v_c), (l', v', v_c'))$. Otherwise, $\mathsf{obs}(\pi) = \mathsf{obs}(\pi'(l, v, v_c))$.

For $\pi_1, \pi_2 \in \widetilde{S} \times (\widetilde{S} \times \widetilde{S})^*$ we define $\pi_1 \equiv \pi_2$ iff $|\mathsf{obs}(\pi_1)| = |\mathsf{obs}(\pi_2)|$ and:

- if $\mathsf{obs}(\pi_1)[0] = (l_1^0, v_1^0, v_{c1}^0)$ and $\mathsf{obs}(\pi_2)[0] = (l_2^0, v_2^0, v_{c2}^0)$, then we have $l_1^0 =^\mathsf{L}_\mathsf{o} l_2^0$, ${v_c}_1^0 = {v_c}_1^0$ and for every $x \in X_\mathsf{o}$ it holds that $v_1^0(x) = v_2^0(x)$, and
- for every $0 < i < |\mathsf{obs}(\pi_1)|$ with $\mathsf{obs}(\pi_1)[i] = ((l_1, v_1, v_{c1}), (l_1', v_1', v_{c1}'))$ and $\mathsf{obs}(\pi_2)[i] = ((l_2, v_2, v_{c2}), (l_2', v_2', v_{c2}'))$, we have $l_1 =^\mathsf{L}_\mathsf{o} l_2$, $l_1' =^\mathsf{L}_\mathsf{o} l_2'$, $v_{c1} = v_{c2}$, ${v_c}_1' = {v_c}_2'$, and for every $x \in X_\mathsf{o}$, $v_1(x) = v_2(x)$ and $v_1'(x) = v_2'(x)$.

A control strategy f for the plant $\mathcal{P}$ defines a set of *controlled paths* $\mathcal{CP}(f,\mathcal{P}) = \widetilde{S} \times ((\widetilde{\Sigma} \times \widetilde{S})^* \cup (\widetilde{\Sigma} \times \widetilde{S})^\omega)$, where $\pi \in \mathcal{CP}(f,\mathcal{P})$ iff $\pi[0] = (l_0, 0, 0)$ and for every $0 < i < |\pi| - 1$ with $\pi[i-1] = (l, v, v_c)$, $\pi[i] = \sigma$ and $\pi[i+1] = (l', v', v_c')$ we have:

- if $\sigma \in \mathbb{R}_{>0}$, then $(l, v) \xrightarrow{\sigma} (l', v')$, $v_c' = v_c + \sigma$ and either $f(\pi[0, i-1]) = \bot$ or $f(\pi[0, i-1]) = (a, x, c)$ and $v'(x) \leq c$ (time can elapse only until x reaches c);
- if $\sigma \in \Sigma_{\mathsf{c}}$, then $(l, v) \xrightarrow{\sigma} (l', v')$, $v_c' = v_c$ and either $f(\pi[0, i-1]) = \sigma$ or $f(\pi[0, i-1]) = (\sigma, x, c)$ and $v(x) = c$ (σ is taken immediately or x has reached c);
- if $\sigma \in \Sigma_{\mathsf{u}}$, then $f(\pi[0, i-1]) \notin \Sigma_{\mathsf{r}}$, $(l, v) \xrightarrow{\sigma} (l', v')$ and $v_c' = v_c$;
- if $\sigma = reset_Z$, then $f(\pi[0, i-1]) = reset_Z$, $v_c' = v_c[0/Z]$, $v' = v$, $l' = l$.

A location $l \in \mathsf{Loc}$ is *reachable* in $\mathcal{CP}(f,\mathcal{P})$ iff there exists a finite path $\pi \in \mathcal{CP}(\pi)$ such that $\mathsf{last}(\pi) = (l, v, v_c)$ for some $v \in \mathbb{R}^X_{\geq 0}$ and $v_c \in \mathbb{R}^{X_{\mathsf{c}}}_{\geq 0}$.

We can now state the *timed safety control synthesis problem with partial observability*: Given a partially observable plant $\mathcal{P} = (\mathcal{A}, \Sigma_{\mathsf{c}}, \Sigma_{\mathsf{u}}, X_{\mathsf{o}}, X_{\mathsf{u}}, =^{\mathsf{L}}_{\mathsf{o}})$ with underlying timed automaton $\mathcal{A} = (\mathsf{Loc}, X, \Sigma, \mathsf{Inv}, R, l_0)$, an *error location* $l_{\mathsf{bad}} \in \mathsf{Loc}$, for which $[l_{\mathsf{bad}}]_{=^{\mathsf{L}}_{\mathsf{o}}} = \{l_{\mathsf{bad}}\}$, and a finite set X_{c} of clocks with $X_{\mathsf{c}} \cap X = \emptyset$, find a finite-state X_{c}-control strategy f for $\mathcal{P}$ such that l_{bad} is not reachable in $\mathcal{CP}(f,\mathcal{P})$ or determine that there does not exist a X_{c}-control strategy for $\mathcal{P}$.

3 Await-Time Games

In this section we introduce *await-time games* and show that the timed safety controller synthesis problem with partial observability reduces to the problem of finding a winning strategy for the controller in an await-time game against its environment (the plant).

Let $\mathcal{P} = (\mathcal{A}, \Sigma_{\mathsf{c}}, \Sigma_{\mathsf{u}}, X_{\mathsf{o}}, X_{\mathsf{u}}, =^{\mathsf{L}}_{\mathsf{o}})$ where $\mathcal{A} = (\mathsf{Loc}, X, \Sigma, \mathsf{Inv}, R, l_0)$ be a partially observable plant fixed for the rest of the paper, together with an error location $l_{\mathsf{bad}} \in \mathsf{Loc}$, for which $[l_{\mathsf{bad}}]_{=^{\mathsf{L}}_{\mathsf{o}}} = \{l_{\mathsf{bad}}\}$. Let X_{c} be a fixed finite set of clocks with $X_{\mathsf{c}} \cap X = \emptyset$.

An await-time game models the interaction between a X_{c}-control strategy $\mathsf{Player_c}$, and the partially observable plant $\mathcal{P}$, i.e, the controller's environment $\mathsf{Player_e}$, in a turn-based manner. Whenever it is his turn, $\mathsf{Player_c}$ has the possibility to propose what controllable action should be executed and when. Then, $\mathsf{Player_e}$ can do one or more transitions executing the actual actions of the plant, i.e., updating the location and all clocks, respecting the choice of $\mathsf{Player_c}$. Since the controller and the plant synchronize when a controllable action is executed or a discrete change in the state-based observation has occurred, the turn is back to $\mathsf{Player_c}$ as soon as this happens. More precisely, $\mathsf{Player_c}$ can choose **(C1)** an action $\sigma \in \Sigma_{\mathsf{c}}$ to be executed after a positive delay, or **(C2)** an action $\sigma \in \Sigma_{\mathsf{c}}$ to be executed without delay, or **(C3)** to remain idle, or **(C4)** a set of clocks from X_{c} to be reset immediately. $\mathsf{Player_e}$ can do transitions that correspond to **(E1)** the time-elapsing transitions in the plant, **(E2)** the discrete controllable and **(E3)** uncontrollable transitions in the plant, as well as transitions that let $\mathsf{Player_e}$ **(E4)** reset the controllable clocks selected by $\mathsf{Player_c}$, or **(E5)** give the turn to $\mathsf{Player_c}$.

Formally, an *await-time game* $\mathcal{I}(\mathcal{P}, l_{\mathsf{bad}}, X_{\mathsf{c}}) = (V_{\mathsf{c}}, V_{\mathsf{o}}, V_{\mathsf{u}}, \iota, \mathcal{T}_{\mathsf{c}}, \mathcal{T}_{\mathsf{e}}, \varphi_{\mathsf{bad}})$ is a tuple consisting of pairwise disjoint sets V_{c}, V_{o} and V_{u} of *controllable*, *observable* and *unobservable* variables respectively, and formulas ι, φ_{bad}, $\mathcal{T}_{\mathsf{c}}$ and $\mathcal{T}_{\mathsf{e}}$ that denote the sets of initial and error states and the transition relations for $\mathsf{Player_c}$ and $\mathsf{Player_e}$ respectively.

The states of the game are described by the finite set $V = V_\mathsf{c} \dot{\cup} V_\mathsf{o} \dot{\cup} V_\mathsf{u}$ of variables. We assume a designated boolean variable $t \in V_\mathsf{c}$ that determines which player choses a successor (i.e, updates his variables) in a given state. The transition relations of the players are given as formulas over V and the set V' of primed versions of the variables.

$\mathsf{Player}_\mathsf{c}$ updates the variables in V_c, which model the decisions of the controller. A variable $act \in V_\mathsf{c}$ with $\mathsf{Dom}(act) = \Sigma_\mathsf{c} \cup \{\bot\}$ indicates the selected controllable action in cases **(C1)** and **(C2)** (and is $\bot$ in cases **(C3)** and **(C4)**). In case **(C1)**, $\mathsf{Player}_\mathsf{c}$ also proposes for at least one clock variable $x \in X_{\mathsf{o}+\mathsf{c}}$ a positive constant, called *await point*, indicating that he wants to execute the selected action as soon as x reaches this value, which must be strictly greater than the current value of x. For this, V_c contains a subset $SC = \{c_x \mid x \in X_{\mathsf{o}+\mathsf{c}}\}$ of variables, called *symbolic constants*, ranging over $\mathbb{Q}_{\geq 0}$. V_c contains variables $wait$ and $reset$ with $\mathsf{Dom}(wait) = \mathbb{B}$ and $\mathsf{Dom}(reset) = 2^{X_\mathsf{c}}$.

$\mathsf{Player}_\mathsf{e}$ updates the variables in $V_\mathsf{o} \dot{\cup} V_\mathsf{u} \dot{\cup} \{t\}$. The set V_o contains the clocks in $X_{\mathsf{o}+\mathsf{c}}$ and a variable $oloc$ for modeling the equivalence class of the current location of the plant. The set V_u contains the clocks in X_u and a variable loc for modeling the plant's location. The auxiliary boolean variable $er \in V_\mathsf{o}$ indicates in which states $\mathsf{Player}_\mathsf{c}$ can choose to reset clocks in X_c and the auxiliary boolean variable $et \in V_\mathsf{u}$ is false in states where $\mathsf{Player}_\mathsf{e}$ has disabled further time-elapse transitions.

$\mathsf{Player}_\mathsf{c}$ has incomplete information about the state of the game, which includes the state of the plant – location and clock valuations. He observes only the variables in $V_{\mathsf{o}+\mathsf{c}} = V_\mathsf{c} \dot{\cup} V_\mathsf{o}$ and is thus oblivious to the current location and valuation of X_u.

Since a controller for a partially observable plant does not observe the plant continuously, but only at the points of synchronization, we need to ensure that $\mathsf{Player}_\mathsf{c}$ cannot win the game only by basing his strategic choices on the number of unobservable steps in the play so far, i.e, that if he can win the game then he can do so with a *stuttering invariant strategy* [4]. To this end, we include in the game $\mathcal{I}$ a *skip*-transition for $\mathsf{Player}_\mathsf{e}$, which is enabled in each state that belongs to $\mathsf{Player}_\mathsf{e}$ and allows for making a transition without changing the values of any variables. This transformation is sound for games with safety winning conditions defined by a set of bad states. That way, we will ensure that winning strategies for $\mathsf{Player}_\mathsf{c}$ correspond to X_c-control strategies for $\mathcal{P}$.

For the rest of the paper, we denote with $\mathcal{I}$ the await-time game $\mathcal{I}(\mathcal{P}, l_\mathsf{bad}, X_\mathsf{c})$.

The formulas ι and φ_bad assert respectively that all variables are properly initialized and that $loc = l_\mathsf{bad}$. The formulas $\mathcal{T}_\mathsf{c}$ and $\mathcal{T}_\mathsf{e}$ assert that the players update their variables according to the rules above. Instead of the respective formulas, we give the transition relations of the corresponding explicit game $G(\mathcal{I}) = (Q_\mathsf{c}, Q_\mathsf{e}, q_0, T_\mathsf{c}, T_\mathsf{e}, =_\mathsf{o}, B)$.

(C1)	$q'(act) \in \Sigma_\mathsf{c}$, $q'(wait) = \mathsf{true}$, $q'(reset) = \emptyset$, and $q'(c_x) > 0$ for some $x \in X_{\mathsf{o}+\mathsf{c}}$, and for every $x \in X_{\mathsf{o}+\mathsf{c}}$ with $q'(c_x) > 0$ we have $q'(c_x) > q(x)$
(C2)	$q'(act) \in \Sigma_\mathsf{c}$, $q'(wait) = \mathsf{false}$, $q'(reset) = \emptyset$ and $q'(c) = 0$ for all $c \in SC$
(C3)	$q'(act) = \bot$, $q'(wait) = \mathsf{true}$, $q'(reset) = \emptyset$ and $q'(c) = 0$ for all $c \in SC$
(C4)	$q(er) = \mathsf{true}$ (resetting controllable clocks is allowed only after a controllable action or discrete change of the observation), $q(x) > 0$ for every $x \in q'(reset)$, $q'(act) = \bot$, $q'(wait) = \mathsf{false}$, $q'(reset) \in 2^{X_\mathsf{c}} \setminus \{\emptyset\}$ and $q'(c) = 0$ for all $c \in SC$

Fig. 3. Transition relation T_c of $\mathsf{Player}_\mathsf{c}$: for states q and q', $(q, q') \in T_\mathsf{c}$ iff $q(t) = \mathsf{true}$, $q'(t) = \mathsf{false}$, $q'|_{(V_\mathsf{o} \cup V_\mathsf{u})} = q|_{(V_\mathsf{o} \cup V_\mathsf{u})}$ and one of the conditions **(C1)**, **(C2)**, **(C3)**, **(C4)** holds

(E1)	$q(wait) = \mathsf{true}$, $q(et) = \mathsf{true}$ (time-elapse trans. enabled) and for some $\sigma \in \mathbb{R}_{>0}$: – $(q(loc), q\vert_X) \xrightarrow{\sigma} (q'(loc), q'\vert_X)$, $q'(oloc) = q(oloc)$, and $q'\vert_{X_\mathsf{c}} = q\vert_{X_\mathsf{c}} + \sigma$, – if $q(act) \neq \bot$, then for every $x \in X_{\mathsf{o+c}}$ with $q(x) < q(c_x)$, we have $q'(x) \leq q(c_x)$ (cannot let time pass beyond an await point if Player$_\mathsf{c}$ chose an action in Σ_c), – if $q(act) \neq \bot$, then $q'(et) = \mathsf{false}$ iff $q'(x) \geq q(c_x)$ and $q(x) < q(c_x)$ for some $x \in X_{\mathsf{o+c}}$ with $q(c_x) > 0$ (disable time-elapse transitions upon reaching an await point), – $q'(t) = \mathsf{false}$, $q'(er) = q(er)$
(E2)	$q(wait) = \mathsf{false}$ or $q(et) = \mathsf{false}$ (time-elapse trans. disabled), $q(act) = \sigma \in \Sigma_\mathsf{c}$ and: – $(q(loc), q\vert_X) \xrightarrow{\sigma} (q'(loc), q'\vert_X)$, $q'(oloc) = [q'(loc)]_{=^{\mathsf{L}}_{\mathsf{o}}}$, and $q'\vert_{X_\mathsf{c}} = q'\vert_{X_\mathsf{c}}$, – $q'(t) = \mathsf{true}$ (the turn is back to Player$_\mathsf{c}$), $q'(er) = \mathsf{true}$, $q'(et) = \mathsf{true}$
(E3)	$q(reset) = \emptyset$ and for some $\sigma \in \Sigma_\mathsf{u}$: – $(q(loc), q\vert_X) \xrightarrow{\sigma} (q'(loc), q'\vert_X)$, $q'(oloc) = [q'(loc)]_{=^{\mathsf{L}}_{\mathsf{o}}}$, and $q'\vert_{X_\mathsf{c}} = q'\vert_{X_\mathsf{c}}$, – $q'(t) = \mathsf{true}$ iff $q'(oloc) \neq q(oloc)$ or $q(x) > 0$ and $q'(x) = 0$ for some $x \in X_\mathsf{o}$ (the turn is back to Player$_\mathsf{c}$ iff the observation changed), and if $q'(t) = \mathsf{true}$, then $q'(er) = \mathsf{true}$ and $q'(et) = \mathsf{true}$, otherwise $q'(er) = q(er)$ and $q'(et) = q(et)$
(E4)	$q(reset) = Z \neq \emptyset$ (Player$_\mathsf{c}$ chose to reset the clocks in Z) and: – $q'(x) = 0$ for every $x \in Z$ and $q'(x) = q(x)$ for every $x \in (X \cup X_\mathsf{c}) \setminus Z$, – $q'(loc) = q(loc)$, $q'(oloc) = q(oloc)$, $q'(t) = \mathsf{true}$, $q'(er) = \mathsf{false}$, $q'(et) = \mathsf{true}$ (the turn is back to Player$_\mathsf{c}$ and resetting controllable clocks is disabled)
(E5)	$q(wait) = \mathsf{true}$, $q(act) = \bot$, $q(et) = \mathsf{false}$ (Player$_\mathsf{c}$ chose to remain idle and Player$_\mathsf{e}$ has disabled further time-elapse transitions, after making at least one) and: – $q'(loc) = q(loc)$, $q'(oloc) = q(oloc)$ and $q'(x) = q(x)$ for every $x \in X \cup X_\mathsf{c}$, – $q'(t) = \mathsf{true}$ (give the turn back to Player$_\mathsf{c}$), $q'(er) = \mathsf{false}$, $q'(et) = \mathsf{true}$

Fig. 4. Transition relation T_e of Player$_\mathsf{e}$: for states q and q', $(q, q') \in T_\mathsf{e}$ iff $q(t) = \mathsf{false}$, $q'|_{(V_\mathsf{c} \setminus \{t\})} = q|_{(V_\mathsf{c} \setminus \{t\})}$ and one of the conditions **(E1)**, **(E2)**, **(E3)**, **(E4)**, **(E5)** holds, or $q' = q$

The states of $G(\mathcal{I})$ are valuations of V, i.e., elements of the set $\mathsf{Vals}(V)$ that consists of all total functions $q : V \to \bigcup_{x \in V} \mathsf{Dom}(x)$ such that $q(x) \in \mathsf{Dom}(x)$ for every $x \in V$. For $q \in \mathsf{Vals}(V)$ and $U \subseteq V$, we denote $q|_U$ the projection of q onto U.

The states $Q = Q_\mathsf{c} \dot{\cup} Q_\mathsf{e} = \mathsf{Vals}(V)$ are partitioned into those $Q_\mathsf{c} = \{q \in Q \mid q(t) = \mathsf{true}\}$ that belong to Player$_\mathsf{c}$ and those $Q_\mathsf{e} = \{q \in Q \mid q(t) = \mathsf{false}\}$ that belong to Player$_\mathsf{e}$. The initial state $q_0 \in Q_\mathsf{c}$ is the unique state that satisfies ι and the set B of error states consists of all states in Q_e that satisfy φ_bad.

The *observation equivalence* $=_\mathsf{o}$ on Q is defined by the partitioning of the variables in V as follows: $q_1 =_\mathsf{o} q_2$ iff $q_1|_{V_{\mathsf{o+c}}} = q_2|_{V_{\mathsf{o+c}}}$.

The transition relation $T = T_\mathsf{c} \dot{\cup} T_\mathsf{e}$ is partitioned into the transition relations T_c for Player$_\mathsf{c}$ and T_e for Player$_\mathsf{e}$, which are defined in Fig. 3 and Fig. 4, respectively.

A *path* in $\mathcal{I}$ is a finite or infinite sequence $\pi \in Q^* \cup Q^\omega$ of states such that for all $1 \leq n < |\pi|$, we have $(\pi[n-1], \pi[n]) \in T$. We call a path π maximal iff π is infinite or $\mathsf{last}(\pi) \in B$. A *play* (*prefix*) in $\mathcal{I}$ is a maximal path (finite path) π in $\mathcal{I}$ such that $\pi[0] = q_0$. The extension of $=_\mathsf{o}$ to paths is straightforward. We denote with $\mathsf{prefs}_\mathsf{c}(\mathcal{I})$ ($\mathsf{prefs}_\mathsf{e}(\mathcal{I})$) the set of prefixes π in $\mathcal{I}$ such that $\mathsf{last}(\pi) \in Q_\mathsf{c}$ ($\mathsf{last}(\pi) \in Q_\mathsf{e}$).

A *strategy for* Player$_\mathsf{c}$ is a total function $f_\mathsf{c} : \mathsf{prefs}_\mathsf{c}(\mathcal{I}) \to \mathsf{Vals}(V_\mathsf{c})$ mapping prefixes to valuations of V_c such that for every $\pi \in \mathsf{prefs}_\mathsf{c}(\mathcal{I})$ there exists $q \in Q$ with $(\mathsf{last}(\pi), q) \in T_\mathsf{c}$ such that $f_\mathsf{c}(\pi) = q|_{V_\mathsf{c}}$, and which is consistent w.r.t. $=_\mathsf{o}$: for all $\pi_1, \pi_2 \in \mathsf{prefs}_\mathsf{c}(\mathcal{I})$ with $\pi_1 =_\mathsf{o} \pi_2$ it holds that $f_\mathsf{c}(\pi_1) = f_\mathsf{c}(\pi_2)$. A strategy for

$\mathsf{Player}_\mathsf{e}$ is a total function $f_\mathsf{e} : \mathsf{prefs}_\mathsf{e}(\mathcal{I}) \to \mathsf{Vals}(V_\mathsf{o} \cup V_\mathsf{u} \cup \{t\})$ such that for every $\pi \in \mathsf{prefs}_\mathsf{e}(\mathcal{I})$ there is a $q \in Q$ with $(\mathsf{last}(\pi), q) \in T_\mathsf{e}$ and $f_\mathsf{e}(\pi) = q|_{(V_\mathsf{o} \cup V_\mathsf{u} \cup \{t\})}$.

The *outcome* of a strategy f_c for $\mathsf{Player}_\mathsf{c}$ is the set $\mathsf{outcome}(f_\mathsf{c})$ of plays such that $\pi \in \mathsf{outcome}(f_\mathsf{c})$ iff for every $0 < n < |\pi|$ with $\pi[n-1] \in Q_\mathsf{c}$ it holds that $\pi[n]|_{V_\mathsf{c}} = f_\mathsf{c}(\pi[0, n-1])$. A strategy f_c for $\mathsf{Player}_\mathsf{c}$ is *winning* if for every $\pi \in \mathsf{outcome}(f_\mathsf{c})$ and every $n \geq 0$, $\pi[n] \notin B$. The outcome of a strategy f_e for $\mathsf{Player}_\mathsf{e}$ is defined analogously and f_e is *winning* if for every $\pi \in \mathsf{outcome}(f_\mathsf{e})$ there exists a $n \geq 0$ such that $\pi[n] \in B$. For a strategy f we denote with $\mathsf{prefs}(f)$ the set of all prefixes in $\mathsf{outcome}(f)$.

We can reduce the timed safety control synthesis problem with partial observability to finding a finite-state winning strategy for $\mathsf{Player}_\mathsf{c}$ in $\mathcal{I}$.

Proposition 1. *There exists a finite-state X_c-control strategy for the partially observable plant $\mathcal{P}$ with error location l_bad, such that l_bad is not reachable in $\mathcal{CP}(f, \mathcal{P})$ iff* $\mathsf{Player}_\mathsf{c}$ *has a finite-state winning strategy in the await-time game $\mathcal{I}(\mathcal{P}, l_\mathsf{bad}, X_\mathsf{c})$.*

4 Abstracting Await-Time Games

In this section we describe an abstraction-based approach to solving await-time games. A finite-state abstract game with perfect information is constructed in two steps. In the first step we construct an await-time game with fixed action points $\mathcal{F}$ by abstracting away the symbolic constants, and thus leaving $\mathsf{Player}_\mathsf{c}$ with a finite state of possible choices in each of its states, and letting $\mathsf{Player}_\mathsf{e}$ resolve the resulting nondeterminism. In the second step we do predicate abstraction of $\mathcal{F}$ w.r.t. a finite set $\mathcal{AP}$ of predicates and thus completely fix the set of (observation) predicates that the controller can track.

Step 1: Fixing the action points. A *finite-choice await-time game* $\mathcal{F}(\mathcal{I}, \xi)$ for the game $\mathcal{I}$ is defined by an *action-point function* $\xi : X_{\mathsf{o}+\mathsf{c}} \to 2^{\mathbb{Q}_{>0}}$. For each clock $x \in X_{\mathsf{o}+\mathsf{c}}$, the set $\xi(x) \subseteq \mathbb{Q}_{>0}$ is a *finite* set of positive rational constants called *action points for* x. The action points for a clock $x \in X_{\mathsf{o}+\mathsf{c}}$ are used to replace the symbolic constant c_x from $\mathcal{I}$ in $\mathcal{F}(\mathcal{I}, \xi)$ as we describe below.

Formally, $\mathcal{F}(\mathcal{I}, \xi) = (V_\mathsf{c}^\mathsf{f}, V_\mathsf{o}, V_\mathsf{u}, \iota^\mathsf{f}, \mathcal{T}_\mathsf{c}^\mathsf{f}, \mathcal{T}_\mathsf{e}^\mathsf{f}, \varphi_\mathsf{bad})$ is a symbolic game that differs from $\mathcal{I}$ in the set of controllable variables, the formulas for the transition relations and the formula describing the initial state. We define $V_\mathsf{c}^\mathsf{f} = V_\mathsf{c} \setminus SC$. Thus, the formula ι^f and the transition formula $\mathcal{T}_\mathsf{c}$ for $\mathsf{Player}_\mathsf{c}$ do not contain assignments to SC.

The possible options for $\mathsf{Player}_\mathsf{c}$ in $\mathcal{F}$ are the same as in $\mathcal{I}$ except for **(C1)**, which is replaced by **(C1$^\mathsf{f}$)** where $\mathsf{Player}_\mathsf{c}$ selects an action $\sigma \in \Sigma_\mathsf{c}$ to be executed after a delay determined by $\mathsf{Player}_\mathsf{e}$. As now $\mathsf{Player}_\mathsf{c}$ updates only the finite-range variables V_c^f, from each $\mathsf{Player}_\mathsf{c}$-state he can choose among finitely many possible successors.

The nondeterminism resulting from replacing **(C1)** by **(C1$^\mathsf{f}$)**, i.e, regarding exactly how much time should elapse before the selected controllable action σ is executed, is resolved by $\mathsf{Player}_\mathsf{e}$. The action σ can be fired at any time *up to (and including) the first action point* reached after a positive amount of time has elapsed. This is achieved by replacing **(E1)** by **(E1$^\mathsf{f}$)**, where $\mathsf{Player}_\mathsf{e}$ can choose to disable further delay transitions at any point. Furthermore, according to **(E1$^\mathsf{f}$)** the duration of the time-elapse transitions is constrained by the action points, regardless of whether $\mathsf{Player}_\mathsf{c}$ has chosen to execute a controllable action after a delay or to remain idle. This allows $\mathsf{Player}_\mathsf{c}$ to remain idle until reaching an action point and then choose to execute a controllable action.

By the definition, in the game $\mathcal{F}(\mathcal{I}, \xi)$ $\mathsf{Player}_\mathsf{e}$ is more powerful than in the game $\mathcal{I}$, while $\mathsf{Player}_\mathsf{c}$ is weaker. Thus, $\mathcal{F}(\mathcal{I}, \xi)$ soundly abstracts the await-time game $\mathcal{I}$.

Proposition 2. *For every action-point function $\xi : X_{\mathsf{o+c}} \to 2^{\mathbb{Q}_{>0}}$, if* $\mathsf{Player}_\mathsf{c}$ *has a (finite-state) winning strategy in the finite-choice await-time game $\mathcal{F}(\mathcal{I}, \xi)$, then* $\mathsf{Player}_\mathsf{c}$ *has a (finite-state) winning strategy in the await-time game $\mathcal{I}$.*

Step 2: Predicate abstraction. We now consider a finite set $\mathcal{AP}$ of predicates that contains at least the atomic formulas occurring in some of ι^f and φ_{bad}. We further require that $\mathcal{AP}$ is precise w.r.t. the (finitely many) choices of $\mathsf{Player}_\mathsf{c}$ in the game $\mathcal{F}$ and w.r.t. every finite-range $v \in V_\mathsf{o}$. That is, $\mathcal{AP}$ contains all boolean variables from $V_{\mathsf{o+c}}$ plus the predicate $v = d$ for every finite-range $v \in V_{\mathsf{o+c}}$ and $d \in \mathsf{Dom}(v)$.

We ensure that $\mathcal{AP}$ is precise w.r.t. the action points in $\mathcal{F}$ by including the predicates $x \leq c$ and $x \geq c$ for each $x \in X_{\mathsf{o+c}}$ and $c \in \xi(x)$. Thus, the observable predicates in $\mathcal{AP}$ (i.e., those referring only to variables in $V_{\mathsf{o+c}}$) are exactly the observation predicates the controller can track in the current abstraction.

We employ the abstraction procedure from [8] to construct a *finite-state perfect-information* abstract game $G^a(\mathcal{F}, \mathcal{AP}) = (Q^a_\mathsf{c}, Q^a_\mathsf{e}, q^a_0, T^a_\mathsf{c}, T^a_\mathsf{e}, B^a)$, which is an explicit safety game. The set of abstract states $Q^a = Q^a_\mathsf{c} \dot\cup Q^a_\mathsf{e}$ is partitioned into the sets of states Q^a_c and Q^a_e that belong to $\mathsf{Player}_\mathsf{c}$ and $\mathsf{Player}_\mathsf{e}$ respectively. The game has a unique initial state q^a_0, and set of error states B^a. The abstract transition relation $T^a = T^a_\mathsf{c} \dot\cup T^a_\mathsf{e}$ is partitioned into T^a_c and T^a_e for the two players and is such that $T^a_\mathsf{c} \subseteq Q^a_\mathsf{c} \times Q^a_\mathsf{e}$ and $T^a_\mathsf{e} \subseteq Q^a_\mathsf{e} \times Q^a$ and each state in Q^a has a successor.

Here a strategy for $\mathsf{Player}_\mathsf{p}$, where $p \in \{\mathsf{c}, \mathsf{e}\}$ is a total function $f^a_\mathsf{p} : \mathsf{prefs}_\mathsf{p}(G^a) \to Q^a$ such that for every $\pi \in \mathsf{prefs}_\mathsf{p}(G^a)$, if $f^a(\pi) = q$, then $(\mathsf{last}(\pi), q) \in T^a_\mathsf{p}$.

The soundness of this abstraction guarantees that if $\mathsf{Player}_\mathsf{c}$ has a winning strategy f^a_c in $G^a(\mathcal{F}, \mathcal{AP})$, then there exists a finite-state concretization f_c of f^a_c which is a winning strategy for $\mathsf{Player}_\mathsf{c}$ in $\mathcal{F}$ (and hence $\mathsf{Player}_\mathsf{c}$ has a winning strategy in $\mathcal{I}$).

5 Counterexample-Guided Observation Refinement

We now present a procedure for automatically refining the observation predicates in case of a spurious abstract counterexample. This procedure takes into account the two steps of the abstraction phase. Since the predicate abstraction procedure is part of the CEGAR-loop from [8], we refer the reader to the interpolation-based refinement method described there to generate new predicates for $\mathcal{AP}$ in the case when the abstract counterexample does not correspond to a counterexample in the game $\mathcal{F}$. In the following, we focus on the case when the predicate abstraction cannot be further refined due to the fact that the abstract counterexample does correspond to a counterexample in the game $\mathcal{F}$. In this case, we check if it actually corresponds to a concrete counterexample in the game $\mathcal{I}$. We now define a symbolic characterization of the counterexamples that are concretizable in $\mathcal{I}$, and develop a refinement procedure for $\mathcal{F}(\mathcal{I}, \xi)$.

Let $\xi : X_{\mathsf{o+c}} \to 2^{\mathbb{Q}_{>0}}$ be an action-point function, and let $\mathcal{AP}$ be a finite set of predicates. Let $\mathcal{F} = \mathcal{F}(\mathcal{I}, \xi)$ be the corresponding finite-choice await-time game, and let $G^a(\mathcal{F}, \mathcal{AP}) = (Q^a_\mathsf{c}, Q^a_\mathsf{e}, q^a_0, T^a_\mathsf{c}, T^a_\mathsf{e}, =^a_\mathsf{o}, B^a)$ be its abstraction w.r.t. $\mathcal{AP}$. Suppose that there exists a winning strategy f^a_e for $\mathsf{Player}_\mathsf{e}$ in $G^a(\mathcal{F}, \mathcal{AP})$.

Abstract Counterexample Strategies. A winning strategy f^a_{e} for $\mathsf{Player}_{\mathsf{e}}$ in $G^a(\mathcal{F},\mathcal{AP})$ is an *abstract counterexample*. For a sequence $\rho \in (2^Q)^* \cup (2^Q)^\omega$ of sets of states in a game, we define $\gamma(\rho)$ as the set of paths π such that $|\pi| = |\rho|$ and for every $0 \le i < |\pi|$ it holds that $\pi[i] \in \rho[i]$. For $\rho_1, \rho_2 \in (2^Q)^* \cup (2^Q)^\omega$ we write $\rho_1 \subseteq \rho_2$ iff $|\rho_1| = |\rho_2|$ and $\gamma(\rho_1) \subseteq \gamma(\rho_2)$. We denote with $\kappa(\mathcal{I})$ and $\kappa(\mathcal{F})$ the perfect-information games for $\mathcal{I}$ and $\mathcal{F}$ defined by knowledge-based subset construction [11].

We say that the strategy f^a_{e} is *concretizable in* $\mathcal{F}$ iff there exists a winning strategy f^κ_{e} for $\mathsf{Player}_{\mathsf{e}}$ in $\kappa(\mathcal{F})$ such that for every $\pi^\kappa \in \mathsf{prefs}(f^\kappa_{\mathsf{e}})$ there exists a $\pi^a \in \mathsf{prefs}(f^a_{\mathsf{e}})$ such that $\pi^\kappa \subseteq \pi^a$. *Concretizability of* f^a_{e} *in* $\mathcal{I}$ is defined analogously.

Since $G^a(\mathcal{F},\mathcal{AP})$ is a finite-state *safety* game, the strategy f^a_{e} can be represented as a finite tree $\mathsf{Tree}(f^a_{\mathsf{e}})$ called *strategy tree* for f^a_{e}, which can be used for constructing a logical formula characterizing the concretizability of f^a_{e}.

Each node in $\mathsf{Tree}(f^a_{\mathsf{e}})$ is identified by a unique $n \in \mathbb{N}$ and is labeled with a state in Q^a denoted $\mathsf{state}(n)$. For a node n, we denote with $\mathsf{children}(n)$ the set of all children of n, with $\mathsf{path}(n)$ the sequence of nodes on the path from the root to n and with $\mathsf{pref}(n)$ the prefix in $G^a(\mathcal{F},\mathcal{AP})$ formed by the sequence of states corresponding to $\mathsf{path}(n)$.

The tree contains a root node 0 labeled with the initial abstract state q^a_0. For each edge from n to m in $\mathsf{Tree}(f^a_{\mathsf{e}})$ it holds that $(\mathsf{state}(n), \mathsf{state}(m)) \in T^a$. If $n \in \mathsf{Tree}(f^a_{\mathsf{e}})$, $\mathsf{state}(n) \notin B^a$ and $\mathsf{state}(n) \in Q^a_{\mathsf{e}}$, then there exists a single child m of n in $\mathsf{Tree}(f^a_{\mathsf{e}})$ and $\mathsf{state}(m) = f^a_{\mathsf{e}}(\mathsf{pref}(n))$. If $n \in \mathsf{Tree}(f^a_{\mathsf{e}})$, $\mathsf{state}(n) \notin B^a$ and $\mathsf{state}(n) \in Q^a_{\mathsf{c}}$, then for every $s \in Q^a$ with $(\mathsf{state}(n), s) \in T^a_{\mathsf{c}}$ there exists exactly one child m of n in $\mathsf{Tree}(f^a_{\mathsf{e}})$ and $\mathsf{state}(m) = s$. If $\mathsf{state}(n) \in B^a$ then $\mathsf{children}(n) = \emptyset$.

Counterexample Strategies That are Spurious in $\mathcal{F}$. If the counterexample-analysis from [8] reports that f^a_{e} is not concretizable in $\mathcal{F}$, we refine the set $\mathcal{AP}$ with the predicates generated by the interpolation-based refinement procedure described there and continue. Otherwise, $\mathsf{Player}_{\mathsf{e}}$ has a winning strategy in $\kappa(\mathcal{F})$, which implies that $\mathsf{Player}_{\mathsf{c}}$ does not have a winning strategy in $\mathcal{F}$. This fact does not imply that f^a_{e} is concretizable in $\mathcal{I}$, as the action-point function ξ might just be too imprecise.

Counterexample Strategies That are Concretizable in $\mathcal{I}$. We now provide a logical characterization of winning strategies for $\mathsf{Player}_{\mathsf{e}}$ in $G^a(\mathcal{F},\mathcal{AP})$ that are concretizable in $\mathcal{I}$. The result is a linear arithmetic formula with alternating universal and existential quantifiers corresponding to the alternating choices of the two players. The variables updated by $\mathsf{Player}_{\mathsf{e}}$ are existentially quantified and the variables updated by $\mathsf{Player}_{\mathsf{c}}$, including the symbolic constants, are universally quantified.

The label $\mathsf{state}(n)$ of each node n in $\mathsf{Tree}(f^a_{\mathsf{e}})$ is a state in $G^a(\mathcal{F},\mathcal{AP})$ and is thus a set of valuations of the abstraction predicates $\mathcal{AP}$. We associate with $\mathsf{state}(n)$ a boolean combination of elements of $\mathcal{AP}$ and thus a formula $\psi^n_{\mathsf{st}}[V^n]$ (V^n consists of indexed versions of the variables in V and $\mathsf{Dom}(x^n) = \mathsf{Dom}(x)$ for $x \in V$). The formula $\psi^n_{\mathsf{st}}[V^n]$, which is the disjunction of all conjunctions representing the valuations in $\mathsf{state}(n)$, characterizes the set of states in $\mathcal{I}$ that are in the concretization of $\mathsf{state}(n)$.

As the abstraction is precise w.r.t. the choices of $\mathsf{Player}_{\mathsf{c}}$ in $\mathcal{F}$, for each node n in $\mathsf{Tree}(f^a_{\mathsf{e}})$, $\mathsf{state}(n)$ defines a valuation $\mathsf{contr}(n) = \mathsf{state}(n)|_{V^{\mathsf{f}}_{\mathsf{c}}}$ of the variables in $V^{\mathsf{f}}_{\mathsf{c}}$.

For two nodes n and m such that $m \in \mathsf{children}(n)$ and $\mathsf{state}(n) \in Q^a_p$, where $p \in \{\mathsf{c}, \mathsf{e}\}$, we define the formula $\psi^{n,m}_{\mathsf{tr}}$ that denotes the transitions in $\mathcal{I}$ from states that satisfy ψ^n_{st}: $\psi^{n,m}_{\mathsf{tr}}[V^n, V^m] \equiv \psi^n_{\mathsf{st}} \wedge \mathcal{T}_p[V^n/V, V^m/V']$.

We now turn to the definition of the *quantified strategy-tree formula* $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ that characterizes the concretizability of f^a_e in $\mathcal{I}$. We annotate in a bottom-up manner each node $n \in \mathsf{Tree}(f^a_\mathsf{e})$ with a quantified linear arithmetic formula φ^n and define $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e})) = \varphi^{n_0}[0/SC^{n_0}]$ where n_0 is the root of $\mathsf{Tree}(f^a_\mathsf{e})$. If $\mathsf{path}(n) = n_0n_1 \ldots n_r$, the formula φ^n will have free variables in $SC^n \cup \bigcup_{i=0}^{r-1}(V^{n_i}_{\mathsf{o+c}} \cup SC^{n_i})$ and thus $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ will be a closed formula.

The annotation formula φ^n for a node n describes the set of prefixes in $\kappa(\mathcal{I})$ that are subsumed by $\mathsf{pref}(n)$ and lead to a state from which $\mathsf{Player_e}$ has a winning strategy in $\kappa(\mathcal{I})$ contained in the corresponding subtree of n. Thus, the formula $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ is satisfiable iff $\mathsf{Player_e}$ has a winning strategy in $\kappa(\mathcal{I})$ subsumed by f^a_e.

– For a leaf node n (with $\mathsf{state}(n) \in B^a$) with $\mathsf{path}(n) = n_0n_1 \ldots n_r$ we define:

$$\varphi^n \equiv \exists V^n_\mathsf{o} \exists V^{n_0}_\mathsf{u} \ldots \exists V^{n_r}_\mathsf{u} \Big(\big(\bigwedge_{i=0}^{r-1} \psi^{n_i,n_{i+1}}_\mathsf{tr} \wedge \psi^{n_r}_\mathsf{st} \wedge loc^{n_r} = l_\mathsf{bad} \big) [\mathsf{contr}(n)/V^{\mathsf{f}^n}_\mathsf{c}] \Big).$$

– For a non-leaf node n with $\mathsf{state}(n) \in Q^a_\mathsf{e}$ and (single) child m, we define:

$$\varphi^n \equiv \exists V^n_\mathsf{o} \big(\varphi^m [SC^n/SC^m, \mathsf{contr}(n)/V^{\mathsf{f}^n}_\mathsf{c}] \big).$$

– For a non-leaf node n with $\mathsf{state}(n) \in Q^a_\mathsf{c}$, we first define a formula $\varphi^{n,m}$ for each node $m \in \mathsf{children}(n)$. The definition of $\varphi^{n,m}$, i.e., the treatment of the symbolic constants SC^m in φ^m, depends on $\mathsf{contr}(m)$, i.e., on the choice made by $\mathsf{Player_c}$ in the game $\mathcal{F}$. For successors m where $\mathsf{Player_c}$ chose to allow $\mathsf{Player_e}$ to decide when to execute the controllable action, we quantify universally over the variables in SC^m, adding a condition which restricts their values to ones that are valid choices of await points for $\mathsf{Player_c}$ in $\mathcal{I}$. In order to ensure intermediate action points, we further require that each c^m_x is smaller than the smallest action point in $\xi(x)$ that is larger than the current value of x. This gives a condition θ^m on the symbolic constants SC^m at node m and we define $\varphi^{n,m} \equiv \forall SC^m(\theta^m \rightarrow \varphi^m)$, where

$$\theta^m \equiv \big(\bigvee_{x \in X_{\mathsf{o+c}}} c^m_x{>}0 \big) \wedge \bigwedge_{x \in X_{\mathsf{o+c}}} (c^m_x{>}0 \rightarrow c^m_x{>}x^n) \wedge \bigwedge_{\substack{x \in X_{\mathsf{o+c}} \\ c \in \xi(x)}} \big(x^n{<}c \rightarrow c^m_x{<}c \big).$$

For successors m where $\mathsf{Player_c}$ chose to execute a controllable action immediately, to reset some controllable clocks, or to remain idle, we substitute SC^m by 0, i.e., $\varphi^{n,m} \equiv \varphi^m[0/SC^m]$ (which agrees with the transition relation $\mathcal{T}_\mathsf{c}$ in $\mathcal{I}$). Finally:

$$\varphi^n \equiv \exists V^n_\mathsf{o} \big(\bigwedge_{m \in \mathsf{children}(n)} \varphi^{n,m} [\mathsf{contr}(n)/V^{\mathsf{f}^n}_\mathsf{c}] \big).$$

The formula $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ characterizes the concretizability of f^a_e in $\mathcal{I}$. Thus, if $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ is satisfiable, then $\mathsf{Player_c}$ has no finite-state winning strategy in $\mathcal{I}$.

Proposition 3. *For every winning strategy f^a_e for* $\mathsf{Player_e}$ *in $G^a(\mathcal{F}, \mathcal{AP})$, the formula* $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ *is satisfiable iff the strategy f^a_e is concretizable in $\mathcal{I}$.*

Extracting Refinement Action Points from a Model. We now consider the case when the formula $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ is unsatisfiable. Since $\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ is a closed formula, its negation $\Phi = \neg\mathsf{QTF}(\mathsf{Tree}(f^a_\mathsf{e}))$ is satisfiable. In Φ all symbolic constants (indexed accordingly) are existentially quantified. Our goal is to compute witnesses for the symbolic constants that can be used for refining the action-point function ξ to eliminate in the resulting finite-choice game the winning strategies for $\mathsf{Player_e}$ subsumed by f^a_e.

Consider a block $\exists SC^n$ of existentially quantified symbolic constants in Φ. The block $\exists SC^n$ is preceded by the blocks of universal quantifiers $\forall V_\mathsf{o}^{n_i}$ for $i = 0, \ldots, r$, where $\mathsf{path}(n) = n_0 n_1 \ldots n_r$. Thus, a witness $\mathsf{w}(c)$ for a symbolic constant $c \in SC^n$ for the satisfiability of Φ is a function $\mathsf{w}(c) : \mathsf{Vals}(V_\mathsf{o}^{n_0}) \times \ldots \times \mathsf{Vals}(V_\mathsf{o}^{n_r}) \to \mathbb{Q}_{\geq 0}$.

Assume for now that we have a tuple of witness functions for the variables in SC^n of the following form. For some $k \in \mathbb{N}_{>0}$, there are k positive rational constants $a_1, \ldots, a_k$ and each a_i is associated with some variable $x \in X_{\mathsf{o+c}}$. The functions are such that we have a case split with k cases according to the valuation v of the observable variables along the prefix, such that in case i we have $\mathsf{w}(c_x)(v) = a_i$, where x is the variable associated with a_i and $\mathsf{w}(c_y)(v) = 0$ for all other $c_y \in SC^n$.

Let $\mathsf{id} : X_{\mathsf{o+c}} \to \mathbb{N} \cap [1, |X_{\mathsf{o+c}}|]$ be an indexing function for the clock variables $X_{\mathsf{o+c}}$. Thus, each a_i is associated with an index $d_i \in [1, |X_{\mathsf{o+c}}|]$ of a variable in $X_{\mathsf{o+c}}$.

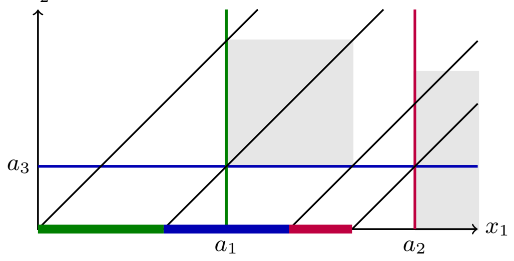
Example. Consider an example with $X_{\mathsf{o+c}} = \{x_1, x_2\}$, where in the abstract state $\mathsf{state}(n_r)$ we know that the value of $x_1^{n_r}$ is in $[0, 5]$ and the value of $x_2^{n_r}$ is 0, and the "good" values for SC^n are depicted as the gray sets on Fig. 5. The figure shows an example where $k = 3$ and each a_i is associated with the shown variable-index d_i. □

More formally, we assume the existence of a function $b : \mathsf{Vals}(V_\mathsf{o}^{n_0}) \times \ldots \times \mathsf{Vals}(V_\mathsf{o}^{n_r}) \to \mathbb{N} \cap [1, k]$, such that for each $c_x^n \in SC^n$, the witness function $\mathsf{w}(c_x^n)$ is such that for every valuation $v \in \mathsf{Vals}(V_\mathsf{o}^{n_0}) \times \ldots \times \mathsf{Vals}(V_\mathsf{o}^{n_r})$ we have $\mathsf{w}(c_x^n)(v) = a_i$ for some i iff $b(v) = i$ and $d_i = \mathsf{id}(x)$ and $\mathsf{w}(c_x^n)(v) = 0$ otherwise.

We can then refine ξ as follows: For each $x \in X_{\mathsf{o+c}}$ we add to the set $\xi(x)$ all positive constants a such that $\mathsf{w}(c_x^n)(v) = a$ for some node n and valuation v. The form of the function $\mathsf{w}(c_x^n)$ implies that the number of these constants is finite.

For a given $k \in \mathbb{N}_{>0}$, we restrict the possible witnesses for SC^n to the above form by strengthening the formula Φ. To this end, we replace the condition θ^n for SC^n that was used in the construction of Φ a stronger one, $\theta^n \wedge \theta_k^n$ where θ_k^n is defined below.

The formula θ_k^n refers to a fresh bounded integer variable b^n with domain $\mathbb{N} \cap [1, k]$, and the variables from a set $A_k^n = \{a_1^n, \ldots, a_k^n\}$ of k fresh rational variables and from a set $D_k^n = \{d_1^n, \ldots, d_k^n\}$ of k fresh bounded integer variables. The variable b^n is



$a_1 = 3, d_1 = \mathsf{id}(x_1)$
$a_2 = 6, d_2 = \mathsf{id}(x_1)$
$a_3 = 1, d_3 = \mathsf{id}(x_2)$

$x_1^{n_r} \in [0, 2) \mapsto c_{x_1}^n = 3, c_{x_2}^n = 0$
$x_1^{n_r} \in [2, 4) \mapsto c_{x_1}^n = 0, c_{x_2}^n = 1$
$x_1^{n_r} \in [4, 5) \mapsto c_{x_1}^n = 6, c_{x_2}^n = 0$

Fig. 5. Example of witnesses for symbolic constants $c_{x_1}^n$ and $c_{x_2}^n$

existentially quantified together with the symbolic constants in SC^n. The variables in A_k^n and D_k^n are free in the resulting formula. We define the formula θ_k^n as:

$$\theta_k^n \equiv \exists b^n \bigwedge_{i=1}^{k} \Big(b^n = i \rightarrow \bigwedge_{x \in X_{\mathsf{o+c}}} \big((\mathsf{id}(x) = d_i^n \rightarrow c_x^n = a_i^n) \wedge (\mathsf{id}(x) \neq d_i^n \rightarrow c_x^n = 0) \big) \Big).$$

The refinement procedure iterates over the values of $k \geq 1$, at each step constructing a formula Φ_k by replacing each constraint θ^n in Φ by $\theta^n \wedge \theta_k^n$. For each $k \geq 1$, Φ_k is a strengthening of Φ and Φ_{k+1} is weaker than Φ_k. The procedure terminates if a k for which the formula Φ_k is satisfiable is reached. In this case we use the values of the newly introduced variables from A_k^n to refine the action-point function ξ. Thus, if it terminates, Algorithm 1 returns a new action-point function ξ' such that for every $x \in X_{\mathsf{o+c}}$, we have $\xi'(x) \supseteq \xi(x)$. The new action points for x are extracted from a model for Φ_k as the values of those variables a_i^n for which d_i^n is equal to $\mathsf{id}(x)$, and they suffice to eliminate all strategies f_{e}^{κ} winning for $\mathsf{Player_e}$ in $\mathcal{F}$ that are in the concretization of f_{e}^a.

Algorithm 1. Computation of refinement action points

```
Input: satisfiable $\Phi$ with $\theta^{n_i}$ for $SC^{n_i}$, for $i = 1, \ldots, m$; function $\xi : X_{\mathsf{o+c}} \rightarrow 2^{\mathbb{Q}_{>0}}$
Output: function $\xi' : X_{\mathsf{o+c}} \rightarrow 2^{\mathbb{Q}_{>0}}$
$\xi'(x) := \xi(x)$ for every $x \in X_{\mathsf{o+c}}$; sat := false; k := 0;
while sat == false do { k++; construct $\Phi_k$; sat := check($\Phi_k$); }
$\mathcal{M}$ = model($\Phi_k$);
foreach $(n, i) \in (\{n_1 ... n_m\} \times \{1, ..k\})$ with $\mathcal{M}(a_i^n) > 0$ do
    forall the $x \in X_{\mathsf{o+c}}$ with $\mathsf{id}(x) = \mathcal{M}(d_i^n)$ do $\xi'(x) := \xi'(x) \cup \{\mathcal{M}(a_i^n)\}$;
end
return $\xi'$
```

Proposition 4. *Let f_{e}^a be a winning strategy for* $\mathsf{Player_e}$ *in $G^a(\mathcal{F}, \mathcal{AP})$. If Algorithm 1 terminates, it returns an action-point function ξ' such that for the await-time game with fixed action points $\mathcal{F}' = \mathcal{F}(\mathcal{I}, \xi')$,* $\mathsf{Player_e}$ *has no winning strategy f_{e}^{κ} in $\kappa(\mathcal{F}')$ such that for every $\pi^{\kappa} \in \mathsf{prefs}(f_{\mathsf{e}}^{\kappa})$ there exists a $\pi^a \in \mathsf{prefs}(f_{\mathsf{e}}^a)$ with $\pi^{\kappa} \subseteq \pi^a$.*

6 Results and Conclusions

We developed a prototype implementation of the presented extension of the CEGAR procedure from [8] to the case of await-time games. We applied our prototype to the safety controller synthesis problem for the Box Painting Production System and the Timed Game For Sorting Bricks examples due to Cassez et al. [4]. We encoded the synthesis problems as await-time games and, starting with empty sets of action points, applied our method to compute observations for which the plants are controllable.

In Table 1 we report on the results from our experiments preformed on an Intel Core 2 Duo CPU at 2.53 GHz with 3.4 GB RAM. We present the maximal number of explored states in the intermediate abstractions, the size of the abstract strategy, the number of action points in the final game, as well as the number of refinement iterations for the await-time game with fixed action points and the number of refinement

Table 1. Results from experiments with our prototype on Box Painting Production System and Timed Game For Sorting Bricks: number of states in largest intermediate abstraction, size of abstract strategy for the controller, number of action points in final abstraction, number of iterations of the respective refinement loops and running time (in seconds). Results from experiments with UPPAAL-TIGA with fixed observations (controllable cases): running time (in seconds)

	A. States	A. Strategy	Act. Points	OBS Iter.	CEGAR Iter.	Time	TIGA
Paint	626	55	2	2	8	73.50	0.08
Paint-100	573	49	2	2	5	29.65	3.57
Paint-1000	573	49	2	2	5	29.53	336.34
Paint-10000	560	76	2	2	7	54.85	> 1800
Paint-100000	614	55	2	2	7	52.88	> 1800
Bricks	1175	125	3	3	3	24.85	0.05
Bricks-100	1175	175	3	3	3	25.16	2.63
Bricks-1000	1175	176	3	3	3	25.29	302.08
Bricks-10000	1175	176	3	3	3	25.83	> 1800
Bricks-100000	1175	175	3	3	3	25.40	> 1800

iterations of the CEGAR loop. In order to demonstrate that our method performs well in situations where fine granularity is needed to win the game, i.e., when the constraints occurring in the plant involve large constants and the differences between certain guards and invariants are small, we constructed multiple instances of each example. Instances Bricks$-N$ and Paint$-N$, where $N \in \{100, 1000, 10000, 100000\}$ were obtained by adding the constant N to all positive constants occurring in the plants.

The results show that the size of the abstract games and strategies generated by our approach depend on the number of action points and predicates and not on the size of the constants in the plant. This is in contrast with approaches based on fixed granularity, where strategies involve counting modulo the given granularity.

Since the problem of synthesizing observation predicates for timed games under incomplete information is out of the scope of existing synthesis tools, a relevant comparison is not possible. However, we used the tool UPPAAL-TIGA, which supports timed games with partial observability and fixed observations, on the problem instances constructed as explained above. For the Box Painting Production System we used observation $y \in [0, 1)$ and for the Timed Game For Sorting Bricks we used observation $[0, 0.5)$ (given as $y \in [0, 1)$ by scaling accordingly). One can see in Table 1 that, although on the small instances the running times are better compared to our approach, on instances where fine granularity is needed, our approach *synthesizes* good observations considerably faster than it takes UPPAAL-TIGA to solve the game with given fixed granularity.

Conclusions. We presented a method to automatically compute observation predicates for timed controllers with safety objectives for partially observable plants. Our approach is based on the CEGAR-paradigm and can be naturally integrated into the CEGAR-loop for games under incomplete information. The observation refinement procedure could be beneficial to methods for solving timed games with fixed observations that are not necessarily CEGAR-based. The bottleneck in such approaches is the enumeration of granularities, which leads to a dramatic increase in the number of state-sets, that need to be explored, and the size of the resulting strategies. As we demonstrated, in some cases, when a reasonable number of action points suffices for controllability, our approach can be extremely successful. This opens up a promising opportunity for synergies between the CEGAR-paradigm and specialized techniques for timed systems.

Acknowledgement. This work is partially supported by the German Research Foundation (DFG) as part of SFB/TR 14 AVACS and by a Microsoft Research PhD Scholarship.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed Control with Partial Observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
3. Cassez, F.: Efficient On-the-Fly Algorithms for Partially Observable Timed Games. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 5–24. Springer, Heidelberg (2007)
4. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed Control with Observation Based and Stuttering Invariant Strategies. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 192–206. Springer, Heidelberg (2007)
5. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for Omega-Regular Games with Imperfect Information'. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
6. de Alfaro, L., Roy, P.: Solving Games Via Three-Valued Abstraction Refinement. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 74–89. Springer, Heidelberg (2007)
7. De Wulf, M., Doyen, L., Raskin, J.-F.: A Lattice Theory for Solving Games of Imperfect Information. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 153–168. Springer, Heidelberg (2006)
8. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In: Proc. FSTTCS 2008. Dagstuhl Seminar Proceedings, vol. 08004 (2008)
9. Finkbeiner, B., Peter, H.-J.: Template-Based Controller Synthesis for Timed Systems. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 392–406. Springer, Heidelberg (2012)
10. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-Guided Control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003)
11. Reif, J.H.: The complexity of two-player games of incomplete information. J. Comput. Syst. Sci. 29(2), 274–301 (1984)

Confidence Bounds for Statistical Model Checking of Probabilistic Hybrid Systems

Christian Ellen[1], Sebastian Gerwinn[1], and Martin Fränzle[1,2]

[1] OFFIS, R&D Division Transportation, Escherweg 2 - 26121 Oldenburg - Germany
{ellen,gerwinn}@offis.de
[2] Carl von Ossietzky Universität, 26111 Oldenburg - Germany
fraenzle@informatik.uni-oldenburg.de

Abstract. Model checking of technical systems is a common and demanding task. The behavior of such systems can often be characterized using hybrid automata, leading to verification problems within the first-order logic over the reals. The applicability of logic-based formalisms to a wider range of systems has recently been increased by introducing quantifiers into satisfiability modulo theory (SMT) approaches to solve such problems, especially randomized quantifiers, resulting in stochastic satisfiability modulo theory (SSMT). These quantifiers combine non-determinism and stochasticity, thereby allowing to represent models such as Markov decision processes. While algorithms for exact model checking in this setting exist, their scalability is limited due to the computational complexity which increases with the number of quantified variables. Additionally, these methods become infeasible if the domain of the quantified variables, randomized variables in particular, becomes too large or even infinite. In this paper, we present an approximation algorithm based on confidence intervals obtained from sampling which allow for an explicit trade-off between accuracy and computational effort. Although the algorithm gives only approximate results in terms of confidence intervals, it is still guaranteed to converge to the exact solution. To further increase the performance of the algorithm, we adopt search strategies based on the upper bound confidence algorithm UCB originally used to solve a similar problem, the multi-armed bandit. Preliminary results show that the proposed algorithm can improve the performance in comparison to existing SSMT solvers, especially in the presence of many randomized quantified variables.

1 Introduction

In safety analysis, one is often interested in guaranteeing certain behavioral properties of complex systems. Such systems are usually described using hybrid automata, which are capable of expressing the continuous dynamics of an environment using differential equations, together with discrete/continuous controllers. As the exact dynamics may not be known, these hybrid automata can contain non-deterministic choices, which have to be resolved. Additionally, due to environmental influences or failure probabilities, the system is likely to be exposed to stochastic type of non-determinism leading to probabilistic hybrid systems[1]. Safety properties for these systems can be formalized as reachability problems, that is, unsafe sets of states must not be reached. For the corresponding verification, a common technique is to use bounded model checking [1,2], which evolves the dynamics of the system up to a given number of transitions and checks the reachability for this unrolled depth. As transitions usually reflect only the switching decision, the reachability problem still has to respect the continuous dynamics of the system. The formalism of Satisfiability Modulo Theories (SMT) together with the corresponding solvers can in turn be used to solve this remaining satisfiability problem.

Recent developments with Conflict-Driven Clause Learning (CDCL) solvers enabled the analysis of large hybrid systems by learning conflicts which provide information from one path about a set of other paths. Formally, these hybrid systems can be modeled and analyzed using a combination of SMT and bounded model checking [3,4]. Analogous to Markov decision processes, it still remains to decide which transition path to evaluate in the presence of probabilism and non-determinism. To this end the formalism of SMT can be extended to Stochastic Satisfiability Modulo Theory (SSMT) which contains quantifiers that allow the encoding of different type of transitions [5]. Such SSMT solvers iterate the tree of possible assignments to the quantified variables and solve the resulting SMT problems at the leaves. The exponential size of this decision tree is one of the main problems for SSMT solvers. Therefore, methods are needed which search the tree efficiently and do not expand the tree completely.

Mathematically, the probability of satisfaction can also be formulated as a nested optimization. In this formulation, the problem of one existential followed by a randomized quantifier is known as the multi-armed bandit problem [6], where bandits are choices of the existential quantifier and expected rewards are the averages from the randomized quantifiers. For this problem, there exists a number of algorithms, most notably the Upper Bound Confidence algorithm (UCB), which has been proven to solve the multi-armed bandit problem in a minimax-optimal way [7].

In this paper, we present an algorithm which combines a sampling based approach for the generation of confidence intervals with the exploitation-scheme of the UCB-algorithm. The resulting algorithm allows for an explicit trade-off between accuracy (desired confidence level and precision in terms of the width of the confidence interval) and efficiency (solving of SMT formulas at the leaves of a decision tree). Additionally, due to the sampling of random variables, the algorithm is also applicable for randomized quantifiers with large or even infinite domains.

This paper is organized as follows. In Section 2 we give a short introduction to the SSMT-formalism and briefly review related work on SSMT-solvers as well as the relevant statistics literature followed by the description of the proposed algorithm including the bound propagation and the selection rules. In Section 3

[1] As an illustrative example for such a system, we use a simple cooling controller throughout this paper, see Figure 1.

we evaluate the algorithm on randomly generated SSMT formulas and on the example hybrid system in Figure 1. We conclude and give an outlook to future work in Section 4.

2 Methods

As mentioned previously, we are interested in satisfiability problems concerning probabilistic hybrid systems. which we illustrate with a simple example in Figure 1. For this particular system, we might be interested in guaranteeing that the probability of overheating is lower than a given threshold. To this end, we use satisfiability modulo theory (SMT) to formalize the satisfiability problem, given a particular path of transitions/decisions $(x_1, \ldots, x_n)$ obtained by unrolling the dynamics for a given number n of transitions. Note that this does not necessarily imply that the time within a state is fixed. For the example in Figure 1, transitions are only possible between `cooling` and `not cooling`. We denote the SMT formula which indicates the satisfiability for the given transition by $\phi(x_1, \ldots, x_n)$. The main task for the SMT solver is then to determine the existence of a satisfying variable assignment for a given SMT formula $\phi(x_1, \ldots, x_n)$. In the cooling example, this corresponds to an assignment of a temperature trajectory, given a state and a starting temperature.

SSMT is an extension of (SMT) [4] consisting of a decision problem for first-order logical formulas over a given background theory (*e.g.,* the arithmetic theories over real numbers, including multiplication). Specifically, an SSMT formula Φ extends an SMT formula ϕ by adding a prefix of quantified variables $Q_1 X_i, ..., Q_n X_n$. Every quantifier Q_i of the prefix binds one variable X_i of ϕ and

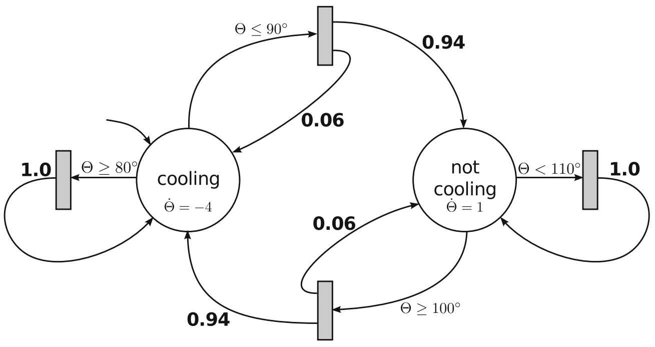


Fig. 1. Example of a probabilistic hybrid system modeling a simple cooling system. A cooling device can either be in the state `cooling` or `not cooling`. Withing the cooling state, the temperature Θ is decreased constantly whereas in the `not cooling` state, the temperature rises. Bold numbers at the edges reflect transition probabilities for the given probabilistic transitions, which can be activated once the guards (inequalities) hold. For this system, we might be interested in the probability of overheating (*e.g.,* $\Theta > 115°$) within a given time-frame.

is either randomized ($\rotatebox[origin=c]{180}{R}_{X_i}$), existential ($\exists_{X_i}$), or universal ($\forall_{X_i}$). Every quantified variable has a finite domain: $\mathcal{X}_i$. In the randomized case, every value $x_j \in \mathcal{X}_i$ is associated with a probability $P(X_i = x_j)$, modeling the likelihood that the corresponding transition is chosen. The other quantifier types model different ways to resolve non-determinism: $\exists$ by maximizing the probability of satisfying the remaining formula over all domain values and analogously $\forall$ by minimizing the probability (see definition 1).

Definition 1. *The semantics on an SSMT formula Φ are defined recursively using Q for the remainder for the quantifier prefix (cf. [5]):*

$$\begin{array}{rlcl}
1. & P(\quad \epsilon : \phi \;) & = & 0 \text{ if } \phi \text{ is unsatisfiable.}\\
2. & P(\quad \epsilon : \phi \;) & = & 1 \text{ if } \phi \text{ is satisfiable.}\\
3. & P(\;\exists_{X_i} Q : \phi \;) & = & \max_{x \in \mathcal{X}_i} P\left(Q : \phi[X_i = x]\right).\\
4. & P(\;\forall_{X_i} Q : \phi \;) & = & \min_{x \in \mathcal{X}_i} P\left(Q : \phi[X_i = x]\right).\\
5. & P(\;\rotatebox[origin=c]{180}{R}_{X_i} Q : \phi \;) & = & \sum_{x \in \mathcal{X}_i} P(X_i = x | X_{\setminus i}) \cdot P\left(Q : \phi[X_i = x]\right).
\end{array}$$

Here, we used the shorthand notation $X_{\setminus i}$ for all variables except the i-th one. From definition 1, we see that the satisfiability problem can be written as a nested optimization-expectation problem. For example, if Φ consists of one existential quantifier followed by a randomized one, the satisfiability problem can be written as follows:

$$P(\exists_x \rotatebox[origin=c]{180}{R}_y \phi(x,y)) = \max_{x \in \mathcal{X}} \mathbb{E}_{y|x}\left[P(\phi(x,y))\right] = \max_{x \in \mathcal{X}} \left(\sum_{y \in \mathcal{Y}} P(\phi(x,y)) P(y|x) \right) \quad (1)$$

For the cooling example, this would correspond to first selecting a transition from a cooling state to one of the randomized states (rectangles in Figure 1) and then choosing a transition at random, according to the probabilities (bold numbers). The existential quantifier corresponds to a pessimistic choice of transitions in terms of consequences with respect to overheating. Note that in equation (1), we used $P(y|x)$ to indicate that the probability distribution associated with the random variable Y potentially depends on the value of other variables within the prefix. Although it is straightforward to extend the formalism developed in this paper to this general case, we assume independence of the different variables to simplify the notation.

Equation (1) suggests we should approximate the expectation via a sampling based scheme, if the set $\mathcal{Y}$ is too large. If we use the average of samples generated according to $P(y|x)$, we observe only a noisy estimate $\hat{\mathbb{E}}_{y|x}$ of the true underlying function $\mathbb{E}_{y|x}$. By using confidence intervals for this estimation, we obtain an uncertainty estimate for the expectation, *i.e.,* randomized quantifier which has then to be propagated through other quantifiers to obtain an overall uncertainty estimate on Φ. Computationally, we are interested in an efficient way of calculating equation (1), that is to efficiently search for promising x to evaluate.

A common representation of an SSMT formula uses a decision tree for the variables in the quantifier prefix, where the nodes are the variables (in order of their occurrence in the prefix) and the decisions are the domain values. The leaves of the tree are replications of ϕ, where every quantified variable is substituted according to the path in the decision tree. Therefore, every leaf represents an SMT problem of its own. As solving these SMT problems at the leaves of the decision tree is a time-consuming problem, most existing work focuses on minimizing the number of evaluations of the leaves by carrying information from one leaf-evaluation to another by using conflict learning.

In the following, we aim at calculating confidence bounds l, u, such that the following holds:

$$P(l \leq P(\Phi) \leq u) \geq 1 - \alpha \tag{2}$$

Where α is a given confidence level, which specifies the quality of the calculated bounds. Here, the outer probability originates from random samples generated during the execution of the algorithm and play the role of classical confidence intervals, while the inner probability is the quantity we wish to estimate, namely the probability of satisfaction.

2.1 Related Work

In the following, we briefly review existing work on SSMT solving on the one hand and on the corresponding statistics literature on the other hand.

SSMT Solving. Based on the iSAT[8] algorithm for SMT problems, an algorithm called SiSAT [5] has been developed to solve SSMT problems efficiently. It implements a fully symbolic solving procedure based on the traversal of the prefix tree, using extended CDCL procedures and pruning rules. The computed probability of satisfaction comes with absolute certainty, that is SiSAT terminates, if the probability is guaranteed to be larger than a given threshold θ or it has been computed exactly. The pruning rules allow SiSAT to ignore parts of the quantifier tree if the outcome of the decisions could be inferred or has no impact on the result (*e.g.,* if the target threshold has already been exceeded or cannot be reached anymore). Otherwise, the algorithm has to perform an exhaustive search over the state space. Due to this exhaustive search, the number of leaves in the tree – and hence the number of SMT problems to solve – depends exponentially on the number of quantified variables in the prefix. Although SiSAT has to examine exponentially many leaves in the worst case, the memory usage is still limited, as the tree is searched in a depth-first manner.

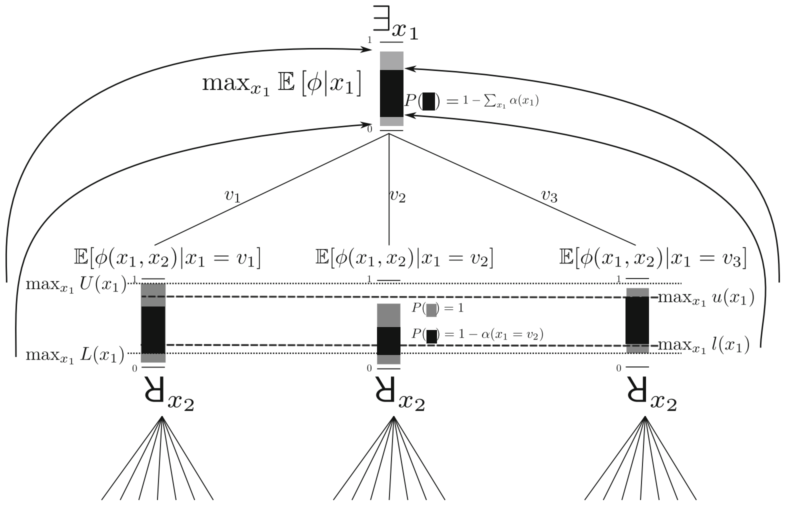
Statistical Model Checking. Inspired from classical hypothesis testing based on a set of data-samples, statistical model checking uses generated traces of the system under investigation to estimate if a given property holds. This can be done, if the system is given only in terms of a trace-generator [9,10,11], or if more structure of the generative model is known (*e.g.,* a continuous time Markov chain [12]). Also, additional information, for example in terms of associated costs, can be incorporated [13,14]. Instead of hypothesis testing, one can also impose a prior distribution on the probability of satisfaction and update a Bayesian believe sequentially as new samples are drawn from the generative model [15]. However, all these approaches are only applicable, if no decisions are needed to resolve other than random non-determinism.

Stochastic Optimization. Problems in the form of equation (1) have been studied extensively in the statistics literature. This special case of an SSMT problem is known as the multi-armed bandit problem [6]. Algorithms which solve this problem approximately have been presented in [7] and also continuous extensions thereof [16]. The idea is to use the stochastic information obtained from a sample $\phi(y,x) \sim \phi(y,x)p(y|x)$ (or any noisy measurement of the expectation) to favor those points x which are more likely to attain the maximal expectation. Due to the sample-based nature of such information, one cannot guarantee hard bounds for the probability of satisfaction. However, even in the finite sample-size regime soft-bounds in terms of confidence intervals are available that are based on Hoeffding's inequality [17]. Extensions to problems similar to nested quantifiers are available for Markov Decision Processes, most notably the upper bound confidence algorithm for trees UCT [18,19]. In the UCT case, the search tree is sequentially expanded if needed. In this roll-out based technique, an estimate of the probability of satisfaction is needed, when a path is not completely traversed to the leaves and hence only parts of the variables are assigned specific values. Additionally, certain drift conditions have to be guaranteed and even if they hold, only asymptotic bounds for the root node are available.

2.2 Bound Propagation

In this section, we describe how the bounds for the full SSMT formula can be obtained from bounds for the intermediate values at the leaves of the decision tree. First, we show how to obtain confidence bounds for the probability of satisfaction at the root node using bound propagation, and then present search strategies for an efficient way to decrease these bounds.

As mentioned previously, we are interested in calculating bounds on the probability of satisfaction. To obtain these bounds (equation (2)), we propagate bounds together with their confidence level from the leaves of the decision tree up to the root (see Figure 2). As the bounds together with confidence levels are propagated from leaves to the root, at each point during the computation, we need bounds as well as confidence levels at the leaves. Without loss of generality, the last quantifier in the decision tree is a randomized quantifier, otherwise we can shift all existential and universal quantifiers into the formula at the leaves as the probability of satisfaction is either 0 or 1. However, in this case we use the SiSAT algorithm to compute these remaining SSMT problems. Therefore, we can obtain the bounds and confidence levels for the leaves of the decision tree by drawing samples from the probability distribution associated with the



$\exists_{x_1}$
$\max_{x_1} \mathbb{E}[\phi|x_1]$
$P(■) = 1 - \sum_{x_1} \alpha(x_1)$
v_1
v_2
v_3
$\mathbb{E}[\phi(x_1, x_2)|x_1 = v_1]$
$\mathbb{E}[\phi(x_1, x_2)|x_1 = v_2]$
$\mathbb{E}[\phi(x_1, x_2)|x_1 = v_3]$
$\max_{x_1} U(x_1)$
$\max_{x_1} L(x_1)$
$P(■) = 1$
$P(■) = 1 - \alpha(x_1 = v_2)$
$\max_{x_1} u(x_1)$
$\max_{x_1} l(x_1)$
R_{x_2}

Fig. 2. Illustration of the bound propagation through the decision tree. Confidence intervals with confidence level $\alpha = \alpha(x_1)$ are obtained at the leaves (here for the random variable x_2) via equation (3). These are propagated to the next level to derive confidence intervals with worse confidence levels $\sum \alpha$. In this illustrative example $v_i, i = 1, 2, 3$ are the possible values for the variable x_1. Additionally to the α-confidence intervals, we could also propagate guaranteed ($\alpha = 1$) confidence bounds.

randomized quantifier and apply a Hoeffding type of inequality similar to the UCT algorithm. If we assume that the domain of the randomized quantifier is finite, we could also get tighter bounds by drawing samples without replacement and using the appropriate inequality (see [20]). However, as such sampling without replacement increases memory usage for large domains, we use the version with replacement [17]. Specifically, we set for the lower l and upper u bounds at the leaves, which depend on the values set for other variables set earlier in the decision tree $x_1, \ldots, x_{k-1}$:

$$\begin{aligned} l(x_{1:k-1}) := \hat{P}(\Phi|x_{1:k-1}) - w, \qquad u(x_{1:k-1}) &:= \hat{P}(\Phi|x_{1:k-1}) + w \\ \hat{P}(\Phi|x_{1:k-1}) &= \frac{1}{m} \sum_{x^i \sim P_k} P\left(\phi(x_1, \ldots, x_{k-1}, x_k = x^i)\right) \\ P(\Phi|x_{1:k-1}) &= \sum_{x_k \in \mathcal{X}_k} P_k(x_k) P\left(\phi(x_1, \ldots, x_{k-1}, x_k)\right), \end{aligned} \tag{3}$$

where w specifies the uncertainty in the estimate $\hat{P}$ of the true (unknown) probability of satisfaction P. m is the number of samples x^i we have drawn from the probability distribution P_k of the random variable X_k. Here, we have written $P_k(x_k)$ as a shorthand for $P(X_k = x_k|X_1 = x_1, \ldots, X_{k-1} = x_{k-1})$. Using these setting and applying Hoeffding's inequality, we have for the confidence level of the specified width:

$$
\begin{aligned}
P\left(l(x_{1:k-1}) \leq P(\varPhi|x_{1:k-1})\right) \geq 1 - e^{\left(-2mw^2\right)} := 1 - \frac{1}{2}\alpha(x_1, \ldots, x_k) \\
P\left(P(\varPhi|x_{1:k-1}) \leq u(x_{1:k-1})\right) \geq 1 - e^{\left(-2mw^2\right)} := 1 - \frac{1}{2}\alpha(x_1, \ldots, x_k)
\end{aligned}
\tag{4}
$$

where α is the confidence level, which depends on the number of samples and the width w. The bound in equation (4) makes a statement about the confidence that we obtain as a consequence of the random sampling. As mentioned previously, we could, obtain a "hard bound" (that holds with probability 1) on the probability of satisfaction if we additionally make use of the explicitly known structure of the randomized quantifier in terms of the associated probability distribution. The confidence bounds apply for the random sampling at the leaves. However, we are interested in calculating bounds for the root of the decision tree. To this end, we state the following propagation rules from level k to the above level $k-1$:

$$
b(x_1, x_2, \ldots, x_{k-1}) = \begin{cases} \max_{x_k \in \mathcal{X}_k} b(x_1, \ldots, x_{k-1}, x_k) & \text{if } Q_k = \exists_{X_k} \\ \min_{x_k \in X_k} b(x_1, \ldots, x_{k-1}, x_k) & \text{if } Q_k = \forall_{X_k} \\ \sum_{x_k \in X_k} P(x_k) b(x_1, \ldots, x_{k-1}, x_k) & \text{if } Q_k = \reflectbox{R}_{X_k} \end{cases} \tag{5}
$$

$$
\alpha(x_1, \ldots, x_{k-1}) = \sum_{x_k \in \mathcal{X}_k} \alpha(x_1, \ldots, x_{k-1}, x_k) \tag{6}
$$

Here, b are the bounds, which are function of the decision variables and could be either lower or upper bounds that we would like to propagate to the preceding level.

This process of propagating intervals is illustrated in Figure 2. In the case of Figure 2 we have two quantifiers, an existential and a randomized one. Given that we have drawn samples for the randomized quantifier, we can calculate confidence intervals via equation (3),(4) for a given confidence level.We can then combine these confidence intervals to obtain confidence intervals for the value at the existential quantifier using equation (5). Note that the confidence level for the "soft" bounds is worse than those at the lower level. In fact, if we want to reach a desired confidence level at the root of the decision tree, *e.g.,* 95%, we have to impose a much higher confidence level at the leaves. For example, in the case of Figure 2, we would have to impose $\alpha(x_1) = \frac{0.05}{3}$ to get the desired 95% confidence interval at the top.

We now show that the bounds calculated in equation (5) indeed hold for the confidence level in equation (6). We start with the bounds in the case of an existential quantifier. In this case, we can simplify the notation, by ignoring variables that are fixed for the propagation and therefore writing $u(x), l(x)$ for the upper and lower confidence bounds for the level that we would like propagate and $\alpha(x)$ for the corresponding confidence level at the lower level. Similarly, we write $\varPhi(x)$ for the true (unknown) values at the leaves of the existential quantifier. With this notation, we have:

Lemma 1. *Let the lower and upper confidence bound hold for each child x of an existential quantifier,* i.e., $P\left(\Phi(x) > u(x)\right) \leq \frac{1}{2}\alpha(x) \forall x$. *Then the following bound hold for the existential quantifier node:*

$$P\left(\max_x l(x) \leq \max_x \Phi(x) \leq \max_x u(x)\right) \geq 1 - \sum_x \alpha(x)$$

Proof. From the lower level, we know $P\left(\Phi(x) > u(x)\right) \leq \frac{1}{2}\alpha(x)$ for each x individually. Applying a simple union bound (cf. [21]), we find:

$$\begin{aligned} P\left(\max_x \Phi(x) > \max_x u(x)\right) &\leq P\left(\exists_x : \Phi(x) > u(x)\right) \\ &= \sum_x \underbrace{P\left(\Phi(x) > u(x)\right)}_{\leq \frac{1}{2}\alpha(x)} - \underbrace{P\left(\forall_x : \Phi(x) > u(x)\right)}_{\geq 0} \\ &\leq \frac{1}{2}\sum_x \alpha(x) \end{aligned} \tag{7}$$

Similarly, we get $P\left(\max_x \Phi(x) < \max_x l(x)\right) \leq \frac{1}{2}\sum_x \alpha(x)$. Combining upper and lower bound completes the proof. □

By maximizing the negative value instead of the minimization for the universal quantifier and inverting the role of upper and lower bound, the same argument can be made for the case of an universal quantifier. For the case of a randomized quantifier, we have:

Lemma 2. *Using the same notation as in Lemma 1, the following bound holds.*

$$P\left(\sum_x P(x)l(x) \leq \sum_x P(x)\Phi(x) \leq \sum_x P(x)u(x)\right) \geq 1 - \sum_x \alpha(x)$$

Proof. As $\forall_x : \Phi(x) \leq u(x) \Rightarrow \sum_x P(x)\Phi(x) \leq \sum_x P(x)u(x)$ $(*)$:

$$\begin{aligned} P\left(\sum_x P(x)\Phi(x) > \sum_x P(x)u(x)\right) &= 1 - P\left(\sum_x P(x)\Phi(x) \leq \sum_x P(x)u(x)\right) \\ &\overset{*}{\leq} 1 - P\left(\forall_x : \Phi(x) \leq u(x)\right) = P\left(\exists_x : \Phi(x) > u(x)\right) \overset{(7)}{\leq} \frac{1}{2}\sum_x \alpha(x) \end{aligned}$$

The lower bounds follow analogously. □

2.3 Search Strategies

So far, we have not considered any search strategies for the optimized selection of children to expand at a quantifier node in order to increase the confidence at the root of the decision tree. By selecting the paths to evaluate in a strategic manner, we aim for gaining as much information as possible about the probability of satisfaction of the SSMT-formula at hand. To this end, we propose several simple search strategies. As an existential quantifier corresponds to a maximum operation (see definition 1), we select the child with the maximal upper confidence bound for exploration. Analogously, we select the child with the minimal confidence bound for universal quantifiers. This strategy is known under the upper confidence bound algorithm. However, additionally to the lower and upper bound from equation (5), we can calculate the upper bound, according to the number of times, the current node x has been visited $n(x)$, compared to the overall number of samples within the current quantifier n analogously to the UCT-algorithm:

$$\mu(x)\pm = \sqrt{\frac{2\log(n)}{n(x)}}, \qquad \mu(x) := l(x) + \frac{u(x) - l(x)}{2}. \tag{8}$$

Note that using the bounds from equation (5) typically leads to smaller bounds compared to the UCB, as it uses more information on the structure below the node. For leaf-nodes, however, they are identical. For a randomized quantifier, we propose the following two strategies:

1. Sample a child to explore according to the associated probability distribution
2. Construct a probability distribution by weighting the probability of each child with the width of its confidence interval

By using combinations of these selection rules, we obtain 4 different selection rules in total.

2.4 Algorithmic Description

Although we have presented the basic components – initial bounds, bound propagation, and search strategies – of our proposed algorithm in the previous sections, we present a more detailed algorithmic description in this section. In particular, this includes how deductions obtained from the solver at the leaves of the decision tree can be used for pruning and incorporated into the calculation of the confidence bounds. We explain the three different phases (see Algorithm 1: selection phase, sampling phase, and propagation phase) in more detail in the following. First, we partition the decision tree into a three parts: A trailing part containing all trailing non-randomized quantifiers, a (non-empty) set of randomized quantifiers, and a leading part which may contain any quantifier. To lighten the description, we collapse the randomized quantifiers in the second part to a single randomized quantifier with more leaves and expand the SMT formula by adding the trailing non-randomized quantifiers to the formula. Hence, we can assume for the following a decision tree/SSMT formula in which the last quantifier is a randomized one. Additionally, to obtain the desired confidence level at the root of the decision tree, we calculate the necessary α for the randomized part by counting the number of leaves L of the first part of the decision tree:

$$\alpha(x_1, \ldots, x_n) = 1 - \frac{1}{L}(1 - \alpha)$$

Algorithm 1. Confidence Bound Generation and Propagation

```
Data: Quantifier Prefix $Q_1, \ldots, Q_n$,
SMT Forumla $\phi(x_1, \ldots, x_n)$,
confidence level $\alpha$
Result: Confidence interval $[\underline{p}, \overline{p}]$ for satisfiability of: $Q_1, \ldots, Q_n \phi(x_1, \ldots x_n)$
        with given confidence
while terminal condition not reached do
    for $i \leftarrow 1$ to $n - 1$ do ;                                // Selection phase
        Select $x_i$ for Variable $X_i | x_1, \ldots, x_{i-1}$;
    end
    for $k \leftarrow 1$ to $m$ do ;                  // Sampling phase: draw $m$ samples
        sample $x_n^k \sim P_n(X_n)$;
        sat$_k \ni \{0, 1\}$= solveSMT($\phi(X_1 = x_1, \ldots, X_n = x_n^k)$)
    end
    update bounds for $u(x_1, \ldots, x_{n-1})$ using eq. (3);
    for $i \leftarrow n - 1$ to 1 do ;                              // Propagation phase
        collect $l(x_1, \ldots, x_i, X_{i+1} = x), u(x_1, \ldots, x_i, X_{i+1} = x)$ for all $x$;
        use deductions to collapse some of these bounds to 0;
        use equation (5) to calculate $l(x_1, \ldots, x_i), u(x_1, \ldots, x_i)$;
    end
end
```

Selection Phase. Within the selection phase, the quantifiers of first part are traversed and a new evaluation is selected based on the type of quantifier and the valuations so far:

- $\exists, \forall$ These quantifiers are decided by selecting the value with the largest or smallest confidence bounds for existential or universal quantifiers respectively. Either the confidence bounds are used directly or they are computed based on the UCB-rule, see equation (8). Note that for some of the variables the bounds might be collapsed due to deduction from DPLL solvers for the SMT formula. The corresponding values will be ignored for the selection rule.
- Ɐ As mentioned in section 2.3, there are two available selection rules for randomized quantifiers. To construct a weighted version of the probability distribution associated with the randomized variable X_k, we use the following:

$$P_s(x_k) = \frac{P_w(x_k)}{\sum_x P_w(x)}, \qquad P_w(x_k) = (u(x_k) - l(x_k))\, P(x_k) \tag{9}$$

Sampling Phase. In this phase, we generate m samples for last randomized quantifier. For each generated sample, we solve the remaining SMT formula and use the resulting data to calculate initial bounds or update the confidence bounds, if the current node as already been sampled in the past and has been selected for further evaluation.

Propagation Phase. After updated confidence bounds for the current path have been generated in the selection phase, the new information needs to be propagated to the root node. The updated bounds are propagated according to equation (5) and the number of visits for each existential and universal quantifier is increased by one. Additional to the evidence from the generated samples, DPLL solvers provide extra information in terms of deductions indicating subtrees which are guaranteed to evaluate to *false*. This supplemental information can be incorporated in the propagation of confidence bounds by collapsing the corresponding bound of the subtree to 0. As bound updates only occur on the current path, the influence of a deduction is only taken into account for this path. The effects of deductions for other subtrees not yet sampled will only be calculated once this subtree is visited.

Terminal Condition. As the algorithm never reaches a point interval with 100% confidence, we have to decide when to abort the sampling procedure. By adjusting combinations of confidence level and desired width of the computed confidence interval, we can set an explicit trade-off between accuracy (width of the interval together with desired confidence) and speed (number of SMT-evaluations). Sometimes we might be interested in verifying that the probability of satisfaction is larger, or lower than a certain threshold, independently of the width of the confidence interval. Therefore, we use the following two possibilities: Terminate if either the confidence interval for the full formula has reached a given width, or if the lower (upper) confidence bound has crossed a predefined threshold.

3 Evaluation

The computational complexity of Monte-Carlo sampling based methods usually do not scale with the dimensionality of the probability distribution at hand, but scale only with the number of samples (see equation (3)). Therefore, we expect the bound propagation to work best when the number of randomized quantified variables for the last part is large, as we obtain the initial confidence bound within this part. Additionally, as we do not use the full information available via deductions of the SMT-solver at the leaves of the decision tree, we expect existing solvers like SiSAT [5] to outperform the bound propagation algorithm in cases of only small proportions of randomized quantified variables. To test these hypotheses, we first evaluated the bound propagation on scenarios of randomly generated SMT formulas. Specifically, we used the same generation mechanism as in [22] called `makewff` to generate random formulas with 24 variables, 20 clauses and 3 variables per clause. By using these settings, we generated formulas which are likely to be satisfiable for some yet not all assignments. For the quantifier prefix, we tested two different settings, one with a large proportion of randomized quantifiers and one with a small proportion. For the first setting, we constructed a quantifier prefix with a high fraction of randomized quantifiers consisting of twice an $\exists - ꓤ$ pair followed by 21 randomized quantifiers. The second setting is particularly disadvantageous for the bound propagation and consisted eight $(\exists - \exists - ꓤ)$ triplets. For each of these

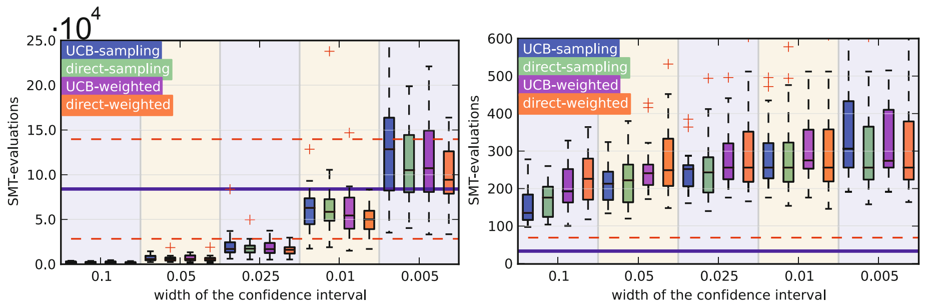


Fig. 3. Performance comparison as a function of the width of the confidence interval with fixed confidence level of 95%. **Left**: Results for the setting with high fraction of randomized quantifiers. **Right**: Disadvantageous setting with more existential quantifiers. The blue line shows the number of evaluations for the SiSAT algorithm (averaged across 16 randomly generated formulas) and in red the standard deviation over these repetitions. For each width we compared different selection rules, UCB stands for an UCB selection rule for the existential quantifier whereas 'direct' indicates a confidence bound selection rule on the confidence bounds according to equation (5). 'weighted' indicates that the probability distribution is weighed with the width of the confidence interval before a sample is drawn.

settings, we compared the SiSAT algorithm and the bound propagation in terms of the number of evaluations of the leaves, *i.e.,* number of solver calls averaged across 16 repetitions (as both the result as well the formula depend on stochastic quantities). For a fixed confidence level, the number of leaf-evaluations depends mainly on the width of the confidence interval used as termination condition. Therefore, we show the performance as a function of the width of the confidence interval in Figure 3. In the left panel, the results are shown for the advantageous setting of large proportions of randomized quantifiers, whereas in the right panel the disadvantageous setting is used. For the second setting (right panel) the probability of satisfaction can be inferred by only a few SMT-evaluations, as a large proportion of the tree can be pruned due to deductions made by the SMT solver. In fact, due to the maximum operation of the existential quantifier, only a few paths contribute to the root value, *i.e.,* it is sufficient to show the satisfiability of these few paths. As the bound propagation algorithm still samples some paths for the trailing randomized quantifier, it needs more SMT evaluations than the SiSAT algorithm. As the number of randomized variables is small, the sampling procedure is likely to sample the same valuation multiple times. These samples can be cached and hence no SMT solver needs to be evaluated. Therefore, we report only the number of actual SMT evaluations neglecting the evaluations, that can be obtained from the cache. For the first setting (left panel), both SiSAT and the four statistical variants need much more evaluations compared to the setting in the right panel, as in this case nearly all paths contribute to the root value. For this setting with a large proportion of randomized quantifiers, the bound propagation decreases the number of SMT evaluations tremendously compared to the SiSAT algorithm. If we fix, however, the width of the confidence

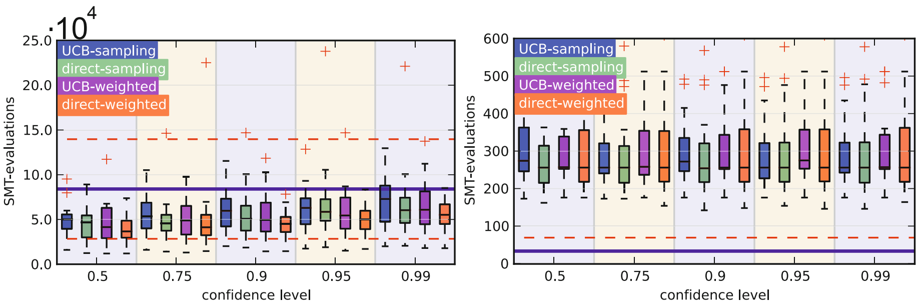


Fig. 4. Performance comparison as a function of the confidence level chosen. For all experiments the number of evaluations is counted until the confidence interval reached a width of ≤ 0.01. Left and right panels show the results for the different settings, analogous to Figure 3.

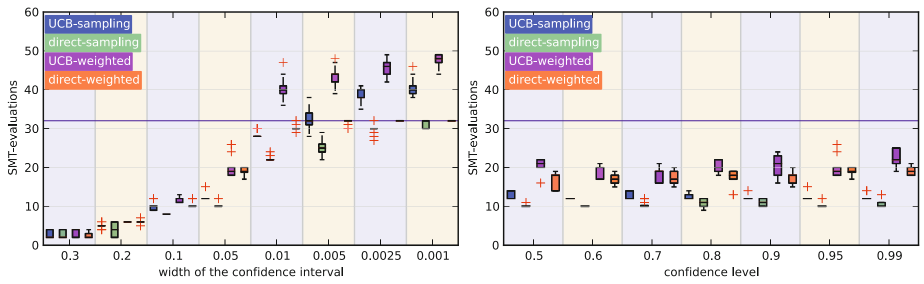


Fig. 5. Performance for the illustrative example of Figure 1. **Left**: Number of SMT evaluations with fixed level of confidence (95%) used as terminal condition. **Right**: Performance of the different search strategies as a function of the confidence level with fixed width of the confidence interval (0.05). For each combination of existential and randomized selection rule the statistics over 100 repetitions are plotted.

interval, the number of evaluations depends on the confidence level, see Figure 4. Analogous to Figure 3, the left and right panel show the results for the different proportions of the type of quantifiers. For this setting the same overall observation holds that the bound propagation gives a superior performance for the first set of quantifiers order (left panel). However, we note, that the width of the confidence interval has a much bigger impact on the number of SMT evaluations than the confidence level. Finally, we performed the same type of analysis for the illustrative example shown in Figure 1, the results for which can be found in Figure 5. The difficulty in this scenario can be increased by increasing the bounding depth in the BMC problem, *i.e.,* number of transition. The results plotted in Figure 5 are obtained by using a bounding depth of 8 steps. For this example, a similar behavior of the different algorithms can be observed. However, for this setting, much more information can be obtained through deductions, as can be seen by the small number of SMT evaluations needed to achieve the goal of the combination of interval width with a given confidence level. In the right panel of Figure 5, we see that the number of SAT evaluations does not vary with the chosen confidence level. The actual confidence level is 100% due to deductions available during the processing, hence once the width of the confidence interval is below the chosen threshold, no more samples are needed to achieve the confidence level. Additionally, it can be observed that the direct confidence bound selection rule performs better than the UCB strategy, although the difference is not as big as in the previous setting.

4 Conclusion

We have presented an SSMT-solving algorithm based on statistical methods used for solving multi-armed bandit problems. As the algorithm allows to specify the desired accuracy in terms of confidence width and confidence level, the trade-off between accuracy and computational effort can be adjusted explicitly. By using the proposed search and sampling strategy, we were able to gain efficiency for certain SSMT problems. The improvement compared to existing SSMT-solvers (SiSAT) is larger, the higher the proportion of randomized quantifiers is within the tree. Indeed, the presented sampling based technique can also be extended to handle continuous valued random variables and thereby extends the model class which can be analyzed to hybrid systems including stochastic differential equations. Importantly, we can also make use of the pruning procedures, which are typically used in CDCL-based solvers. For the implementation, however, not all deductions obtained from these solvers can be exploited to prune the decision tree. Currently, we use the pruning rules to collapse the confidence intervals with confidence level α, although we could collapse the 100% confidence bounds thereby allowing to use worse confidence levels in other subtrees. Exploiting more of these pruning rules is subject to future research.

Acknowledgments. This research was supported by the European Commission under the project MoVeS, FP7-ICT-257005.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Groote, J., van Vlijmen, S., Koorn, J.: The safety guaranteeing system at station hoorn-kersenboogerd. In: Proceedings of the Tenth Annual Conference on Systems Integrity, Software Safety and Process Security, Computer Assurance, COMPASS 1995, pp. 57–68. IEEE (1995)
3. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying Industrial Hybrid Systems with MathSAT. Electronic Notes in Theoretical Computer Science 119(2), 17–32 (2005)
4. Fränzle, M., Herde, C.: HySAT: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design 30(3), 179–198 (2007)

5. Teige, T., Eggers, A., Fränzle, M.: Constraint-based analysis of concurrent probabilistic hybrid systems: An application to networked automation systems. Nonlinear Analysis: Hybrid Systems (2010)
6. Robbins, H.: Some aspects of the sequential design of experiments. Bulletin of the American Mathematical Society 58(5), 527–536 (1952)
7. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2), 235–256 (2002)
8. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1(3-4), 209–236 (2007)
9. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
10. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
11. Younes, H., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Information and Computation 204(9), 1368–1409 (2006)
12. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)
13. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
14. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical Model Checking for Networks of Priced Timed Automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
15. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 243–252. ACM, Stockholm (2010)
16. Bubeck, S., Munos, R., Stoltz, G., Szepesvári, C.: $\mathcal{X}$-armed bandits. Journal of Machine Learning Research 12, 1655–1695 (2011)
17. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, 13–30 (1963)
18. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
19. Szepesvári, C.: In: Reinforcement Learning Algorithms for MDPs. John Wiley & Sons, Inc. (2010)
20. Serfling, R.J.: Probability inequalities for the sum in sampling without replacement. The Annals of Statistics 2(1), 39–48 (1974)
21. Srinivas, N., Krause, A., Kakade, S., Seeger, M.: Gaussian process optimization in the bandit setting: No regret and experimental design. In: Fürnkranz, J., Joachims, T. (eds.) Proceedings of the 27th International Conference on Machine Learning (ICML 2010), pp. 1015–1022. Omnipress, Haifa (2010)
22. Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic boolean satisfiability. Journal of Automated Reasoning 27(3), 251–296 (2001)

Region-Based Analysis of Hybrid Petri Nets with a Single General One-Shot Transition

Hamed Ghasemieh[1], Anne Remke[1],
Boudewijn Haverkort[1,2], and Marco Gribaudo[3]

[1] Design and Analysis of Communication Systems, University of Twente, The Netherlands
{h.ghasemieh,a.k.i.remke,b.r.h.m.haverkort}@utwente.nl
[2] Embedded System Institute, Eindhoven, The Netherlands
[3] Dipartimento Di Elettronica E Informazione, Ingegneria dell'Informazione, Italy
marco.gribaudo@polimi.it

Abstract. Recently, hybrid Petri nets with a single general one-shot transition (HPnGs) have been introduced together with an algorithm to analyze their underlying state space using a conditioning/deconditioning approach. In this paper we propose a considerably more efficient algorithm for analysing HPnGs. The proposed algorithm maps the underlying state-space onto a plane for all possible firing times of the general transition s and for all possible systems times t. The key idea of the proposed method is that instead of dealing with infinitely many points in the t-s-plane, we can *partition* the state space into several regions, such that all points inside one region are associated with the same system state. To compute the probability to be in a specific system state at time τ, it suffices to find all regions intersecting the line $t = \tau$ and decondition the firing time over the intersections. This partitioning results in a considerable speed-up and provides more accurate results. A scalable case study illustrates the efficiency gain with respect to the previous algorithm.

1 Introduction

In a recent study we have evaluated the impact of system failures and repairs on the productivity of a fluid critical infrastructure, in particular, a water treatment plant [8]. In that study, we have developed an analysis algorithm for a class of Hybrid Petri nets [6] with a single general one-shot transition. Despite the current restriction to a single general one-shot transition this class turns out to be very useful for this application field. However, the algorithm proposed in [8] requires a discretization of the support of the distribution that determines the firing time of the general transition. This is on the one hand computationally very expensive for small step sizes, and on the other hand may lead to less accurate results for larger step sizes.

This paper presents a considerably more efficient algorithm that partitions the underlying state space of an HPnG into regions with equivalent markings, depending on the current time t and the firing time of the general transition s. We provide a graphical representation of these regions, a so-called Stochastic Time Diagram (STD), which consists of two main parts, namely, the deterministic and the stochastic part. In the first part, the evolution of the system and the continuous marking solely depends on t, since the general transition has not fired, yet. In the second part, however, the continuous marking and the remaining firing time of deterministic transitions may depend linearly on the values s and t. The main advantage of our new method is that for each system time, we can easily find all possible states of the system. Instead of dealing with infinitely many points in the t-s plane, for computing the probability to be in a specific system state at time τ, it suffices to find all regions intersecting the line $t = \tau$ and decondition the firing time of the general transition over the intersections that correspond to a the specific system state. The partitioning into regions with the same system state avoids accuracy problems of the old algorithm that stem from a discretization with fixed step sizes and also significantly decreases the computation time.

The idea of a partitioning the underlying state-space of hybrid systems is not new and, e.g., the underlying state-space of Timed Automata (TA) [3,2] can be partitioned into zones, where each zone represents a symbolic system state. However, due to the fact that all real-valued clocks have an identical drift of 1 the shape of a zone is much more restricted than the shape of the regions resulting for HPnGs. Similarly, a partitioning of the state-space of hybrid systems has been introduced in [1]. The difference to our work is that we partition according to time and the support of the general transition instead of the values of the continuous variables. Also for Dense-Time Reactive Systems [12,11] that are a generalization of Timed Petri Nets [5] a partitioning of the state space into state classes has been introduced. whereas, such systems do not include continuous variables they allow to equip timed transitions with an interval indicating their firing time. Again, since time evolves linearly with derivative 1, the shape of state classes is similarly restricted as zones for TA. Dynamical Systems having Piecewise-Constant Derivatives (PCDs) [4] represent a class of hybrid systems where the evolution of the continuous variables is piecewise-linear and the control component of a state is fully determined by the values of the continuous variables. This also results in a set of regions, where each region is associated with a constant vector field which identifies the rates at which the various variables change. Similarly to HPnGs, PCDs allow different slopes for the continuous variables within one region and more general guards for discrete transitions. However, in contrast to PCDs, the discrete state of an HPnG is not fully described by the values of the continuous variables and hence allows for a more general discrete component.

This paper is further organized as follows: In Section 2 we provide a brief description of the modelling formalism and system evolution of HPnG and introduce some notation that is used throughout the paper. In Section 3, we intuitively describe the idea behind our new algorithm, and illustrate the idea with a simple example. In Section 4, we formalize the details of the algorithm, and prove that the partitioning results in polygons. Section 5 addresses the computation of measures of interests, e.g., the probability of having an empty storage at a given time. To study the efficiency of the new algorithm, Section 6 compares the run time with the existing algorithm [8] on a scalable case study.

2 Hybrid Petri Nets with General One-Shot Transitions

A HPnG is defined as a tuple $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{m}_0, \mathbf{x}_0, \Phi)$, where $\mathcal{P} = \mathcal{P}^D \cup \mathcal{P}^C$ is a set of *places* that can be divided into two disjoint sets $\mathcal{P}^D$ and $\mathcal{P}^C$ for the discrete and continuous places, respectively. The discrete marking $\mathbf{m}$ is a vector that represents the number of tokens $m_P \in \mathbf{N}$ for each discrete place $P \in \mathcal{P}^D$ and the continuous marking $\mathbf{x}$ is a vector that represents the non-negative level of fluid $x_P \in \mathbf{R}_0^+$ for each continuous place $P \in \mathcal{P}$. The initial marking is given by $(\mathbf{m}_0, \mathbf{x}_0)$.

Four types of *transitions* are possible, as follows. The set of immediate transitions, the set of deterministically timed transitions, the set of general transitions, and the set of continuous transitions together form the finite set of transitions $\mathcal{T} = \mathcal{T}^I \cup \mathcal{T}^D \cup \mathcal{T}^G \cup \mathcal{T}^C$. Note that in this paper the number of general transitions is restricted to $|\mathcal{T}^G| = 1$. Also the set of *arcs* $\mathcal{A}$ consists of four sets: The set of discrete input and output arcs $\mathcal{A}^D$, connects discrete places and discrete transitions and the set of continuous input and output arcs $\mathcal{A}^C$ connects continuous places and continuous transitions. The set of inhibitor arcs $\mathcal{A}^I$ and the set of test arcs $\mathcal{A}^T$, both connect discrete places to all kinds of transitions.

The tuple $\Phi = (\phi_b^{\mathcal{P}}, \phi_p^{\mathcal{T}}, \phi_d^{\mathcal{T}}, \phi_f^{\mathcal{T}}, \phi_g, \phi_w^{\mathcal{A}}, \phi_s^{\mathcal{A}}, \phi_p^{\mathcal{A}})$ contains 8 *functions*. Function $\phi_b^{\mathcal{P}} : \mathcal{P}^C \to \mathbf{R}^+ \cup \infty$ assigns an upper bound to each continuous place. In contrast to the definition of HPnG in [8] in the following $\phi_p^{\mathcal{T}} : \mathcal{T}^D \cup \mathcal{T}^I \to \mathbf{N}$ specifies a *unique priority* to each immediate and deterministic transition to resolve firing conflicts, as in [10]. Deterministic transitions have a constant firing time defined by $\phi_d^{\mathcal{T}} : \mathcal{T}^D \to \mathbf{R}^+$ and continuous transitions have a constant nominal flow rate defined by $\phi_f^{\mathcal{T}} : \mathcal{T}^C \to \mathbf{R}^+$. The general transition is associated with a random variable s with a cumulative probability distribution function (CDF) $\phi_g(s)$, and its probability density function (PDF) is denoted $g(s)$. We assign to all arcs except continuous arcs the weight: $\phi_w^{\mathcal{A}} : \mathcal{A} \setminus \mathcal{A}^C \to \mathbf{N}$ which defines the amount of tokens that is taken from or added to connected places upon firing of the transition.

Conflicts in the distribution of fluid occur when a continuous place reaches one of its boundaries. To prevent overflow, the fluid input has to be reduced to match the output, and to prevent underflow the fluid output has to be reduced to match the input, respectively. The firing rate of fluid transitions is then adapted according to the share $\phi_s^{\mathcal{A}} : \mathcal{A}^C \to \mathbf{R}^+$ and priority $\phi_p^{\mathcal{A}} : \mathcal{A}^C \to \mathbf{N}$ that is assigned to the continuous arcs that connect the transition to the place. This is done by distributing the available fluid over all continuous arcs. Those with highest priority are considered first and if there is enough fluid available, all transitions with the highest priority can still fire at their nominal speed. Otherwise, their fluid rates are adapted according to the firing rate of the connected transitions and the share of the arc, according to [6]. The adaptation of fluid rates in these cases, results in a piecewise constant fluid derivative per continuous place.

The *state* of an HPnG is defined by $\Sigma = (\mathbf{m}, \mathbf{x}, \mathbf{c}, \mathbf{d}, \mathcal{G})$, where vector $\mathbf{c} = (c_1, \ldots, c_{|\mathcal{T}^D|})$ contains a clock c_i for each deterministic transition that represents the time that T_i^D has been enabled. Vector $\mathbf{d} = (d_1, \ldots, d_{|\mathcal{P}^C|})$ indicates the drift, i.e., the change of fluid per time unit for each continuous place. Note that even though this vector d is determined uniquely by x and m, it is included in the definition of a state to make it more descriptive. A general transition is only allowed to fire once, hence, the flag $\mathcal{G} \in \{0, 1\}$ indicates whether the general transition has already fired. So, the initial state of the system is $\Sigma_0 = (\mathbf{m}_0, \mathbf{x}_0, \mathbf{0}, \mathbf{d}_0, 0)$. For a more detailed description HPnGs and their evolution, we refer to [9].

3 Graphical Representation of the System Evolution

The evolution of an HPnG can be represented by a so-called *Stochastic Time Diagram* (STD) that illustrates the system state at each time conditioned on the firing time of the general transition. Section 3.1 introduces STDs and shows how to generate this diagram for a simple example in Section 3.2.

3.1 Stochastic Time Diagram

Given an initial state of an HPnG and a predefined value for the firing time of the general transition (denoted s) the state of the system can be determined for all the future times t starting from a given initial state. Hence, in order to characterize the system state, we consider a two-dimensional diagram with t on the vertical axis and s on the horizontal axis. Each point in this diagram is associated with a unique system state. A generic version of this diagram is shown in Figure 1. The *stochastic* area contains all states for which we assume that the general transition has fired, i.e., the current system time is larger than the firing time of general transition, $t > s$. The *deterministic* area, in contrast, represents all states where the general transition has not fired yet, i.e., $t < s$. In this area the evolution of the system is independent of parameter s.

To compute measures of interest for HPnGs, the state space needs to be deconditioned with a probability density function $g(s)$. The main idea of the proposed method is that instead of dealing with infinitely many points in the t-s-plane, we can *partition* the state space into several regions, such that all points inside one region are associated with the same system state. More formally a *system state Γ* is defined as a set of HPnG states with the same discrete marking m, drift d and general transition flag $\mathcal{G}$, where the continuous marking and the clock values linearly depend on s and t according to the same equations. Then to compute the probability to be in a specific system state at time τ, it suffices to find all regions intersecting the line $t = \tau$ that correspond to the specific system state and integrate $g(s)$ over the intersection. This idea is illustrated for a given partitioning in Figure 2.

While the generic STD from Figure 1 holds for every HPnG, the partitioning into invariant regions as shown in Figure 2, depends on the structure of the model at hand. These invariant regions exist, because the state of the system does not change until an *event* occurs. At each system state two types of potential events should be considered: a fluid place reaching its lower/upper boundary or an enabled deterministic transition reaching its firing time. Both events induce

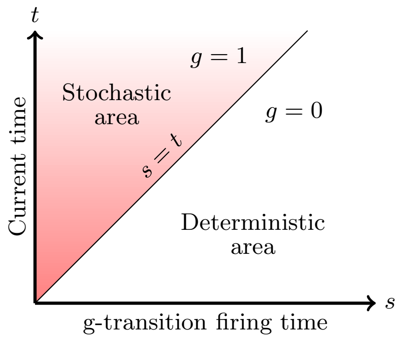


Fig. 1. Generic presentation of STD

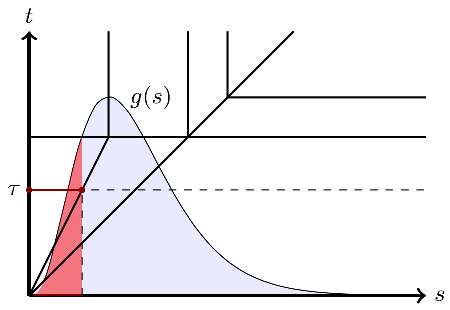


Fig. 2. Deconditioning according to the probability density function $g(s)$

a state change, i.e., the system enters another region. Therefore, the boundary between regions represents the occurrence of an event. Section 4 presents an algorithm that constructs the STD for a given HPnG and proves that for HPnGs all boundaries are linear.

3.2 Reservoir Example

In order to illustrate the above concepts, we construct the STD for an example HPnG taken from [8]. Using the same graphical representation as in [8], Figure 3 shows an HPnG model of a reservoir that is filled by a pump and drained due to some demand. The reservoir C_r can contain at most 10 units of fluid (say, m^3) and as long as the discrete places P_p contains a token and the reservoir is not full, fluid is pumped in with rate 2. As long as P_d contains a token and the reservoir is not empty, fluid is taken from the reservoir at rate 1. The demand is deterministically switched off after 5 time units by transition D_e and the pump fails according to an arbitrary probability distribution. At $t = 0$, the reservoir is empty.

Assuming that the general transition has not been fired, i.e., $s > t$, there are two possible events: either transition D_e fires at time 5 or reservoir C_r reaches its upper boundary. Since the overall rate of change (drift) of fluid into C_r is 1 in this sense, it takes 10 time units to become full. So the first occurring event is D_e, firing at time 5. This event is represented in Figure 4 by the horizontal line $t = 5$, labelled D_e. Then, since transition F_d is no longer enabled, the drift at C_r becomes 2. Since the reservoir contains 5 units of fluid, it takes 2.5 time units to reach its upper boundary, which occurs at time 7.5. In Figure 4, this is shown by line $t = 7.5$, labelled C_r. After entering the area above this line, no deterministic event is possible anymore, i.e., we reach an *absorbing* region.

After partitioning the deterministic area, the line $t = s$, is divided into three segments, as illustrated in Figure 4. Then, in order to partition the stochastic area, all of these segments have to be considered as possible firing times of the

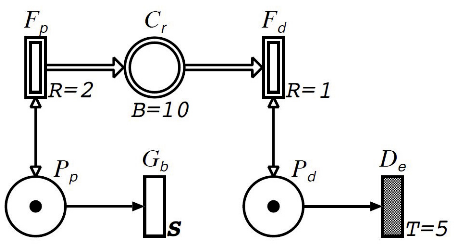


Fig. 3. Reservoir model

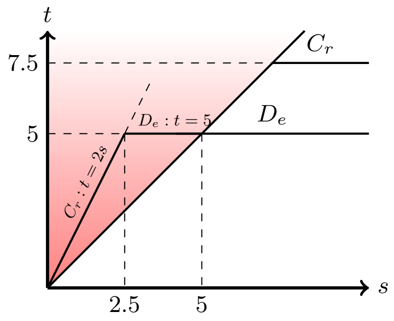


Fig. 4. Polygon over the interval $[0, 5)$

general transition. Starting from the initial state of the system, first consider that the general transition fires at time $s \in [0, 5)$. Hence, by passing the line $t = s$, the system enters the stochastic area. Then two events are possible: either D_e fires or C_r reaches its lower boundary. Before the general transition fired, place C_r had drift 1 and since s time units have passed it now contains s units of fluid. After the general transition has fired, the transition F_p is disabled, and the drift at C_r becomes -1.

Now, either the reservoir becomes empty or the deterministic transition fires, which stops the demand. To find out which of these events is going to occur first, we have to compare their occurrence time t, which may depend on s. Let Δt be the time needed for C_r to become empty, we have: $\Delta t = s$. The previous event has occurred at time s, so $\Delta t = t - s$ and the reservoir becomes empty at $t = 2s$. Since transition D_e fires at time 5, the occurrence time equation of this event is simply $t = 5$ and does not depend on s. The minimum of both equations then determines the next event, as shown in the shaded area in Figure 4. The procedure forms a polygon over the segment $t = s$ for $s \in [0, 5)$. Note that each side of this polygon represents the occurrence time of an event, hence, the procedure can be repeated recursively for each of them ,i.e., we can form another polygon over each side, and continue this procedure until we have obtained the complete partitioning of the stochastic area, up to the maximum analysis time. Figure 2 shows the complete STD for the reservoir example with nine different regions in the stochastic area. After all regions have been determined, measures of interests can be computed by deconditioning over the distribution function $g(s)$.

4 Generating the Diagram

We now present a formal algorithm for the generation of the STD. It consists of two main phases: partitioning the deterministic area (described in Section 4.1), and partitioning the stochastic area (described in Section 4.2).

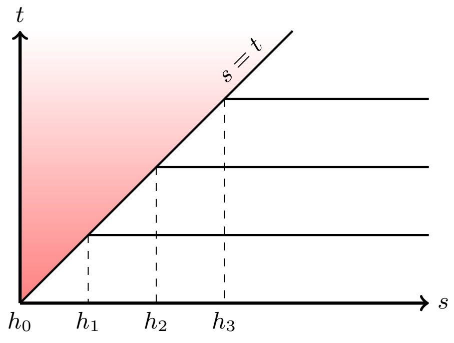


Fig. 5. Deterministic regions

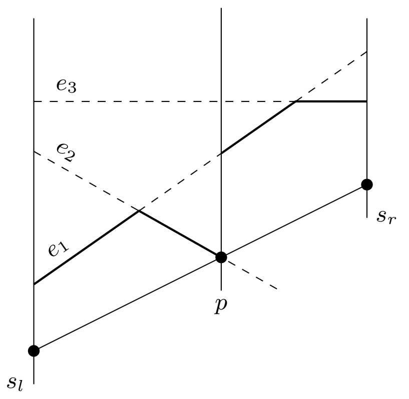


Fig. 6. Formation of invariant polygons

4.1 Partitioning the Deterministic Area

In this phase the evolution of the system is purely deterministic, since by assumption the general transition has not fired yet. Therefore the so-called *deterministic regions* are constructed with lines parallel to the s-axis, as shown in Figure 5. Each deterministic region $\mathcal{R}_i$ is determined uniquely by the interval $[h_i, h_{i+1})$, where h_i is the occurrence time of the event that changes the state of the system from region $\mathcal{R}_{i-1}$ into region $\mathcal{R}_i$.

The procedure for partitioning the deterministic area is outlined in Algorithm 2. Until the system reaches T_{max}, the procedure findNDtEvent, c.f. Algorithm 3, provides the next event for each marking of the system. During the evolution of the system at each point, two types of events are possible: an enabled deterministic transition reaches its firing time or a continuous place reaches a boundary. We iterate over all continuous places, and find the time at which each reaches its lower and upper boundaries (lines 2-8, Algorithm 3). Also for all enabled deterministic transitions we have to find the remaining time to fire (lines 9-13, Algorithm 3). Finally, the next event is the one with the smallest remaining time to occur, denoted e and Δt_e, respectively. A new region $\mathcal{R}_i$ is created and added to the set of deterministic regions $\mathcal{R}^D$ (lines 6-8, Algorithm 2). Then the current system sate is updated (Algorithm 6) and the current system time is advanced by Δt_e time units (lines 9-10).

4.2 Partitioning the Stochastic Area

In this phase we partition the area above the line $t = s$. As shown in Figure 5, in the previous phase the line $t = s$ has been segmented into several line segments, by deterministic regions. We iterate over all these line segments, and if the general transition was enabled in the system state of the corresponding deterministic region, it is fired, and the area above the corresponding line segment is further

Algorithm 1. genDiagram()

```
1: $\mathcal{P}^{\mathcal{S}}, \mathcal{R}^{\mathcal{D}} \leftarrow \emptyset$
2: $\mathcal{R}^{\mathcal{D}} \leftarrow$ partDtrmArea($\Gamma_0$)
3: for all $\mathcal{R}_i \in \mathcal{R}^{\mathcal{D}}$ do
4:     if $T^G$ in enabled in the $\mathcal{R}_i.\Gamma$ then
5:         $\delta.[s_l, s_r) \leftarrow R_i.[h_i, h_{i+1})$
6:         $\delta.eq \leftarrow t = s$
7:         $\Gamma' \leftarrow$ update($T^G, \Delta t = s - R_i.h_i, R_i.\Gamma$)
8:         $\mathcal{P}^{\mathcal{S}} \leftarrow \mathcal{P}^{\mathcal{S}} \cup$ partStochArea($\delta, \Gamma'$)
```

Algorithm 2. partDtrmArea(Γ_0)

```
Require: Initial system state $\Gamma_0$.
Ensure: Set of deterministic regions.
1: $\Gamma \leftarrow \Gamma_0$
2: $\mathcal{R}^D \leftarrow \emptyset$
3: $t \leftarrow 0$; $i \leftarrow 0$
4: while $t < t_{max}$ do
5:     $(e, \Delta t_e) \leftarrow$ findNDtEvent($\Gamma$)
6:     $R_i.[h_i, h_{i+1}) \leftarrow [t, t + \Delta t_e)$
7:     $R_i.\Gamma \leftarrow \Gamma$
8:     $\mathcal{R}^D \leftarrow \mathcal{R}^D \cup \{R_i\}$
9:     $\Gamma \leftarrow$ update($e, \Delta t = \Delta t_e, \Gamma$);
10:    $t \leftarrow t + \Delta t_e$; $i \leftarrow i + 1$
```

Algorithm 3. findNDtEvent(Γ)

```
Require: The current system state $\Gamma$.
Ensure: Next event and its remaining time to occur.
1: $\Delta t_{min} \leftarrow \infty$
2: for all $P_i \in \mathcal{P}^C$ do
3:     if $\Gamma.d_i > 0$ then
4:         $\Delta t_e \leftarrow \frac{\phi_b^{\mathcal{P}}(P_i) - \Gamma.x_i}{\Gamma.d_i}$
5:     if $\Gamma.d_i < 0$ then
6:         $\Delta t_e \leftarrow \frac{\Gamma.x_i}{\Gamma.d_i}$
7:     if $\Delta t_e < \Delta t_{min}$ then
8:         $(e, \Delta t_{min}) \leftarrow (P_i, \Delta t_e)$
9: for all $T_i \in \mathcal{T}^D$ do
10:    if $T$ is enabled then
11:        $\Delta t_e \leftarrow \phi_d^{\mathcal{T}}(T_i) - \Gamma.c_i$
12:        if $\Delta t_e < \Delta t_{min}$ then
13:            $(e, \Delta t_{min}) \leftarrow (T_i, \Delta t_e)$
14: return $(e, \Delta t_{min})$
```

partitioned. An arbitrary segment δ is defined by equation $\delta.eq : t = \alpha s + b$ and endpoints $\delta.[s_l, s_r)$. Each segment δ corresponds to an event in a way that the equation $\delta.eq$ represents its occurrence time, and the general transition had to fire between the endpoints $\delta.[s_l, s_r)$. For example if the general transition fires when the system is in the deterministic region $\mathcal{R}_i$, we enter the area above the segment with equation $t = s$ and endpoints $[h_i, h_{i+1})$.

Proposition 1. *After firing the general transition at time s, for each system state the occurrence time of the next events are linear functions of s.*

Proof. We prove the proposition by structural induction on associated segments of events, with firing of the general transition as the basis. W.l.o.g. assume that before firing the general transition, the system is at the deterministic region $\mathcal{R}_i$. When the general transition fires at time s, we have been in this region for $\Delta t = s - h_i$ time units, so at the firing time of the general transition the fluid level of continuous place P_k^C linearly depends on s, as follows $x'_k = d_k(s - h_i) + x_k$. Also the clock value of an enabled deterministic transition T_k^D is $c'_k = (s - h_i) + c_k$. Therefore, at the very moment after firing of the general transition, all fluid levels and clock values are linear functions of s, and hence the occurrence time of the next events are linear function of s too, as shown for the general case below.

As the inductive step, w.r.t. induction hypothesis, suppose that an event has occurred at time $t = \alpha s + \beta$ for $s \in [s_l, s_r)$. Recall, that two events are possible: a continuous place reaches its lower / upper boundary or an enabled deterministic transition reaches its firing time. Let the fluid level in a continuous place P_k^C be $x_k = a_k^p s + b_k^p$. The amount of time this place needs to reach one of its boundary is denoted by Δt_k^p and can be calculated as follows:

$$d_k \Delta t_k^p = \begin{cases} \phi_b^{\mathcal{P}}(P) - x_k & \text{if } d_k > 0, \\ -x_k & \text{if } d_k < 0. \end{cases}$$

According to the occurrence of the previous event, we have $\Delta t_k^p = t_k^p - (\alpha s + \beta)$ for $s \in [s_l, s_r)$, where t_k^p is the occurrence time of this fluid event. As a result, above the line $t = \alpha s + \beta$ and $s \in [s_l, s_r)$, the considered fluid event occurs at time t_k^p as follows:

$$t_k^p = \begin{cases} (\alpha - \frac{a_k^p}{d_k})s + (\frac{-b_k + \phi_b^{\mathcal{P}}(P_k^C)}{d_k} + \beta) & \text{if } d_k > 0, \\ (\alpha - \frac{a_k^p}{d_k})s + (-\frac{b_k}{d_k} + \beta) & \text{if } d_k < 0. \end{cases} \tag{1}$$

The firing time for a deterministic transition can also be derived in a same way. Let the clock value of the deterministic transition T_k^D be $c_k^t = a_k^t s + b_k^t$. The firing time can be calculated as follows:

$$t_k^t = (\alpha - a_k^t)s + (\phi_d^{\mathcal{T}}(T_k^D) + \beta - b_k^t). \tag{2}$$

Therefore, occurrence time of both types of events linearly depend on s. Now, with the same argument, if we set $\alpha = 1$ and $\beta = 0$ the basis of the induction is also satisfied, and hence the proof is complete. □

As an immediate result of the above proposition we can state that all regions in the stochastic area are polygons. Since the system state in these polygons does not change, we call them *invariant polygons* in the following. We present the algorithm for partitioning the area above an arbitrary *underlying segment* δ. The algorithm for this procedure is outlined in Algorithm 4. At first, we check whether the maximum analysis time has not been reached (line 1). Then we identify the potential events that can occur in the current marking of the system. For this we consider all continuous places and enabled deterministic transitions.

The procedure for finding all potential events, at each system state, is called findPotEvents and shown in Algorithm 5. It uses the same arguments as in the proof of Proposition 1. In lines 2-7 we iterate over all continuous places, and for each with a non-zero drift, the time for reaching its boundary is computed, according to Equation (1). Also in lines 8-11 this is repeated for all enabled deterministic transitions, according to Equation (2). Finally the set of all events and the equations of their occurrence time is returned.

In order to find the occurrence times of the next events conditioned on the value of $s \in [s_l, s_r)$, we have to take the minimum over all these linear equations. Taking the minimum over a set of lines results in several convex polygon(s) over the underlying segment δ. Note that, these equations are only valid in the area above the underlying segment δ. An example with three possible events: e_1, e_2 and e_3 is presented in Figure 6. Event e_2 intersects with the underlying segment at point p, so e_2 can not occur for $s > p$ and in the minimum taking procedure after this point e_2 does not have to be considered any more. As a result two polygons will be formed over the underlying segment.

The procedure that identifies the set of next events over an underlying segment, is called findNEvents, it simply iterates over all lines indicating the firing time of potential events and for each $s \in [s_l, s_r)$ finds the minimum line. It returns a set of segments from which the invariant polygon(s) over the underlying segment can be formed by iteration over the set of segments, this is done in procedure createPolygons. These two procedures are described in detail in [7].

The procedures findPotEvents, findNEvents and createPolygons, are called in lines 3-5, in Algorithm 4. Now, having obtained the set of segments of all next events, we can partition the area above each of these segments. Through line 6-8 we iterate over all these segments, and recursively call the function partStochArea for each segment, after updating the system state.

The procedure for updating the system state is provided in Algorithm 6. This procedure needs the event e and the amount of time Δt that can possibly depend on s, to advance the marking. In lines 2-3, for each continuous place P_i it alters the fluid level in that place according to Δt and the fluid drift d_i. Also in lines 4-5, it adds Δt to the clock value of each enabled deterministic transition. Moreover, if the event e is a transition it is fired, in line 8, to update the discrete marking. Finally we update the fluid drifts for the new marking, by calling the function

Algorithm 4. PARTSTOCHAREA(δ, Γ)

```
Require: The underlying segment $\delta$ and the current system state $\Gamma$.
Ensure: Partitioning of the area above the given underlying segment $\delta$.
 1: if $\delta.s_l > T_{max}$ then
 2:     return
 3: potentialSet ← FINDPOTEVENTS($\delta, \Gamma$)
 4: nextEvSet ← FINDNEVENTS(potentialSet, $\delta$)
 5: polygonSet ← polygonSet ∪ CREATEPOLYGONS(nextEvSet, $\delta, \Gamma$)
 6: for all $(e, \delta_e) \in$ nextEvSet do
 7:     $\Gamma' \leftarrow$ UPDATE(e, $\Delta t = \delta_e$.eq $-\delta$.eq, $\Gamma$)
 8:     polygonSet ← polygonSet ∪ PARTSTOCHAREA($\delta_e, \Gamma'$)
 9: return polygonSet
```

Algorithm 5. FINDPOTEVENTS(δ, Γ)

```
Require: The underlying segment $\delta$ and the current system state $\Gamma$.
Ensure: Set of potential events.
 1: potSet ← $\emptyset$
 2: for all $P_i \in \mathcal{P}^C$ do
 3:     if $\Gamma.d_i > 0$ then
 4:         $eq \leftarrow t = \delta.eq - \frac{\Gamma.x_i}{\Gamma.d_i} + \frac{\phi_b^{\mathcal{P}}(P_i)}{\Gamma.d_i}$
 5:     if $\Gamma.d_i < 0$ then
 6:         $eq \leftarrow t = \delta.eq - \frac{\Gamma.x_i}{\Gamma.d_i}$
 7:     potSet ← potSet $\cup\{(P_i, \text{eq})\}$
 8: for all $T_i \in \mathcal{T}^D$ do
 9:     if $T$ is enabled then
10:         $eq \leftarrow t = \delta.eq - \Gamma.c_i + \phi_d^{\mathcal{T}}(T_i)$
11:         potSet ← potSet $\cup\{(T, \text{eq})\}$
12: return potSet
```

Algorithm 6. UPDATE$(e, \Delta t = as + b, \Gamma)$

```
Require: The event $e$ to be committed, the $s$-dependent equation $\Delta t$ of time to ad-
    vance, and the marking $\Gamma$ to be updated.
Ensure: Advancement of system marking for the specified time.
 1: $\Gamma' \leftarrow \Gamma$
 2: for all $P_i \in \mathcal{P}^C$ do
 3:     $\Gamma'.x_i \leftarrow \Gamma.x_i + \Delta t \times \Gamma.d_i$
 4: for all $T_i \in \mathcal{T}^D$ do
 5:     if $T_i$ is enabled then
 6:         $\Gamma'.c_i \leftarrow \Gamma.c_i + \Delta t$
 7: if $e$ is a transition then
 8:     $\Gamma' \leftarrow fire(\text{e}, \Gamma)$
 9: $\Gamma' \leftarrow$ upadteDrifs($\Gamma'$)
10: return $\Gamma'$
```

updateDrifts. This is done according to the new discrete and continuous marking and rate adaptation in rules described in [9].

Finally, Algorithm 1 generates the STD. First the procedure partDtrmArea is called, and the deterministic regions are saved in the set $\mathcal{R}^D$. Then for each region $\mathcal{R}_i$, if the general transition is enabled, the segment with equation $t = s$ with the interval $[h_i, h_{i+1})$ is created (lines 4-6). The marking of the system is updated in line 7, by calling procedure update. Since the general transition should be fired we pass T^G as argument. Also the time that has passed after entering region $\mathcal{R}_i$ until firing the general transition, is $\Delta t = s - h_i$, as it is passed as second argument. Finally the procedure partStochArea is called with two arguments the created segment and updated system state.

5 Computing Measures

After the STD has been generated, the state of the system depends on the distribution of the firing time of the general transition, $g(s)$ and on the system time. By deconditioning s over the values of $g(s)$, the state probability distribution can be derived, as briefly sketched in Section 3.1. In order to compute more sophisticated measures of interest, we introduce property ψ, see below, which is defined as a combination of discrete and continuous markings. Note that this property is an extended version of what has appeared in [8]. The main difference is that we add negation which makes it complete and the result more expressive:

$$\psi = \neg\psi \mid \psi \wedge \psi \mid n_p = a \mid x_k \leq b. \qquad (3)$$

To compute the probability of being in a system state for which property ψ holds at time τ, at first we have to identify all invariant polygons and deterministic regions the system can be in, i.e. all regions intersecting line $t = \tau$. Then we verify whether property ψ holds for any of these regions, and if so determine the intervals in which the property is satisfied. Finally, $g(s)$ is integrated over all these regions.

An atomic property which reasons about discrete places, either holds in the complete invariant polygon or not at all. Recall, that the amount of fluid in a continuous place may linearly depend on s. Hence, an atomic property explaining the amount of fluid in a continuous place, may be valid only in a certain part of the considered polygon. More specifically, let the amount of fluid in the place P^c_k be $x_k = \alpha s + \beta$. For the computation of the probability to be in a system state for which $x_k \leq b$ holds, $s^* = (b - \beta)/\alpha$ defines the threshold value of s where the validity of the property changes. In case s^* lies inside a given polygon, depending on the sign of α the property is satisfied either before or after s^*.

Let the line $t = \tau$ intersect the polygon $\mathcal{P}_i$ in the interval $[s^i_1, s^i_2]$, then to negate a property we need to find the complement of the interval within $[s^i_1, s^i_2]$ for which the original property holds. For the conjunction of two properties we need to find the intersection of the two intervals that are associated with the two original properties. Therefore, a nested property ψ may be satisfied in a set of intervals.

Let $\mathcal{P}^\tau$ be the set of all invariant polygons intersecting the line $t = \tau$. For a given invariant polygon $\mathcal{P}_i \in \mathcal{P}^\tau$, the set of intervals in which the property ψ holds is indicated by $\mathcal{S}^i$ and each interval in this set is denoted $[s_l^i, s_r^i)$. Trivially $\mathcal{S}^i$ is empty if the property is not satisfied in $\mathcal{P}_i$. Let $\mathcal{R}_i^\tau$ denote the deterministic region for which $\tau \in [h_i, h_{i+1})$. Also, let $\mathcal{I}^\psi(s, \tau)$ be the characteristic function for condition ψ at the point (s, τ), which evaluates to 1 or 0 whether ψ holds or not, respectively. Furthermore, $\mathcal{I}^\psi(\mathcal{R}_i^\tau)$ indicates whether condition ψ is satisfied in $\mathcal{R}_i^\tau$. So, the probability distribution to be in a system state for which property ψ is satisfied at time τ can be computed as follows:

$$\begin{aligned}
\pi^\psi(\tau) &= \int_0^\infty \mathcal{I}^\psi(s,\tau) g(s) ds \\
&= \int_0^\tau \mathcal{I}^\psi(s,\tau) g(s) ds + \int_\tau^\infty \mathcal{I}^\psi(s,\tau) g(s) ds \\
&= \sum_{\mathcal{P}_i \in \mathcal{P}^\tau} \int_{\mathcal{S}^i} g(s) ds + \mathcal{I}^\psi(\mathcal{R}_i^\tau) \int_\tau^\infty g(s) ds \\
&= \left(\sum_{\mathcal{P}_i \in \mathcal{P}^\tau} \sum_{[s_l^i, s_r^i) \in \mathcal{S}^i} \left(\phi_g(s_r^i) - \phi_g(s_l^i) \right) \right) + \mathcal{I}^\psi(\mathcal{R}_i^\tau)(1 - \phi_g(\tau)) \qquad (4)
\end{aligned}$$

The above set of equations shows how the partitioning into regions can be used for smarter deconditioning. Equation (4) consists of two terms. The first term expresses the probability of holding ψ at time τ, in the stochastic area, by simply iterating over all invariant polygons intersecting the line $t = \tau$ and summing the probability over all intervals in which the property holds. The second term expresses the probability of being in $\mathcal{R}_i^\tau$ if the property ψ holds in it.

6 Case Study

The complexity of the proposed algorithm clearly depends on the structure of the model. The process of computing measures of interest also linearly depends on the number of regions. In the following we show the scalability and efficiency of the proposed method using the case study as in [8]. We scale the number of transitions and continuous places in the case study, and discuss its influence on the number of regions and the computation time of the algorithm.

Figure 7, presents a model of a water treatment facility with different phases. The continuous place C_i represents the storage of a water softening phase. By design this is a slow process with large storage. The continuous place C_f and transition F_f, represent a generic water filtering phase. Opposed to the softening phase this filtration phase is a fast process with small storage. The continuous place C_s, represents the final storage from which water is distributed to the customers with different rates, depending on the time of the day. The deterministic transition T_b represents a failure at time α, in the softening phase. When it fires, the continuous transition F_i is disabled and the general transition G_r becomes enabled. G_r models the time it takes to repair the system failure according to

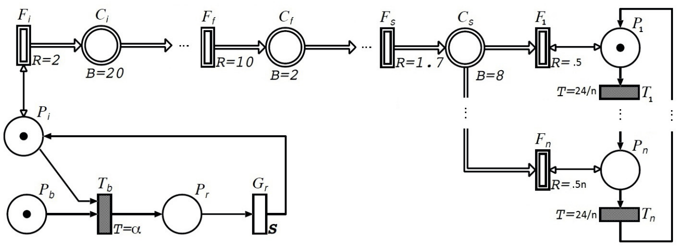


Fig. 7. HPnG model for a symbolic water treatment facility

Table 1. Scaling the filtration phases. All times in milliseconds.

	Region-based			Param. reach.	
#Filters	#Region	STD	MCT	Tree	MCT
1	327	43	161	10	11106
2	433	80	239	19	13153
3	539	69	294	19	15415
4	663	77	373	22	17188
5	769	86	461	25	19352
6	903	95	509	26	21501
7	1026	106	586	30	23385
8	1159	121	662	31	25875

Table 2. Scaling the demand rates. All times in milliseconds.

	Region-based			Param. reach.	
#Demands	#Region	STD	MCT	Tree	MCT
2	202	26	104	15	32348
3	403	76	202	21	43874
4	909	72	431	23	52526
5	1204	82	538	38	66793
6	1624	91	711	49	79479
7	1797	90	681	30	69484
8	2225	115	1004	99	115542
9	2776	125	1195	102	120129
10	3457	143	1451	133	136896

the arbitrary probability density $g(s)$. Note that discrete place P_b restricts the model such that the failure can occur only once.

The model presented in Figure 7 is made scalable in two ways. First, by cascading more filtration phases, and second by dividing the day into more intervals with different demand rates. In order to show the efficiency of our algorithm, we scale the model in these two ways, and for each instance, compute the probability distribution for the amount of fluid in place C_s. This is an important measure of interest, because an empty final storage C_s means failure to deliver water to the consumers. Moreover, to provide a comparison with the parametric reachability algorithm, as presented in [8], we also calculate this probability distribution using this algorithm. All the computations have been performed on a machine equipped with a 2.0 GHz intel® CORE™ i7 processor, 4 GB of RAM, and Windows 7. The results are shown in Tables 1 and 2.

Scaling in both dimensions increases the number of regions, as shown in the second column of both tables. The time needed to construct the STD and the tree with all parametric locations, are given in the third and fifth columns of both tables. The time needed to compute the measures of interest is denoted MCT (Measure Computation Time). When scaling the number of filters the generation of the STD takes about 3 to 4 times longer than the construction of the parametric locations. This is due to the more involved computations that are necessary to construct the polygons in the STD. When scaling the number of demands the generation of the STD takes about 2 to 3 times longer than the construction of the parametric locations. This is, however, more than compensated for when the measures of interest are computed. The new algorithm is, depending on the size of the model, between 20 and 100 times faster than the algorithm in [8]. Apparently, for smaller models the speed up is larger than for bigger models. This is because the complexity of the old algorithm is logarithmic in number of parametric locations. Furthermore, in case a closed form of the CDF exists, the choice of the distribution does not influence the complexity of the region-based algorithm. Clearly, the MCT of the parametric reachability algorithm depends on the chosen discretization step. The results presented in the tables have been obtained for a discretization step of 0.005. A larger discretization step reduces the MCT, but also decreases the accuracy of the results. For a discretization step of 0.005, the maximum difference between the results from both algorithms is 0.5%. Running the parametric location algorithm with a discretization step of 0.2 leads to approximately the same MCT with both algorithms, the resulting maximum relative error, however is 3%.

7 Conclusions

This paper presents an algorithm for the analysis of HPnGs that partitions the state space into regions, where all the states in a given region have the same deterministic marking and the continuous marking and the remaining firing time for all states in the same region follow the same linear function of s and t.

The restrictions of the model class to a single one-shot transition and the requirement of a unique priority assignment to each deterministic and immediate transition ensure that the computed partitioning is a single two-dimensional STD. Relaxing the requirement of the unique priority assignment potentially leads to concurrency between timed transitions. In [8] this has been resolved by a probabilistic choice between transitions with the same minimum firing time. Since the firing of different transitions leads to a different further evolution of the system, a different STD is needed. To compute measures of interest, the deconditioning then needs to take several STDs into account and weight them according to the probabilities assigned to the firing of each transition. Future work will investigate how this can be done efficiently. Also allowing more general transitions or relaxing the one-shot restriction will change the resulting STD. Each firing of a general transition will add an extra dimension to the STD and the deconditioning then needs to be done for several dimension. This is also an interesting line for future research.

Even though the model class currently is restricted in several ways, it is still very useful for the application field of fluid critical infrastructures, since the physical processes are fairly deterministic and stochasticity is only needed to model failures and repairs. To the best of our knowledge no analyzable model class exists that allows for an arbitrary amount of continuous places without resetting the amount of fluid upon discrete changes, as needed in this field. Furthermore, we would like to emphasis that the presented algorithm presents an enormous improvement with respect to the parametric reachability analysis in [8], it allows for a much quicker analysis and due to the partitioning the obtained results are also more accurate.

Acknowledgement. This work has been supported by the ROCKS project through the NWO grant DN 63-257 and . Anne Remke is funded by a NWO Veni grant.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Yovine, S., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
2. Alur, R., Courcoubetis, C., Henzinger, T.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. Hybrid Systems 736, 209–229 (1993)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Asarin, E., Maler, O.: Reachability analysis of dynamical systems having piecewise-constant derivatives. Theoretical Computer Science 138(1), 35–65 (1995)
5. Berthomieu, B.: Modeling and verification of time dependent systems using time Petri nets. IEEE Transactions on Software Engineering 17(3), 259–273 (1991)
6. David, R., Alla, H.: Discrete, Continuous, and Hybrid Petri Nets, 2nd edn. Springer (2010)
7. Ghasemieh, H., Remke, A., Haverkort, B., Gribaudo, M.: Region-based analysis of hybrid Petri nets with a single general one-shot transition: extended version. Technical report, University of Twente (2012), http://wwwhome.cs.utwente.nl/~anne/techreport/std.pdf
8. Gribaudo, M., Remke, A.: Hybrid Petri Nets with General One-Shot Transitions for Dependability Evaluation of Fluid Critical Infrastructures. In: 2010 IEEE 12th International Symposium on High Assurance Systems Engineering, pp. 84–93. IEEE CS Press (November 2010)
9. Gribaudo, M., Remke, A.: Hybrid petri nets with general one-shot transitions: model evolution. Technical report, University of Twente (2010), http://wwwhome.cs.utwente.nl/~anne/techreport/hpng.pdf
10. Kartson, D., Balbo, G., Donatelli, S., Franceschinis, G., Conte, G.: Modelling with Generalized Stochastic Petri Nets, 1st edn. John Wiley & Sons, Inc. (1994)
11. Vicario, E.: Static analysis and dynamic steering of time-dependent systems. IEEE Transactions on Software Engineering 27(8), 728–748 (2001)
12. Vicario, E., Sassoli, L., Carnevali, L.: Using stochastic state classes in quantitative evaluation of dense-time reactive systems. IEEE Transactions on Software Engineering 35(5), 703–719 (2009)

Reducing Quasi-Equal Clocks in Networks of Timed Automata

Christian Herrera, Bernd Westphal, Sergio Feo-Arenis, Marco Muñiz, and Andreas Podelski

Albert-Ludwigs-Universität Freiburg, 79110 Freiburg, Germany

Abstract. We introduce the novel notion of *quasi-equal* clocks and use it to improve the verification time of networks of timed automata. Intuitively, two clocks are *quasi-equal* if, during each run of the system, they have the same valuation except for those points in time where they are reset. We propose a transformation that takes a network of timed automata and yields a network of timed automata which has a smaller set of clocks and preserves properties up to those not comparing *quasi-equal* clocks. Our experiments demonstrate that the verification time in three transformed real world examples is much lower compared to the original.

1 Introduction

Modelling the local timing behaviour of components and the synchronisation between them is the natural way to model distributed real-time systems by networks of timed automata [1]. This is achieved in a straightforward manner by using independent clocks.

For designs where a set of clocks is intended to be synchronized, e.g., the class of TDMA-based protocols [2], using independent clocks causes unnecessary verification overhead for those specifications where the order of resets of synchronised clocks is not relevant. For instance, in network traversal time requirements, resetting synchronised clocks does not contribute to the time lapses being measured. The unnecessary overhead is caused by the interleaving semantics of timed-automata where the automata do their resets one by one and thereby induce a set of reachable intermediate configurations which grows exponentially in the number of components in the system. Although the interleaving semantics of timed-automata offers a practical solution for model checking in tools like *Uppaal* [3], it artificially introduces intermediate states that are explored when verifying models of physical systems, although they may be irrelevant for the property being verified.

The overhead could be eliminated by manually optimising models for verification. Nonetheless, modelling without technicalities — in particular without manual optimizations for verification — is desired to improve on the readability and maintainability of the models. We aim to bridge the gap between efficiency and readability by enabling the modelling engineer to use more natural representations of a system. Unnecessary overhead can be mechanically removed as per our approach, thus enabling both readable models and efficient model checking.

To this end, we characterise clocks intended to be synchronised in the real world by the novel notion of *quasi-equality*. Intuitively, two clocks are *quasi-equal* if, during each run of the system, they have the same valuation except for those points in time where they are reset. We call the properties which, for a given network of timed automata, are independent from the ordering in which the *quasi-equal* clocks are reset, *validable*. Sets of quasi-equal clocks induce *equivalence classes* in networks of timed-automata. We present an algorithm that replaces all clocks from an *equivalence class* of *quasi-equal* clocks by a representative clock. The result is a network of timed automata with a smaller set of clocks. It is weakly bisimilar to the original network and thus preserves validable properties. We show that when performing clock replacement alone, properties are not necessarily preserved. Our algorithm introduces a new automaton in the network and uses auxiliary variables and *broadcast* synchronization channels to ensure that the semantics of the original network are preserved, up to configurations where quasi-equal clocks have different valuations. The use of our algorithm can lead to significant improvements in the verification cost of validable properties compared to the cost of verifying them in the original network.

This paper is organized as follows. In Section 2, we provide basic definitions. Section 3 introduces the formal definition of *quasi-equal* clocks, presents the algorithm that implements our approach on the set of well-formed networks, and proves its correctness. In Section 4, we compare the verification time of three real world examples before and after applying our approach. In Section 5, we draw conclusions and propose future work.

1.1 Related Work

The reduction of the state space to be explored in order to speed up the verification of properties of a system, is a well-known research topic. Diverse techniques have been proposed to achieve such a reduction, many of them by using static analysis over timed automata [4,5,6,7]. One method that uses static analysis is presented in [8], originally defined for single automata and later generalized for networks of timed automata [9]. This method reduces the number of clocks in single timed automata by detecting *equal* clocks. Two clocks are equal in a location if both are reset at the same time and by the same edge, or both are set to clocks that are themselves equal in the source location. Equal clocks always have the same valuation, so just one clock for every set of equal clocks in a given location is necessary to determine the behavior of the system at that location. The case studies we considered for experiments do not have equal clocks therefore applying the method in [8] would not reduce clocks.

In sequential timed automata [10], one set of quasi-equal clocks is syntactically declared. Those quasi-equal clocks are implicitly reduced by applying the sequential composition operator, which also exploits other properties of sequential timed automata, and thereby achieves further improvements in verification time.

In [11], clocks are reduced while abstracting systems composed of timed components which represent processes. Each process uses one internal clock for its internal operations. The approach exploits the fact that each component works in a sort of sequence in order to process events, and each internal clock is only used for a small fraction of time during such a sequence, thus only one clock can be used instead. The networks we consider can in general not be reduced to this approach as we do not assume a working sequence.

The technique in [4,5] is based on so called *observers*, which are single components representing safety or liveness requirements of a given network of timed automata. For each location of the observer, the technique deactivates (or ignores) irrelevant components (clocks or even a whole automaton) if such components do not play a role in the future evolution of such an observer. The networks of our experiments do not have observers, therefore we can not use this technique. However, their case studies may benefit from our approach if observer clocks are quasi equal to component clocks.

2 Preliminaries

Following the presentation in [12], we here recall timed automata definitions.

Let $\mathcal{X}$ be a set of *clocks*. The set $\Phi(\mathcal{X})$ of *simple clock constraints* over $\mathcal{X}$ is defined by the grammar $\varphi ::= x \sim y \mid x - y \sim c \mid \varphi_1 \wedge \varphi_2$ where $x, y \in \mathcal{X}$, $c \in \mathbb{Q}_{\geq 0}$, and $\sim \in \{<, \leq, \geq, >\}$. Let $\Phi(\mathcal{V})$ be a set of *integer constraints* over *variables* $\mathcal{V}$. The set $\Phi(\mathcal{X}, \mathcal{V})$ of *constraints* comprises $\Phi(\mathcal{X})$, $\Phi(\mathcal{V})$, and conjunctions of clock and integer constraints. We use $\mathit{clocks}(\varphi)$ to denote the set of clocks occurring in a constraint φ. We assume the canonical satisfaction relation "$\models$" between *valuations* $\nu : \mathcal{X} \cup \mathcal{V} \to \mathit{Time} \cup \mathbb{Z}$ and constraints, with $\mathit{Time} = \mathbb{R}_{\geq 0}$.

A timed automaton $\mathcal{A}$ is a tuple $(L, B, \mathcal{X}, \mathcal{V}, I, E, \ell_{ini})$, which consists of a finite set of *locations* L, a finite set B of *actions* comprising the *internal action* τ, finite sets $\mathcal{X}$ and $\mathcal{V}$ of clocks and variables, a mapping $I : L \mapsto \Phi(\mathcal{X})$, that assigns to each location a *clock constraint*, and a set of *edges* $E \subseteq L \times B \times \Phi(\mathcal{X}, \mathcal{V}) \times \mathcal{R}(\mathcal{X}, \mathcal{V}) \times L$. An edge $e = (\ell, \alpha, \varphi, \vec{r}, \ell') \in E$ from location ℓ to ℓ' involves an action $\alpha \in B$, a *guard* $\varphi \in \Phi(\mathcal{X}, \mathcal{V})$, and a *reset vector* $\vec{r} \in \mathcal{R}(\mathcal{X}, \mathcal{V})$. A reset vector is a finite, possibly empty sequence of *clock resets* $x := 0$, $x \in \mathcal{X}$, and *assignments* $v := \psi_{int}$, where $v \in \mathcal{V}$ and ψ_{int} is an integer expression over $\mathcal{V}$. We write $\mathcal{X}(\mathcal{A})$, $\ell_{ini}(\mathcal{A})$, etc. to denote the set of clocks, the initial location, etc. of $\mathcal{A}$, and $\mathit{clocks}(\vec{r})$ to denote the set of clocks occurring in $\vec{r}$.

A finite sequence $\mathcal{A}_1, \dots, \mathcal{A}_N$ of timed automata with pairwise disjoint sets of clocks and pairwise disjoint sets of locations together with a set $\mathcal{B} \subseteq \bigcup_{i=1}^{N} B(\mathcal{A}_i)$ of *broadcast channels* is called *network (of timed automata)*. To indicate that $\mathcal{N}$ consists of $\mathcal{A}_1, \dots, \mathcal{A}_N$, we write $\mathcal{N}(\mathcal{A}_1, \dots, \mathcal{A}_N)$, and we write $\mathcal{A} \in \mathcal{N}$ if and only if $\mathcal{A} \in \{\mathcal{A}_1, \dots, \mathcal{A}_N\}$. Given a set of clocks $X \subseteq \mathcal{X}(\mathcal{N})$, we use $\mathcal{RES}_X(\mathcal{N})$ to denote the set of automata in $\mathcal{N}$ which have an outgoing edge that resets a clock from X, i.e. $\mathcal{RES}_X(\mathcal{N}) = \{\mathcal{A} \in \mathcal{N} \mid \exists\, (\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{A}) \bullet \mathit{clocks}(\vec{r}) \cap X \neq \emptyset\}$.

The operational semantics of the network $\mathcal{N}$ is the labelled transition system $\mathcal{T}(\mathcal{N}) = (\mathit{Conf}(\mathcal{N}), \mathit{Time} \cup B, \{\xrightarrow{\lambda} \mid \lambda \in \mathit{Time} \cup B\}, \mathcal{C}_{ini})$. The set of configurations $\mathit{Conf}(\mathcal{N})$ consists of pairs of *location vectors* $\langle \ell_1, \dots, \ell_N \rangle$ from $\times_{i=1}^{N} L(\mathcal{A}_i)$ and valuations of $\bigcup_{1 \leq i \leq N} \mathcal{X}(\mathcal{A}_i) \cup \mathcal{V}(\mathcal{A}_i)$ which satisfy the constraint $\bigwedge_{i=1}^{N} I(\ell_i)$.

We write $\ell_{s,i}$, $1 \leq i \leq N$, to denote the location which automaton $\mathcal{A}_i$ assumes in configuration $s = \langle \ell_s, \nu_s \rangle$ and $\nu_{s,i}$ to denote $\nu_s|_{\mathcal{V}(\mathcal{A}_i) \cup \mathcal{X}(\mathcal{A}_i)}$. Between two configurations $s, s' \in \mathit{Conf}(\mathcal{N})$ there can be three kinds of transitions. There is a *delay transition* $\langle \ell_s, \nu_s \rangle \xrightarrow{t} \langle \ell_{s'}, \nu_{s'} \rangle$ if $\nu_s + t' \models \bigwedge_{i=1}^{N} I_i(\ell_{s,i})$ for all $t' \in [0, t]$, where $\nu_s + t'$ denotes the valuation obtained from ν_s by time shift t'. There is a *synchronization transition* $\langle \ell_s, \nu_s \rangle \xrightarrow{\tau} \langle \ell_{s'}, \nu_{s'} \rangle$ if there are $1 \leq i, j \leq N$, $i \neq j$, a channel $b \in B(\mathcal{A}_i) \cap B(\mathcal{A}_j)$, and edges $(\ell_{s,i}, b!, \varphi_i, \vec{r}_i, \ell_{s',i}) \in E(\mathcal{A}_i)$ and $(\ell_{s,j}, b?, \varphi_j, \vec{r}_j, \ell_{s',j}) \in E(\mathcal{A}_j)$ such that $\ell_{s'} = \ell_s[\ell_{s,i} := \ell_{s',i}][\ell_{s,j} := \ell_{s',j}]$, $\nu_s \models \varphi_i \wedge \varphi_j$, $\nu_{s'} = \nu_s[\vec{r}_i][\vec{r}_j]$, and $\nu_{s'} \models I_i(\ell_{s',i}) \wedge I_j(\ell_{s',j})$. Let $b \in \mathcal{B}$ be a broadcast channel and $1 \leq i_0 \leq N$ such that $(\ell_{s,i_0}, b!, \varphi_{i_0}, \vec{r}_{i_0}, \ell_{s',i_0}) \in E(\mathcal{A}_{i_0})$. Let $1 \leq i_1, \ldots, i_k \leq N$, $k \geq 0$, be those indices different from i_0 such that there is an edge $(\ell_{s,i_j}, b?, \varphi_{i_j}, \vec{r}_{i_j}, \ell_{s',i_j}) \in E(\mathcal{A}_{i_j})$. There is *broadcast transition* $\langle \ell_s, \nu_s \rangle \xrightarrow{\tau} \langle \ell_{s'}, \nu_{s'} \rangle$ in $\mathcal{T}(\mathcal{N})$ if $\ell_{s'} = \ell_s[\ell_{s,i_0} := \ell_{s',i_0}] \cdots [\ell_{s,i_k} := \ell_{s',i_k}]$, $\nu_s \models \bigwedge_{j=0}^{k} \varphi_{i_j}$, $\nu_{s'} = \nu_s[\vec{r}_{i_0}] \cdots [\vec{r}_{i_k}]$, and $\nu_{s'} \models \bigwedge_{j=0}^{k} I_{i_j}(\ell_{s',i_j})$.

A finite or infinite sequence $\sigma = s_0 \xrightarrow{\lambda_1} s_1 \xrightarrow{\lambda_2} s_2 \ldots$ is called *transition sequence* (starting in $s_0 \in \mathcal{C}_{ini}$) of $\mathcal{N}$. Sequence σ is called *computation* of $\mathcal{N}$ if and only if it is infinite and $s_0 \in \mathcal{C}_{ini}$. We denote the set of all computations of $\mathcal{N}$ by $\Pi(\mathcal{N})$. A configuration s is called *reachable* (in $\mathcal{T}(\mathcal{N})$) if and only if there exists a computation $\sigma \in \Pi(\mathcal{N})$ such that s occurs in σ. A timed automaton or network configuration s is called *timelocked* if and only if there is no delay transition in any transition sequence starting at s.

A *basic formula* over $\mathcal{N}$ is either $\mathcal{A}_i.\ell$, $1 \leq i \leq n$, $\ell \in L(\mathcal{A}_i)$, or a constraint φ from $\Phi(\bigcup_{i=1}^{N} \mathcal{X}(\mathcal{A}_i), \bigcup_{i=1}^{N} \mathcal{V}(\mathcal{A}_i))$. It is satisfied by a configuration $s \in \mathit{Conf}(\mathcal{N})$ if and only if $\ell_{s,i} = \ell$ or $\nu_s \models \varphi$, respectively. A *reachability query EPF* over $\mathcal{N}$ is $\exists\Diamond\, \mathit{CF}$ where CF is a *configuration formula* over $\mathcal{N}$, i.e. any logical connection of basic formulae. $\mathcal{N}$ satisfies $\exists\Diamond\, \mathit{CF}$, denoted by $\mathcal{N} \models \exists\Diamond\, \mathit{CF}$, if and only if there is a configuration s reachable in $\mathcal{T}(\mathcal{N})$ such that $s \models \mathit{CF}$. We write $\mathcal{N} \models_{\neg\text{timelock}} \exists\Diamond\, \mathit{CF}$ if and only if CF is satisfied by a reachable, not timelocked configuration of $\mathcal{T}(\mathcal{N})$.

3 Reducing Clocks in Networks of Timed Automata

3.1 Quasi-Equal Clocks

Definition 1 (Quasi-Equal Clocks). *Let $\mathcal{N}$ be a network with clocks $\mathcal{X}$. Two clocks $x, y \in \mathcal{X}$ are called quasi-equal, denoted by $x \simeq y$, if and only if for all computation paths of $\mathcal{N}$, the valuations of x and y are equal, or the valuation of one of them is equal to 0, i.e., if*

$$\forall\, s_0 \xrightarrow{\lambda_1} s_1 \xrightarrow{\lambda_2} s_2 \cdots \in \Pi(\mathcal{N})\ \forall\, i \in \mathbb{N}_0 \bullet \nu_{s_i} \models (x = 0 \vee y = 0 \vee x = y).$$

For example, consider a distributed chemical plant controller. At the end of every minute, the controller fills two containers with gas, one for at most 10 seconds and one for at most 20 seconds. In Figure 1, a model of the system, the network $\mathcal{N}$ which is composed of automata $\mathcal{A}_1$ and $\mathcal{A}_2$, is shown. Both automata start

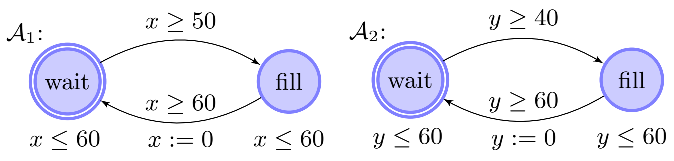


Fig. 1. The model of a chemical plant controller with quasi-equal clocks

in a waiting phase and after filling the containers, they wait for the next round. Both clocks, x and y, are reset when their valuation is equal to 60. Yet, in the strict interleaving semantics of networks of timed automata, the resets occur one after the other. According to Definition 1, x and y are *quasi-equal* because their valuations are only different from each other when one of the clocks has already been reset and the other still has value 60.

Lemma 1. *Let $\mathcal{N}$ be a network with clocks $\mathcal{X}$. The quasi-equality relation $\simeq \subseteq \mathcal{X} \times \mathcal{X}$ is an equivalence relation.*

Proof. For transitivity, show $(x = 0 \vee y = 0 \vee x = y) \wedge (x = 0 \vee z = 0 \vee x = z) \wedge (y = 0 \vee z = 0 \vee y = z)$ by induction over indices in a computation. □

In the following, we use $\mathcal{EC}_\mathcal{N}$ to denote the set $\{Y \in \mathcal{X}/\simeq \mid 1 < |Y|\}$ of equivalence classes of *quasi-equal* clocks of $\mathcal{N}$ with at least two elements. For each $Y \in \mathcal{X}/\simeq$, we assume a designated representative $\mathit{rep}(Y) \in Y$. We may write $\mathit{rep}(x)$ to denote $\mathit{rep}(Y)$ if $x \in Y$ is the representative clock of Y.

Given a constraint $\varphi \in \Phi(\mathcal{X}, \mathcal{V})$, we write $\Gamma(\varphi)$ to denote the constraint that is obtained by syntactically replacing in φ each occurrence of a clock $x \in \mathcal{X}$ by the representative $\mathit{rep}(x)$.

3.2 Transformational Reduction of Quasi-Equal Clocks

In the following we present an algorithm which reduces a given set of quasi-equal clocks in networks of timed automata. For simplicity, we limit the discussion to the syntactically characterised class of well-formed networks. The syntactical rules, although restrictive at first sight, still enabled us to apply our approach to relevant real world examples.

Definition 2 (Well-formed Network). *A network $\mathcal{N}$ is called* well-formed *if and only if it satisfies the following restrictions for each set of quasi-equal clocks $Y \in \mathcal{EC}_\mathcal{N}$:*

(R1) *An edge resetting a clock $x \in Y$ is not a loop and has a guard of the form $x \geq C_Y$, and the source location of such an edge has an invariant $x \leq C_Y$ for some constant $C_Y > 0$, i.e.,*

$$\exists\, C_Y \in \mathbb{N}^{>0}\ \forall\, \mathcal{A} \in \mathcal{N}\ \forall\, (\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{A})\ \forall\, x \in \mathit{clocks}(\vec{r}) \bullet$$
$$(\mathit{clocks}(\vec{r}) \cap Y \neq \emptyset)$$
$$\implies (\varphi = (x \geq C_Y) \wedge I(\ell) = (x \leq C_Y) \wedge \ell \neq \ell_{ini}(\mathcal{A}) \wedge \ell \neq \ell').$$

(R2) *A location having an outgoing edge resetting a clock $x \in Y$, does not have other outgoing edges, and such an edge only resets a single clock, i.e.,*

$$\begin{aligned}&\forall\, e_1 = (\ell_1, \alpha_1, \varphi_1, \vec{r}_1, \ell_1'), e_2 = (\ell_2, \alpha_2, \varphi_2, \vec{r}_2, \ell_2') \in E(\mathcal{N}) \bullet \\ &\quad (\ell_1 = \ell_2 \wedge (\mathit{clocks}(\vec{r}_1) \cup \mathit{clocks}(\vec{r}_2)) \cap Y \neq \emptyset) \\ &\qquad \implies e_1 = e_2 \wedge |\mathit{clocks}(\vec{r}_1)| = 1 \wedge \exists\, x \in Y \bullet \vec{r}_1 = (x := 0).\end{aligned}$$

(R3) *An edge resetting a clock $x \in Y$ is unique per automaton, i.e.,*

$$\begin{aligned}&\forall\, \mathcal{A} \in \mathcal{N}\ \forall\, (\ell_1, \alpha_1, \varphi_1, \vec{r}_1, \ell_1'), (\ell_2, \alpha_2, \varphi_2, \vec{r}_2, \ell_2') \in E(\mathcal{A}) \bullet \\ &\quad ((\mathit{clocks}(\vec{r}_1) \cup \mathit{clocks}(\vec{r}_2)) \cap Y \neq \emptyset) \implies e_1 = e_2.\end{aligned}$$

(R4) *A location having an outgoing edge resetting a clock $x \in Y$ has at least one incoming, non-looped edge, i.e.,*

$$\begin{aligned}&\forall\, \mathcal{A} \in \mathcal{N}\ \forall\, (\ell_2, \alpha_2, \varphi_2, \vec{r}_2, \ell_2') \in E(\mathcal{A}) \bullet \\ &\quad (\mathit{clocks}(\vec{r}_2) \cap Y) \neq \emptyset \implies \exists\, (\ell_1, \alpha_1, \varphi_1, \vec{r}_1, \ell_1') \in E(\mathcal{A}) \bullet \ell_1' = \ell_2 \wedge \ell_1 \neq \ell_2.\end{aligned}$$

(R5) *The action of an edge resetting a clock $x \in Y$ is τ, i.e.,*

$$\forall\, (\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{N}) \bullet (\mathit{clocks}(\vec{r}) \cap Y \neq \emptyset) \implies \alpha = \tau.$$

(R6) *At most one clock from Y occurs in the constraint of any edge, i.e.,*

$$\forall\, (\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{N}) \bullet |\mathit{clocks}(\varphi) \cap Y| \leq 1.$$

By rules *R1*, *R2*, and *R3* there is a unique reset edge per equivalence class and automaton, and a constant describing the reset times of quasi-equal clocks from the same equivalence class. By *R4* guarantees the existence of an edge that can be used to encode blocking multicast synchronisation. Rules *R2*, *R5*, and *R6* guarantee that the behaviour of a well-formed network is independent from the order of resets of quasi-equal clocks.

These rules should be relaxed to cover a broader class of networks of timed automata. For example, *R3* could be weakened to allow quasi-equal clocks with more than one reset point to be reduced. This would make, however, the transformation algorithm and its prove of correctness more involved.

Our transformation mainly operates on the source and destination locations of the edges resetting quasi-equal clocks, so-called reset locations.

Definition 3 (Reset Location). *Let $\mathcal{N}$ be a well-formed network. Let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of clocks of $\mathcal{N}$. Let $(\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{N})$ be an edge that resets a clock from Y, i.e. $\mathit{clocks}(\vec{r}) \cap Y \neq \emptyset$. Then ℓ (ℓ') is called* reset (successor) location *wrt. Y. We use $\mathcal{RL}_Y$ ($\mathcal{RL}_Y^+$) to denote the set of reset (successor) locations wrt. Y in $\mathcal{N}$ and we set $\mathcal{RL}_\mathcal{N} := \bigcup_{Y \in \mathcal{EC}_\mathcal{N}} \mathcal{RL}_Y$ and similarly $\mathcal{RL}_\mathcal{N}^+$.*

In the following we describe the transformation function $\mathcal{K}$. It works with two given inputs: a well-formed network $\mathcal{N}$ and the set of equivalence classes $\mathcal{EC}_\mathcal{N}$ of quasi-equal clocks in $\mathcal{N}$. $\mathcal{K}$ outputs a transformed network $\mathcal{N}' = \mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$ by performing in $\mathcal{N}$ the following steps for each equivalence class $Y \in \mathcal{EC}_\mathcal{N}$:

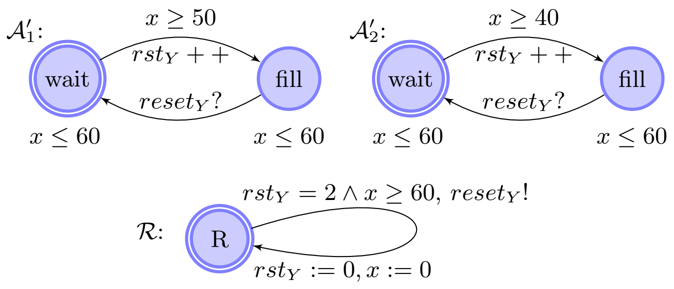


Fig. 2. The model of the chemical plant controller after applying $\mathcal{K}$

– Delete each reset of a clock from Y.
– For each edge resetting a clock from Y, replace the guard by *true*.
– In each invariant and each guard, replace each clock $x \in Y$ by $rep(x)$.
– Add to $\mathcal{N}$ a broadcast channel $reset_Y$ and add the input action $reset_Y?$ to each edge resetting a clock from Y.
– Add a counter variable rst_Y to $\mathcal{N}$ and add the increment $rst_Y := rst_Y + 1$ to the reset sequence of each incoming edge of a reset location $\ell \in \mathcal{RL}_Y$.

As a final step, add a new automaton $\mathcal{R}$ with a single location $\ell_{ini,\mathcal{R}}$ *if* $\mathcal{EC}_\mathcal{N} \neq \emptyset$. For each $Y \in \mathcal{EC}_\mathcal{N}$, add an edge $(\ell_{ini,\mathcal{R}}, \alpha, \varphi, \vec{r}, \ell_{ini,\mathcal{R}})$ to $\mathcal{R}$ with action $reset_Y!$, guard $\varphi = (rst_Y = n_Y \wedge rep(Y) \geq C_Y)$, and reset vector $\vec{r} = rst_Y := 0, rep(Y) := 0$. In the guard φ, n_Y is the number of automata that reset the clocks of Y, i.e. $n_Y = |\mathcal{RES}_Y(\mathcal{N})|$, and C_Y is the time at which the clocks in Y are reset, i.e., the constant C_Y as described in R1. The result of applying the transformation to the example from Figure 1 is shown in Figure 2.

Note that well-formedness together with the counter variables rst_Y enforce *blocking* multicast synchronisation, that is, always all automata from $\mathcal{RES}_Y(\mathcal{N})$ participate in the reset. Furthermore, $\mathcal{N}'$ is equal to $\mathcal{N}$ if there are no quasi-equal clocks in $\mathcal{N}$.

3.3 A Semantical Characterisation of $\mathcal{N}'$

Following our discussion, we can distinguish two kinds of configurations in the transition system of a well-formed network $\mathcal{N}$. A configuration is unstable if there are quasi-equal clocks with different values, and stable otherwise.

In the following, we observe that our algorithm yields a network whose configurations directly correspond to the stable configurations of $\mathcal{N}$. In addition, we observe that transition sequences in $\mathcal{N}'$ correspond to transition sequences in $\mathcal{N}$ where reset phases of different equivalence classes do not overlap. To this end, we formally define stability of configurations and different notions of reset sequences, in particular full pure reset sequences, i.e., those where reset phases do not overlap.

Definition 4 (Stable Configuration). *Let $\mathcal{N}$ be a well-formed network and let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of quasi-equal clocks.*

A configuration $s \in \mathit{Conf}(\mathcal{N})$ is called stable wrt. Y *if and only if all clocks in Y have the same value in s, i.e., if $\forall\, x \in Y \bullet \nu_s(x) = \nu_s(\mathit{rep}(x))$.*

We use $\mathcal{SC}_Y$ to denote the set of all configurations that are stable wrt. Y and $\mathcal{SC}_\mathcal{N}$ to denote the set $\bigcap_{Y \in \mathcal{EC}_\mathcal{N}} \mathcal{SC}_Y$ of globally stable *configurations.*

Definition 5 (Reset Sequence). *Let $\mathcal{N}$ be a well-formed network and let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of quasi-equal clocks. Let $\sigma = s_0 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} s_n$ be a transition sequence of $\mathcal{N}$ such that s_0 and s_n are stable wrt. Y, $s_1, \ldots, s_{n-1}$ are unstable wrt. Y, and the valuation of every clock in Y at s_n is 0. Then σ is called* Y-reset sequence. *We use $\mathcal{RS}_Y$ to denote the set of Y-reset sequences of $\mathcal{N}$.*

A suffix of σ starting at s_i, $1 \leq i \leq n$, is called a pure Y-reset sequence *if and only if the valuation for some clock of Y changes with every transition, i.e. if $\forall\, i < j \leq n \bullet \nu_{s_{j-1}}|_Y \neq \nu_{s_j}|_Y$. We use $\mathcal{RS}_Y^{pure}$ to denote the set of pure Y-reset sequences of $\mathcal{N}$.*

If σ is in $\mathcal{RS}_Y^{pure}$ and starts in a globally stable configuration, i.e., $s_0 \in \mathcal{SC}_\mathcal{N}$, then σ is called a full pure Y-reset sequence. *We use $\mathcal{RS}_Y^{full}$ to denote the set of full pure Y-reset sequences of $\mathcal{N}$. The smallest suffix of σ which is not in $\mathcal{RS}_Y^{pure}$ is called an* impure Y-reset sequence. *We use $\mathcal{RS}_Y^{impure}$ to denote the set of impure Y-reset sequences of $\mathcal{N}$.*

Let $\sigma = s_0 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} s_n \xrightarrow{\lambda_{n+1}} s_{n+1}$ be a transition sequence of $\mathcal{N}$ such that the prefix of σ up to and including s_n is in $\mathcal{RS}_Y^{pure}$ and s_n and s_{n+1} coincide on Y, i.e. $\nu_{s_n}|_Y = \nu_{s_{n+1}}|_Y$. Then σ is called pure Y-reset sequence-δ.

Proposition 1. *For all well-formed networks $\mathcal{N}$, if $s_0 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} s_n$ is a full pure Y-reset sequence of $\mathcal{N}$, then s_n is globally stable, i.e., $s_n \in \mathcal{SC}_\mathcal{N}$, and the valuation at s_n of each clock $x \in Y$ is 0, i.e., $\nu_{s_n}|_Y = 0$.*

Formally, the relation between a stable configuration $s \in \mathit{Conf}(\mathcal{N})$ of a well-formed network $\mathcal{N}$ and a configuration $r \in \mathit{Conf}(\mathcal{N}')$ of the network $\mathcal{N}' = \mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$ is characterised by the function *reverseQE*. It removes the following from r: the unique location of the automaton $\mathcal{R}$; each counter variable rst_Y, $Y \in \mathcal{EC}_\mathcal{N}$; and it assigns to each clock $x \in Y$ the value of the clock $\mathit{rep}(x) \in Y$.

Definition 6 (*reverseQE* and Consistency). *Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network and let $\mathcal{N}' = \mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$. The function reverseQE : $\mathit{Conf}(\mathcal{N}') \to \mathit{Conf}(\mathcal{N})$ is defined point-wise as follows. Let $r = \langle(\ell_1, \ldots, \ell_n, \ell_{ini,\mathcal{R}}), \nu\rangle \in \mathit{Conf}(\mathcal{N}')$. Then reverseQE$(r) = \langle(\ell_1, \ldots, \ell_n), \tilde{\nu}\rangle$ where*

$$\tilde{\nu} = \big(\nu \cup \textstyle\bigcup_{Y \in \mathcal{EC}_\mathcal{N}} \{x \mapsto \nu(\mathit{rep}(x)) \mid x \in Y\}\big) \setminus \{\mathit{rst}_Y \mapsto \nu(\mathit{rst}_Y) \mid Y \in \mathcal{EC}_\mathcal{N}\}.$$

The configuration r is called Y-consistent *if and only if $\nu_r(\mathit{rst}_Y)$ is the number of reset locations wrt. Y assumed in r, i.e., if $\nu_r(\mathit{rst}_Y) = |\{\ell_1, \ldots, \ell_n, \ell_{ini,\mathcal{R}}\} \cap \mathcal{RL}_Y|$. We use CONS_Y to denote the set of Y-consistent configurations of $\mathcal{N}'$ and $\mathit{CONS}_{\mathcal{N}'}$ to denote the set $\bigcap_{Y \in \mathcal{EC}_\mathcal{N}} \mathit{CONS}_Y$ of* consistent *configurations.*

Proposition 2. *Let $\mathcal{N}$ be a well-formed network. Then reverseQE is a bijection between $\mathit{CONS}_{\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})}$ and $\mathcal{SC}_\mathcal{N}$.*

In the following we define a special transition relation for well-formed networks, which relates stable configurations by collapsing full pure reset sequences. A special transition in $\mathcal{N}$ corresponds to a single transition in $\mathcal{N}' = \mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$.

Definition 7 ($\overset{\lambda}{\Rightarrow}$). *Let $\mathcal{N}$ be a well-formed network and let $s, s' \in \mathcal{SC}_\mathcal{N}$ be two globally stable configurations of $\mathcal{N}$. There is a transition $s \overset{\lambda}{\Rightarrow} s'$ if and only if there is either a delay or τ-transition, or a full pure Y-reset sequence for some $Y \in \mathcal{EC}_\mathcal{N}$ from s to s' in $\mathcal{T}(\mathcal{N})$, i.e., if*

$$s \xrightarrow{\lambda} s' \wedge \lambda \in \mathit{Time} \cup \{\tau\}$$
$$\vee \left(\lambda = \tau \wedge \exists\, s_0 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_n} s_n \in \mathcal{RS}_Y^{full} \bullet s = s_0 \wedge s_n = s'\right).$$

Configuration s is called $\Rightarrow$-reachable *if and only if there are configurations $s_0, \dots, s_n \in \mathit{Conf}(\mathcal{N})$ such that $s_0 \in \mathcal{C}_{ini}$, $s_n = s$, and $s_0 \overset{\lambda_1}{\Longrightarrow} s_1 \dots s_{n-1} \overset{\lambda_n}{\Longrightarrow} s_n$ for some $\lambda_1, \dots, \lambda_n \in \mathit{Time} \cup \{\tau\}$.*

Definition 8 (Weak Bisimulation). *Let $\mathcal{N}_1$ be a well-formed network and $\mathcal{N}_2$ a network with the same set of locations, i.e., $L(\mathcal{N}_1) = L(\mathcal{N}_2)$. Let $\mathcal{T}(\mathcal{N}_i) = (\mathit{Conf}(\mathcal{N}_i), \Lambda, \{\xrightarrow{\lambda}_i \mid \lambda \in \Lambda\}, \mathcal{C}_{ini,i})$, $i = 1, 2$, be the corresponding transition systems restricted to $\Lambda = \mathit{Time} \cup \{\tau\}$.*

A weak bisimulation *is a relation $\mathcal{S} \subseteq \mathit{Conf}(\mathcal{N}_1) \times \mathit{Conf}(\mathcal{N}_2)$ such that*

1. *$\forall\, s \in \mathcal{C}_{ini,1}\ \exists\, r \in \mathcal{C}_{ini,2} \bullet (s, r) \in \mathcal{S}$ and $\forall\, r \in \mathcal{C}_{ini,2}\ \exists\, s \in \mathcal{C}_{ini,1} \bullet (s, r) \in \mathcal{S}$,*
2. *for all $(s, r) \in \mathcal{S}$,*
   - *(a) for all configuration formulae CF over $\mathcal{N}_i$, $s \models CF$ iff $r \models \Gamma(CF)$,*
   - *(b) if $s \xrightarrow{\lambda}_1 s'$, there exists $r' \in \mathit{Conf}(\mathcal{N}_2)$ such that $r \xrightarrow{\lambda}_2 r'$ and $(s', r') \in \mathcal{S}$,*
   - *(c) if $r \xrightarrow{\lambda}_2 r'$, there exists $s' \in \mathit{Conf}(\mathcal{N}_1)$ such that $s \xrightarrow{\lambda}_1 s'$ and $(s', r') \in \mathcal{S}$.*

The networks $\mathcal{N}_1, \mathcal{N}_2$ are called weakly bisimilar*, if and only if there exists a weak bisimulation $\mathcal{S}$ for $\mathcal{T}(\mathcal{N}_1)$ and $\mathcal{T}(\mathcal{N}_2)$.*

Proposition 3. *Let $\mathcal{N}$ be a well-formed network. Let $\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$ be the network output by $\mathcal{K}$. Let $r \in \mathit{Conf}(\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N}))$ and $s \in \mathit{Conf}(\mathcal{N})$ be two configurations, such that $s = \mathit{reverseQE}(r)$. Then, for every $\ell \in \mathcal{L}(\mathcal{N})$, ℓ is the i-th location of ℓ_s if and only if ℓ is the i-th location of ℓ_r, i.e., if*

$$\forall\, \ell \in \mathcal{L}(\mathcal{N}) \bullet \ell = \ell_{s,i} \Leftrightarrow \ell_{r,i} = \ell.$$

Proposition 4. *Let $\mathcal{N}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ be a well-formed network. Let $s \in \mathit{Conf}(\mathcal{N})$ and $r \in \mathit{Conf}(\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N}))$ be two configurations such that $s = \mathit{reverseQE}(r)$. Let $CF_\mathcal{N}$ be the set of configuration formulae over $\mathcal{N}$. Then*

$$\forall\, CF \in CF_\mathcal{N} \bullet s \models CF \iff r \models \Gamma(CF).$$

Theorem 1. *Any well-formed network $\mathcal{N}$ is weakly bisimilar to $\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$.*

Proof. $\{(\mathit{reverseQE}(r), r) \mid r \in \mathit{CONS}_{\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})}\}$ is a weak bisimulation by Definition 8, and Propositions 2, 3 and 4. □

Corollary 1 (Reachability of Stable Configurations). *Let $s \in \mathcal{SC}_\mathcal{N}$ be a stable configuration of the well-formed network $\mathcal{N}$. s is $\Rightarrow$-reachable in $\mathcal{T}(\mathcal{N})$ if and only if $\mathit{reverseQE}^{-1}(s)$ is reachable in $\mathcal{T}(\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N}))$.*

Proof. Theorem 1 and Proposition 2. □

3.4 Handling Impurities and Unstable Configurations

While Corollary 1 allows us to conclude from reachability of a configuration in $\mathcal{N}'$ to the reachability of a corresponding configuration in $\mathcal{N}$, we cannot conclude in the opposite direction. The reason is that $\mathcal{N}'$ misses two things: firstly, computation paths of $\mathcal{N}$ that contain overlapping reset phases are not simulated by any computation path of $\mathcal{N}'$, i.e., $\mathcal{N}'$ would not reach a stable configuration if it were only reachable by computation paths with overlapping reset phases. Secondly, reachability of unstable configurations of $\mathcal{N}$ is not reflected in $\mathcal{N}'$.

In the following we approach the two issues as follows. We argue that, under the additional assumption that the reset edges in $\mathcal{N}$ are pre/post delayed, impure computation paths without timelock can always be reordered into pure ones. Regarding unstable configurations, we firstly observe that – not surprisingly – $\mathcal{N}'$ reflects reachability queries which explicitly ask for stable configurations. In addition, we syntactically characterise the class of local queries, which are reflected by $\mathcal{N}'$ because they cannot distinguish stable and unstable configurations.

Definition 9 (Delayed Edge). *An edge e of a timed automaton $\mathcal{A}$ in network $\mathcal{N}$ is called* delayed *if and only if time must pass before e can be taken, i.e., if*

$$\begin{aligned}&\forall\, s_0 \xrightarrow{\lambda_1}_{E_1} s_1 \ldots s_{n-1} \xrightarrow{\lambda_n}_{E_n} s_n \in \Pi(\mathcal{N}) \bullet e \in E_n\\ &\qquad \implies \exists\, 0 \le j < n \bullet \lambda_j \in \mathit{Time} \setminus \{0\} \wedge \forall\, j \le i < n \bullet E(\mathcal{A}) \cap E_i = \emptyset\end{aligned}$$

where we write $s_i \xrightarrow{\lambda_i}_{E_i} s_{i+1}$, $i \in \mathbb{N}^{>0}$, to denote that the transition $s_i \xrightarrow{\lambda_i} s_{i+1}$ is justified by the set of edges E_i; E_i is empty for delay transitions, i.e. if $\lambda_i \in \mathit{Time}$.

Definition 10 (Reset Pre/Post Delay). *Let $\mathcal{N}$ be a well-formed network. We say $\mathcal{EC}_\mathcal{N}$-*reset edges are pre/post delayed *in $\mathcal{N}$ if and only if all edges originating in reset or reset successor locations are delayed, i.e. if*

$$\forall\, e = (\ell, \alpha, \varphi, \vec{r}, \ell') \in E(\mathcal{N}) \bullet \ell \in \mathcal{RL}_\mathcal{N} \cup \mathcal{RL}_\mathcal{N}^+ \implies e \textit{ is delayed.}$$

There are *sufficient* syntactic criteria for an edge $e = (\ell_1, \alpha_1, \varphi_1, \vec{r}_1, \ell_2)$ being delayed. For instance, if $(\ell_0, \alpha_0, \varphi_0, \vec{r}_0, \ell_1)$ is the only incoming edge to ℓ_1 and if $\varphi_0 = (x \ge C \wedge x \le C)$ and $\varphi_1 = (x \ge D \wedge x \le D)$ and $C < D$, then e is delayed. It is also delayed if $(\ell_0, \alpha_0, \varphi_0, \vec{r}_0, \ell_1)$ is the only incoming edge to ℓ_1, $\vec{r}_0$ is resetting x, and $\varphi_1 = (x > 0)$.

Both patterns occur, e.g., in the FSN case-study (cf. Section 4). There, the reset location is entered via an edge following the former pattern, and the edges originating at the reset successor location follow the latter pattern. Thus $\mathcal{EC}_\mathcal{N}$-reset edges are pre/post delayed in FSN.

Proposition 5. *Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network where $\mathcal{EC}_\mathcal{N}$-resets are pre/post delayed and let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of quasi-equal clocks.*

Let $s \in \mathit{Conf}(\mathcal{N})$ be a reachable configuration of $\mathcal{N}$ which is not timelocked and not stable wrt. Y. Then all automata in $\mathcal{RES}_Y(\mathcal{N})$ are either in a reset or in a reset successor location in s, i.e.

$$\begin{aligned}&\forall\, Y \in \mathcal{EC}_\mathcal{N}\ \forall\, s \in \mathit{Conf}(\mathcal{N}) \setminus \mathcal{SC}_Y\ \forall\, 1 \le i \le n\, \bullet\\ &\qquad \mathcal{A}_i \in \mathcal{RES}_Y(\mathcal{N}) \implies \ell_{s,i} \in \mathcal{RL}_Y \cup \mathcal{RL}_Y^+.\end{aligned}$$

In order to precisely define the concept of reordering of configurations in computation paths, we introduce the following notion of congruence of configurations. Reordering then means that for each impure computation path there exists a pure computation path over congruent configurations.

Definition 11 (Congruent Modulo Y)
Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network with variables $\mathcal{V}$. Two configurations $s_1, s_2 \in \mathit{Conf}(\mathcal{N})$ are called congruent modulo $Y \in \mathcal{EC}_\mathcal{N}$*, denoted by $s_1 \equiv_Y s_2$, if and only if they coincide on values of all variables and if, for each $\mathcal{A} \in \mathcal{N}$, either s_1, s_2 coincide on the location of $\mathcal{A}$ and on the values of clocks from $\mathcal{X}(\mathcal{A})$, or $\mathcal{A}$ is in a reset location in s_1 and in the corresponding reset successor location in s_2, and s_2 has the effect of taking a reset edge, i.e. if*

$$\begin{aligned}&\nu_{s_1}|_\mathcal{V} = \nu_{s_2}|_\mathcal{V} \wedge \big(\forall\, 1 \leq i \leq n \bullet \big(\ell_{1,i} = \ell_{2,i} \wedge \nu_{s_1}|_{\mathcal{X}(\mathcal{A}_i)} = \nu_{s_2}|_{\mathcal{X}(\mathcal{A}_i)}\big)\\ &\quad \vee \textstyle\bigvee_{j=1,2}(\ell_{s_j,i} \in \mathcal{RL}_Y \wedge \ell_{s_{3-j},i} \in \mathcal{RL}_Y^+\\ &\qquad \wedge \exists\, (\ell_{s_j,i}, \alpha, \varphi, \vec{r}, \ell_{s_{3-j},i}) \in E(\mathcal{A}_i) \bullet \nu_{s_j}|_{\mathcal{X}(\mathcal{A}_i)}[\vec{r}] = \nu_{s_{3-j}}|_{\mathcal{X}(\mathcal{A}_i)})\big).\end{aligned}$$

We write $s_1 \equiv_{Y_1,\ldots,Y_n} s_2$ if and only if $s_1 \equiv_{Y_1\cup\cdots\cup Y_n} s_2$ for $Y_1, \ldots, Y_n \in \mathcal{EC}_\mathcal{N}$.

Lemma 2. *Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network where $\mathcal{EC}_\mathcal{N}$-resets are pre/post delayed and let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of quasi-equal clocks.*

Each impure reset sequence which starts in a reachable and ends in a not timelocked configuration can be "reordered" into a reset sequence-δ of Y, i.e.,

$$\forall Y \in \mathcal{EC}_\mathcal{N}\ \forall s_0 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} s_n \in \mathcal{RS}_Y^{impure}\ \exists r_0, \ldots, r_n \in \mathit{Conf}(\mathcal{N}) \bullet r_0 = s_0 \wedge$$
$$r_n = s_n \wedge (\forall\, 0 \leq i < n \bullet r_i \equiv_Y s_0) \wedge r_0 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} r_{n-1} \in \mathcal{RS}_Y^{pure} \wedge r_{n-1} \xrightarrow{\lambda_1} r_n.$$

Proof. Proposition 5 and, by Definition 2 (well-formedness), the reset edges wrt. Y are independent from the edges justifying the transition from s_0 to s_1. □

In Figure 3 we provide an illustration of the reordering of reset sequences from Lemma 2. Subfigure a) represents a Y-reset sequence from s_0 to s_4, where filled circles represent stable configurations and stars unstable configurations. The transition marked with (!) represents an impurity. The suffix from s_1 to s_4 is an impure Y-reset sequence. Subfigure b) shows the result after reordering. The sequence from r_0 to r_4 is a pure Y-reset sequence-δ.

Lemma 3. *Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network where $\mathcal{EC}_\mathcal{N}$-resets are pre/post delayed and let $Y \in \mathcal{EC}_\mathcal{N}$ be a set of quasi-equal clocks.*

Each reset sequence $s_0 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} s_n \in \mathcal{RS}_Y$ of Y where s_0 is reachable and s_n is not timelocked can be "reordered" into a computation path

$$r_0 \xrightarrow{\lambda_{k_1}} r_1 \ldots r_{n-1} \xrightarrow{\lambda_{k_n}} r_n$$

where $k_1, \ldots, k_n$ is a reordering of $1, \ldots, n$, $r_0 = s_0$, and where there exists an index $0 \leq j \leq n$ such that $s_0 \equiv_Y r_i$ for all $0 \leq i \leq j$, $s_i \equiv_Y r_{k_i}$ for all $j < i \leq n$, and $r_0 \xrightarrow{\lambda_{k_1}} \ldots \xrightarrow{\lambda_{k_j}} r_j \in \mathcal{RS}_Y^{pure}$.

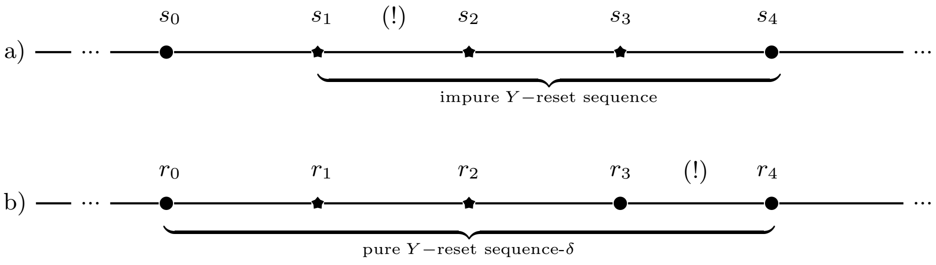


Fig. 3. Reordering of reset sequences

Proof. Apply Lemma 2 inductively from right to left. That is, if the reset sequence of Y has m impure reset sequences of Y, we start the reordering with the m-th impure reset sequence (the shortest one), and finalize with the first (and longest) impure reset sequence of Y. $\square$

Definition 12 (Stability Query). *A configuration formula CF over the well-formed network $\mathcal{N}$ is called* stability query *iff CF is exactly satisfied in stable configurations of $\mathcal{N}$, i.e., if $\forall\, s \in \mathit{Conf}(\mathcal{N}) \bullet s \models \mathit{CF} \iff s \in \mathcal{SC}_{\mathcal{N}}$.*

Proposition 6. *Let $\mathcal{N}$ be a well-formed network.*
Then $\mathit{CF} = \bigwedge_{Y \in \mathcal{EC}_{\mathcal{N}}} \bigwedge_{x \in Y} (x - \mathit{rep}(x) = 0)$ is a stability query over $\mathcal{N}$.

Note that if reset locations are known for a specific network, a stability query could also be stated in terms of those.

Theorem 2. *Let $\mathcal{N}$ be a well-formed network where $\mathcal{EC}_{\mathcal{N}}$-resets are pre/post delayed, let CF be a configuration formula and 'stable' a stability query over $\mathcal{N}$. Then*

$$\mathcal{K}(\mathcal{N}, \mathcal{EC}_{\mathcal{N}}) \models \exists\Diamond\, \Gamma(\mathit{CF}) \iff \mathcal{N} \models \exists\Diamond(\mathit{CF}) \wedge \mathit{stable}.$$

Proof. Use Lemma 3 to obtain a transition sequence which $\Rightarrow$-reaches the witness configuration, then Corollary 1. $\square$

In the following, we bring together Lemmata 2 and 3 to handle computation paths with arbitrary overlaps of reset phases. This allows us to conclude that $\mathcal{N}'$ reflects queries which cannot distinguish stable and unstable configurations.

Lemma 4. *Let $\mathcal{N}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a well-formed network where $\mathcal{EC}_{\mathcal{N}}$-edges are pre/post delayed. Let*

$$\sigma = s_0 \xrightarrow{\lambda_{1,1}} s_{1,1} \ldots \xrightarrow{\lambda_{1,m_1}} s_{1,m_1} \xrightarrow{\lambda_1} s_1 \ldots s_{n-1} \xrightarrow{\lambda_{n,1}} s_{n,1} \ldots \xrightarrow{\lambda_{n,m_n}} s_{n,m_n} \xrightarrow{\lambda_n} s_n$$

be a transition sequence where s_0 is reachable and s_n is not timelocked, where s_i is globally stable, i.e., $s_i \in \mathcal{SC}_{\mathcal{N}}$, $0 \le i \le n$, where for each sub-sequence $\sigma_i = s_{i-1} \xrightarrow{\lambda_{i,1}} s_{i,1} \ldots s_{i,m_i} \xrightarrow{\lambda_i} s_i$, $0 < i \le n$, either s_i has a globally stable *successor, i.e.* $m_i = 0$, *and* $\lambda_i = \tau$, *or* σ_i *is a full pure reset sequence of some* $Y \in \mathcal{EC}_\mathcal{N}$, *and where at* s_0 *starts a full pure reset sequence, i.e.* $m_1 > 0$.

Then σ *can be "reordered" into the transition sequence*

$$\sigma' = r_0 \xrightarrow{\hat{\lambda}_1} r_1 \ldots r_{K-1} \xrightarrow{\hat{\lambda}_K} r_K$$
$$\xrightarrow{\hat{\lambda}_{K+1,1}} r_{K+1,1} \ldots r_{K+1,m_{K+1}} \xrightarrow{\hat{\lambda}_{K+1}} r_{K+1} \ldots r_{n-1} \xrightarrow{\hat{\lambda}_{n,1}} r_{n,1} \ldots r_{n,p_n} \xrightarrow{\hat{\lambda}_{n,p_n}} r_n$$

where $1 \leq i_1 < \cdots < i_K \leq n$ *are the indices of the configurations with globally stable successors, where* $0 \leq j_1, \ldots, j_N \leq n$ *are the indices such that there is a full pure reset sequence of* $Y_k \in \mathcal{EC}_\mathcal{N}$ *between* s_{j_k} *and* s_{j_k+1} *in* σ, *and where the actions not belonging to full pure reset sequences occur first, followed by the full pure reset sequences of* $Y_1, \ldots, Y_N$ *in this order, i.e.* $r_0 = s_0$, $r_k \equiv_{Y_1,\ldots,Y_N} s_{i_k}$ *for* $1 \leq k \leq K$, *and* $r_k \xrightarrow{\hat{\lambda}_{k+1,1}} r_{k+1,1} \ldots r_{k+1,m_{j_k}} \xrightarrow{\hat{\lambda}_{k+1,m_{j_k}}} r_{k+1} \in \mathcal{RS}^{pure}_{Y_{j_k}}$.

Proof. Similar to Lemma 2 using the independence of reset edges from other edges implied by Definition 2, then induction similar to Lemma 3. □

Definition 13 (Local Query). *Let* $\mathcal{N}$ *be a well-formed network. A reachability query* $\exists\Diamond\, CF$ *over* $\mathcal{N}$ *is called* local query *wrt.* $\mathcal{A} \in \mathcal{N}$ *if and only if* CF *is in disjunctive normal form, i.e.* $CF = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} BF_{i,j}$, *and if each atom* $BF_{i,j}$ *is of the form* $\mathcal{A}.\ell$ *with* $\ell \in L(\mathcal{A})$, *or a constraint* φ *with* $clocks(\varphi) \subseteq \mathcal{X}(\mathcal{A})$.

Theorem 3. *Let* $\mathcal{N}$ *be a well-formed network where* $\mathcal{EC}_\mathcal{N}$*-reset edges are pre/post delayed and let* $\exists\Diamond\, CF$ *be a reachability query which is local to* $\mathcal{A} \in \mathcal{N}$.

If there is a configuration reachable in $\mathcal{N}$ *which satisfies* CF *and which is not timelocked, then there is a configuration reachable in* $\mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N})$ *which satisfies* $\Gamma(CF)$, *i.e.* $\mathcal{N} \models_{\neg timelock} \exists\Diamond\, CF \implies \mathcal{K}(\mathcal{N}, \mathcal{EC}_\mathcal{N}) \models \exists\Diamond\, \Gamma(CF)$.

Proof. Use Lemma 4 to obtain a stable configuration which satisfies CF, then Theorem 2 applies. □

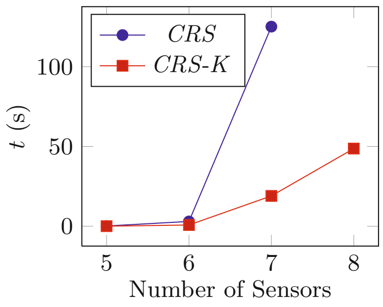
4 Experimental Results

We applied our approach manually to three real world case studies, one of which is an industrial case, and the other two were obtained from the scientific literature. Initially, the six restrictions of well-formedness were motivated by the industrial case, and later generalised to increase the applicability of our approach.

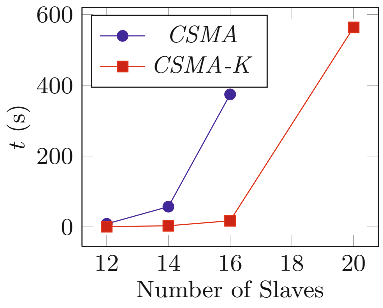
CRS-N is the cascaded ride sharing protocol [13] with N sensors organized in the form of a spanning tree. There exists a sink node that collects data from every sensor. We verified the local query *lessMaxFail*, which states that if a sensor has at least one working communication path, the data sent to the sink node is correctly aggregated. In *lessMaxFail* we only use variables and locations of the sink node.

Table 1. Row 'Clk.' gives the number of clocks in the model, 'States' the number of visited states, 'M.' the memory usage in MB, and 't (s)' the runtime in seconds. (Env.: Intel i3, 2.3GHz, 3GB, Ubuntu 11.04, verifyta 4.1.3.4577 with default options.)

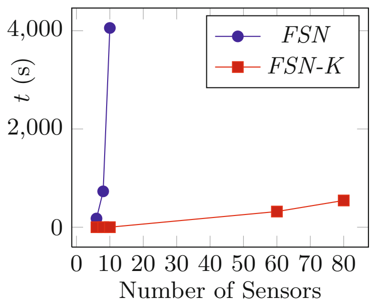
Network	Clk.	States	M. (MB)	t (s)
CRS-5	5	14.9k	17.3	0.2
CRS-5K	1	5.0k	16.2	0.1
CRS-6	6	264.5k	74.7	3.0
CRS-6K	1	64.9k	43.1	0.8
CRS-7	7	7,223.8k	1,986.3	142.1
CRS-7K	1	1,266.4k	693.5	19.2
CRS-8	8	-	-	
CRS-8K	1	2,530.2k	1,543,200	48.7



Network	Clk.	States	M. (MB)	t (s)
CSMA-12	14	688.2k	230.1	8.5
CSMA-12K	1	49.3k	36.3	0.5
CSMA-14	16	3,670.1k	1,257.6	57.0
CSMA-14K	1	229.5k	135.6	3.1
CSMA-16	18	18,874.5k	7,051.7	374.3
CSMA-16K	1	1,048.7k	597.5	17.2
CSMA-20	22	-	-	-
CSMA-20K	1	20,971.7k	10,589.0	563.5



Network	Clk.	States	M. (MB)	t (s)
FSN-6	12	2,149.9k	302.1	176.6
FSN-6K	5	0.9k	16.6	0.1
FSN-8	14	5,084.3k	643.1	729.9
FSN-8K	5	0.9k	17.0	0.1
FSN-10	16	17,474.7k	2,069.4	4057.1
FSN-10K	5	0.9k	17.4	0.1
FSN-60K	5	4,239.4k	454.4	318.4
FSN-80K	5	5,604.4k	611.3	543.9



CSMA-N is the model of the CSMA/CD protocol [14] with N slaves and one master. We verified the local query *noCollision*, which states that no collision occurs when slaves send data to the master. We have represented the occurrence of collisions by a location of the master.

FSN-N is a custom TDMA-based wireless fire alarm system with N sensors [15]. We verified the local query *300seconds*, which states that a sensor malfunction is detected by the central unit (main sensor) in at most 300 seconds. In this query we use a clock and a location from the central unit.

Table 1 gives the figures for verification before and after applying the transformation from Section 3.2, the latter figures are indicated by suffix K at the network name.

5 Conclusion and Future Work

We have presented a transformation approach to mechanically remove verification overhead from networks of timed automata where clocks in the real world are intended to be synchronized. We formally introduced the notion of *quasi-equal* clocks to characterise such clocks. We propose a transformation that goes beyond simple syntactical replacement. We formally prove the correctness of our approach and define a class of timed automata networks and reachability queries for which it is applicable. Although well-formedness imposes a set of restrictions over the networks where we can apply our approach, this is reasonable since the semantics of well-formed networks are preserved after transformation, up to configurations where quasi-equal clocks have different valuations.

Experiments with real-world case studies show the feasibility of reducing clocks in networks of timed automata based on quasi-equal clocks. Significant gains in the computational cost of model checking using *Uppaal* for transformed models are achieved, once eliminated the unnecessary overhead caused when well-formed networks generate intermediate configurations by resetting quasi-equal clocks one by one. We enable an increase in decoupling between modelling as a design and documentation activity, and model optimization for verification. Thus, the approach effectively narrows the gap between readable, maintainable models and model checking efficiency.

In the future, we would like to enlarge the spectrum of networks that can be treated by our approach by investigating relaxations of the well-formedness criteria presented. Additionally, we would like to extend the types of queries supported by the transformation by providing a broader syntax for validable queries beyond simple reachability. Finally, an automatic detection of quasi-equal clocks would increase the mechanisation of our approach.

References

1. Alur, R., Dill, D.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Rappaport, T.S.: Wireless communications: principles and practice, vol 2. Prentice Hall (2002)
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Braberman, V., Garbervestky, D., Kicillof, N., Monteverde, D., Olivero, A.: Speeding Up Model Checking of Timed-Models by Combining Scenario Specialization and Live Component Analysis. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 58–72. Springer, Heidelberg (2009)
5. Braberman, V.A., Garbervetsky, D., Olivero, A.: Improving the Verification of Timed Systems Using Influence Information. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 21–36. Springer, Heidelberg (2002)
6. Bozga, M., Fernandez, J.-C., Ghirvu, L.: State Space Reduction Based on Live Variables Analysis. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 164–178. Springer, Heidelberg (1999)

7. Lugiez, D., Niebert, P., Zennou, S.: A Partial Order Semantics Approach to the Clock Explosion Problem of Timed Automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 296–311. Springer, Heidelberg (2004)
8. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata. In: RTSS, pp. 73–81. IEEE (1996)
9. Daws, C., Tripakis, S.: Model Checking of Real-Time Reachability Properties Using Abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
10. Muñiz, M., Westphal, B., Podelski, A.: Timed Automata with Disjoint Activity. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 188–203. Springer, Heidelberg (2012)
11. Salah, R., Bozga, M., Maler, O.: Compositional timing analysis. In: EMSOFT, pp. 39–48. ACM (2009)
12. Olderog, E.-R., Dierks, H.: Real-time systems - formal specification and automatic verification. Cambridge University Press (2008)
13. Gobriel, S., Khattab, S., Mossé, D., Brustoloni, J., Melhem, R.: Ridesharing: Fault tolerant aggregation in sensor networks using corrective actions. the 3rd annual. In: IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), pp. 595–604 (2006)
14. Jensen, H., Larsen, K., Skou, A.: Modelling and analysis of a collision avoidance protocol using SPIN and Uppaal. In: 2nd SPIN Workshop (1996)
15. Dietsch, D., Feo-Arenis, S., Westphal, B., Podelski, A.: Disambiguation of industrial standards through formalization and graphical languages. In: RE, pp. 265–270 (2011)

SMT-Based Induction Methods for Timed Systems

Roland Kindermann, Tommi Junttila, and Ilkka Niemelä

Aalto University
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
{Roland.Kindermann,Tommi.Junttila,Ilkka.Niemela}@aalto.fi

Abstract. Modeling time-related aspects is important in many applications of verification methods. For precise results, it is necessary to interpret time as a dense domain, e.g. using timed automata as a formalism, even though the system's resulting infinite state space is challenging for verification methods. Furthermore, fully symbolic treatment of both timing related and non-timing related elements of the state space seems to offer an attractive approach to model checking timed systems with a large amount of non-determinism. This paper presents an SMT-based timed system extension to the IC3 algorithm, a SAT-based novel, highly efficient, complete verification method for untimed systems. Handling of the infinite state spaces of timed system in the extended IC3 algorithm is based on suitably adapting the well-known region abstraction for timed systems. Additionally, k-induction, another symbolic verification method for discrete time systems, is extended in a similar fashion to support timed systems. Both methods are evaluated and experimentally compared to a booleanization-based verification approach that uses the original discrete time IC3 algorithm.

1 Introduction

In many application areas of model checking, such as analysis of safety instrumented systems, modeling and analyzing in the presence of dense time constructions such as timers and delays is essential. Compared to finite state systems, such timed systems add an extra layer of challenge for model checking tools. In many cases, timed automata [1,2,3] are a convenient formalism for describing and model checking timed systems. There are many tools, Uppaal [4] to name just one, for timed automata and model checking algorithms for timed automata have been studied extensively during the last two decades, see, e.g., [3] for an overview. Most state-of-the-art model checking systems for timed automata use the so-called region abstraction to make a finite state abstraction of the dense time clocks in the automata. These regions are then manipulated symbolically with difference bounded matrices or decision diagram structures (see, e.g., [5]).

In this paper our focus is on model checking of safety instrumented systems (see, e.g., [6]). Such systems have features that are challenging for the classic timed automata based approach described above. First, safety instrumented systems do typically involve a substantial number of timing related issues. However, such systems are often not best described using automata-like control structures but with a sequential circuit-like control logic. This makes the use of timed automata rather inconvenient in modeling. Second, such systems tend to have a relatively large amount of non-deterministic input signals which are computationally challenging for model checking tools based on explicit state representation of discrete components (i.e. control location and data).

Hence, we are interested in developing model checking techniques that complement the automata based methods to address these issues. Instead of timed automata, we use a more generic symbolic system description formalism [7] which can be seen as an extension of the classic symbolic transition systems [8] with dense time clock variables and constraints. In our previous work [7,9], we have experimented with (i) SMT-based bounded model checking (BMC) [10,11], and (ii) BDD-based model checking based on booleanization of the region abstracted model. These methods were not totally satisfactory as (i) BMC can, in practice, only find bugs, not prove correctness of the system, and (ii) the BDD-based method does not seem to scale well to realistically sized models.

In order to address the computational challenge to develop model checking techniques that can handle timing as well as a substantial amount of non-deterministic input signals and prove correctness, we turn to inductive techniques. The motivation here is the success of temporal induction [12,13] and, especially, of the IC3 algorithm [14] in the verification of finite state hardware systems. Our approach is to employ SMT solvers instead of SAT solvers as the basic constraint solver technology and apply symbolic region abstraction to handle the dense time clocks in the models. We extend IC3 to timed systems by using linear arithmetics instead of propositional logic and by lifting the concrete states found by the SMT solver to symbolic region level constraints that are further used in the subsequent steps to constrain the search. As a result we obtain a version of IC3 that does not explicitly construct the symbolic region abstracted system but still can exclude whole regions of states at once. We also describe an SMT-based extension of the k-induction algorithm to these kinds of timed systems. In addition, we develop optimizations that allow us to exclude more regions at a time in the SMT-based IC3 algorithm, and to use stronger "simple path" constraints in k-induction.

Our experimental results indicate that SMT-based IC3 can indeed prove much more properties and on much larger models than were possible with our earlier approaches or with SMT-based timed k-induction. Furthermore, when comparing to the approach of using the original propositional IC3 on booleanized region abstracted model, we observe that using richer logics in the SMT framework makes the IC3 algorithm scale much better for timed systems. However, IC3 seems to perform worse than k-induction (and thus BMC) in finding counter-examples to properties that do not hold. This is probably due to its backwards DFS search nature, and leads us to the conclusion of recommending the use of a portfolio approach combining SMT-based BMC and IC3 when model checking these kinds of safety instrumented systems.

2 Symbolic Timed Transition Systems and Regions

To model timed systems we use symbolic timed transition systems[1] (STTS) [7], a generic formalism allowing modeling of arbitrary control logic structures, data manipulation, and non-deterministic external inputs. In a nutshell, STTSs can be seen as

[1] Not to be confused with similarly named concepts, e.g., the symbolic transition relation of a timed automaton [15].

symbolic transition systems [8] extended with real-valued clocks and associated constraints; they are also quite similar to the "Finite State Machines with Time" of [16]. By using encoding techniques similar to those in [10,11], timed automata (and networks of such) can be efficiently translated into STTS [7].

In the following, we use standard concepts of propositional and first-order logics. We assume typed (i.e., sorted) logics, and that formulas are interpreted modulo some background theories (in particular, linear arithmetics over reals); see, e.g., [17] and references therein. If $Y = \{y_1, ..., y_l\}$ is a set of variables and ϕ formula over Y, then $Y' = \{y'_1, ..., y'_l\}$ is the set of corresponding similarly typed *next-state variables* and ϕ' is obtained from ϕ by replacing each variable y_j with y'_j. Similarly, if ψ is a formula over $Y \cup Y'$, then, for each $i \in \mathbb{N}$, the formula $\psi^{[i]}$ is obtained by replacing y_j with $y_j^{[i]}$ and y'_j with $y_j^{[i+1]}$, of the same types. For example, if $\psi = (c'_2 \leq c_1 + \delta) \land x'_1$, then $\psi^{[4]} = (c_2^{[5]} \leq c_1^{[4]} + \delta^{[4]}) \land x_1^{[5]}$.

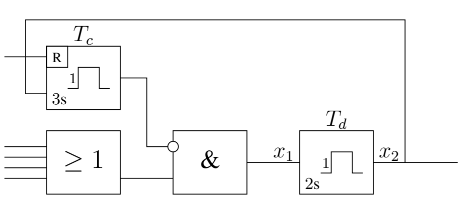
An STTS (or simply a system) is a tuple $\langle X, C, \mathit{Init}, \mathit{Invar}, T, R\rangle$, where

- $X = \{x_1, ..., x_n\}$ is a finite set of finite domain *state variables*,
- $C = \{c_1, ..., c_m\}$ is a finite set of real-valued clock variables (or simply *clocks*),
- Init is a formula over X describing the initial states of the system,
- Invar is a formula over $X \cup C$ specifying a state invariant (throughout the paper, we assume the state invariants to be convex, as defined later),
- T is the *transition relation formula* over $X \cup C \cup X'$, and
- R associates with each clock $c \in C$ a *reset condition* formula r_c over $X \cup C \cup X'$.

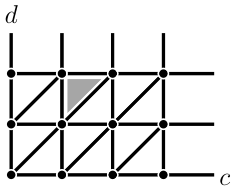
Like in timed automata context, we require that in all the formulas in the system the use of clock variables is restricted to atoms of the form $c \bowtie n$, where $c \in C$ is a clock variable, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{Z}$. Observe that, as in the timed automata context as well, one could use rational constants in systems and then scale them to integers in a behavior and property preserving way. A system is *untimed* if it does not have any clock variables. For the sake of readability only, we do not consider the so-called urgency constraints [7] in this paper.

The semantics of an STTS is defined by its states and how they may evolve to others. A *state* is simply an interpretation over $X \cup C$. A state s is *valid* if it respects the state invariant, i.e. $s \models \mathit{Invar}$. A state s is an *initial state* if it is valid, $s \models \mathit{Init}$, and $s(c) = 0$ for each clock $c \in C$. Given a state s and $\delta \in \mathbb{R}_{\geq 0}$, we denote by $s + \delta$ the state where clocks have increased by δ, i.e. $(s + \delta)(c) = s(c) + \delta$ for each clock $c \in C$ and $(s+\delta)(x) = s(x)$ when $x \in X$. A valid state s may evolve into a successor state u, denoted by $s \longrightarrow u$, if u is also valid and either of the following holds:

1. *Discrete step*: (i) the current and next state interpretations evaluate the transition relation to true, i.e. $\gamma \models T$ where $\gamma(y) = s(y)$ when $y \in X \cup C$ and $\gamma(x') = u(x)$ when $x' \in X'$, and (ii) each clock either resets or keeps its value: for each clock $c \in C$, $u(c) = 0$ if $\gamma \models r_c$ and $u(c) = s(c)$ otherwise.
2. *Time elapse step*: (i) some amount of time elapses: $u = s + \delta$ for some $\delta \in \mathbb{R}_{\geq 0}$, and (ii) the state invariant is respected in the states in between: $s + \mu$ is valid for all $0 < \mu \leq \delta$.



(a) A part of a timed safety instrumented system, taken from [18]



(b) Clock regions

Fig. 1. Illustrations of safety instrumented systems and regions

A path is a finite sequence $s_0 s_1 ... s_l$ of states such that $s_i \longrightarrow s_{i+1}$ for each consecutive pair of states in the path. A state is reachable if there is a path from an initial state to that state. A *property* P is a formula over the state variables X and clock variables C, adhering to the same restrictions on the use of clock variables as the system's formulas. In this paper we are interested in solving the problem whether the given state property P is an invariant, i.e. whether P holds in all reachable states of the system.

As in the context of timed automata, we require the state invariants in STTSs to be *convex*: for all states s and for all $0 \leq \eta \leq \delta$ it should hold that whenever $s \models \mathit{Invar}$ and $(s + \delta) \models \mathit{Invar}$, also $(s + \eta) \models \mathit{Invar}$. Thus, a state invariant cannot become false and then true again during a time-elapse step, making condition (ii) of time-elapse steps to always hold. Convexity is easy to test with one call to an SMT solver.

Example 1. Consider an STTS modeling the timer T_d in the safety instrumented system in Fig. 1(a). The STTS has the clock variable d which is reset when a discrete step makes the signal x_1 true, i.e. $r_d = (\neg x_1 \wedge x_1')$, corresponding to the activation of the timer. The output signal x_2 is initially false, i.e. *Init* contains the conjunct $(\neg x_2)$. It changes to true when the signal x_1 does and then stays true for two seconds, captured by the conjunct $(x_2' \Leftrightarrow (\neg x_1 \wedge x_1') \vee (x_2 \wedge (d < 2)))$ in the transition relation T. To force the timer output to be reset after two seconds, *Invar* contains the conjunct $(x_2 \Rightarrow (d \leq 2))$.

Regions. A conceptual tool for handling the infinite state space of an STTS is the region abstraction [1]. Given an $a \in \mathbb{R}_{\geq 0}$, let $\mathrm{fract(a)}$ be its fractional part, i.e. $a = \lfloor a \rfloor + \mathrm{fract(a)}$ and $0 \leq \mathrm{fract(a)} < 1$. Let m_c be the maximum (relevant) value of the clock c, i.e. the largest constant that c is compared to in Invar, T, R or P. A *clock valuation* is an interpretation over C. Two clock valuations v and w belong to the same equivalence class called *region*, denoted by $v \sim w$, if for all clocks $c, d \in C$

1. either (i) $\lfloor v(c) \rfloor = \lfloor w(c) \rfloor$ or (ii) $v(c) > m_c$ and $w(c) > m_c$;
2. if $v(c) \leq m_c$, then $\mathrm{fract(v(c))} = 0$ iff $\mathrm{fract(w(c))} = 0$; and
3. if $v(c) \leq m_c$ and $v(d) \leq m_d$, then $\mathrm{fract(v(c))} \leq \mathrm{fract(v(d))}$ iff $\mathrm{fract(w(c))} \leq \mathrm{fract(w(d))}$.

Essentially, points 1 and 2 ensure that any clock constraint is either satisfied by all interpretations within the same region or violated by all interpretations in that region and 3 ensures that the next region reached as time passes is the same for all interpretations. Figure 1(b) illustrates the region abstraction for an STTS with two clocks, c and d, with $m_c = 3$ and $m_d = 2$. The thick black lines, thick black dots and the areas in between the thick black lines each represent a different region. The region in which $\lfloor v(c) \rfloor = \lfloor v(d) \rfloor = 1$ and $0 < \mathrm{fract(v(c))} < \mathrm{fract(v(d))}$ is highlighted in gray. Two states, s and u, are in the same region, denoted by $s \sim u$, if they agree on the values of the state variables and are in the same region when restricted to clock variables.

Due to the restrictions imposed on the use of clock variables, states in the same region are (i) indistinguishable for predicates, meaning that $u \models \mathit{Init}$ iff $s \models \mathit{Init}$, $u \models \mathit{Invar}$ iff $s \models \mathit{Invar}$, and $u \models P$ iff $s \models P$ whenever $s \sim u$, and (ii) forward bisimilar: if $s \longrightarrow s'$ and $s \sim u$, then there exists a u' such that $u \longrightarrow u'$ and $s' \sim u'$.

Formula Representation with Combined Steps. To simplify the exposition, to reduce amount of redundancy in paths, and to enable some optimizations, we introduce a *formula representation* for STTSs that exploits a well-known observation: for reachability checking, it is enough to consider paths where discrete steps and time elapse steps alternate, as two consecutive time elapse steps can be merged into one and zero duration time elapse steps can be added in between discrete steps. For a given STTS $\langle X, C, \mathit{Init}, \mathit{Invar}, T, R\rangle$, we define the following formulas:

- $\widehat{\mathit{Invar}} := \mathit{Invar} \wedge \bigwedge_{c \in C} c \geq 0$. Now $s \models \widehat{\mathit{Invar}}$ for a state s iff s is a valid state and all clock values are non-negative.
- $\widehat{\mathit{Init}} := \mathit{Init} \wedge \bigwedge_{c \in C} c = \hat{c}$ for a fresh, otherwise unconstrained real-valued variable $\hat{c}$. Now $s \models \widehat{\mathit{Init}}$ iff, forgetting the state validity requirements, s is a state reachable from an initial state with time elapse steps only.
- $\widehat{T} := T \wedge \delta \geq 0 \wedge \bigwedge_{c \in C}(r_c \Rightarrow c' = \delta) \wedge \bigwedge_{c \in C}(\neg r_c \Rightarrow c' = c + \delta)$ with δ being a fresh real-valued variable. Thus, a state u is reachable from a state s with one discrete step followed by one time elapse step iff $\pi \models \widehat{T}$ for the valuation π on $X \cup C \cup X' \cup C'$ mapping each $z \in X \cup C$ to $s(z)$ and each $z' \in X' \cup C'$ to $u(z)$.

3 k-Induction for Timed Systems

The k-induction method [12,13], originally proposed for finite-state systems, inductively proves a reachability property for a system or discovers a counter-example while trying to prove the property. In the following, we extend k-induction to a complete verification method for STTS in a similar way as done in [19].

As the base case of an inductive proof, k-induction shows that no bad state can be reached within k steps starting from an initial state for some $k \in \mathbb{N}$. As the inductive step, k-induction shows that it is impossible to have a path consisting of k good (property-satisfying) states followed by a bad (property-violating) state. Together, these steps prove that the property holds in any reachable state.

For an untimed system $\langle X, \emptyset, \mathit{Init}, \mathit{Invar}, T, \emptyset\rangle$, both the base case and inductive step can be proven using a SAT solver. The base case holds iff the formula $\mathit{Init}^{[0]} \wedge \bigwedge_{i=0}^{k} \mathit{Invar}^{[i]} \wedge \bigwedge_{i=0}^{k-1} T^{[i]} \wedge \bigwedge_{i=0}^{k-1} P^{[i]} \wedge \neg P^{[k]}$ is unsatisfiable. Likewise, the inductive step holds iff the formula $\bigwedge_{i=0}^{k} \mathit{Invar}^{[i]} \wedge \bigwedge_{i=0}^{k-1} T^{[i]} \wedge \bigwedge_{i=0}^{k-1} P^{[i]} \wedge \neg P^{[k]}$ is unsatisfiable. Initially, k-induction attempts an inductive proof with $k = 0$. If unsuccessful, k is increased until the inductive proof succeeds or a counter-example is found while checking the base case. Note that the large overlap both between the formulas for checking base case and inductive step and between the checks before and after increasing k can be exploited by incremental SAT solvers [13].

While correct, the described approach is not complete as the induction step is not guaranteed to hold even if the property checked is satisfied by the system. For finite-state systems k-induction can be made complete by only considering simple (non-looping) paths when checking the inductive step. This can be done by adding a quadratic number disequality constraints to the SAT formula, requiring any pair of states to be distinct. Experimental evidence, however, suggests that it is beneficial to add disequality constraints for pairs of states only when observed that they are actually needed [13].

k-Induction for STTS. Both the base case and the inductive step formula can be applied to an STTS $\langle X, C, \mathit{Init}, \mathit{Invar}, T, R\rangle$ simply by replacing Init, T and Invar in these formulas by $\widehat{\mathit{Init}}$, $\widehat{T}$ and $\widehat{\mathit{Invar}}$ and using an SMT solver instead of a SAT solver. However, unlike for untimed systems, termination is not guaranteed even when adding disequality constraints. For untimed systems, disequality constraints guarantee termination because in a finite state system there are no simple paths of infinite length and, thus, the simple path inductive step check is guaranteed to be unsatisfiable with sufficiently large k. Timed systems, in contrast, typically have no upper bound for the length of a simple path and, thus, disequality constraints are not sufficient for completeness. However, the infinite state space of an STTS can be split into a finite number of regions. Thus, any reasoning made for finite state systems can be applied to regions of states. In particular, k-induction is complete and correct when only paths that do not visit two states belonging to the same region are considered in the inductive step [19]. By enforcing this property on inductive step paths using region-disequality constraints, complete k-induction can be performed using $\widehat{\mathit{Init}}$, $\widehat{T}$ and $\widehat{\mathit{Invar}}$ (almost) without modification.

In order to specify that two states of an STTS belong to different regions, region-disequality constraints need to individually constrain the integer and fractional parts of clock values. As only some SMT-solvers, such as Yices [20], allow referring to integer and fractional parts of real-valued variables, we provide a region-disequality constraint encoding that does not rely on such a feature.[2] Instead, we split each clock variable c (and the difference variable δ) into two variables: c_{int} represents the integer and c_{fract} the fractional part of c's value. This "splitting of clocks" requires rewriting of $\widehat{\mathit{Init}}$, $\widehat{T}$ and $\widehat{\mathit{Invar}}$ by replacing each atom involving a clock with a formula as follows:

Atom	Replacement, $n \in \mathbb{N}$	Atom	Replacement, $n \in \mathbb{N}$
$c < n$	$c_{\text{int}} < n$	$c \leq n$	$c_{\text{int}} < n \lor (c_{\text{int}} = n \land c_{\text{fract}} = 0)$
$c > n$	$c_{\text{int}} > n \lor (c_{\text{int}} = n \land c_{\text{fract}} > 0)$	$c \geq n$	$c_{\text{int}} \geq n$
$c = n$	$c_{\text{int}} = n \land c_{\text{fract}} = 0$	$c = \hat{c}$	$c_{\text{int}} = \hat{c}_{\text{int}} \land c_{\text{fract}} = \hat{c}_{\text{fract}}$
$c' = \delta$	$c'_{\text{int}} = \delta_{\text{int}} \land c'_{\text{fract}} = \delta_{\text{fract}}$		
$c' = c + \delta$	$((c_{\text{fract}} + \delta_{\text{fract}} < 1) \Rightarrow (c'_{\text{int}} = c_{\text{int}} + \delta_{\text{int}} \land c'_{\text{fract}} = c_{\text{fract}} + \delta_{\text{fract}})) \land ((c_{\text{fract}} + \delta_{\text{fract}} \geq 1) \Rightarrow (c'_{\text{int}} = c_{\text{int}} + \delta_{\text{int}} + 1 \land c'_{\text{fract}} = c_{\text{fract}} + \delta_{\text{fract}} - 1))$		

[2] In [9] we give an alternative encoding for region-disequality constraints in a BMC setting.

Then, two states with indices i and j can be forced to be in different regions by the following region-disequality constraint $\mathit{DiffRegion}^{[i,j]}$:

$$\begin{aligned}\bigvee_{x\in X} x^{[i]} \neq x^{[j]} \vee \bigvee_{c\in C} (c^{[i]}_{\text{int}} \neq c^{[j]}_{\text{int}} \wedge (\neg max^{[i]}_c \vee \neg max^{[j]}_c))\\ \vee \bigvee_{c\in C} (\neg max^{[i]}_c \wedge \neg(c^{[i]}_{\text{fract}} = 0 \Leftrightarrow c^{[j]}_{\text{fract}} = 0))\\ \vee \bigvee_{c\in C} \bigvee_{d\in C\setminus\{c\}} (\neg max^{[i]}_c \wedge \neg max^{[i]}_d \wedge \neg((c^{[i]}_{\text{fract}} \leq d^{[i]}_{\text{fract}}) \Leftrightarrow (c^{[j]}_{\text{fract}} \leq d^{[j]}_{\text{fract}})))\end{aligned}$$

where the shorthand $max^{[i]}_c := c^{[i]}_{\text{int}} > m_c \vee (c^{[i]}_{\text{int}} = m_c \wedge c_{\text{fract}} > 0)$ detects whether the clock c exceeds its maximum relevant value m_c.

4 IC3 for Timed Systems

In this section, we first describe the IC3 algorithm [14] for untimed finite state systems (see also [21] for an alternative, complementary account of the algorithm). We then introduce a timed system extension using region abstraction and SMT solvers.

Like k-induction, the IC3 algorithm tries to generate an inductive proof for a given state property P on an untimed system $S = \langle X, \emptyset, \mathit{Init}, \mathit{Invar}, T, \emptyset\rangle$. But unlike the unrolling-based approach used by k-induction, proofs generated by the IC3 algorithm only consists of a single formula Proof satisfying three properties: (a) Proof is satisfied by any initial state of S, (b) Proof is satisfied by any successor of any state satisfying Proof, and (c) $\mathit{Proof} \Rightarrow P$. Properties (a) and (b) serve as base case and inductive step for showing that the set of states satisfying Proof is an over-approximation of the states reachable in S while property (c) proves that any reachable state satisfies P.

In order to generate a proof, the IC3 algorithm builds a sequence of sets of formulas $F_0 \ldots F_k$ satisfying certain properties. Eventually, one of these sets becomes the proof Proof. Each F-set represents the set of states satisfying all its formulas. The properties satisfied by the sequence are (i) $\mathit{Init} \wedge \mathit{Invar} \Rightarrow F_0$, (ii) $F_i \Rightarrow F_{i+1}$, (iii) $F_i \Rightarrow P$, and (iv) $F_i \wedge \mathit{Invar} \wedge T \wedge \mathit{Invar}' \Rightarrow F'_{i+1}$. The basic strategy employed by the IC3 algorithm is to add clauses to the F_i-sets in a fashion that keeps properties (i) to (iv) intact until $F_k \wedge \mathit{Invar} \wedge T \wedge \mathit{Invar}' \Rightarrow P'$. In this situation, k can be increased by appending $\{P\}$ to the sequence. The algorithm terminates once $F_i = F_{i+1}$ for some i and provides F_i as a proof. Upon termination, properties (i) and (ii) imply proof-property (a), property (iv) and the termination condition $F_i = F_{i+1}$ imply property (b) and property (iii) implies property (c). Note that, in practice, property (ii) is enforced by adding any formula added to a given F-set also to all F-sets with lower index, i.e. $F_i \subseteq F_{i-1}$.

After sketching the basic strategy, we will now take a closer look at the algorithm. Note, however, that only a simplified version of the algorithm which focuses on aspects relevant for the extension for STTS is described here. Figure 2(a) shows the main loop of the IC3 algorithm. Each iteration, the algorithm first checks whether it is currently possible to extend the sequence of F-sets by appending P. Note that as appending P will never result in properties (i) to (iii) being violated, it is sufficient to check whether

```
1: loop
2:     if $F_k \wedge Invar \wedge T \wedge Invar' \wedge \neg P'$ is UNSAT then
3:         $k := k + 1$
4:         add $F_k \leftarrow \{P\}$ to sequence of $F$-sets
5:         propagate()
6:         if $F_i = F_{i+1}$ for some $i$ then
7:             return true {Property holds}
8:     else
9:         $s \leftarrow$ predecessor of a bad state extracted from the model
10:        $success \leftarrow blockState(s)$
11:        if $\neg success$ then
12:            return false {Prop. violated}
```

(a) The main loop of IC3

```
1: $Q \leftarrow$ priority queue containing $\langle s, k \rangle$
2: while $Q$ not empty do
3:     $s, i \leftarrow Q.popMin()$
4:     if $i = 0$ then
5:         return false {Counter-example found}
6:     if $F_{i-1} \wedge \neg s \wedge T \wedge Invar \wedge Invar' \wedge s'$ is SAT then
7:         $z \leftarrow$ predecessor of $s$ extracted from model
8:         $Q.add(\langle s, i \rangle)$
9:         $Q.add(\langle z, i-1 \rangle)$
10:    else
11:        $t \leftarrow generalize(\neg s)$
12:        Add $t$ to $F_0 \dots F_i$
13:        if $i < k$ then
14:            $Q.add(\langle s, i+1 \rangle)$
15: return true
```

(b) The $blockState(s)$ sub-routine

Fig. 2. The IC3 algorithm

extending the sequence would violate property (iv). A corresponding SAT call is made in Line 2 of Fig. 2(a). If this call indicates that the sequence can safely be extended, it is extended in Lines 3 and 4. In the next step, clauses may be propagated from F-sets to subsequent sets in the sequence. While this step is vital for termination, a more detailed description is omitted here for space limitations. After propagation, the algorithm's termination condition is checked in Line 6.

Of course, the SAT check in Line 2 may as well indicate that the F-sequence may currently not be extended without violating property (iv). In this case, a state s that satisfies F_k and has a bad successor can be extracted from the model returned by the SAT solver. As s prevents the sequence from being extended, the algorithm attempts to drop s from (the set of states represented by) F_k by adding a clause that implies $\neg s$.[3] The corresponding subroutine call, $blockState(s)$, may need to add further clauses also to other F-sets than F_k to ensure that the properties of the sequence remain satisfied.

The $blockState(s)$ subroutine, outlined in Fig. 2(b), operates on a list of proof obligations, each being a pair of a state and an index. An obligation $\langle s, i \rangle$ indicates that it is necessary to drop s from F_i before the main loop of the algorithm can continue. Initially, the only proof obligation is to drop the state provided as an argument from F_k. For any proof obligation $\langle s, i \rangle$, the $blockState$ subroutine in Line 6 checks whether s has a predecessor z in F_{i-1}. Such a predecessor prevents s from being excluded from F_i without violating property (iv). Thus, if a predecessor is found, the obligation $\langle s, i \rangle$ can not be fulfilled immediately and is added to the set of open obligations again in Line 8. Furthermore, z has to be excluded from F_{i-1} before s can be excluded from F_i. Thus, obligation $\langle z, i-1 \rangle$ also being added to the open obligations in Line 9.

If the SAT call in Line 6 is unsatisfiable, then s has no predecessor in F_{i-1} and can safely be excluded from F_i without violating property (iv). Then, s is excluded by adding a generalization of the clause $\neg s$ to F_i. More precisely, the algorithm attempts to drop literals from $\neg s$ in a way that preserves properties (i) and (iv) before adding the resulting clause to F_i. Without this generalization step, states would be excluded one at a time from the F-sets resulting in a method akin to explicit state model checking.

[3] Abusing the notation, we interpret a state s as formula $\bigwedge_{y \in C \cup X} y = s(y)$ where appropriate.

So far, it has been assumed that P holds. If this is not the case, the main loop will eventually pass a predecessor of a bad state reachable in S to $blockState$. Then, $blockState$ essentially performs a backwards depth-first search that eventually leads to an initial state of S, which is detected in Line 4. It is straightforward to extract a counter-example from the proof obligations if it is detected that P does not hold.

Note that, while sufficient for explaining our extensions, only a simplified version of the IC3 algorithm has been described. Most notably, the complete version of the algorithm additionally aims to satisfy proof obligations for multiple successive F-sets at a time if possible and performs generalization based on unsatisfiable cores obtained from SAT calls in various locations. For a description of these techniques as well as complete arguments for correctness and completeness of the approach refer to [14,21].

Extending IC3 for timed systems. As was the case with k-induction, the key to extending the IC3 algorithm to timed systems is the region abstraction. Again, we will use an SMT-solver instead of a SAT-solver and the combined step encoding $\widehat{Init}$, $\widehat{Invar}$, and $\widehat{T}$ on an STTS will replace $Init$, $Invar$ and T. To operate on the region level, we lift each concrete state in a satisfying interpretation returned by the SMT solver into to the region level in the IC3 algorithm code whenever it is passed back to the SMT solver again. To do this, given a state s, we construct a conjunction $\tilde{s}$ of atoms such that $\tilde{s}$ represents all the states in the same region as s, i.e. for any state u it holds $u \models \tilde{s}$ iff $u \sim s$. Formally, $\tilde{s}$ is the conjunction of the atoms given by the following rules:

1. For each state variable $x \in X$, add the atom $(x = s(x))$.
2. For each clock c with $s(c) > m_c$, add the atom $(c > m_c)$.
3. For each clock c with $s(c) \leq m_c$ and $\mathrm{fract(s(c))} = 0$, add the atoms $(c \leq s(c))$ and $(c \geq s(c))$. Two atoms are added instead of $(c = s(c))$ so that the clause generalization sub-routine has more possibilities for relaxing $\neg\tilde{s}$.
4. For each clock c with $s(c) < m_c$ and $\mathrm{fract(s(c))} \neq 0$, add the atoms $(c > \lfloor s(c) \rfloor)$ and $(c < \lceil s(c) \rceil)$.
5. For each pair c, d of distinct clocks with $s(c) \leq m_c$, and $s(d) \leq m_d$,
   (a) if $\mathrm{fract(s(c))} = \mathrm{fract(s(d))}$, add the atoms $(d \leq c - \lfloor s(c) \rfloor + \lfloor s(d) \rfloor)$ and $(d \geq c - \lfloor s(c) \rfloor + \lfloor s(d) \rfloor)$, and
   (b) if $\mathrm{fract(s(c))} < \mathrm{fract(s(d))}$, add the atom $(d > c - \lfloor s(c) \rfloor + \lfloor s(d) \rfloor)$.

where points 2 to 4 encode condition 1 in the region definition in Sect. 2, 3 and 4 encode condition 2 and 5 encodes condition 3. Conveniently, unlike in the region-disequality constraints in k-induction, there is no need to directly access the integral and fractional parts of clock variables in $\tilde{s}$ because $\tilde{s}$ considers one fixed region. Indeed, all the atoms concerning clock variables will fall in the difference logic fragment of linear arithmetics over reals, having very efficient decision procedures available [22,23].

We now let the IC3 algorithm operate as in the untimed case except the satisfiability calls are changed to operate on the region level. Especially, the formula in Line 6 of Fig. 2(b) is modified to $F_{i-1} \wedge \neg\tilde{s} \wedge T \wedge Invar \wedge Invar' \wedge \tilde{s}'$ to operate on the region level, searching a predecessor in F_{i-1} for *any* state in the same region as s. Likewise, in Line 11 $\neg\tilde{s}$, a clause representing all states not in the same region as s, is passed to clause generalization excluding at least all states in the region of s from F_i. Inside the clause generalization mechanisms, clock atoms are handled in the same way as state variable literals. Note that clauses from which clock atoms have been dropped during generalization correspond to excluding a convex union of regions, called a zone.

Example 2. Consider the STTS for the system in Fig. 1(a) discussed in Ex. 1. For the state $s = \{x_1 \mapsto \mathbf{false}, x_2 \mapsto \mathbf{true}, c \mapsto 1.4, d \mapsto 1.65, ...\}$ we obtain the conjunction $\tilde{s} = (\neg x_1 \wedge x_2 \wedge (c > 1) \wedge (c < 2) \wedge (d > 1) \wedge (d < 2) \wedge (d > c) \wedge ...)$ that represents all the states in the region of s. The clause that excludes the whole region of s is simply $(x_1 \vee \neg x_2 \vee (c \leq 1) \vee (c \geq 2) \vee (d \leq 1) \vee (d \geq 2) \vee (d \leq c) \vee ...)$.

The soundness of the timed IC3 algorithm can be argued as follows. We say that a formula ϕ over $X \cup C$ *respects regions* if for all states s and u, $s \sim u$ implies that $\phi \models s$ iff $\phi \models u$. By construction, a state property P as well as $\tilde{s}$ and $\neg\tilde{s}$ for any state s all respect regions. Furthermore, any sub-clause of $\neg\tilde{s}$ returned by the clause generalization sub-routine also respects regions. As a result all clauses in the F-sets respect regions and, thus, the F-sets exclude whole regions only. Furthermore, the modified formula $F_{i-1} \wedge \neg\tilde{s} \wedge T \wedge \mathit{Invar} \wedge \mathit{Invar}' \wedge \tilde{s}'$ in Line 6 of Fig. 2(b) is unsatisfiable iff no state in the same region as s can be reached from the F_{i-1}-set; thus excluding the whole region in Line 11 is correct.

Because the number of regions is finite, only a finite number of clauses can be added to any F-set. As a result, the argument for termination given in [21] can be applied to the timed IC3 algorithm as well.

5 Optimizations by Excluding Multiple Regions

We now describe optimizations for timed IC3 and k-induction that often allow us to exclude more regions (i.e., zones of a certain form) at once during clause generalization and in the region-disequality constraints, respectively. They exclude time-predecessor regions, i.e. regions from whose states one can reach a certain region by just letting time pass. For example, all the light gray regions and the dark gray region (with $c = 3$ and $d > 2$) in Fig. 3 are time-predecessors of the dark gray region. A state u belongs to a time-predecessor region of another state s, denoted by $u \precsim s$, if the states agree on the values of the state variables and, when restricted to the clock variables, u belongs to a time-predecessor region of s; $\precsim$ is formally defined in [24].

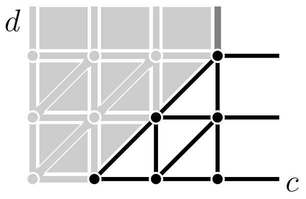


Fig. 3. A time-predecessor zone

Application to IC3. The timed variant of the IC3 algorithm described in Sect. 4 excludes an entire region from an F-set once a state inside that region (and thus the whole region) has been found to be unreachable from the previous F-set. In this section, we will argue that it is actually possible to exclude all the time-predecessor regions at the same time. By excluding more than one region, the F-sets potentially shrink faster which can lead to improved execution times. This optimization to the IC3 algorithm is based on the following lemma, the proof of which can be found in [24]:

Lemma 1. *Let s be a valid state. If none of the states in the region of s can be reached from an initial state with one time elapse step followed by n combined steps, then none of the valid states in the time-predecessor regions of s can, either.*

Applying Lemma 1 to a state found unreachable by the IC3 algorithm justifies dropping all the time-predecessor regions at the same time. For this purpose we construct, for a given state s, a conjunction $\tilde{s}_{\precsim}$ representing all the states in the time-predecessor regions of s. Then $\tilde{s}_{\precsim}$ and $\neg\tilde{s}_{\precsim}$ can be used instead of s and $\neg s$ in SMT calls and as argument for clause generalization. For a precise definition of $\tilde{s}_{\precsim}$, please refer to [24].

Example 3. Consider again the STTS for the system in Fig. 1(a) discussed in Ex. 1. For the state $s = \{x_1 \mapsto \mathbf{false}, x_2 \mapsto \mathbf{true}, c \mapsto 3.0, d \mapsto 2.7, ...\}$ in the dark gray clock region in Fig. 3, we get the conjunction $\tilde{s}_{\precsim} = (\neg x_1 \wedge x_2 \wedge (c \leq 3) \wedge (d > c - 1) \wedge ...)$ representing all the states in the time-predecessor regions.

Application to *k*-induction. The idea of excluding time-predecessor regions can also be applied to k-induction. This is based on the following lemma (again proven in [24]), stating that a path of combined steps can be compressed into a shorter, region-equivalent path ending in a region-equivalent state if a state in it is in the time-predecessor region of a later state:

Lemma 2. *Let $s_0 s_1^{\mathrm{d}} s_1 \ldots s_{i-1}^{\mathrm{d}} s_{i-1} s_i^{\mathrm{d}} s_i \ldots s_j^{\mathrm{d}} s_j \ldots s_k^{\mathrm{d}}$ be a path such that (i) s_0 is an initial state, (ii) $s_i^{\mathrm{d}} \precsim s_j^{\mathrm{d}}$, (iii) each step between s_l and s_{l+1}^{d} is a time elapse step, and (iv) each step between s_l^{d} and s_l is a discrete step. Then $s_0 s_1^{\mathrm{d}} s_1 \ldots s_{i-1}^{\mathrm{d}} s_{i-1} u_j^{\mathrm{d}} u_j \ldots u_k^{\mathrm{d}}$ with $u_j^{\mathrm{d}} \sim s_j^{\mathrm{d}}$ for all $j \leq l \leq k$ and $u_j \sim s_j$ for all $j \leq l < k$ is also a path.*

Now this implies that in timed k-induction we can use, instead of the region-disequality formula $\mathit{DiffRegion}^{[i,j]}$, a stronger formula $\mathit{DiffRegion}_{\precsim}^{[i,j]}$ excluding the state $s^{[i]}$ from being in a time-predecessor region of the state $s^{[j]}$ when $i < j$. We omit the details but this formula can be obtained from the definition of the $\precsim$ relation in a similar way as the $\mathit{DiffRegion}^{[i,j]}$ formula was obtained from the definition of $\sim$ in Sect. 3.

6 Experiments

To determine the usefulness of the described methods, they were evaluated experimentally. Specifically, we were interested in the following questions: how do the methods perform and scale (i) in the area they were designed for, i.e. timed systems with a large number of state variables and a large number of discrete step successors for most states; (ii) compared to each other; (iii) compared to using discrete time verification methods in a semantics-preserving way; and (iv) outside the area they were designed for, i.e. on models with a small number of state variables and small number of discrete step successors for most states.

Setup. Timed k-induction and the timed IC3 algorithm were implemented in Python, each supporting both region encoding variants. Using a more efficient programming language like C would likely yield only moderate execution time improvements as a

Table 1. Verification times in seconds for industrial benchmarks. Properties that require parts of the system that are omitted in the submodels could not be verified on those submodels.

		Full					Medium size submodel					Small submodel				
Property	Satisfied	Timed IC3, extended r.	Timed IC3, basic regions	k-induction, basic regions	k-induction, extended r.	Booleaniza-tion + IC3	Timed IC3, extended r.	Timed IC3, basic regions	k-induction, basic regions	k-induction, extended r.	Booleaniza-tion + IC3	Timed IC3, extended r.	Timed IC3, basic regions	k-induction, basic regions	k-induction, extended r.	Booleaniza-tion + IC3
1	yes	0.55	0.55	**0.50**	0.53	181.95	0.32	0.33	0.42	**0.29**	16.43	0.21	0.21	0.29	**0.19**	8.51
2	yes	0.55	0.51	**0.47**	0.50	*timeout*	**0.30**	0.32	0.33	0.34	*timeout*	**0.21**	**0.21**	0.23	0.22	1459.60
3	yes	0.57	0.56	**0.49**	**0.49**	*timeout*	**0.32**	0.33	0.36	0.36	*timeout*	0.22	**0.21**	0.23	0.24	*timeout*
4	yes	0.54	0.57	**0.48**	0.58	*timeout*	**0.33**	0.34	0.35	0.34	*timeout*					
5	yes	0.68	0.55	0.62	**0.54**	191.81	0.32	0.31	**0.30**	0.31	18.10					
6	yes	0.57	0.58	0.60	**0.50**	*timeout*	0.34	**0.33**	0.37	0.38	*timeout*					
7	yes	0.57	0.60	**0.51**	0.55	*timeout*										
8	no	*timeout*	*timeout*	**2.21**	2.24	*timeout*	0.30	**0.29**	0.35	0.33	*timeout*	0.21	**0.20**	0.33	0.23	2367.16
9	no	0.62	**0.56**	0.70	0.65	194.25	0.32	0.31	0.29	**0.27**	4.47	0.21	0.23	0.29	**0.20**	2.05
10	no	*timeout*	*timeout*	2.36	**2.15**	165.16	**0.31**	0.32	0.41	**0.31**	17.07	**0.20**	0.23	0.21	**0.20**	8.58
11	no	**0.53**	0.54	0.62	0.65	169.91	0.33	0.35	0.41	**0.32**	17.53					
12	no	*timeout*	*timeout*	2.21	**2.11**	*timeout*	0.32	**0.31**	0.34	0.33	*timeout*					
13	no	*timeout*	*timeout*	2.24	**2.17**	*timeout*	0.32	**0.31**	0.46	0.35	*timeout*					
14	no	**0.57**	**0.57**	0.63	0.65	170.87	**0.32**	0.36	0.42	0.33	17.80					
15	no	**0.56**	0.59	0.85	0.66	81.07										

significant fraction of the time is spent by the SMT solver. As an SMT-solver, Yices [20] version 1.0.31 was used. All experiments were executed on GNU/Linux computers with AMD Opteron 2435 CPUs limited to one hour of CPU time and 2 GB of RAM. The prototype implementation and the benchmarks used are available on the first author's website (`http://users.ics.aalto.fi/kindermann/`).

Industrial benchmark. The first benchmark used is a model of an emergency diesel generator intended for the use in a nuclear power plant. The full model and two sub-models, which are sufficient verifying some properties, were used. The numbers of clocks and state variables are 24 and 130 for the full model, 7 and 64 for the medium size and 6 and 36 for the small sub-model. The industrial model has been studied previously but not as a whole verified using real time. To allow for discrete time verification using the model checker NuSMV [25], the model first had to be split into multiple subcomponents that were then verified individually [26]. To allow for real time verification using the model checker Uppaal [4], further simplification by removing the possibility of individually components breaking non-deterministically, which is present in the original model, was necessary [26]. A booleanization-based attempt to verify the smallest sub-model was unable to verify all properties [7]. Furthermore, the industrial benchmark was used to compare different variants of bounded model checking of timed automata [9].

All four variants of the methods introduced in this paper were applied to the industrial model. Additionally, the original IC3 implementation [14] combined with a semantics-preserving booleanization approach [7] was used. Evaluating the fully symbolic approach of [16] on this benchmark is left for future as the tool fsmtMC seems to be not yet publicly available. Table 1 shows the resulting execution times. k-induction did not exceed 3 seconds for any property. The timed IC3-approaches performed mostly similarly but timed out four times. Both real-time verification methods performed significantly better than the booleanization / IC3 combination, illustrating that using specialized real time verification methods is worthwhile.

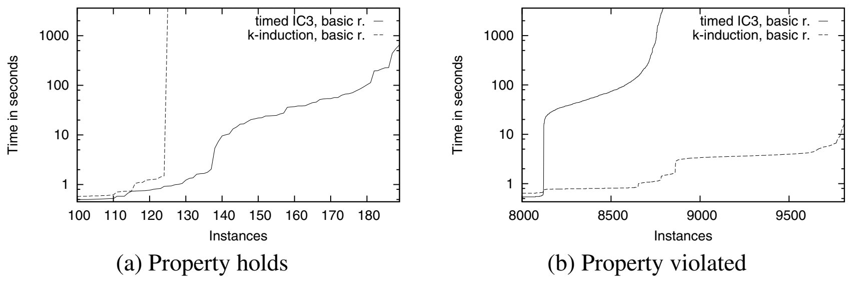


Fig. 4. Time required to verify by numbers of properties for randomly generated properties. A point (x, y) indicates that for x properties y or less time was needed (each). Effects of using the extended region encoding were minor, the corresponding lines were omitted for better readability.

Random properties. While the industrial benchmark showed that the methods work well in the area they were designed for, execution times were generally too small to compare the different methods and variants. Therefore, 10000 additional random properties were generated each for the full model and the medium size sub-model. More precisely, literals containing both clock and state variables and random clauses consisting of up to three such literals were generated. A property was considered to hold, if the corresponding clause is satisfied by all reachable states. Figure 4 shows a comparison of the resulting execution times on the full size model using the basic region encoding. The results for the medium size model are very similar and have been omitted due to a lack of space. The booleanization approach performed poorly on the original properties for the industrial model. Short, non-systematic tests indicated that the same behavior would have been observed on random properties as well and, consequently, booleanization-based approaches were not applied to the random properties.

For violated properties, k-induction performed very well, due to its bounded model checking component. For holding properties timed IC3 performed significantly better. Executing both methods (or timed IC3 and BMC) in parallel could combine their strengths. The plot for timed IC3 for non-holding properties shows a steep vertical climb, due to the fact that finding counter-examples with three or more states is significantly harder than one or two-state counter-examples for the IC3-method. In the presence of one or two-state counter-examples, the SMT queries made by timed IC3 are the same that would be executed by BMC. If the counter-example has three or more states, in contrast, the method finds a predecessor of a bad state and then searches an initial state that is predecessor of the found state. In many cases, such an initial predecessor does not exist. Thus, that in order to find a longer counter-example, the timed IC3 may need to search for a suitable middle state for the counter-example.

Figures 5(a) and 5(b) compare the execution times on random properties. Even though timed IC3 generally performed much better on holding properties than k-induction, k-induction was faster for a few properties. For violated properties, in contrast, k-induction performed consistently better than timed IC3. Figures 5(c) and 5(d) compare basic and extended regions (cf. Sect. 5) for the random properties on the full model and Fig. 5(e) shows the same comparison for the timed IC3 algorithm on the medium size model. The k-induction / medium size model results are similar to k-induction /

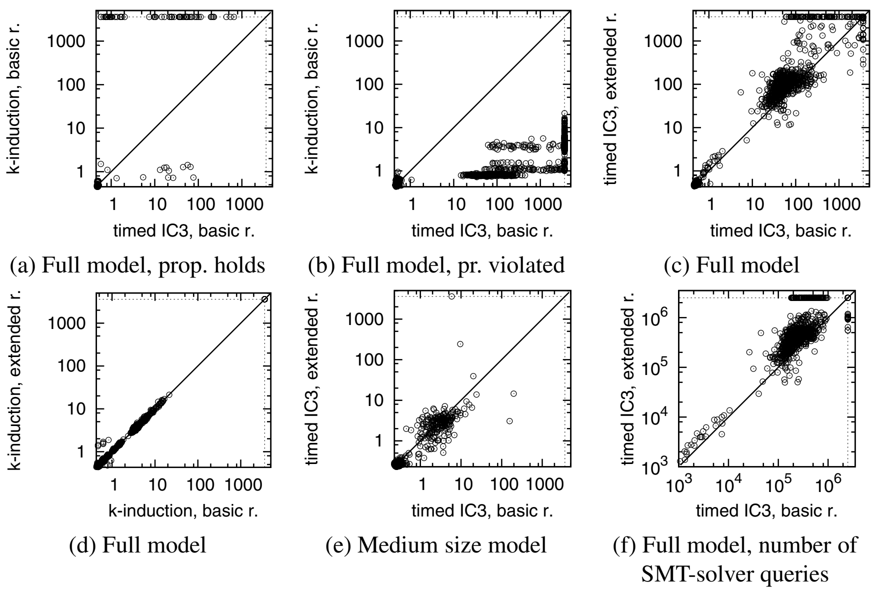


Fig. 5. Comparison between basic and extended regions (execution time in seconds except for (f), combining holding and non-holding properties except for (a) and (b))

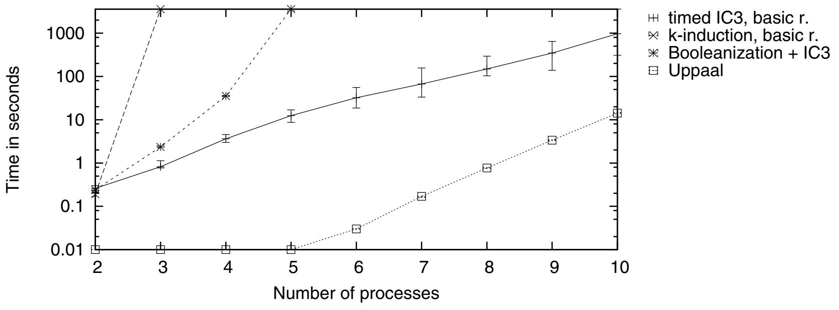


Fig. 6. Verification time the Fischer protocol (min, max and median of 11 executions)

full model and are omitted here due to space restrictions. Using time-predecessor regions made no difference for k-induction. For the timed IC3 algorithm, their effect depended on the size of the model used. On the medium size model, performance slightly increased while slightly decreasing on the full model. A likely explanation is the large number of clocks in the full model. While the time-predecessor region encoding uses fewer literals referring to a single clock, it contains more literals comparing two clocks (cf. [24]). Thus, the clause size grows quicker in the number of clocks for time-predecessor regions, which eventually outweighs the gain of excluding more states at once. Figure 5(f) shows the number of SMT-solver queries needed by timed IC3 for the 1766 properties for which at least 1000 queries were needed. Similarly to the effect on execution time, using extended regions slightly increased the number of queries.

Fischer protocol. As a third benchmark, the Fischer mutual exclusion protocol, a standard benchmark for timed verification, was used. In addition to the five methods used for the industrial method, Uppaal [4] version 4.0.11, a model checker for networks of timed automata, was used. Unlike the industrial benchmark, the Fischer protocol is fairly deterministic with respect to discrete steps possible from any given state and, thus, could be expected to favor Uppaal over the fully-symbolic methods. Figure 6 shows the execution times for verifying the Fischer model with a varying number of processes. While timed IC3 was, unsurprisingly, significantly slower than Uppaal, it scaled similarly, i.e. the runtime increased at a similar rate. k-induction timed out at three processes already while the booleanization-based approach showed rapid runtime growth and timed out at five processes.

7 Conclusion

This paper introduces two verification methods for symbolic timed transition systems: a timed variant of the IC3 algorithm and an adapted version of k-induction. Furthermore, a potential optimization to both methods is devised.

Both methods were able to verify properties on an industrial model verification of which had been found in previous attempts intractable and outperformed a booleanization-based approach significantly. Random properties on the same model revealed that the timed IC3 variant performs better for satisfied properties while timed k-induction performs better on violated properties. The experiments suggest that executing timed IC3 in parallel with bounded model checking would yield excellent performance for the verification of large, non-deterministic real-time systems.

Additionally, the proposed methods were evaluated on another family of benchmark, the Fischer mutual exclusion protocol with a varying number of processes. This family has only a small amount of non-determinism and the runtime of the methods was higher than that of the timed automata model checker Uppaal. However, the timed IC3 algorithm was found to have similarly good scaling as Uppaal.

As a final remark, an another extension of IC3 capable of verifying timed systems has been presented very recently in [27]; they do not use region abstraction directly but apply interpolants.

Acknowledgments. This work has been financially supported by the Academy of Finland under the Finnish Centre of Excellence in Computational Inference (COIN) and project 128050.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Alur, R.: Timed Automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
3. Bengtsson, J.E., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)

4. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
5. Wang, F.: Efficient verification of timed automata with BDD-like data structures. International Journal on Software Tools for Technology Transfer 6(1), 77–97 (2004)
6. Gruhn, P., Cheddie, H.L.: Safety Instrumented Systems: Design, Analysis, and Justification. ISA (2006)
7. Kindermann, R., Junttila, T., Niemelä, I.: Modeling for symbolic analysis of safety instrumented systems with clocks. In: Proc. ACSD 2011, pp. 185–194. IEEE (2011)
8. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1992)
9. Kindermann, R., Junttila, T., Niemelä, I.: Beyond Lassos: Complete SMT-Based Bounded Model Checking for Timed Automata. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 84–100. Springer, Heidelberg (2012)
10. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded Model Checking for Timed Systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 243–259. Springer, Heidelberg (2002)
11. Sorea, M.: Bounded model checking for timed automata. Electronic Notes in Theoretical Computer Science 68(5), 116–134 (2002)
12. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
14. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
15. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static Guard Analysis in Timed Automata Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
16. Morbé, G., Pigorsch, F., Scholl, C.: Fully Symbolic Model Checking for Timed Automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 616–632. Springer, Heidelberg (2011)
17. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
18. Björkmann, K., Frits, J., Valkonen, J., Heljanko, K., Niemelä, I.: Model-based analysis of a stepwise shutdown logic. VTT Working Papers 115. VTT Technical Research Centre of Finland, Espoo (2009)
19. de Moura, L., Rueß, H., Sorea, M.: Bounded Model Checking and Induction: From Refutation to Verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
20. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
21. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of IWLS, IEEE/ACM (2011)
22. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
23. Cotton, S., Maler, O.: Fast and Flexible Difference Constraint Propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)

24. Kindermann, R., Junttila, T., Niemelä, I.: SMT-based induction methods for timed systems. arXiv.org document arXiv:1204.5639v1(cs.LO) (2012)
25. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
26. Lahtinen, J., Björkman, K., Valkonen, J., Frits, J., Niemelä, I.: Analysis of an emergency diesel generator control system by compositional model checking. VTT Working Papers 156. VTT Technical Research Centre of Finland, Espoo (2010)
27. Hoder, K., Bjørner, N.: Generalized Property Directed Reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)

Timed Automata with Disjoint Activity

Marco Muñiz, Bernd Westphal, and Andreas Podelski

Albert-Ludwigs-Universität Freiburg, 79110 Freiburg, Germany
{muniz,westphal,podelski}@informatik.uni-freiburg.de

Abstract. The behavior of timed automata consists of idleness and activity, i.e. delay and action transitions. We study a class of timed automata with periodic phases of activity. We show that, if the phases of activity of timed automata in a network are disjoint, then location reachability for the network can be decided using a concatenation of timed automata. This reduces the complexity of verification in Uppaal-like tools from quadratic to linear time (in the number of components) while traversing the same reachable state space. We provide templates which imply, by construction, the applicability of sequential composition, a variant of concatenation, which reflects relevant reachability properties while removing an exponential number of states. Our approach covers the class of TDMA-based (Time Division Multiple Access) protocols, e.g. FlexRay and TTP. We have successfully applied our approach to an industrial TDMA-based protocol of a wireless fire alarm system with more than 100 sensors.

1 Introduction

Timed real world applications may include a large number of components. These components can often be modeled using timed automata [1] and their composition. The behavior of timed automata consists of idleness and activity, i.e. delay and action transitions. In many cases the activity of the timed automata (which model an application) is disjoint i.e. one automaton is active while the other ones are idle, except at time points where they may synchronize. In timed automata with disjoint activity their parallel product will introduce many edges and locations which are not relevant given the disjoint activity assumption (unreachable locations). These many edges are unnecessarily evaluated and increase the verification costs in tools like Uppaal [3].

In this paper, we formalize a notion of timed automata with disjoint activity and characterize a class of timed automata with periodic cycles of activity. Then, we define a semantic concatenation operator which, when applied to timed automata with disjoint activity, produces automata bisimilar to the one obtained by their parallel composition. The automaton obtained by the concatenation operator has a reduced number of edges and locations than the one obtained by the parallel composition operator. Identifying the periodic cyclic phases of activity of a timed automata, and if these activity phases are disjoint can be as hard as verifying a property. Therefore, we introduce templates for timed automata which by construction ensure periodic phases of activity. In addition, a syntactic check in the instances of the templates suffices to ensure that their corresponding phases of activity are disjoint. We used our approach for verifying a real world *wireless fire alarm* system with up to 125 sensors. By using our approach in Uppaal the verification times decreased from quadratic to linear on the number of sensors.

An important class of systems which can be verified using our approach is the class of real-time network protocols which use the *Time Division Multiple Access* design principle. Well-known examples in this class of *TDMA protocols* include the Bluetooth protocol of [6], the time triggered protocol of [12], and the time triggered architectures of [11,7,10].

1.1 Related Work

Since the cost of model checking for a network of timed automata increases exponentially in the number of components, much research has been directed towards techniques that demonstrate a potentially exponential speedup in interesting applications (see, e.g., [2]). It turns out that, for the class of timed automata with disjoint activity , the cost increases quadratically in the number of components (which, as confirmed in our experiments, can be bad enough with an increasing number of components); thus, the best one can hope for, is to be able to demonstrate a potentially linear speedup (see Figure 1, page 202). Little research has been devoted to the (comparatively modest, albeit occasionally relevant) goal of a linear speedup; in particular, we are not aware of techniques that are directly related to our approach. Still, let us note that the technique of active clock reduction of [4] and its generalisation in [2], which may seem relevant in this context, are orthogonal to our approach; in fact, when we present our experimental evaluation, we evaluate the improvement obtained by the syntactic transformation (for a non-optimized model) with respect to the execution time for an optimized model which has only one clock.

In [5,9,14] Communication-Closed Layers and timed automata are studied. The approach presented in [14] and ours are complementary. The main differences are that the approach in [14] is action based, whereas our approach is time based. In addition, we consider cyclic timed automata, whereas in [14], automata can not perform actions after reaching their corresponding final location.

In Section 6 we introduce sequential timed automata and present in Definition 9 the notion of an overclock for two clocks. We use this notion to reduce the number of clocks in sequential timed automata. In [8] this notion is generalized to quasi-equal clocks and a more general reduction method for quasi-equal clocks is presented. However, we show that for the context of sequential timed automata, the Sequentialisation method proposed in Section 6 yields an Automaton in which verification can be carried out in linear time on the number of sequential timed automata, whereas by using the method proposed in [8] the verification will be carried out in quadratic time. This is illustrated in our case study in Section 7.

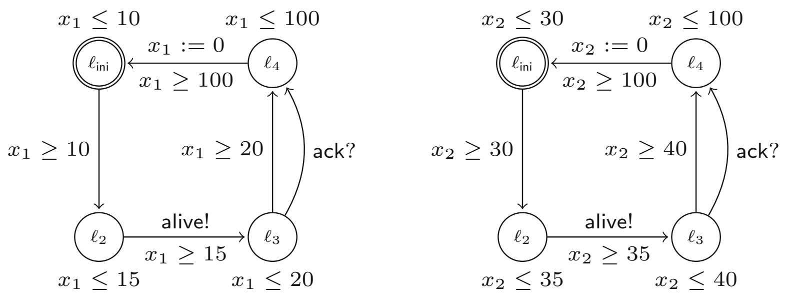


Fig. 1. Timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ modeling the behavior of two simplified sensors. The constants $10, 20$ and $30, 40$ denote the start resp. the end of the i-th time slot interval. The i-th sensor waits for the start of its designated time slot, sends its alive signal and waits for the ack signal from the central unit to arrive before the end.

2 Applications

First, we elucidate our method by explaining it through an example. Next, we describe how our method could be applied to a Time-Triggered Architecture system.

2.1 Example: Fire Alarm System

In the simplified (and simplistic) version that we consider for the example of a TDMA protocol, the wireless fire alarm system is a network of a central unit and a number n of sensors; here, $n = 10$. Figure 1 shows two timed automata which model two (simplified) sensors. The protocol operates in cycles of fixed length of time, say 100. The cycle is split in n time slot intervals, each of the same length of time. In each cycle, the central unit listens at the i-th time slot to an alive message from the i-th sensor. If the alive message is received, the central unit replies with an ack message.

Figure 2 shows the timed automaton that is 'equivalent' to the parallel product of the timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$; it is obtained from applying the operation that we will introduce in Section 6. We note three phenomena that we observe on the example (and describe the concepts that we will introduce in the corresponding section in order to investigate the phenomena). These phenomena constitute the premises of our approach.

(1) The initial location of $\mathcal{A}_i$ is visited infinitely often, with a regular period (in Section 4, we define notions of cyclicity and periodicity of timed automata).

(2) The i-th sensor mostly (but not exclusively!) performs action transitions at the i-th time slot (in Section 5, we formally characterize the notion of activity for timed automata, give a semantics-based definition of two timed automata being sequentialisable, and introduce the concatenation '$\cdot$' of sequentialisable timed automata).

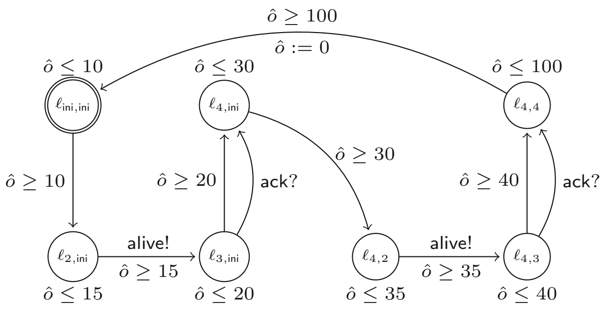


Fig. 2. The timed automaton $\mathcal{A}_1 \fatsemi \mathcal{A}_2$ that is 'equivalent' to the parallel product of the timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ from Figure 1, obtained from applying the operation that we will introduce in Section 6.

(3) In the interleaving semantics of the parallel composition of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$, the two different (zero-time) action transitions with the reset of the clock x_1 resp. the reset of the clock x_2 diverge, i.e., they lead to states where the values for x_1 and x_2 are different (in Section 6, when we define the sequential composition '$\fatsemi$' of sequentialisable timed automata, we recover the determinacy of the behavior of the parallel product through the concept of the *overclock*, i.e., the introduction of a new clock $\hat{o}$).

In addition, we observe that the automaton $\mathcal{A}_1||\mathcal{A}_2$, has an increased number of edges in comparision to $\mathcal{A}_1 \fatsemi \mathcal{A}_2$. The reduced number of edges yield a linear reduction in the verification time, justified by Lemma 3.

2.2 Potential Application: Steer-By-Wire Architecture Using TTP/C

In order to illustrate the applicability of our method, we propose an informal model for a system based on the Time-Triggered Architecture.

In the Time-Triggered Architecture every node consist of a local cpu, a Communication Network Interface and a TTP/C controller. The data communication over TTP/C is organized in TDMA rounds. A TDMA round is divided into slots and every node is assigned to a slot. A recurring sequence of TDMA rounds constitutes a cluster. The system can be globally monitored based on bus tracing.

A steer-by-wire system would require from 8 to 30 nodes. Interesting properties to verify might include: If a node fails, is this node detected as malfunctioning in within a TDMA round; or if a node fails, does the system still satisfy a given property.

Let us consider that the system has n nodes and one global monitor. The monitor could be modeled by one timed automaton $\mathcal{A}_m$. Every node could be modeled by Two timed Automata; one for the cpu $\mathcal{A}_{cpu}$ and one for the TTP/C communication $\mathcal{A}_{TTP}$. The automata corresponding to a node could communicate via shared variables. A TDMA round would have the following form, $\mathcal{A}_m||\mathcal{A}_{cpu_1}||\dots||\mathcal{A}_{cpu_n}||\mathcal{A}_{TTP_1}||\dots||\mathcal{A}_{TTP_n}$. Since Automaton $\mathcal{A}_{TPP_i}$ will only perform actions on its corresponding slot for $i \in \{1, \dots, n\}$, we can apply our method and obtain: $\mathcal{A}_m || \mathcal{A}_{cpu_1} || \dots || \mathcal{A}_{cpu_n} || (\mathcal{A}_{TTP_1} \fatsemi \dots \fatsemi \mathcal{A}_{TTP_n})$. Which we believe, would lead to an improvement on the verification time. Given a fixed number of TDMA rounds we can use the above method to construct a cluster.

3 Preliminaries

The formal basis for our work are timed automata [1]. In the following, we briefly recall the main definitions, our presentation follows [13]. Note that, in contrast to [13], we do not distinguish transition sequences and (time-stamped) computation paths. Here, the configurations of the labelled transition system are already time-stamped.

Let $\mathbb{X}$ be a set of clocks. The set $\Phi(\mathbb{X})$ of simple clock constraints over $\mathbb{X}$ is defined by the grammar $\varphi ::= x \sim C \mid x - y \sim C \mid \varphi_1 \land \varphi_2$ where $x, y \in \mathbb{X}$, $C \in \mathbb{Q}_0^+$, and $\sim \in \{<, \leq, \geq, >\}$. We assume the canonical satisfaction relation "$\models$" between *valuations* of the clocks $\nu : \mathbb{X} \to \mathsf{Time}$ and simple clock constraints, with $\mathsf{Time} = \mathbb{R}_{>0}$.

A Timed Automaton (TA) $\mathcal{A}$ is a tuple $(L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$, which consists of a finite set of *locations* L, with typical element ℓ, a finite set Σ of *actions* comprising the *internal action* τ, a finite set of *clocks* $\mathbb{X}$, a mapping $I : L \to \Phi(\mathbb{X})$, that assigns to each location a *clock constraint*, and a set of *edges* $E \subseteq L \times \Sigma \times \Phi(\mathbb{X}) \times \mathcal{P}(\mathbb{X}) \times L$. An edge $e = (\ell, \alpha, \varphi, Y, \ell') \in E$ from ℓ to ℓ' involves an action $\alpha \in \Sigma$, a *guard* $\varphi \in \Phi(\mathbb{X})$, and a *reset set* $Y \subseteq \mathbb{X}$.

The operational semantics of the timed automaton $\mathcal{A}$ is the labelled transition system $\mathcal{TS}(\mathcal{A}) = (\mathit{Conf}(\mathcal{A}), \mathsf{Time} \cup \Sigma, \{\xrightarrow{\lambda} \mid \lambda \in \mathsf{Time} \cup \Sigma\}, C_{\mathsf{ini}})$. The set of *configurations* $\mathit{Conf}(\mathcal{A}) = \{(\langle \ell, \nu \rangle, t) \in L \times (\mathbb{X} \to \mathsf{Time}) \times \mathsf{Time} \mid \nu \models I(\ell)\}$ consists of time-stamped pairs of a location $\ell \in L$ and a valuation of the clocks $\nu : \mathbb{X} \to \mathsf{Time}$ which satisfies the clock constraint $I(\ell)$. The set of initial configurations is $C_{\mathsf{ini}} = \{(\langle \ell_{\mathsf{ini}}, \nu_{\mathsf{ini}} \rangle, 0)\} \cap \mathit{Conf}(\mathcal{A})$ where $\nu_{\mathsf{ini}}(x) = 0$ for all clocks $x \in \mathbb{X}$. There is a *delay transition* from configuration $\langle \ell, \nu \rangle, t$ to $\langle \ell, \nu + t' \rangle, t + t'$, i.e. $\langle \ell, \nu \rangle, t \xrightarrow{t'} \langle \ell, \nu + t' \rangle, t + t'$, if and only if $\nu + t'' \models I(\ell)$ for all $t'' \in [0, t']$, where $\nu + t'$ denotes the valuation obtained from ν by time shift t'. There is an *action transition* between $\langle \ell, \nu \rangle, t$ and $\langle \ell', \nu' \rangle, t$, i.e $\langle \ell, \nu \rangle, t \xrightarrow{\alpha} \langle \ell', \nu' \rangle, t$, if and only if there exists an edge $(\ell, \alpha, \varphi, Y, \ell') \in E$ with $\nu \models \varphi$, $\nu' = \nu[Y := 0]$, and $\nu' \models I(\ell')$, where $\nu[Y := 0]$ denotes the valuation obtained from ν by resetting exactly the clocks in Y. We write $\ell(c)$, $\nu(c)$, and $t(c)$, to denote the location ℓ, valuation ν, and time-stamp t of a configuration $c = \langle \ell, \nu \rangle, t$.

An infinite or maximally finite sequence $\pi = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} c_2 \dots$ is called a *computation* of $\mathcal{A}$ if and only if $c_0 \in C_{\mathsf{ini}}$ and $(c_i, c_{i+1}) \in \xrightarrow{\lambda}$ for all $i \in \mathbb{N}_0$. We write π_j to denote the j-th configuration $c_j = \langle \ell_j, \nu_j \rangle, t_j$ in π, and λ_j^π to denote the label of j-th transition in π, or simply λ_j if π is clear from the context. We write $\pi \in \mathcal{TS}(\mathcal{A})$ if and only if π is a computation of $\mathcal{A}$.

The *parallel composition* of two timed automata $\mathcal{A}_i = (L_i, \Sigma_i, \mathbb{X}_i, I_i, E_i, \ell_{\mathsf{ini},i})$, $i = 1, 2$, with disjoint sets of clocks $\mathbb{X}_1$ and $\mathbb{X}_2$ yields the timed automaton $\mathcal{A}_1 \| \mathcal{A}_2 \overset{\mathsf{def}}{=} (L_1 \times L_2, \Sigma_1 \cup \Sigma_2, \mathbb{X}_1 \cup \mathbb{X}_2, I, E, (\ell_{\mathsf{ini},1}, \ell_{\mathsf{ini},2}))$ where $I(\ell_1, \ell_2) :=$

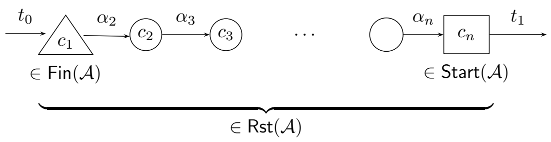

t_0
α_2
α_3
c_1
c_2
c_3
$\cdots$
α_n
c_n
t_1
$\in \mathsf{Fin}(\mathcal{A})$
$\in \mathsf{Start}(\mathcal{A})$
$\in \mathsf{Rst}(\mathcal{A})$

Fig. 3. Start configurations are in the initial location and have an action predecessor and a delay successor, final configurations a delay predecessor and an action successor. Configurations on an action-only path between a final and a start configuration are called restart configurations.

$I_1(\ell_1) \wedge I_2(\ell_2)$, for each $\ell_1 \in L_1, \ell_2 \in L_2$, and where E consists of *handshake* and *asynchronous edges* defined as follows. There is a handshake transition $((\ell_1, \ell_2), \tau, \varphi_1 \wedge \varphi_2, Y_1 \cup Y_2, (\ell_1', \ell_2')) \in E$ if there are *complementary* actions α and $\bar{\alpha}$ in $\Sigma_1 \cup \Sigma_2$ such that $(\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1$ and $(\ell_2, \bar{\alpha}, \varphi_2, Y_2, \ell_2') \in E_2$. For each edge $(\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1$ and each location $\ell_2 \in L_2$, there is an asynchronous transition $((\ell_1, \ell_2), \alpha, \varphi_1, Y_1, (\ell_1', \ell_2)) \in E$, and analogously for each transition in E_2.

4 Periodic Cyclic Timed Automata

Timed automata models of, e.g., TDMA-based protocols can be cyclic and periodic in the following sense. Intuitively, a timed automaton is cyclic if the initial location is visited infinitely often on all computations, the corresponding configurations are called start configuration. A timed automaton is periodic with period pt if configurations containing the initial location are reached only at integer multiples of the period and are reached from a unique final location.

In the following, we formally define periodic cyclic timed automata in terms of the new notions of start, restart, and final configurations (cf. Figure 3).

Definition 1 (Start and Final Configuration). *Let* $\mathcal{A} = (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ *be a timed automaton. The set* $\mathsf{Start}(\mathcal{A})$ *of* start configurations *of* $\mathcal{A}$ *consists of those configuration of* $\mathcal{TS}(\mathcal{A})$ *that are at location* ℓ_{ini} *and occur in a computation* $\pi \in \mathcal{TS}(\mathcal{A})$ *as source of a delay transition and as destination of an action transition, i.e.*

$$\mathsf{Start}(\mathcal{A}) \overset{\mathsf{def}}{=} \{c = \langle \ell_{\mathsf{ini}}, \nu \rangle, t \in \mathit{Conf}(\mathcal{A}) \mid \exists \pi \in \mathcal{TS}(\mathcal{A}), m \in \mathbb{N}_0 \bullet \\ \pi_m = c \wedge \lambda_m \in \mathsf{Time} \wedge (\lambda_{m-1} \in \Sigma \vee m = 0)\}.$$

The set $\mathsf{Rst}(\mathcal{A})$ *of* restart configurations *consists of those configuration of* $\mathcal{TS}(\mathcal{A})$ *that occur in a computation* $\pi \in \mathcal{TS}(\mathcal{A})$ *as action-predecessor of a start configuration, i.e.*

$$\mathsf{Rst}(\mathcal{A}) \overset{\mathsf{def}}{=} \{c \in \mathit{Conf}(\mathcal{A}) \mid \exists \pi \in \mathcal{TS}(\mathcal{A}), m, i \in \mathbb{N}_0 \bullet m \leq i \wedge \pi_m = c \\ \wedge \pi_i \in \mathsf{Start}(\mathcal{A}) \wedge \pi_m \xrightarrow{\lambda_m} \ldots \xrightarrow{\lambda_{i-1}} \pi_i \wedge \forall m \leq j \leq i \bullet \lambda_j \in \Sigma\}.$$

The set $\mathsf{Fin}(\mathcal{A})$ *of* final configurations *consists of the maximal restart configurations of* $\mathcal{TS}(\mathcal{A})$*, that is, restart configurations which are the destination of a delay transition, i.e.*

$$\begin{aligned}\mathsf{Fin}(\mathcal{A}) \overset{\mathsf{def}}{=} \{c \in \mathsf{Rst}(\mathcal{A}) \mid \exists \pi \in \mathcal{TS}(\mathcal{A}), m \in \mathbb{N}_0 \bullet \\ \pi_m = c \wedge (\lambda_{m-1} \in \mathsf{Time} \vee m = 0)\}.\end{aligned}$$

The set L_{rst} *of* restart locations *consists of those locations that occur in a restart configuration, i.e.*

$$L_{\mathsf{rst}} \overset{\mathsf{def}}{=} \{\ell \in L \mid \exists \nu : \mathbb{X} \to \mathsf{Time}, t \in \mathsf{Time} \bullet \langle \ell, \nu \rangle, t \in \mathsf{Rst}(\mathcal{A})\}.$$

Definition 2 (Periodic Cyclic). *A timed automaton* $\mathcal{A} = (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ *is called* periodic cyclic *with period* $pt \in \mathsf{Time}$ *if and only if each computation comprises infinitely many start configurations which occur at a regular period of time and if there is a unique* final location ℓ_{fin}*, i.e.*

$$\begin{aligned}\mathsf{peCy}(\mathcal{A}, pt) \overset{\mathsf{def}}{\iff} & \forall \pi \in \mathcal{TS}(\mathcal{A}), p \in \mathbb{N}_0 \ \exists c = (\langle \ell, \nu \rangle, t) \in \mathsf{Start}(\mathcal{A}), i \in \mathbb{N}_0 \bullet \\ & \quad \pi_i = c \wedge t = pt \cdot p \ \wedge \\ & \forall \pi \in \mathcal{TS}(\mathcal{A}), c = (\langle \ell, \nu \rangle, t) \in \mathsf{Start}(\mathcal{A}), i \in \mathbb{N}_0 \ \bullet \\ & \quad \pi_i = c \Rightarrow \exists p \in \mathbb{N}_0 \bullet t = pt \cdot p \ \wedge \\ & \exists \ell_{\mathsf{fin}} \in L \ \forall (\langle \ell, \nu \rangle, t) \in \mathsf{Fin}(\mathcal{A}) \bullet \ell = \ell_{\mathsf{fin}}.\end{aligned}$$

Theorem 1. *Let* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *be periodic cyclic timed automata with period* $pt \in \mathsf{Time}$*. Then* $\mathcal{A}_1 \| \mathcal{A}_2$ *is periodic cyclic with period* pt*.*

5 Concatenation of Sequentialisable Timed Automata

We say a timed automaton is active at a point in time if there exists a computation where an action transition is taken at that time. Two periodic cyclic timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are called sequentialisable if, they have the same period and within each period, all activity of $\mathcal{A}_1$ lies strictly before all activity of $\mathcal{A}_2$, except for integer multiples of the period. In the following, we formally define activity and sequentialisability. We define the concatenation of two sequentialisable timed automata and show in Theorem 2 that the result satisfies exactly the same reachability and leads-to properties as the parallel composition of the two. In Lemma 3, we discuss the relation between outgoing and enabled edges in the parallel composition and our concatenation.

Definition 3 (Activity). *The set of* activity points $\mathsf{Active}(\mathcal{A}) \subseteq \mathsf{Time}$ *of a timed automaton* $\mathcal{A} = (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ *consists of those points in time at which action transitions take place in some computation, i.e.*

$$\mathsf{Active}(\mathcal{A}) \overset{\mathsf{def}}{=} \{t \in \mathsf{Time} \mid \exists \pi \in \mathcal{TS}(\mathcal{A}), j \in \mathbb{N} \bullet \lambda_j \in \Sigma \wedge t(\pi_j) = t\}.$$

Definition 4 (Sequentialisable). *Two timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are called* sequentialisable *if and only if*

1. *$\mathcal{A}_1$ and $\mathcal{A}_2$ have disjoint sets of clocks,*
2. *$\mathcal{A}_1$ and $\mathcal{A}_2$ are periodic cyclic with period $pt \in \mathsf{Time}$, and*
3. *for each $p \in \mathbb{N}_0$,* within *the p-th period, $\mathcal{A}_1$ is active strictly before $\mathcal{A}_2$, i.e.*

$$sup(\mathsf{Active}_p(\mathcal{A}_1)) < inf(\mathsf{Active}_p(\mathcal{A}_2))$$

where $\mathsf{Active}_p(\mathcal{A}_i) \overset{\mathsf{def}}{=} \mathsf{Active}(\mathcal{A}_i) \cap \,]pt \cdot p, pt \cdot (p+1)[$, $i = 1, 2$.

Note 1. Within the p-th period, $p \in \mathbb{N}_0$, the activity points of two sequentialisable timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint, i.e.

$$\mathsf{Active}(\mathcal{A}_1) \cap \mathsf{Active}(\mathcal{A}_2) \cap \,]pt \cdot p, pt \cdot (p+1)[\, = \emptyset.$$

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are sequentialisable. Then on each period, first $\mathcal{A}_1$ is active and reaches its final location while $\mathcal{A}_2$ is at its initial location. Subsequently, $\mathcal{A}_2$ is active and reaches its final location while $\mathcal{A}_1$ is at its final location. At the end of the period both $\mathcal{A}_1$ and $\mathcal{A}_2$ are active at locations corresponding to their reset configurations.

Lemma 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequentialisable timed automata with period pt.*

1. *For all points of time different than the integer multiples of pt and within the activity of $\mathcal{A}_1$, $\mathcal{A}_2$ is in its initial location ℓ_{ini_2}, i.e.*

$$\begin{aligned}&\forall p \in \mathbb{N}_0, t \in \mathsf{Time} \bullet t \in [inf(\mathsf{Active}_p(\mathcal{A}_1)), sup(\mathsf{Active}_p(\mathcal{A}_1))]\\ &\implies \forall \pi \in \mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)\ \exists \langle(\ell_1, \ell_2), \nu\rangle, t \in Conf(\mathcal{A}_1 \| \mathcal{A}_2), j \in \mathbb{N}_0 \bullet\\ &\qquad \pi_j = \langle(\ell_1, \ell_2), \nu\rangle, t \wedge \ell_2 = \ell_{\mathsf{ini}_2}.\end{aligned}$$

2. *For all points of time different than the integer multiples of pt and within the activity of $\mathcal{A}_2$, $\mathcal{A}_1$ is in its final location ℓ_{fin_1}, i.e.*

$$\begin{aligned}&\forall p \in \mathbb{N}_0, t \in \mathsf{Time} \bullet t \in [inf(\mathsf{Active}_p(\mathcal{A}_2)), sup(\mathsf{Active}_p(\mathcal{A}_2))]\\ &\implies \forall \pi \in \mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)\ \exists \langle(\ell_1, \ell_2), \nu\rangle, t \in Conf(\mathcal{A}_1 \| \mathcal{A}_2), j \in \mathbb{N}_0 \bullet\\ &\qquad \pi_j = \langle(\ell_1, \ell_2), \nu\rangle, t \wedge \ell_1 = \ell_{\mathsf{fin}_1}.\end{aligned}$$

3. *In each computation, both, $\mathcal{A}_1$ and $\mathcal{A}_2$, are simultaneously at a restart location at integer multiples of pt, i.e.*

$$\begin{aligned}&\forall p \in \mathbb{N}_0, t \in \mathsf{Time} \bullet t = pt \cdot p\\ &\implies \forall \pi \in \mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)\ \exists \langle(\ell_1, \ell_2), \nu\rangle, t \in Conf(\mathcal{A}_1 \| \mathcal{A}_2), j \in \mathbb{N}_0 \bullet\\ &\qquad \pi_j = \langle(\ell_1, \ell_2), \nu\rangle, t \wedge \ell_1 \in L_{\mathsf{rst}} \wedge \ell_2 \in L_{\mathsf{rst}}.\end{aligned}$$

For sequentialisable automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with period pt, Lemma 1 suggests that for time points different than $pt \cdot p$ for some p. It is not necessary to compute the product of the locations on $\mathcal{A}_1$ and $\mathcal{A}_2$. The following concatenation operator exploits this fact and computes the product of locations only if both $\mathcal{A}_1$ and $\mathcal{A}_2$ are active. i.e. at time points $pt \cdot p$.

Definition 5 (Concatenation)
Let $\mathcal{A}_1 = (L_1, \Sigma_1, \mathbb{X}_1, I_1, E_1, \ell_{\mathsf{ini}_1})$ and $\mathcal{A}_2 = (L_2, \Sigma_2, \mathbb{X}_2, I_2, E_2, \ell_{\mathsf{ini}_2})$ be sequentialisable timed automata with period $pt \in \mathsf{Time}$. Let ℓ_{fin_i} denote the final location and let L_{rst_i} denote the set of restart locations of automaton $\mathcal{A}_i$, $i \in \{1, 2\}$.

The concatenation *of $\mathcal{A}_1$ and $\mathcal{A}_2$ yields the timed automaton*

$$\mathcal{A}_1 \cdot \mathcal{A}_2 \overset{\mathsf{def}}{=} (L, \Sigma_1 \cup \Sigma_2, \mathbb{X}_1 \cup \mathbb{X}_2, I, E, \ell_{\mathsf{ini}})$$

where

- $L = (L_1 \times \{\ell_{\mathsf{ini}_2}\}) \cup (\{\ell_{\mathsf{fin}_1}\} \times L_2) \cup (L_{\mathsf{rst}_1} \times L_{\mathsf{rst}_2})$
- $I(\ell_1, \ell_2) = I_1(\ell_1) \wedge I_2(\ell_2)$, $\ell_1 \in L_1, \ell_2 \in L_2$,
- $E = \{((\ell_1, \ell_2), \alpha, \varphi_1, Y_1, (\ell_1', \ell_2)) \mid (\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1 \wedge \ell_2 \in L_{\mathsf{rst}_2}\}$
  $\cup \{((\ell_1, \ell_2), \alpha, \varphi_2, Y_2, (\ell_1, \ell_2')) \mid (\ell_2, \alpha, \varphi_2, Y_2, \ell_2') \in E_2 \wedge \ell_1 \in L_{\mathsf{rst}_1}\}$, *and*
- $\ell_{\mathsf{ini}} = (\ell_{\mathsf{ini}_1}, \ell_{\mathsf{ini}_2})$.

Definition 6 (Bisimulation). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be timed automata and*

$$\mathcal{TS}_i(\mathcal{A}_i) = (\mathit{Conf}(\mathcal{A}_i), \mathsf{Time} \cup \Sigma_i, \{\xrightarrow{\lambda^i} \mid \lambda^i \in \mathsf{Time} \cup \Sigma_i\}, C_{\mathsf{ini}_i})$$

the corresponding labelled transition systems. A relation $\mathcal{R} \subseteq \mathit{Conf}(\mathcal{A}_1) \times \mathit{Conf}(\mathcal{A}_2)$ is called bisimulation *of $\mathcal{A}_1$ and $\mathcal{A}_2$ if and only if it satisfies the following conditions.*

1. $\forall c_1 \in C_{\mathsf{ini}_1}\ \exists c_2 \in C_{\mathsf{ini}_2} \bullet (c_1, c_2) \in \mathcal{R}$ *and* $\forall c_2 \in C_{\mathsf{ini}_2}\ \exists c_1 \in C_{\mathsf{ini}_1}) \bullet (c_1, c_2) \in \mathcal{R}$
2. *for all* $(c_1 = (\langle \ell_1, \nu_1 \rangle, t_1),\ c_2 = (\langle \ell_2, \nu_2 \rangle, t_2)) \in \mathcal{R}$,
   (a) $\nu_1 = \nu_2$, $t_1 = t_2$,
   (b) $\forall c_1 \xrightarrow{\lambda^1} c_1'\ \exists c_2 \xrightarrow{\lambda^2} c_2 \bullet (c_1', c_2') \in \mathcal{R}$
   (c) $\forall c_2 \xrightarrow{\lambda^2} c_2'\ \exists c_1 \xrightarrow{\lambda^1} c_1' \bullet (c_1', c_2') \in \mathcal{R}$.

$\mathcal{A}_1$ is called bisimilar to $\mathcal{A}_2$ iff there exists a bisimulation of $\mathcal{A}_1$ and $\mathcal{A}_2$.

For sequentialisable timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$, the implications of Lemma 1 and the definition of the concatenation operator imply that the transition system $\mathcal{TS}(\mathcal{A}_1 \cdot \mathcal{A}_2)$ corresponds to the reachable part of $\mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)$.

Theorem 2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequentialisable timed automata. Then $\mathcal{TS}(\mathcal{A}_1 \cdot \mathcal{A}_2)$ is bisimilar to $\mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)$.*

Theorem 2, ensures that the start configurations of $\mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2)$ are in $\mathcal{TS}(\mathcal{A}_1 \cdot \mathcal{A}_2)$. Therefore, the following theorem holds.

Theorem 3. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequentialisable timed automata with period pt. Then $\mathcal{A}_1 \cdot \mathcal{A}_2$ is periodic cyclic with period pt.*

Lemma 2 (Bisimulation). *Reachability properties are preserved under bisimulation, i.e. given bisimilar timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ and a state assertion φ, i.e., an expression over clock constraints and locations, we have*

$$(\exists \pi \in \mathcal{TS}(\mathcal{A}_1)\ \forall j \in \mathbb{N}_0 \bullet \pi_j \models \varphi) \iff (\exists \pi \in \mathcal{TS}(\mathcal{A}_2)\ \forall j \in \mathbb{N}_0 \bullet \pi_j \models \varphi)$$
$$(\forall \pi \in \mathcal{TS}(\mathcal{A}_1)\ \forall j \in \mathbb{N}_0 \bullet \pi_j \models \varphi) \iff (\forall \pi \in \mathcal{TS}(\mathcal{A}_2)\ \forall j \in \mathbb{N}_0 \bullet \pi_j \models \varphi).$$

Definition 7 (Enabled Edges). *Let $\mathcal{A} = (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ be a timed automaton and $c \in \mathit{Conf}(\mathcal{A})$ a configuration.*

We use $\mathit{out}(c)$ to denote the set of outgoing edges in c, i.e. the edges $e = (\ell, \alpha, \varphi, Y, \ell') \in E$ where $\ell = \ell(c)$.

Edge e is called enabled *if and only if its guard is satisfied and the effect of resets satisfies the clock constraint of the destination of e, i.e., if $\nu(c) \models \varphi$ and $\nu(c)[Y := 0] \models I(\ell')$. We use $\mathit{enab}(c)$ to denote the set of edges enabled in c.*

Note 2. Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be timed automata with pairwise disjoint edge sets and let $c = \langle (\ell_1, \ldots, \ell_n), \nu \rangle, t \in \mathit{Conf}(\mathcal{A}_1 \| \ldots \| \mathcal{A}_n)$ be a configuration.

1. Enabled edges are in particular outgoing, i.e. $\mathit{enab}(c) \subseteq \mathit{out}(c)$.
2. The set of outgoing edges in the parallel composition is determined by the components, i.e.

$$\mathit{out}(c) = \bigcup_{1 \le i \le n} \mathit{out}(\langle \ell_i, \nu|_{\mathbb{X}_i} \rangle, t), \mathit{enab}(c) = \bigcup_{1 \le i \le n} \mathit{enab}(\langle \ell_i, \nu|_{\mathbb{X}_i} \rangle, t),$$

thus (with disjoint edge sets)

$$|\mathit{out}(c)| = \Sigma_{1 \le i \le n} |\mathit{out}(\langle \ell_i, \nu|_{\mathbb{X}_i} \rangle, t)|, |\mathit{enab}(c)| = \Sigma_{1 \le i \le n} |\mathit{enab}(\langle \ell_i, \nu|_{\mathbb{X}_i} \rangle, t)|.$$

Since, outgoing edges are evaluated, a reduction on the number of outgoing edges yields a reduction on time complexity. This reduction can go from quadratic to linear time, as the following lemma shows.

Lemma 3. *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be sequentialisable timed automata with period pt with disjoint edge sets and with exactly one outgoing edge per location.*

1. *Let $c \in \mathit{Conf}(\mathcal{A}_1 \| \ldots \| \mathcal{A}_n)$ be a configuration of the parallel composition of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ where the time-stamp is not an integer multiple of the period pt, i.e. where $\nexists p \in \mathbb{N}_0 \bullet t(n) = p \cdot pt$.*
   Then $|\mathit{out}(c)| = n$ and $|\mathit{enab}(c)| = 1$.
2. *Let $c \in \mathit{Conf}(\mathcal{A}_1 \cdot \ldots \cdot \mathcal{A}_n)$ be a configuration of the concatenation of $\mathcal{A}_1, \ldots,$ $\mathcal{A}_n$ where the time-stamp is not an integer multiple the period pt.*
   Then $|\mathit{out}(c)| = |\mathit{enab}(c)| = 1$.

6 Sequential Composition of Sequential Timed Automata

As the decision whether a given pair of timed automata is sequentialisable is in general at least as difficult as the considered analysis problem, we provide a syntactical pattern such that instances of the pattern are sequentialisable and such that a specialized sequential composition applies.

Definition 8 (Sequential Timed Automaton). *A* sequential timed automaton *(STA) is a tuple*

$$\mathcal{A} = (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}}, \ell_{\mathsf{fin}}, \mathsf{sta}, \mathsf{fin}, pt, e_{\mathsf{fin}}, \hat{x})$$

where $\mathcal{A}_0 \overset{\mathsf{def}}{=} (L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ *is a timed automaton,* ℓ_{fin} *is a final location,* $\mathsf{sta}, \mathsf{fin} \in \mathbb{Q}_0^+$ *are* start *and* final time, $pt \in \mathbb{Q}_0^+$ *is a period,* $e_{\mathsf{fin}} \in E$ *is an edge of the form* $(\ell_{\mathsf{fin}}, \varnothing, \hat{x} \geq pt, Y \cup \{\hat{x}\}, \ell_{\mathsf{ini}})$ $\hat{x} \in \mathbb{X}$ *is a master clock, which satisfies the following syntactical constraints:*

– *the start time is positive and strictly smaller than the final time, which is strictly smaller than the period, i.e.*

$$0 < \mathsf{sta} \land \mathsf{sta} < \mathsf{fin} \land \mathsf{fin} < pt, \qquad \text{(saActive)}$$

– *the initial location is left if* $\hat{x}$ *reaches the start time, i.e.*

$$I(\ell_{\mathsf{ini}}) = \hat{x} \leq \mathsf{sta}, \qquad \text{(saStart)}$$
$$\forall (\ell, \alpha, \varphi, Y, \ell') \in E \bullet \ell = \ell_{\mathsf{ini}} \implies \varphi = \hat{x} \geq \mathsf{sta}, \qquad \text{(saStartTime)}$$

– *locations connected by an edge to the final location are only assumed until* $\hat{x}$ *reaches the final time, i.e.*

$$\forall (\ell, \alpha, \varphi, Y, \ell') \in E \bullet \ell' = \ell_{\mathsf{fin}} \Rightarrow I(\ell) = \hat{x} \leq \mathsf{fin}, \qquad \text{(saFinalTime)}$$

– *the final location is only assumed until* $\hat{x}$ *reaches the period, i.e.*

$$I(\ell_{\mathsf{fin}}) = \hat{x} \leq pt, \qquad \text{(saPeriod)}$$

– *the master clock is reset exactly on edge* e_{fin}*, i.e.*

$$\forall (\ell, \alpha, \varphi, Y, \ell') \in E \bullet \hat{x} \in Y \Rightarrow (\ell, \alpha, \varphi, Y, \ell') = e_{\mathsf{fin}}, \qquad \text{(saOneReset)}$$

– ℓ_{fin} *and* ℓ_{ini} *are connected exactly by edge* e_{fin}*, i.e.*

$$\forall (\ell, \alpha, \varphi, Y, \ell') \in E \bullet \ell = \ell_{\mathsf{fin}} \land \ell' = \ell_{\mathsf{ini}} \implies (\ell, \alpha, \varphi, Y, \ell') = e_{\mathsf{fin}}, \quad \text{(saOneFin)}$$

and the following semantical constraint:

– *whenever the initial location is assumed, the final location is finally reached, i.e.*

$$\begin{aligned} &\forall \pi \in \mathcal{TS}(\mathcal{A}_0), j \in \mathbb{N}_0, \langle \ell, \nu \rangle, t \in \mathit{Conf}(\mathcal{A}_0) \bullet \pi_j = \langle \ell, \nu \rangle, t \land \ell = \ell_{\mathsf{ini}} \\ &\quad \implies \exists k \in \mathbb{N}_0, \langle \ell', \nu' \rangle, t' \in \mathit{Conf}(\mathcal{A}_0) \bullet \\ &\qquad\quad k \geq j \land \pi_k = \langle \ell', \nu' \rangle, t' \land \ell' = \ell_{\mathsf{fin}} \end{aligned} \qquad \text{(saCyclic)}$$

Let $\mathcal{A}'$ be the automaton obtained by replacing synchronization transitions in $\mathcal{A}$ by internal transitions. Then, saCyclic can be checked for $\mathcal{A}'$ in a model checker. Figure 4 depicts a purely syntactically restricted template which ensures the instances to be sequential automata.

Theorem 4. *Let* $(L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}}, \ell_{\mathsf{fin}}, \mathsf{sta}, \mathsf{fin}, pt, e_{\mathsf{fin}}, \hat{x})$ *be a sequential timed automaton. Then* $(L, \Sigma, \mathbb{X}, I, E, \ell_{\mathsf{ini}})$ *is periodic cyclic with period pt.*

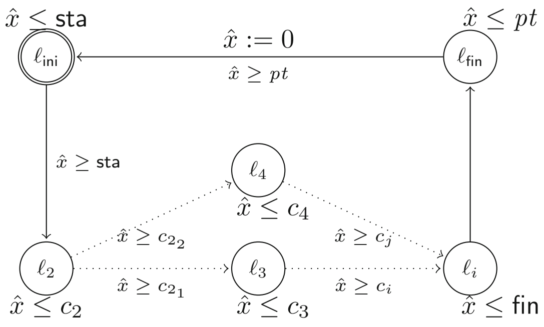


Fig. 4. A syntactical pattern for sequential timed automata

Before applying the sequential operator we must be sure that the activity phases of the automata are disjoint. In sequential automata sequentialisability can be syntacticaly proven, as the following lemma shows.

Lemma 4. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequential timed automata with the same period pt and final time fin_1 of $\mathcal{A}_1$ strictly smaller than the start time sta_2 of $\mathcal{A}_2$, i.e. $\mathsf{fin}_1 < \mathsf{sta}_2$. Then $\mathcal{A}_1$ and $\mathcal{A}_2$ are sequentialisable.*

Consider the parallel product or the concatenation of n sequential automata (see Figure 4) which are sequentialisable. The automaton will include a diamond like structure corresponding to the product of the n final locations. This structure will have 2^n locations. In this locations the master clocks $\hat{x}_1, \ldots, \hat{x}_n$ will not be equal, but note that this happends in zero time. Therefore, we define a new clock *overclock* $\hat{o}$ which preserves the quasi-equalitiy of the clocks, and allows to further optimize the resulting automaton.

Definition 9 (Overclock). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be timed automata with clocks x_1 and x_2, respectively. A clock $\hat{o}$ of $\mathcal{A}_1$ or $\mathcal{A}_2$ is an* overclock *for x_1 and x_2 in $\mathcal{A}_1 \| \mathcal{A}_2$ if and only if*

$$\forall \pi \in \mathcal{TS}(\mathcal{A}_1 \| \mathcal{A}_2), j \in \mathbb{N}_0 \langle (\ell_1, \ell_2), \nu \rangle, t \in Conf(\mathcal{A}_1 \| \mathcal{A}_2) \bullet \pi_j = \langle (\ell_1, \ell_2), \nu \rangle, t$$
$$\implies \nu \models (x_1 = \hat{o} \wedge x_2 = \hat{o}) \vee \ell_1 \in L_{\mathsf{rst}_1} \vee \ell_2 \in L_{\mathsf{rst}_2}.$$

Lemma 5. *Given sequential timed automata, $\mathcal{A}_1, \mathcal{A}_2$ with period pt and masterclocks $\hat{x}_1$, $\hat{x}_2$ respectively. Then, there exist an overclock $\hat{o}$ for $\hat{x}_1$ and $\hat{x}_2$.*

The diamond structure which we described above, will have 2^n locations and $n!$ zero time interleaving sequences, which occur at time points $pt \cdot p$ for $p \in \mathbb{N}_+$. Note that clocks $\hat{x}_1, \ldots, \hat{x}_n$ are always equal up to the time points $pt \cdot p$. The sequential composition operator which we define below, will remove this diamond structure and will replace all the clocks by an overclock.

Definition 10 (Sequential Composition). *Let $\mathcal{A}_i$,*

$$\mathcal{A}_i = (L_i, \Sigma_i, \mathbb{X}_i, I_i, E_i, \ell_{\mathsf{ini}_i}, \ell_{\mathsf{fin}_i}, \mathsf{sta}_i, \mathsf{fin}_i, pt, e_{\mathsf{fin}_i}, \hat{x}_i), i = 1, 2,$$

be sequential timed automata.

Then the sequential composition *of $\mathcal{A}_1$ and $\mathcal{A}_2$ yields the tuple*

$$\mathcal{A}_1 \fatsemi \mathcal{A}_2 \overset{\mathsf{def}}{=} (L, \Sigma_1 \cup \Sigma_2, \mathbb{X}_1 \cup \mathbb{X}_2, I, E, (\ell_{\mathsf{ini}_1}, \ell_{\mathsf{ini}_2}), (\ell_{\mathsf{fin}_1}, \ell_{\mathsf{fin}_2}), \mathsf{sta}_1, \mathsf{fin}_2, pt, e_{\mathsf{fin}}, \hat{o})$$

where

- *the master clock $\hat{o}$ is an overclock for $\hat{x}_1$, $\hat{x}_2$.*
- *the set of locations consists of pairs where $\mathcal{A}_2$ assumes an initial or $\mathcal{A}_1$ assumes a final location, i.e.*

$$L = ((L_1 \setminus \{\ell_{\mathsf{fin}_1}\}) \times \{\ell_{\mathsf{ini}_2}\}) \cup (\{\ell_{\mathsf{fin}_1}\} \times L_2),$$

- *the final edge e_{fin} is*

$$((\ell_{\mathsf{fin}_1}, \ell_{\mathsf{fin}_2}), \varnothing, \varphi_1 \wedge \varphi_2, Y_1 \cup Y_2, (\ell_{\mathsf{ini}_1}, \ell_{\mathsf{ini}_2}))$$

  given the final edges $e_{\mathsf{fin}_i} = (\ell_{\mathsf{fin}_i}, \varnothing, \varphi_i, Y_i, \ell_{\mathsf{ini}_i})$, $i = 1, 2$,
- *the clock constraint of location (ℓ_1, ℓ_2) is the conjunction of the corresponding clock constraints in $\mathcal{A}_1$ and $\mathcal{A}_2$ where substitute each $\hat{x}_1$ and $\hat{x}_2$ is syntactically substituted by $\hat{o}$, i.e.*

$$I(\ell_1, \ell_2) = (I_1(\ell_1) \wedge I(\ell_2))[\hat{x}_1/\hat{o}, \hat{x}_2/\hat{o}],$$

- *the set of edges comprises e_{fin} and compositions of $\mathcal{A}_1$ and $\mathcal{A}_2$ edges where $\hat{x}_1$ and $\hat{x}_2$ are substituted by $\hat{x}$ in guards and reset sets, i.e.*

$$E = \{e_{\mathsf{fin}}\} \cup \{((\ell_1, \ell_{\mathsf{ini}_2}), \alpha, \tilde{\varphi}_1, \tilde{Y}_1, (\ell_1', \ell_{\mathsf{ini}_2})) \mid (\ell_1, \alpha, \varphi_1, Y_1, \ell_1') \in E_1 \setminus \{e_{\mathsf{fin}_1}\}\}$$
$$\cup \{((\ell_{\mathsf{fin}_1}, \ell_2), \alpha, \tilde{\varphi}_2, \tilde{Y}_2, (\ell_{\mathsf{fin}_1}, \ell_2')) \mid (\ell_2, \alpha, \varphi_2, Y_2, \ell_2') \in E_2 \setminus \{e_{\mathsf{fin}_2}\}\}$$

  where $\tilde{\varphi}_i = \varphi_i[\hat{x}_1/\hat{o}, \hat{x}_2/\hat{o}]$, $i = 1, 2$, and $\tilde{Y}_i = Y_i[\hat{x}_1/\hat{o}, \hat{x}_2/\hat{o}]$, $i = 1, 2$.

Theorem 5. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequential timed automata with the same period pt and final time fin_1 of $\mathcal{A}_1$ strictly smaller than the start time sta_2 of $\mathcal{A}_2$, i.e. $\mathsf{fin}_1 < \mathsf{sta}_2$. Then $\mathcal{A}_1 \fatsemi \mathcal{A}_2$ is periodic cyclic with period pt.*

In Section 5, we have shown that For sequential automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ which are sequentialisable $\mathcal{A}_1 || \ldots || \mathcal{A}_n$ and $\mathcal{A}_1 \cdot \cdots \cdot \mathcal{A}_n$ are bisimilar. In what follows, we will show that $\mathcal{A}_1 || \ldots || \mathcal{A}_n$ and $\mathcal{A}_1 \fatsemi \ldots \fatsemi \mathcal{A}_n$ are weak-bisimilar (which reflects the effect of removing the diamond like structure in the sequential composition of the n sequential automata). We use the following definition in order to simplify a definition of weak-bisimulation.

Definition 11 (Action Reachability). *Let $\mathcal{A}$ be a timed automaton and $c, c' \in Conf(\mathcal{A})$ configurations. We say c' is* reachable *in $\mathcal{A}$ from c* via action transitions*, denoted by $\mathsf{actReach}(c, c', \mathcal{A})$, if and only if*

$$\mathsf{actReach}(c, c', \mathcal{A}) \overset{\mathsf{def}}{\iff} \exists c_0, c_1, c_2, \ldots, c_n \in Conf(\mathcal{A}), i \in \mathbb{N}_0 \bullet c_0 = c \wedge c_n = c'$$
$$\wedge \forall 0 \leq j < n \in \mathbb{N}_0 \bullet c_j \xrightarrow{\lambda_j} c_{j+1} \wedge \lambda_j \in \Sigma.$$

Definition 12 (Weak-bisimulation). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequential automata with $\hat{o} \in \mathbb{X}_1$ and $\hat{x}_1, \hat{x}_2 \in \mathbb{X}_2$ such that $\hat{o}$ is an overclock for $\hat{x}_1, \hat{x}_2$ and let*

$$\mathcal{TS}_i(\mathcal{A}_i) = (\mathit{Conf}(\mathcal{A}_i), \mathsf{Time} \cup \Sigma_i, \{\xrightarrow{\lambda^i} \mid \lambda^i \in \mathsf{Time} \cup \Sigma_i\}, C_{\mathsf{ini}_i}), i = 1, 2,$$

be the corresponding labelled transition systems.

A relation $\mathcal{W} \subseteq \mathit{Conf}(\mathcal{A}_1) \times \mathit{Conf}(\mathcal{A}_2)$ is called weak-bisimulation *of $\mathcal{A}_1$ and $\mathcal{A}_2$ if and only if it satisfies the following conditions.*

1. $\forall c_1 \in C_{\mathsf{ini}_1}\, \exists c_2 \in C_{\mathsf{ini}_2} \bullet (c_1, c_2) \in \mathcal{W}$ *and* $\forall c_2 \in C_{\mathsf{ini}_2}\, \exists c_1 \in C_{\mathsf{ini}_1}) \bullet (c_1, c_2) \in \mathcal{W}$
2. *for all* $(c_1 = (\langle \ell_1, \nu_1 \rangle, t_1), c_2 = (\langle \ell_2, \nu_2 \rangle, t_2)) \in \mathcal{W}$,
   (a) $\beta(\nu_1) = \nu_2$, $t_1 = t_2$ *with* $\beta(\nu) = \nu|_{\hat{o}} \cup \{\nu_{\hat{x}_1} \mapsto \nu(\hat{o}), \nu_{\hat{x}_2} \mapsto \nu(\hat{o})\}$,
   (b) $\forall c_1 \xrightarrow{\lambda^1} c_1' \bullet (\exists c_2 \xrightarrow{\lambda^2} c_2' \bullet (c_1', c_2') \in \mathcal{W}) \vee (\ell(c_1) = \ell_{\mathsf{fin}_1} \wedge$
   $$\exists c_2'' \bullet \mathsf{actReach}(c_2, c_2'', \mathcal{A}_2) \wedge \ell(c_2'') = \ell_{\mathsf{ini}_2} \wedge (c_1', c_2'') \in \mathcal{W}),$$
   (c) $\forall c_2 \xrightarrow{\lambda^2} c_2'\ \exists c_1 \xrightarrow{\lambda^1} c_1' \bullet (c_1', c_2') \in \mathcal{W} \vee (\ell(c_2) = \ell_{\mathsf{fin}_2} \wedge$
   $$\exists c_2'' \bullet \mathsf{actReach}(c_2, c_2'', \mathcal{A}_2) \wedge \ell(c_2'') = \ell_{\mathsf{ini}_2} \wedge (c_1', c_2'') \in \mathcal{W}),$$

$\mathcal{A}_1$ is called weak-bisimilar to $\mathcal{A}_2$ iff there is a weak-bisimulation of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Theorem 6. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequential timed automata with the same period pt and final time fin_1 of $\mathcal{A}_1$ strictly smaller than the start time sta_2 of $\mathcal{A}_2$, i.e. $\mathsf{fin}_1 < \mathsf{sta}_2$. Then $\mathcal{A}_1 \,⨟\, \mathcal{A}_2$ is weak-bisimilar to $\mathcal{A}_1 \| \mathcal{A}_2$.*

7 Case Study

For the example in Section 2.1, we have used a simplified version of a fire alarm system which monitors the well functioning of n sensors by sending and receiving alive messages. For our case study, we consider a real world fire alarm system which we denote by *FAS* (the system is being developed by a German company; an anonymized version of a model of the system will be made public). The *FAS* monitors n sensors using m channels. In order, for *FAS* to obtain an EU quality certificate it has to be conform, among others, with the following condition: If a sensor is malfunctioning, it has to be recognized in less than 300 seconds. We denote this property by AG *less300*.

In addition, the certifying institution is able to (i) block an arbitrary channel for any number of seconds, then (ii) release the blocked channel for at least 1 second and repeat (i), (ii) any number of times.

In order to model the above mentioned situations, we constructed a sensor switcher *SW* which non-deterministically turns off any sensor. We constructed a channel blocker *CB*, which models the blocking of channels as described above. Now, let *FAS-CB* denote the fire alarm system together with the channel blocker *CB*. Let *FAS-SW* be *FAS* together with the sensor switcher. Let *FAS-CB-SW* be *FAS* together with the channel blocker and a sensor switcher.

Table 1. Verification times (AMD Opteron 6174 2.2GHz, 64Gb RAM) and visited states for satisfied properties $\varphi_1 = (\mathsf{AG}\ \textit{not deadlock})$ and $\varphi_2 = (\mathsf{AG}\ \textit{less300})$ for the fire alarm system with 125 sensors using Uppaal, and verification times of *FAS-CB-SW* for property (AG *less300*) over number of sensors

System	Q	Broadcast		Sequential	
		t (s)	states	t (s)	states
FAS-CB-SW	φ_1	1103.9	5947.4k	318.1	5971.5k
	φ_2	2240.5	11508.3k	704.2	11614.5k
FAS-CB	φ_1	196.4	1184.7k	120.3	1189.5k
	φ_2	272.6	165.7k	150.3	1666.9k
FAS-SW	φ_1	13.7	104.1k	87.4	104.7k
	φ_2	10.62k	145.5	87.4	146.4k
FAS	φ_1	2.5	20.7k	87.5	20.8k
	φ_2	1.3	20.7k	85.6	20.8k

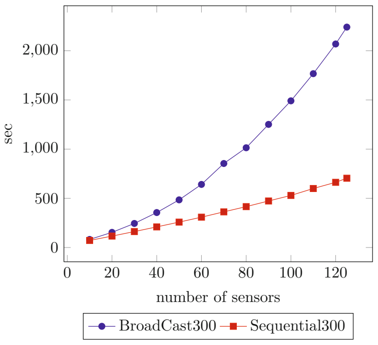


Table 1, show the verification results for the satisfied properties AG *not deadlock* and AG *less300* for the corresponding system with 125 sensors by using Uppaal. The times include the parsing time of Uppaal templates. The attempt of modeling a sensor, with its own clock, did not scale to more than 10 sensors. Therefore, our modelers manually optimized the system, such that all 125 sensors share one clock, and synchronizations are performed via a broadcast channel. This optimized systems correspond to the column Broadcast. In addition, the system *FAS-CB-SW* Broadcast corresponds to the system obtained by applying the technique presented in [8].

We observe that for a large state space, sequential is much faster that broadcast as expected by Lemma 3. However, for small state space such as *FAS* broadcast is faster; in this context, note that the parsing time for the large template consisting of 125 sequentialized automata is taking about 85 sec.

Considering the verification times of the system *FAS-CB-SW* and property AG *less300* for 10, 20, ..., 120, and finally 125 sensors, the curve for broadcast is comparable with the statement of Lemma 3 (cf. Table 1).

8 Conclusion and Future Work

We have presented an approach for optimizing the timed model checking method for the class of timed automata with disjoint activity. We have presented a syntactic transformation, by which parallel composition of its component automata is replaced by the application of a new sequential composition operator. The approach uses the syntactic transformation as a preprocessing step with an existing timed model checking method. We have implemented the approach (using Uppaal) and applied it to verify a wireless fire alarm system with more than 100 sensors. The experimental evaluation indicates the practical potential of the approach for improving upon the time cost in a useful manner.

For future work we may consider richer forms of expressing the property of sequentialisability, for example by means of handshaking communication.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static Guard Analysis in Timed Automata Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata. In: IEEE Proc. RTSS 1996, pp. 73–81. IEEE Computer Society Press (1996)
5. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2(3), 155–173 (1982)
6. Haartsen, J.C.: The Bluetooth radio system. IEEE Personal Communications 7(1), 28–36 (2000)
7. Heiner, G., Thurner, T.: Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In: FTCS, pp. 402–407 (1998)
8. Herrera, C., Westphal, B., Feo-Arenis, S., Muñiz, M., Podelski, A.: Reducing Quasi-Equal Clocks in Networks of Timed Automata. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 155–170. Springer, Heidelberg (2012)
9. Janssen, W., Poel, M., Xu, Q., Zwiers, J.: Layering of Real-Time Distributed Processes. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 393–417. Springer, Heidelberg (1994)
10. Kopetz, H.: The time-triggered approach to real-time systems design. In: Randell, B., et al. (eds.) Predictably Dependable Computing Systems. Springer (1995)
11. Kopetz, H., Bauer, G.: The Time-Triggered Architecture. Proc. IEEE, 112–126 (2003)
12. Kopetz, H., Grünsteidl, G.: TTP - a time-triggered protocol for fault-tolerant real-time systems. In: FTCS, pp. 524–533 (1993)
13. Olderog, E.R., Dierks, H.: Real-Time Systems - Formal Speci cation and Automatic Veri cation. Cambridge University Press (2008)
14. Kopetz, H., Grünsteidl, G.: TTP - a time-triggered protocol for fault-tolerant real-time systems. In: FTCS, pp. 524–533 (1993)

The Complexity of Bounded Synthesis for Timed Control with Partial Observability

Hans-Jörg Peter[1,2] and Bernd Finkbeiner[2]

[1] Advanced Research Center
Atrenta Inc.
38000 Grenoble, France
[2] Department of Computer Science
Saarland University
66123 Saarbrücken, Germany

Abstract. We revisit the synthesis of timed controllers with partial observability. Bouyer et al. showed that timed control with partial observability is undecidable in general, but can be made decidable by fixing the granularity of the controller, resulting in a 2ExpTime-complete problem. We refine this result by providing a detailed complexity analysis of the impact of imposing a bound on the size of the controller, measured in the number of locations. Our results identify which types of bounds are useful (and which are useless) from an algorithmic perspective. While bounding the number of locations without fixing a granularity leaves the problem undecidable, bounding the number of locations *and* the granularity reduces the complexity to NExpTime-complete. If the controller is restricted to be a discrete automaton, the synthesis problem becomes PSpace-complete, and, for a fixed granularity of the plant, even NPTime-complete. In addition to the complexity analysis, we also present an effective synthesis algorithm for location-bounded discrete controllers, based on a symbolic fixed point computation. Synthesis of bounded controllers is useful even if the bound is not known in advance. By iteratively increasing the bound, the synthesis algorithm finds the smallest, and therefore often most useful, solutions first.

1 Introduction

The theory of timed automata has made it possible to extend the algorithms for automatic verification and controller synthesis from discrete systems to real-time systems. An open challenge is, however, to effectively synthesize real-time controllers under partial observability, i.e., in situations where there are some events in the plant that the controller cannot observe.

Since the synthesis problem of real-time controllers under partial observability is in general undecidable [4], synthesis algorithms must focus on restricted classes of controllers. Bouyer et al. studied, for example, the synthesis problem with *fixed granularity*, where the number of clocks and the precision of the guards is limited in advance [11,4]. While this restriction ensures decidability, it unfortunately does not suffice to obtain an effective algorithm, because the synthesis problem

Table 1. Overview on the complexities of bounded synthesis for timed controllers with partial observability. The results written in **bold face** are established in this paper, the other results are taken from [4].

Bound \ Granularity	Unspecified	Fixed	Discrete
Unbounded	Undecidable	2ExpTime-complete	**2ExpTime-complete**
Locations	**Undecidable**	**NExpTime-complete**	**PSpace-complete / NPTime-complete**
Clocks	Undecidable	2ExpTime-complete	—

remains intractably expensive (2ExpTime-complete). Finding restrictions on the timed controllers that lead to a significant reduction in complexity thus remained an open question.

In this paper, we undertake a systematic study of the impact of various restrictions on the complexity of the controller synthesis problem. We introduce a bound on the *size* of the controller and limit the search to only those controllers that fall below the bound. In the setting of discrete systems, this idea is known as *bounded synthesis* [30]. For plants given as timed automata, natural adaptations of the bounded synthesis approach are to search for a controller with a bounded number of locations.

We analyze the complexity of the bounded synthesis problem under different types of bounds, and under different restrictions on the granularity. The results are summarized in Table 1. Some restrictions do not help: bounding the number of locations without fixing a granularity leaves the problem undecidable. Fixing both the granularity and a bound on the number of locations, however, reduces the complexity from 2ExpTime-complete to NExpTime-complete. Most interesting is the restriction to discrete controllers, where all clocks are located in the plant and the untimed controller only reacts to discrete events. Here, the complexity reduces to PSpace, i.e., the problem is exactly as hard as standard model checking. If the granularity of the plant is fixed, the complexity reduces further to NPTime.

Related Work. In his seminal work on discrete two-player games [27], Reif introduced the *knowledge-based subset construction* to transform a game with imperfect information to a game with perfect information. The construction causes an exponential blow-up. The (fully observable) timed controller synthesis problem in the framework of timed automata [1] was defined by Maler et al. by introducing two-player timed games [23,2]. The decidability of the problem was shown by demonstrating that the standard discrete attractor construction [31] on the region graph suffices to obtain timed controllers. Henzinger and Kopke showed that this construction is theoretically optimal by proving that the synthesis problem for safety properties is ExpTime-complete [16]. Controller synthesis against external specifications given as nondeterministic timed automata was considered by D'Souza and Madhusudan [11]. They were the first who discovered that fixing the granularity of the controller leads to decidability. Bouyer et al. extended this work by introducing partial observability for the controller [4]. A more pragmatic approach was investigated by Cassez et al. by restricting the choices and the observability of the controller so that the implementation of zone-based synthesis algorithms becomes possible [7]. An extension of this work uses *alternating timed simulation relations* to efficiently control partially observable systems [9]. An alternative restriction is to only consider controllers that match a given template. We recently obtained promising experimental results with an implementation that searches for such controllers using automatic abstraction refinement [14].

The idea of a-priori fixing syntactic properties of the system that should be synthesized resembles *bounded synthesis* [30] from the (fully observable, pure discrete) LTL synthesis community. Symbolic implementations based on SMT-solving [15], antichains [13], or BDDs [12] followed. In these works, one fixes the maximal number of states that the synthesized system may have. Recently, Kupferman et al. continued this line of research by distinguishing between bounding the system and/or the environment [19]. Following a similar idea, Lustig et al. proposed synthesizing systems based on component libraries [22].

Laroussinie et al. investigated the impact of bounding the number of clocks for model checking timed automata [20]. Chen and Lu extended this work to the fully observable synthesis setting by bounding the number of clocks in the plant [10], which is in contrast to this paper, where we impose bounds on the controller. For STRIPS planning, Rintanen [28] investigated the impact of no and partial observability on finding discrete plans.

Contributions of the Paper

- We provide the theoretical foundation for an extension of the bounded synthesis approach to the setting of real-time control. Our results identify which types of bounds are useful (and which are useless) from an algorithmic perspective.
- We provide matching lower and upper bounds for the complexity of the various synthesis problems, extending the complexity analysis of Bouyer et al. [4] to a complete picture of the controller synthesis problem for timed systems under partial observability. The proofs require nontrivial extensions of the techniques used in the literature that may also be of interest in other settings. For example, the proof of the NExpTime lower bound of Theorem 6 is based on an insightful connection between timed automata and the theory of problems on succinctly specified graphs.
- We demonstrate that bounded synthesis can be implemented in the setting of standard fixpoint-based verification tools for real-time systems. For this purpose, we present a construction that computes the set of discrete location-bounded controllers symbolically as a least fixed point.

Outline. We first recall the foundations of timed automata and timed controller synthesis with partial observability in Sections 2 and 3, respectively. In Section 4, we investigate the impact of bounding the locations of the controller. Finally, Section 5 introduces discrete controllers and investigates the impact of bounded and unbounded synthesis in this setting. For each lemma and theorem newly established in this paper, we give a brief description of the proof idea in the main part of the paper. The detailed versions of the proofs as well as their underlying technical constructions can be found in the appendix.

2 Timed Automata

In this section, we recall the timed automaton model by Alur and Dill.

Definition. A *timed automaton* [1] is a tuple $A = (L, l_0, \Sigma, \Delta, X)$, where L is a finite set of (control) locations, $l_0 \in L$ is the initial location, Σ is a finite set of actions, $\Delta \subseteq L \times \Sigma \times \mathcal{C}(X) \times 2^X \times L$ is an edge relation, X is a finite set of real valued clocks, and $\mathcal{C}(X)$ is the set of clock constraints over X. A clock constraint $\varphi \in \mathcal{C}(X)$ is of the form

$$\varphi \equiv \textbf{true} \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi_1,$$

where x is a clock in X, φ_1 and φ_2 are clock constraints from $\mathcal{C}(X)$, and c is a constant in $\mathbb{Q}_{\geq 0}$ encoded in binary. A *clock valuation* $\boldsymbol{t} : X \to \mathbb{R}_{\geq 0}$ assigns a nonnegative value to each clock and can also be represented by a $|X|$-dimensional vector $\boldsymbol{t} \in \mathcal{R}$, where $\mathcal{R} = \mathbb{R}^X_{\geq 0}$ denotes the set of all clock valuations. We write $l \xrightarrow{a,\varphi,\lambda} l'$ to refer to a tuple $(l, a, \varphi, \lambda, l')$ in Δ. We say that A is *deterministic* if, for any two distinct edges $l \xrightarrow{a,\varphi,\lambda} l'$ and $l \xrightarrow{a',\varphi',\lambda'} l''$, it holds that $a = a' \Rightarrow \varphi \wedge \varphi' \equiv \textbf{false}$. Following the setting of [2], we assume that timed automata are strongly nonzeno, i.e., there are no cycles where an infinite time-convergent sequence of transitions is possible.

The *(timed) states* of a timed automaton are pairs $(l, \boldsymbol{t})$ of locations and clock valuations. Timed automata have two types of transitions: *timed transitions*, where only time passes and the location remains unchanged, and *discrete transitions*, where no time passes, the current location may change and some clocks can be reset to zero. In a timed transition, denoted by $(l, \boldsymbol{t}) \xrightarrow{a} (l, \boldsymbol{t} + a \cdot \mathbf{1})$, the same nonnegative value $a \in \mathbb{R}_{\geq 0}$ is added to all clocks. A discrete transition, denoted by $(l, \boldsymbol{t}) \xrightarrow{a} (l', \boldsymbol{t}')$ for some $a \in \Sigma$, corresponds to an edge $(l, a, \varphi, \lambda, l')$ of Δ such that $\boldsymbol{t}$ satisfies the clock constraint φ, written as $\boldsymbol{t} \models \varphi$, and $\boldsymbol{t}' = \boldsymbol{t}[\lambda := 0]$ is obtained from $\boldsymbol{t}$ by setting the clocks in λ to 0.

We say that a state s is *forward reachable* if there is an $n \in \mathbb{N}$ and a finite sequence of transitions of the form $s_0 \xrightarrow{a_1} s_1 \dots s_{n-1} \xrightarrow{a_n} s_n$ such that $s_0 = (l_0, \mathbf{0})$ is the initial state (where $\mathbf{0}$ is the zero vector), $s_n = s$, and for all $1 \leq i \leq n$, $s_i = (l_i, \boldsymbol{t}_i)$ are states and $s_{i-1} \xrightarrow{a_i} s_i$ are transitions of the automaton, respectively. We say that the sequence $a_1 a_2 \dots a_n \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$ is a *timed prefix* of A and we define $L(A)$ as the set of all timed prefixes leading to states that are forward reachable.

Granularity. The *granularity* of a timed automaton defines its timing resources [11]. Formally, a granularity is represented by a tuple $\mu = (Y, m, c_{max})$, where Y is a finite set of clocks, $m \in \mathbb{N}_{\geq 1}$, and $c_{max} \in \mathbb{Q}_{\geq 0}$. We say that a timed automaton $A = (L, l_0, \Sigma, \Delta, X)$ is μ-granular if $X = Y$ and, for each constant $c \in \mathbb{Q}_{\geq 0}$ appearing in the clock constraints of the guards of the edges in Δ, it holds that c is an integer multiple of $\frac{1}{m}$ and $c \leq c_{max}$. We call the value of a clock $x \in X$ *maximal* if it is strictly greater than c_{max}.

Composition. Timed automata can be syntactically composed into networks, in which the automata run in parallel and synchronize on shared actions. For two timed automata $A_1 = (L_1, l_0^1, \Sigma_1, \Delta_1, X_1)$ and $A_2 = (L_2, l_0^2, \Sigma_2, \Delta_2, X_2)$, the *parallel composition* $A_1 \| A_2$ is the timed automaton $(L_1 \times L_2, (l_0^1, l_0^2), \Sigma_1 \cup \Sigma_2, \Delta, X_1 \cup X_2)$, where Δ is the smallest set that contains

- $(l_1, l_2) \xrightarrow{a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2} (l_1', l_2')$,
  if $a \in \Sigma_1 \cap \Sigma_2$, $l_1 \xrightarrow{a, \varphi_1, \lambda_1} l_1' \in \Delta_1$ and $l_2 \xrightarrow{a, \varphi_2, \lambda_2} l_2' \in \Delta_2$,
- $(l_1, l_2) \xrightarrow{a, \varphi_1, \lambda_1} (l_1', l_2)$,
  if $a \in \Sigma_1 \setminus \Sigma_2$, $l_1 \xrightarrow{a, \varphi_1, \lambda_1} l_1' \in \Delta_1$, and
- $(l_1, l_2) \xrightarrow{a, \varphi_2, \lambda_2} (l_1, l_2')$,
  if $a \in \Sigma_2 \setminus \Sigma_1$, $l_2 \xrightarrow{a, \varphi_2, \lambda_2} l_2' \in \Delta_2$.

If A_1 is (X, m, c_{max})-granular and A_2 is (X', m', c'_{max})-granular, then the *combined granularity* of $A_1 \| A_2$ is $(X \cup X', m \cdot m', \max(c_{max}, c'_{max}))$.

Finite Semantics. The decidability of the reachability problem of timed automata relies on the existence of the *region equivalence relation* [1] on $\mathcal{R}$ which has a finite index. In the following, we fix a μ-granular timed automaton $A = (L, l_0, \Sigma, \Delta, X)$ with $\mu = (X, m, c_{max})$. We say that two clock valuations $\boldsymbol{t}_1, \boldsymbol{t}_2 \in \mathcal{R}$ are in the same *clock region*, denoted $\boldsymbol{t}_1 \sim \boldsymbol{t}_2$, if

- the set of clocks with maximal value is the same in $\boldsymbol{t}_1$ and in $\boldsymbol{t}_2$ (i.e., $\forall x \in X : \boldsymbol{t}_1(x) > c_{max} \Leftrightarrow \boldsymbol{t}_2(x) > c_{max}$), and
- $m \cdot \boldsymbol{t}_1$ and $m \cdot \boldsymbol{t}_2$ agree (1) on the integer parts of the clock values, (2) on the relative order of the fractional parts of the clock values, and (3) on the equality of the fractional parts of the clock values with 0. That is, for all clocks x and y with nonmaximal value, it holds that
  (1) $\lfloor m \cdot \boldsymbol{t}_1(x) \rfloor = \lfloor m \cdot \boldsymbol{t}_2(x) \rfloor$,
  (2) $fr(m \cdot \boldsymbol{t}_1(x)) \leq fr(m \cdot \boldsymbol{t}_1(y)) \Leftrightarrow fr(m \cdot \boldsymbol{t}_2(x)) \leq fr(m \cdot \boldsymbol{t}_2(y))$, and
  (3) $fr(m \cdot \boldsymbol{t}_1(x)) = 0$ iff $fr(m \cdot \boldsymbol{t}_2(x)) = 0$,
  where $fr(m \cdot \boldsymbol{t}_i(x)) = m \cdot \boldsymbol{t}_i(x) - \lfloor m \cdot \boldsymbol{t}_i(x) \rfloor$ for $i \in \{1, 2\}$.

We denote with $[\boldsymbol{t}] = \{\boldsymbol{t}' \in \mathcal{R} \mid \boldsymbol{t} \sim \boldsymbol{t}'\}$ the clock region $\boldsymbol{t}$ belongs to. We say that two states $s_1 = (l_1, \boldsymbol{t}_1)$ and $s_2 = (l_2, \boldsymbol{t}_2)$ of A are *region-equivalent*, denoted by $s_1 \sim s_2$, if their locations are the same ($l_1 = l_2$) and the clock valuations are in the same clock region ($\boldsymbol{t}_1 \sim \boldsymbol{t}_2$), and denote with $[(l, \boldsymbol{t})] = \{(l, \boldsymbol{t}') \in L \times \mathcal{R} \mid \boldsymbol{t} \sim \boldsymbol{t}'\}$ the equivalence class of region-equivalent states that $(l, \boldsymbol{t})$ belongs to.

Regions are a suitable semantics for the abstraction of timed automata because they essentially preserve the set of time-abstracted prefixes: if there is a discrete transition $s \xrightarrow{a} s'$ from a state s to a state s' of a timed automaton, then there is, for all states t with $t \sim s$, a state t' with $t' \sim s'$ such that $t \xrightarrow{a} t'$ is a discrete transition with the same label. For timed transitions, a slightly weaker property holds: if there is a timed transition $s \xrightarrow{d} s'$ from a state s to a state s', then there is, for all states t with $t \sim s$, a state t' with $t' \sim s'$ such that there is a timed transition $t \xrightarrow{d'} t'$ (but possibly with $d' \neq d$).

The *finite semantics* of a timed automaton $A = (L, l_0, \Sigma, \Delta, X)$ is the finite directed graph $[\![A]\!]_\mu = (Q, q_0, T)$ where

– the symbolic state set $Q = \{[(l, \boldsymbol{t})] \mid (l, \boldsymbol{t}) \in L \times \mathcal{R}\}$ of $[\![A]\!]_\mu$ is the set of equivalence classes of region-equivalent states of A, with
– the initial state $q_0 = [(l_0, \boldsymbol{t}_0)]$, and
– the set $T = \{(q, q') \in Q \times Q \mid \exists t \in q,\ t' \in q', a \in \Sigma \cup \mathbb{R}_{\geq 0}. t \xrightarrow{a} t'\}$ of transitions.

The finite semantics of a timed automaton A is also sometimes called the *region graph* of A.

The finite semantics is reachability-preserving:

Lemma 1. [1] *For a timed automaton $A = (L, l_0, \Sigma, \Delta, X)$ there is a finite path from a state $(l, \boldsymbol{t})$ to a state $(l', \boldsymbol{t}')$ if, and only if, there is a finite path from $\left[(l, \boldsymbol{t})\right]$ to $\left[(l', \boldsymbol{t}')\right]$ in $[\![A]\!]_\mu$.*

Reachability Model Checking. The decidability of checking reachability properties for timed automata relies on the existence of the so called *region abstraction* that yields a *finite semantics*. Applying this abstraction on a given timed automaton gives a finite automaton whose number of states is linear in the locations and exponential in the granularity:

Lemma 2. [1] *For a μ-granular timed automaton $A = (L, l_0, \Sigma, \Delta, X)$ with $\mu = (X, m, c_{max})$, there always exists a finite automaton A' which preserves the reachability information of the states of A. Furthermore, the number of states of A' is bounded by*

$$\begin{aligned} &|L| \cdot |X|! \cdot 2^{|X|} \cdot \prod_{x \in X} O(m \cdot c_{max}) \\ = &|L| \cdot |X|! \cdot O(m \cdot c_{max})^{|X|}. \end{aligned}$$

For a given timed automaton A, we define $\mathsf{Reach}(A)$ as the set of all states forward reachable of A. For a set of states B, characterizing the bad states of A, we use $\mathsf{Safe}(A, B)$ as an abbreviation for $\mathsf{Reach}(A) \cap B = \emptyset$. We assume that B can be compactly represented by a Boolean predicate over the locations and clock values of A. The *model checking problem* (MC) is to decide whether $\mathsf{Safe}(A, B)$ is true. For deciding MC, the region abstraction is a theoretically optimal state space representation:

Theorem 1. *[1] For a timed automaton A and a set of bad states B, deciding* $\mathsf{Safe}(A, B)$ *is* PSPACE-*complete.*

3 Timed Control with Partial Observability

In this section, we recall some known results for timed controller synthesis, which form the starting point of our investigation.

Plants and Controllers. A *partially observable plant* is a tuple $(P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, where P is a timed automaton $(L, l_0, \Sigma, \Delta, X)$, Σ_{in} and $\Sigma_{\mathsf{out}}^{\mathsf{obs}}$ are the input and observable output actions, respectively, with $\Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma$, and $X^{\mathsf{obs}} \subseteq X$ are the observable clocks. For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, with $P = (L_p, l_0^p, \Sigma_p, \Delta_p, X_p)$, a *controller for $\mathcal{P}$* is a deterministic timed automaton $C = (L_c, l_0^c, \Sigma_c, \Delta_c, X_c)$ with $X_c \cap X_p = X^{\mathsf{obs}}$ and $\Sigma_c = \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}$ such that C does neither

(1) *reset plant clocks*: for each $l \xrightarrow{a,\varphi,\lambda} l' \in \Delta_c$, we require that $\lambda \cap X_p = \emptyset$;
(2) *inhibit plant actions*: for all timed prefixes $w \in L(P\|C)$ with $w.u \in L(P)$ and $u \in \Sigma_{\mathsf{out}}^{\mathsf{obs}}$, we require that $w.u \in L(P\|C)$; nor
(3) *introduce timelocks*: for all timed prefixes $w \in L(P\|C)$, we require that there is a $d \in \mathbb{R}_{\geq 0}$ and a $c \in \Sigma_{\mathsf{in}}$ such that $w.d.c \in L(P\|C)$.

We treat the case where the controller has complete information as a special case. We say that $(P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$ is *fully observable*, if

(1) P is deterministic,
(2) $\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}} = \Sigma$, and
(3) $X^{\mathsf{obs}} = X$.

For a (partially or fully observable) plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$ and a set of bad states B, the *controller synthesis problem* is to synthesize a controller C such that $\mathsf{Safe}(P\|C, B)$. Recall that we require timed automata (so the controllers) to be non-zeno. This way, we rule out trivial solutions consisting of a (physically unmeaningful) zeno controller that achieves its safety objective just by executing discrete actions infinitely often in a bounded amount of time.

Controller Synthesis. For the fully observable setting, Maler et al. showed that the controller synthesis problem can be reduced to solving a finite two-player safety game (which is known to be PTIME-complete [17]) on the finite semantics of the given plant. They also showed that a fully informed controller can always be expressed in the granularity of the plant:

Lemma 3. *[23] For a fully observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, where P is μ-granular, and a set of bad states B, if there is a controller C for $\mathcal{P}$, such that* $\mathsf{Safe}(P\|C, B)$*, then there is a μ-controller C' (i.e., with no own clocks) such that* $\mathsf{Safe}(P\|C', B)$.

Henzinger and Kopke proved that the game-theoretic synthesis algorithm based on the finite semantics is theoretically optimal.

Theorem 2. [16] *For a fully observable plant P and a set of bad states B, synthesizing a controller C for P, such that* $\mathsf{Safe}(P \| C, B)$*, is* ExpTime*-complete.*

Bouyer et al. showed that in the presence of partial observability, the general timed synthesis problem becomes undecidable, even when a bound is imposed on the number of clocks of the controller.

Theorem 3. [4,3] *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$ and a set of bad states B, the following holds:*

(1) Synthesizing a controller C for $\mathcal{P}$, such that $\mathsf{Safe}(P \| C, B)$*, is undecidable.*
(2) For a given integer constant $k \in \mathbb{N}_{\geq 1}$, synthesizing a controller C for $\mathcal{P}$ with k clocks, such that $\mathsf{Safe}(P \| C, B)$*, is undecidable.*

However, an important result of their work is that by imposing a granularity bound on the controller, one achieves decidability.

Theorem 4. [4,3] *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, a granularity μ, and a set of bad states B, synthesizing a μ-controller C for $\mathcal{P}$, such that* $\mathsf{Safe}(P \| C, B)$*, is* 2ExpTime*-complete.*

Inspired by the last theorem, our paper continues this line of research by investigating finer bounds on the controller.

This concludes the recalling of the results that can be found in the literature. Based on these results, we start with our investigation.

4 Location-Bounded Controllers

This section starts the presentation of our new results. First, we investigate the impact of bounding only the number of locations while leaving the granularity unspecified. It turns out that this does not bring decidability.

Theorem 5. *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, a set of bad states B, and an integer $k \in \mathbb{N}_{\geq 1}$, synthesizing a controller C for $\mathcal{P}$ with k locations, such that* $\mathsf{Safe}(P \| C, B)$*, is undecidable.*

The proof is based on a reduction from the halting problem of a given two-counter Minsky machine, which is known to be undecidable [24]. The basic idea is to let the synthesis algorithm generate a controller that simulates an accepting run of the machine, or to report that no such controller/run exists. Following the standard construction proposed in [3] (which, in turn, is an extension of the one proposed in [1]) we let the plant nondeterministically and unobservably for the controller verify that he faithfully performs the simulation. The challenge in obtaining the undecidability result of Theorem 5 is to reduce the halting problem to the existence of a controller with a *bounded* number of locations. In the appendix, we give a novel encoding where the computation of the Minsky machine is entirely stored in the clock values. The proof thus reduces the halting problem to the existence of a controller with a *single* location.

When bounding the locations of the controller *and* its granularity, the complexity for controller synthesis drops from 2ExpTime-complete (cf. Theorem 4) to NExpTime-complete.

Theorem 6. *For a partially observable plant* $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$*, a granularity* μ*, a set of bad states* B*, and a bound* $k \in \mathbb{N}$*, synthesizing a* μ*-controller* C *for* $\mathcal{P}$ *with* k *locations, such that* $\mathsf{Safe}(P \| C, B)$*, is* NExpTime*-complete.*

We note that this result does not depend on the (unary or binary) encoding of k and μ. The proof of the NExpTime lower bound is based on a novel proof technique that uses clocks to represent bits for querying an edge relation of a succinctly represented graph. The key idea is to represent exponentially many nodes via only polynomially many clocks. We provide a polynomial-time reduction from Succinct graph coloring, which is known to be NExpTime-complete [21,26,32]. In our reduction, we use an answer of the synthesis problem to decide whether there is a k-coloring of a given undirected graph that is succinctly represented (i.e., the graph's edge relation E is given by a Boolean function).

In the (possibly infinite) interaction between plant and controller, the plant nondeterministically selects a node n and queries a color c from the controller. Then, the plant selects a second node n' and queries a color c'. If n and n' are connected via E and c and c' are the same, the plant enters a bad state. Otherwise, the colors of another two nodes are queried, and so on. A selected node is communicated to the controller by letting him read the values of the clocks representing that node.

For showing the NExpTime upper bound, one can provide a nondeterministic algorithm that guesses a controller in exponential time, and then validates that the guess was correct. For a granularity $\mu = (X, m, c_{max})$, the number of distinct atomic constraints is bounded by

$$\gamma = \prod_{x \in X} O(m \cdot c_{max}) = O(m \cdot c_{max})^{|X|},$$

which is single exponential (recall that m and c_{max} are given in binary using polynomially many bits). Now, in each location admitted to C, for each atomic constraint and each event from Σ, we have to decide (1) which clocks to reset and (2) in which location to change next. Note that, because we require C to be deterministic, we only have to make this decision once for every atomic constraint. Hence, since this choice has to be repeated for every location admitted to C, the number of possible controllers is bounded by

$$\left(k \cdot 2^{|X|}\right)^{\gamma \cdot |\Sigma| \cdot k}$$

and a single controller can thus be represented using

$$\gamma \cdot |\Sigma| \cdot k \cdot \left(\lceil \log k \rceil + |X|\right)$$

(i.e., only single exponentially) many bits. The validation relies on model checking, which is, according to Theorem 1, in PSpace $\subseteq$ NExpTime. Note that, from a complexity-theoretic point of view, this is the best one can do, since any deterministic algorithm would have a double exponential worst-case running time, unless NExpTime = ExpTime.

Concerning the size of the representation of the smallest feasible controller, if there is one at all, the following theorem states that it is highly unlikely[1] that a small controller always exists.

Theorem 7. *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, a granularity μ, a set of bad states B, and a bound $k \in \mathbb{N}$, if there is a μ-controller C for $\mathcal{P}$ with k locations, such that* $\mathsf{Safe}(P \| C, B)$*, then C cannot always be represented polynomially, unless* NExpTime = PSpace.

5 Discrete Controllers

In this section, we investigate the impact of restricting the controller to be a pure discrete system communicating synchronously with an arbitrary timed plant.

Definition. For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, we say that a controller $C = (L_c, l_0^c, \Sigma_c, \Delta_c, X_c)$ is *discrete*, if $|X_c| = 1$ and for each $l \xrightarrow{a,\varphi,\lambda} l' \in \Delta_c$ it holds that $\lambda = X_c$ and either

(1) $a \in \Sigma_{\mathsf{out}}^{\mathsf{obs}}$ and $\varphi \equiv \mathbf{true}$, or
(2) $a \in \Sigma_{\mathsf{in}}$ and $\varphi \equiv x \leq 0$, assuming $X_c = \{x\}$.

Intuitively, discrete controllers only react to discrete observations of the plant. They are not allowed to measure the time between two observed events.

We want to point out that discrete controllers differ from controllers with a fixed sampling rate considered in [16,8]. Obviously, the only meaningful bound which one can impose on discrete controllers is to restrict the number of locations. In the following, we investigate the bounded and the unbounded case.

5.1 Bounded Case

Requiring that the controller should be discrete, and, additionally, bounding the number of locations of the controller, reduces the complexity of the synthesis problem from 2ExpTime-complete (cf. Theorem 4) to PSpace-complete. The problem is thus exactly as hard as model checking (cf. Theorem 1).

Theorem 8. *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, a set of bad states B, and a bound $k \in \mathbb{N}$, synthesizing a discrete controller C for $\mathcal{P}$ with k locations, such that* $\mathsf{Safe}(P \| C, B)$*, is* PSpace*-complete.*

[1] As it is common belief that PSpace $\subsetneq$ ExpTime $\subsetneq$ NExpTime.

The lower bound immediately follows from the PSpace-hardness of timed model checking, which is easily seen to be a special case.

Containment in NPSpace (which is known to coincide with PSpace [29]) can be established through the following nondeterministic algorithm. As the synthesized controller must be discrete and since every controller should be deterministic, a number of bits polynomial in $|\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}| \cdot \log k$ suffices to fully describe a controller. Hence, our algorithm can just guess these bits in polynomial time and then use timed model checking as an oracle to verify the guess. In summary, our algorithm is in $\text{NPTime}^{\text{MC}} \subseteq \text{NPSpace} = \text{PSpace}$.

An Effective Synthesis Algorithm. To illustrate the practical relevance of the PSpace upper bound, we now describe an effective deterministic algorithm for the synthesis of bounded discrete controllers. The algorithm is based on a symbolic fixed point iteration. We use (a polynomial number of) Boolean variables to represent the structure of the controller (i.e., which locations are connected via an edge with a certain action). Sets of locations are represented using Boolean functions over a set of $O(\log k)$ location variables.

Let R be the set of states of the finite semantics of the plant, S be the set of all possible controller structures, and L be the set of all locations for all possible structures. Our algorithm incrementally computes a partial function $f : R \to S \to L$ such that, for each location $l \in f(r, s)$, the combined plant/controller state (r, l) is backward reachable assuming that the controller is of structure s. In an actual implementation, one would represent f as a mapping from regions to tuples from $2^S \times 2^L$, which, in turn, can be efficiently represented using discrete symbolic data structures (such as binary decision diagrams).

Initially, f maps each bad region to **true** (representing all controllers and locations) and each other region to **false** (representing no controllers and no locations). In each step of the fixed point iteration, we backpropagate from each region r the annotated pair of controller structures/locations over all transitions leading to r. When backpropagating a pair, represented by a Boolean formula φ, over a transition t, the resulting formula φ' is obtained by computing the weakest predecessors of φ. For the source region r' of t, we update $f(r') := f(r') \vee \varphi'$.

Once the fixed point is reached, we can derive the feasible controller structures from the annotation of the initial region. For this purpose we quantify the conjunction of the annotation of the initial region with the initial controller location existentially over the location variables. The set of structures characterized by the resulting Boolean function are the *infeasible* controllers. Hence, the negation yields the *feasible* controllers.

Since, according to Lemma 2, there are only single-exponentially many regions, and since a particular region is visited at most single-exponentially often (because there are only single-exponentially many controllers), we obtain in total a single-exponential running time (note that the two exponents multiply). We can therefore conclude that, unless PTime = PSpace, the time-complexity of the deterministic algorithm matches the complexity established in Theorem 8.

Inspired by the argumentation for the upper bound in Theorem 8, one might ask for the complexity of the synthesis problem if we impose a polynomial bound on the finite semantics of the plant. We can show that, in this case, the synthesis problem even becomes NPTime-complete.

Lemma 4. *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$, where the number of regions of P is polynomial in the size of P, a set of bad states B, and a bound $k \in \mathbb{N}$, synthesizing a discrete controller C for P with k locations, such that $\mathsf{Safe}(P \| C, B)$, is* NPTime*-complete.*

The lower bound can be shown by a reduction from Graph coloring, which is known to be NPTime-hard. The reduction goes analogously to the one for establishing the lower bound for Theorem 6 with the difference that, here, we use polynomially many locations (and no clocks) to represent the explicitly given graph in the plant.

Before we prove containment in NPTime, let us first ascertain the following fact that immediately follows from the well-known result that reachability checking on explicitly represented graphs is NLogSpace-complete:

Lemma 5. *[18] For a given directed graph $G = (V, E)$ with nodes V and edges E, and a set of bad nodes $V' \subseteq V$, finding a lasso (i.e., a path leading to and comprising some cycle in G), which avoids any nodes in V', is* NLogSpace*-complete.*

Now, the NPTime upper bound of Lemma 4 can be established by the following nondeterministic algorithm that runs in polynomial time. Analogously to establishing the upper bound for Theorem 8, we first guess a controller in polynomial time. But now, the model checking procedure runs on a region graph of only polynomial size and, according to Lemma 5, requires only logarithmic space. Thus, the problem is in $\text{NPTime}^{\text{NLogSpace}} = \text{NPTime}$.

It is straight forward to see that, according to Lemma 2, the number of regions is only exponential in the granularity, but linear in the number of locations. Also, note that no plant clocks were used in establishing the NPTime lower bound for the last lemma. Consequently, we can state the following corollary.

Corollary 1. *For a fixed granularity μ, the problem of synthesizing a bounded discrete safety controller for a partially observable μ-granular plant is* NPTime*-complete.*

5.2 Unbounded Case

It turns out that the restriction to discrete controllers does not pay off in the unbounded case. In fact, we obtain the same 2ExpTime complexity bounds (cf. Theorem 4) as for the general synthesis problem already investigated in the literature [11,4,3].

Theorem 9. *For a partially observable plant $\mathcal{P} = (P, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, X^{\mathsf{obs}})$ and a set of bad states B, synthesizing a discrete controller C for $\mathcal{P}$ with an unspecified number of locations, such that $\mathsf{Safe}(P \| C, B)$, is* 2ExpTime*-complete.*

The upper bound follows immediately from the upper bound established in Theorem 4. However, we additionally provide a deterministic algorithm that runs in double exponential time. First, we obtain a new plant automaton P' by enriching P by a fresh clock x, which is reset to 0 on every edge with an action $a \in \Sigma_{\mathsf{in}}$. On every edge of P' with an action $a \in \Sigma_{\mathsf{out}}^{\mathsf{obs}}$, we strengthen the guard with $x \leq 0$. Then, we construct the region graph of P', which, according to Lemma 2, is of single exponential size. We hide unobservable action and delay transitions by replacing them by ε-transitions. Finally, we obtain an equivalent finite game with perfect information by constructing the so-called belief space [27], which leads to a second exponential blowup. Since solving pure discrete safety games is PTime-complete [17,16], we conclude that our algorithm requires double exponential time.

The AExpSpace = 2ExpTime lower bound, which is more technically involved, is established by a reduction from the halting problem of an alternating Turing machine whose tape length is bounded exponentially in the size of the input. In our reduction, the Turing machine reaches its final state iff there exists a safe controller. Similar to a proof presented by Rintanen in the reachability planning setting [28], instead of storing the contents of the whole tape, we let the plant, unobservable for the controller, select a dedicated tape cell that should be watched. Unlike in the pure discrete setting of [28], we use polynomially many clocks (instead of Boolean variables) to represent the bits of some integer variables encoding the exponentially large index of the watched tape cell and the current position of the tape head.

Also different to [28], as our interest lies in safety controllers, we need to avoid that a controller is synthesized that never reaches the final state by infinitely looping through some other states. For this, we introduce a counter that keeps track of the number of steps executed so far. Since the maximal number of steps without visiting a state twice corresponds to the number of possible configurations, we can use this maximal number as a general bound, beyond which the plant immediately enters a bad state. Unfortunately, as this number is double exponential in the number of bits (i.e., clocks), we cannot use just another integer variable to represent the step counter (because this variable would comprise exponentially many bits). Instead, we let the plant force the controller to faithfully produce the correct sequence of bits of the step counter. Again, instead of remembering all bits, we let the plant nondeterministically and unobservably for the controller select a dedicated bit that should be watched, whose correct incrementation is verified.

We point out that, for establishing the lower bound, the 2ExpTime-hardness proof given in [11], where the controller is timed and can observe all clocks, does not apply to our setting of discrete controllers with partial observability.

6 Conclusion

In this paper, we have extended the bounded synthesis approach [30] to timed systems. We have established the complexity of timed control with partial observability under different types of bounds, and under different restrictions on the granularity. Our results in particular identify the synthesis of discrete controllers (over timed plants with limited observability) as a special case with significant practical relevance and, at the same time, very reasonable complexity: synthesizing discrete controllers is no harder than model checking, and can, in fact, be implemented with a symbolic fixed point iteration similar to BDD-based model checking [5,6]. Our results thus draw a much more optimistic picture for the synthesis of realistic controllers than previous work on the unbounded synthesis problem, where the introduction of real-time was shown to cause an exponential blow-up and partial observability was shown to make the problem undecidable.

The bounded synthesis approach is useful both when a reasonable bound is fixed *a priori* and when no bound is known in advance and the algorithm must, instead, *search* for the right bound. Bounded synthesis with iteratively increasing bounds is a complete method for the unbounded synthesis problem, with the significant advantage over previously studied approaches that the smallest solutions are found first.

Acknowledgment. This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Lafay, J.-F. (ed.) Proc. 5th IFAC Conference on System Structure and Control, pp. 469–474. Elsevier (1998)
3. Bouyer, P., Chevalier, F.: On the control of timed and hybrid systems. EATCS Bulletin 89, 79–96 (2006)
4. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed Control with Partial Observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
7. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed control with observation based and stuttering invariant strategies. In: Namjoshi, et al. [25], pp. 192–206
8. Cassez, F., Henzinger, T.A., Raskin, J.-F.: A Comparison of Control Problems for Timed and Hybrid Systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)

9. Chatain, T., David, A., Larsen, K.G.: Playing games with timed games. In: Giua, A., Silva, M., Zaytoon, J. (eds.) Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 2009), Zaragoza, Spain (September 2009)
10. Chen, T., Lu, J.: Towards the complexity of controls for timed automata with a small number of clocks. In: Ma, J., Yin, Y., Yu, J., Zhou, S. (eds.) FSKD (5), pp. 134–138. IEEE Computer Society (2008)
11. D'Souza, D., Madhusudan, P.: Timed Control Synthesis for External Specifications. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 571–582. Springer, Heidelberg (2002)
12. Ehlers, R.: Symbolic Bounded Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 365–379. Springer, Heidelberg (2010)
13. Filiot, E., Jin, N., Raskin, J.-F.: An Antichain Algorithm for LTL Realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
14. Finkbeiner, B., Peter, H.-J.: Template-Based Controller Synthesis for Timed Systems. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 392–406. Springer, Heidelberg (2012)
15. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Proceedings of the 2nd Workshop on Automated Formal Methods (AFM 2007), Atlanta, Georgia, USA, November 6, pp. 69–76. ACM Press (2007)
16. Henzinger, T.A., Kopke, P.W.: Discrete-time control for rectangular hybrid automata. Theoretical Computer Science 221(1-2), 369–392 (1999)
17. Immerman, N.: Number of quantifiers is better than number of tape cells. J. Comput. Syst. Sci. 22(3), 384–406 (1981)
18. Jones, N.D.: Space-bounded reducibility among combinatorial problems. J. Comput. Syst. Sci. 11(1), 68–85 (1975)
19. Kupferman, O., Lustig, Y., Vardi, M.Y., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: Schwentick, T., Dürr, C. (eds.) STACS. LIPIcs, vol. 9, pp. 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
20. Laroussinie, F., Markey, N., Schnoebelen, P.: Model Checking Timed Automata with One or Two Clocks. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 387–401. Springer, Heidelberg (2004)
21. Lozano, A., Balcázar, J.L.: The Complexity of Graph Problems for Succinctly Represented Graphs. In: Nagl, M. (ed.) WG 1989. LNCS, vol. 411, pp. 277–286. Springer, Heidelberg (1990)
22. Lustig, Y., Vardi, M.Y.: Synthesis from Component Libraries. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 395–409. Springer, Heidelberg (2009)
23. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
24. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc. (1967)
25. Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.): ATVA 2007. LNCS, vol. 4762. Springer, Heidelberg (2007)
26. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
27. Reif, J.H.: The complexity of two-player games of incomplete information. J. Comput. Syst. Sci. 29(2), 274–301 (1984)

28. Rintanen, J.: Complexity of planning with partial observability. In: Zilberstein, S., Koehler, J., Koenig, S. (eds.) ICAPS, pp. 345–354. AAAI (2004)
29. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. 4(2), 177–192 (1970)
30. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, et al. [25], pp. 474–488
31. Thomas, W.: On the Synthesis of Strategies in Infinite Games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
32. Veith, H.: Languages represented by boolean formulas. Inf. Process. Lett. 63(5), 251–256 (1997)

Static Detection of Zeno Runs in UPPAAL Networks Based on Synchronization Matrices and Two Data-Variable Heuristics

Jonas Rinast and Sibylle Schupp

Hamburg University of Technology, Institute for Software Systems,
D-21073 Hamburg, Germany
{jonas.rinast,schupp}@tu-harburg.de

Abstract. This paper addresses Zeno runs, i.e., transition sequences that can execute arbitrarily fast, in the context of model checking with the UPPAAL tool. Zeno runs conflict with real-world experience where execution always takes time and they may introduce timelocks into the models. We enhance previous work on static detection of Zeno runs by extending synchronization exploitation using a synchronization matrix that not only ensures valid synchronization partners exist but also that their number is correct. Additionally, we introduce two data-variable heuristics into the analysis as in most models data-variable constraints prevent certain Zeno runs. The analysis is implemented in a tool called ZenoTool and empirically evaluated using 13 benchmarks. The evaluation shows that our analysis is more accurate in 3 cases and never less accurate than the analysis results of previous work and that performance and memory overhead are at the same time very low.

Keywords: Zeno Runs, Timed Automata, UPPAAL.

1 Introduction

Timed automata [3] provide an easy-to-use, graphical way to model systems that need to adhere to timing constraints. However, specifications may suffer from erroneous states, e.g., timelocks, which are blocking states analogous to deadlocks in untimed systems. The potential occurrence of Zeno runs—paths in the timed automata system that execute infinitely many action transitions in finite time—complicates the matter as timelocks involving them (so-called *Zeno timelocks*) can not be detected by standard deadlock detection methods. Model-checking tools like UPPAAL can verify liveness properties to ensure the absence of Zeno runs. However, the verification process needs to explore the whole state space of the system, which might render the verification of complex systems infeasible. This paper therefore explores static analysis techniques to enlarge the class of systems that can be proved safe in regards to Zeno runs. We improve the accuracy of the static Zeno run detection technique developed by Gómez and Bowman [14], which uses the strong non-Zenoness (SNZ) property introduced by Tripakis [27], by refining the used method to exploit synchronization. In addition, we introduce two heuristics that enhance the method by considering data-variable valuations.

In brief, our analysis works as follows. First, we extract all simple loops in the timed automata system by modifying an existing cycle detection algorithm [24–26] to process multiple edges between the same locations correctly. Those loops are checked for their strong non-Zenoness property by inspecting relevant clock assignments and constraints. Next, loops are declared safe or unsafe based on external updates to clock variables that may invalidate the SNZ property. We then determine whether valid synchronization scenarios involving a loop exist by solving linear (in)equation systems using our synchronization matrix, and dismiss loops without one. The synchronization matrix represents the synchronization capabilities of all unsafe loops and also considers the number of partners required in contrast to the synchronization-group approach [14]. Concerning data-variable heuristics, we introduce one that extends the safety propagation to data variables such that loops are considered safe if they depend on data-variable valuations that can only be obtained from safe loop iterations. The second heuristics eliminates loops that can not iterate at run time because of conflicting constraints requiring different variable valuations.

We implemented the analysis in a tool named ZenoTool, along with several models from different case studies from the literature [23]. We also implemented the synchronization-group approach [14] and compared it with the results of our method. Of the 13 models analyzed, our analysis is in 3 cases more accurate. Specifically, one model benefits from the introduction of the synchronization matrix and the two other improvements are attributed to the data-variable heuristics. The memory overhead is less than 15% and the hit in performance is less than 17% on average in spite of the fact that we employ more machinery.

This paper is organized as follows. Section 2 introduces UPPAAL's timed automata model as well as Zeno runs. Section 3 presents the analysis by Gómez and Bowman including the strong non-Zenoness property and the notion of loop safety. Section 4 introduces our synchronization-matrix approach and Section 5 deals with the data-variable heuristics. Section 6 focuses on our implementation and the empirical results, Section 7 summarizes related work, and, at last, Section 8 concludes the paper and suggests further research.

2 Zeno Runs in UPPAAL Networks

2.1 UPPAAL's Timed Automata Model

The timed automata model used by UPPAAL is an extended model of the one proposed by Alur and Dill [3]. A system may consist of multiple, possibly concurrent components and every component is modeled as a finite state machine: a set of locations and edges with an initial location forming a directed graph. Communication between concurrent components is achieved by synchronization on user-defined channels and shared, bounded data variables. Timing constraints are realized by special clock variables, which take values in the positive reals: All clock variables advance simultaneously at the same rate but can be reset individually upon firing of an edge. Edges represent action transitions of the system. Basic annotations on edges are updates to variables including clocks (resets), basic boolean expressions on (time) variables (guards), and synchronization labels. Guards state whether or not an edge may be fired at a certain state. Synchronization labels relate multiple edges and indicate that certain edges fire synchronously. UPPAAL defines binary synchronization (1-to-1 relation) where both edges must be available (required synchronization) and broadcast synchronization (1-to-many relation) where receivers need not be available (optional synchronization). Time transitions are implicitly possible in locations. They can therefore be annotated with so-called invariants. The system may only remain in a location while the clock invariant is satisfied. We now formalize relevant parts of the underlying model. For more detail we refer to the paper by Bowman and Gómez [14].

A *timed automaton* is a tuple $A = (L, l_0, Lab, E, I, \mathcal{V})$. L denotes the set of locations; $l_0 \in L$ is the initial location of the timed automaton and Lab the set of synchronization labels used by the automaton. E is the set of edges and I a mapping function, which may assign an invariant to locations. $\mathcal{V}$ is the set of variables the timed automaton uses. As it is common, edges $(l, a, g, u, l') \in E$ will also be written in the format $l \xrightarrow{a,g,u} l'$, where a is the synchronization label, g the guard, and u the update.

A *network of timed automata* is an aggregation of n single automata $N = \langle A_1, \ldots, A_n \rangle$, where $A_i = (L_i, l_{0,i}, Lab_i, E_i, I_i, \mathcal{V}_i)$. A location in the network is represented by the location vector containing the positions in the individual components $\bar{l} = \langle l_1, \ldots, l_\mathrm{n} \rangle,\ l_i \in L_i$.

The semantics of N is given by a *timed transition system* $(S, s_0, (\{\epsilon\} \cup \mathrm{I\!R}^+), T)$, where $S = \langle \bar{l}, v \rangle$ is the set of reachable states using a location $\bar{l}$ and a variable valuation function v that maps variables to their current actual values. States are *reachable* if they can be reached by execution of the underlying network of timed automata. The initial system state is given by s_0, and $T \subseteq S \times (\{\epsilon\} \cup \mathrm{I\!R}^+) \times S$ is the transition relation that links source and destination states with an action indicator (ϵ) or a time delay ($\mathrm{I\!R}^+$). We use $s \overset{\gamma}{\Longrightarrow} s'$, $\gamma \in (\{\epsilon\} \cup \mathrm{I\!R}^+)$, to denote state transitions.

2.2 Zeno Runs and Zeno Timelocks

Wrong usage of synchronization, false time constraints, or invalid application of urgency may introduce flaws into a model. Dead- and timelocks are one category of effects that may arise. Timelocks occur in specifications if time is unable to progress further due to constraints. Zeno timelocks are especially difficult to track down because they can not be detected by standard deadlock detectors as only the delay of time is blocked and action transitions are still possible. We now give a short formalization of Zeno runs and Zeno timelocks.

At first, we define a run $\rho := s_1 \overset{\gamma_1}{\Longrightarrow} s_2 \overset{\gamma_2}{\Longrightarrow} \ldots,\ s_i \in S,\ \gamma_i \in \{\epsilon\} \cup \mathrm{I\!R}^+$ to be a path in the timed transition system. A run may be finite, ending in a state $s_\mathrm{n} \in S$, or infinite. A run ρ is *time-divergent* if the sum of all delays occurring in ρ is infinite.

Definition 1 (Zeno run). *A run ρ is a* Zeno run *if ρ is infinite and ρ is not time-divergent.*

The definition characterizes an infinite path in the transition system where time converges and thus infinite actions are executed in finite time. Such behavior is impossible in real systems and one needs to be closely examine whether a Zeno run approximation is appropriate in a certain model.

Definition 2 (Zeno timelock). *A state s is a* Zeno timelock *if all runs starting in s are not time-divergent and for all finite runs starting in s a Zeno run exists that starts with the complete transition sequence of the corresponding finite run.*

The definition ensures that upon reaching a Zeno timelock state s all runs are not time-divergent and additionally all finite runs can be extended to Zeno runs with the intend of excluding normal deadlocks. An example of a Zeno timelock is given in Fig. 1. The single transition is not constrained and can be executed at all times. Thus, Zeno runs are possible. Time, however, can not advance past five time units and therefore all states in the system are Zeno timelocks.

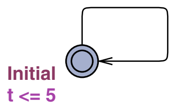


Fig. 1. Zeno run and Zeno timelock

3 Strong Non-zenoness and Loop Safety

Strong non-Zenoness (SNZ) is a static property of loops in a timed transition system. A *loop* or *elementary cycle* is a transition sequence $\langle l_0 \xrightarrow{a_1,g_1,u_1} l_1 \ldots l_{n-1} \xrightarrow{a_n,g_n,u_n} l_n\rangle$, where $l_0 = l_n$ and $l_i \neq l_j$ for all $0 \leq i \neq j \leq$ n. *Observable loops* are loops that synchronize on a channel. Furthermore, a run *covers* a loop if all edges of the loop are fired infinitely often in the run. SNZ was introduced by Tripakis [27] to classify loops according to their ability to contribute to Zeno runs. A loop is strongly non-Zeno if a clock variable t is bounded from below by a guard $(t \geq n, n \geq 1)$ and reset in the same loop $(t = 0)$. This constellation ensures that time needs to pass when the loop is iterated. The guard only becomes enabled after a sufficient amount of time has passed as the clock is reset every iteration.

Gómez and Bowman modified the definition of strong non-Zenoness to allow more loops to be classified accurately [9]. Additionally, they restricted the number of loops to analyze in the network to elementary cycles only. They also proposed the exploitation of synchronization such that not all loops in a network need to be strongly non-Zeno for it to be considered free from Zeno runs.

Advanced modeling features of UPPAAL however may invalidate SNZ. Non-zero clock updates, updates of the form $t = k$ where $k > 0$, may render lower bounds on clocks meaningless when inferring strong non-Zenoness. Even though a clock is bound from below and reset in a loop a Zeno run may occur if the valuation of the clock still satisfies the lower-bound imposing guard after the clock is reset. Also, non-zero clock updates introduce an order dependency when multiple updates to the same clock occur in a loop. A definition of strong non-Zenoness considering non-zero updates is given in the following.

Let Φ be a clock constraint and g a guard. $\Phi \in g$ denotes a constraint Φ that can be inferred from the guard g. Let x and y be clocks, let m be a natural number including zero ($m \in \mathbb{N}^0$), and u be an update annotation on an edge. We use $x = m \in u$ to denote an update u that sets the valuation of x to m. In addition, if e is an edge in a loop lp annotated with a guard g and n is a natural number ($n \in \mathbb{N}$), we define $x_{\sqsupseteq n} \in g$ to express that either $x \sqsupseteq n \in g$ ($\sqsupseteq \in \{=, >, \geq\}$) or $x - y \sqsupseteq n \in g$ ($\sqsupseteq \in \{>, \geq\}$). Using those definitions we define x_{lb} to be the lower bound for x in g if $x_{\sqsupseteq x_{\text{lb}}} \in g$ and there is no $x'_{\text{lb}} > x_{\text{lb}}$ such that $x_{\sqsupseteq x'_{\text{lb}}} \in g$. At last, we use $\mathcal{U}(p, q)$ to denote the set of updates that occur on the edges on the path from p to q ($\langle p \xrightarrow{a_1, g_1, u_1} e_1 \dots e_{\text{n}-1} \xrightarrow{a_\text{n}, g_\text{n}, u_\text{n}} q \rangle$). We rephrase the SNZ definition [14] as follows:

Definition 3 (Strongly non-Zeno loop). *A loop lp is* strongly non-Zeno (SNZ) *if there exists a clock x with lower bound x_{lb}, an edge e_u with an update u, and an edge e_g with a guard g in lp, and there are natural numbers m, m' that adhere to the following constraints:*

$$x = m \in u \qquad \text{clock reset} \qquad (1)$$

$$m < x_{\text{lb}} \qquad \text{valid reset} \qquad (2)$$

$$\forall u \in \mathcal{U}(e_\text{u}, e_\text{g})(\nexists x = m' \in u(m' \geq x_{\text{lb}})) \qquad \text{lower bound invalidation} \qquad (3)$$

Equation (1) ensures the clock variable x is reset in the loop. Equation (2) assures the clock variable is reset to a value lower than the corresponding lower bound and thus time has to pass when iterating the loop. At last, (3) guarantees that there is no update to the clock variable on the path from e_u to e_g that invalidates the clock reset. A clock x that complies with those constraints is called an *SNZ witness* for the loop *lp*.

In UPPAAL an SNZ loop, however, may still contribute to a Zeno run. External updates to non-local clock variables may render possible infinitely fast iterations. If a global clock variable is an SNZ witness for a loop other loops may assign values to the clock variable that satisfy the constraint of the SNZ loop. If such an external update can execute infinitely fast the SNZ loop also is prone to Zeno runs as no time is required to fulfill the clock constraint. A safety property to accommodate external updates is given next.

Definition 4 (Safe loop). *A loop lp in an UPPAAL specification is considered* safe *from Zeno runs if it is a strongly non-Zeno loop (SNZ) and either the loop has a local SNZ witness or all external updates to an SNZ witness of lp originate from safe loops.*[1]

[1] Requiring updating external loops to be safe instead of strongly non-Zeno with a local witness [14] makes the safety property transitive.

Proposition 1. *Any run that covers a safe loop lp is time-divergent.*

Proof. A run that covers a strongly non-Zeno loop is time-divergent by definition unless the SNZ witness clock is externally updated infinitely often without delays. Such updates must originate from loops that exhibit Zeno runs themselves. But such updates are impossible as a safe loop only receives updates from safe loops by definition. Thus, safe loops are free from Zeno runs. □

4 Synchronization Matrices

We now introduce our synchronization-matrix approach. First we discuss the representation of loops, then the construction of the synchronization matrix, and finally the equation system that determines the set of safe loops.

4.1 Loop Modeling

At first, we assign appropriate modeling vectors to all observable, unsafe loops. Let U refer to the set of observable, unsafe loops and let $Sync(S)$ be the set of synchronization channels used in a set of loops S. We denote the set of all synchronization channels used in unsafe, observable loops $\mathcal{L} = Sync(U)$. In addition, we define $Base(lp)$ to be the ordered *base vector* set and $Broad(lp)$ the ordered *broadcast vector* set for a loop lp, and BCh to be the set of broadcast synchronization channels. A vector $\boldsymbol{v}$ in either set has one component per used synchronization channel and those components take values in the integral numbers ($\boldsymbol{v} \in \mathbb{Z}^{|\mathcal{L}|}$). We refer to individual vector components by $x_{\texttt{channel}}$. In our notation, `c` denotes a channel, `a[]` an array, e an expression, `c!` a sending channel, and `c?` a receiving one.

Vectors are assigned to loops in the following way. Initially, for every $lp \in U$ the set $Base(lp)$ consists of a single zero vector. The set $Broad(lp)$ is empty. We then iterate over all loops $lp \in U$ and for every synchronization label lb used in the loop the sets are updated using the matching following rule:

1. *binary channel (send/receive), broadcast channel (receive)*
   $(\texttt{c} \notin BCh) \vee (lb = \texttt{c?})$
   For every vector $\boldsymbol{v} \in Base(lp)$, increase $x_{\texttt{c}}$ by 1, if lb is of the form `c!`, or decrease $x_{\texttt{c}}$ by 1, if lb is of the form `c?`.
2. *broadcast channel (send)*
   $\texttt{c} \in BCh \wedge lb = \texttt{c!}$
   If not already in the set, add a vector $\boldsymbol{v}$ to $Broad(lp)$, where $x_{\texttt{c}}$ is set to 1.
3. *unresolved binary channel array (send/receive), unresolved broadcast channel array (receive)*
   $((\nexists \texttt{c} \in \texttt{a}[\texttt{c} \in BCh]) \vee (lb = \texttt{a[}e\texttt{]?})) \wedge e$ unresolved
   For every channel $\texttt{c} \in \texttt{a}$, create a copy $C_{\texttt{c}}$ of $Base(lp)$. Then, for every vector $\boldsymbol{v} \in C_{\texttt{c}}$, increase $x_{\texttt{c}}$ by 1, if lb is of the form `a[`e`]!`, or decrease $x_{\texttt{c}}$ by 1, if lb is of the form `a[`e`]?`. The resulting base vector set is the union of all copies: $Base(lp) = \bigcup_{\texttt{c}} C_{\texttt{c}}$.

4. *unresolved broadcast channel array (send)*
   $\forall \mathtt{c} \in \mathtt{a}[\mathtt{c} \in \mathit{BCh}] \wedge lb = \mathtt{a[}e\mathtt{]!} \wedge e$ unresolved
   For every channel $\mathtt{c} \in \mathtt{a}$ if not already in the set, add a vector $\boldsymbol{v}$ to $\mathit{Broad}(lp)$, where $x_{\mathtt{c}}$ is set to 1.

Fig. 2 shows one loop per rule. Assume the following channel variables: a is a binary channel, b is a broadcast channel, c is a binary channel array of size two, and d is a broadcast channel array of size two. All loops are unsafe and observable. We use the following form of vectors to represent the loops: $\boldsymbol{v} = (x_{\mathtt{a}}, x_{\mathtt{b}}, x_{\mathtt{c[0]}}, x_{\mathtt{c[1]}}, x_{\mathtt{d[0]}}, x_{\mathtt{d[1]}})^T$. The resulting loop models are:

- Loop A: $\mathit{Base}(lp) = \{\ (1,0,0,0,0,0)^T\ \},\ \mathit{Broad}(lp) = \emptyset$
- Loop B: $\mathit{Base}(lp) = \{\ (0,0,0,0,0,0)^T\ \},\ \mathit{Broad}(lp) = \{\ (0,1,0,0,0,0)\ \}$
- Loop C: $\mathit{Base}(lp) = \{\ (0,0,1,0,0,0)^T, (0,0,0,1,0,0)^T\ \},\ \mathit{Broad}(lp) = \emptyset$
- Loop D: $\mathit{Base}(lp) = \{\ (0,0,0,0,0,0)^T\ \}$,
  $\mathit{Broad}(lp) = \{\ (0,0,0,0,1,0)^T, (0,0,0,0,0,1)^T\ \}$

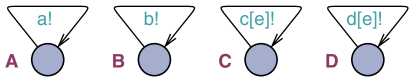


Fig. 2. Synchronization cases during loop model construction

4.2 Matrix Construction

Based on the loop models we now create the synchronization matrices, which will be used to calculate valid synchronization scenarios. The matrix has the following form:

$$\mathbf{S} = \begin{pmatrix} \mathbf{M} \\ \mathbf{C} \end{pmatrix} = \begin{pmatrix} \mathbf{M_1} & \mathbf{M_2} & \dots & \mathbf{M_n} \\ \mathbf{C_1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{C_2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{C_n} \end{pmatrix} \tag{4}$$

$\mathbf{M_i}$ are model matrices for all unsafe, observable loops and $\mathbf{C_i}$ are associated constraint matrices. The model matrices $\mathbf{M_i}$ can be constructed from the loop models:

$$\mathbf{M_i} = \begin{pmatrix} \boldsymbol{v_1} \dots \boldsymbol{v_n} & \boldsymbol{b_1} \dots \boldsymbol{b_m} & \boldsymbol{b_1} \dots \boldsymbol{b_m} \end{pmatrix}, \boldsymbol{v_j} \in \mathit{Base}(lp_{\mathrm{i}}), \boldsymbol{b_k} \in \mathit{Broad}(lp_{\mathrm{i}}) \tag{5}$$

The constraint matrices $\mathbf{C_i}$ relate the broadcast vectors to the base vectors. They are divided into three parts in the same way as the model matrices $\mathbf{M_i}$. They are of the form

$$\mathbf{C_i} = \begin{pmatrix} \mathbf{0} & s \cdot \mathbf{I} & -1 \cdot \mathbf{I} \\ \boldsymbol{l_i} \dots \boldsymbol{l_i} & \mathbf{L}_{\mathrm{i}} & \mathbf{0} \end{pmatrix} \tag{6}$$

Here, $\mathbf{0}$ is the zero matrix, s is the total number of synchronization labels, $\mathbf{I}$ is the identity matrix, $\boldsymbol{l}_\mathbf{i}$ is the *base-link vector*, and $\mathbf{L}_\mathrm{i}$ is the *base-link matrix* of the loop. The upper blocks in $\mathbf{C}_\mathbf{i}$ model *broadcast-link constraints* and the lower blocks model *base-link constraints.* Together, the constraints model the 1-to-n synchronization behavior of broadcast channels. For the construction of the base link vectors and matrices, we iterate through all synchronization labels lb in the loop lp_i and add values to $\boldsymbol{l}_\mathbf{i}$ and rows to $\mathbf{L}_\mathrm{i}$ if one of the following rules applies:

1. *broadcast channel (send)* ($c \in BCh \wedge lb = \mathtt{c!}$)
   Add the number of sending synchronization labels of the form `c!` in the loop lp_i to $\boldsymbol{l}_\mathbf{i}$. Add a zero row to $\mathbf{L}_\mathrm{i}$ and set the value x_p to -1. The index p refers to the position of the vector $\boldsymbol{v} \in \mathit{Broad}(lp_\mathrm{i})$, where $x_\mathtt{c} = 1$.
2. *unresolved broadcast channel array (send)* ($\forall \mathtt{c} \in \mathtt{a}[\mathtt{c} \in BCh] \wedge lb = \mathtt{a[}e\mathtt{]!}$)
   Add the number of sending synchronization labels of the form `a[`e`]!` in the loop lp_i to $\boldsymbol{l}_\mathbf{i}$. Add a zero row to $\mathbf{L}_\mathrm{i}$ and set the values x_j, $j \in P$, to -1. The set P of indices refers to the positions of the vectors $\boldsymbol{v_j} \in \mathit{Broad}(lp_\mathrm{i})$, where $x_\mathtt{c} = 1$ and `c` is a channel in the array `a`.

Equation (7) shows the synchronization matrix $\mathbf{S}$ for the example model shown in Fig. 2. Vertical bars separate different loops and dashed vertical bars separate the base vectors from the broadcast vectors. The horizontal double bar separates the model matrices from the constraint matrices and the simple horizontal bars separate the loop constraint matrices.

$$\mathbf{S} = \left(\begin{array}{c|c:cc|cc|c:cccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline\hline
0 & 0 & 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0
\end{array}\right) \tag{7}$$

Loop `A` only sends on the binary channel `a` and is therefore modeled with a single base vector in the first column. Loop `B` only sends on the broadcast channel `b` and thus is modeled in columns 2-4 by the empty base vector and the duplicated broadcast vector. The vectors are linked using a broadcast-link constraint and a base-link constraint. Columns 5 and 6 model loop `C` that sends on an unresolved binary channel array `c` with 2 elements. Two base vectors are created as synchronization is possible on `c[0]` or `c[1]`. Lastly, loop `D` sends on an unresolved broadcast channel array `d`. Accordingly, the zero base vector is kept and for every channel in the array one broadcast vector was created and then duplicated. The broadcast link constraints bind the duplicates together. In the last row the base-link constraint ensures only either `d[0]` or `d[1]` can synchronize per loop iteration.

Correctness argument The synchronization matrix is a nearly exact representation of the synchronization capabilities in UPPAAL.[2] All required synchronization partners for a loop are captured by its base vector set. Optional synchronizations are possible due to the broadcast vectors and are restricted by the binding constraints such that broadcast synchronizations may synchronize with all potential receivers per loop iteration.

4.3 Synchronization Scenarios

Using the synchronization matrix **S** we can construct valid synchronization scenarios involving all loops. Loops for which no valid synchronization scenario exists can not exhibit Zeno runs and may be removed from the analysis result set. The (in)equation system that needs to be solved is given in (8):

$$\mathbf{M}\boldsymbol{x}^*_{\mathbf{M}} = \mathbf{0}, \quad \mathbf{C}\boldsymbol{x}^*_{\mathbf{C}} \geq \mathbf{0}, \quad \mathbf{S} = \begin{pmatrix} \mathbf{M} \\ \mathbf{C} \end{pmatrix}, \quad \boldsymbol{x}^* = \begin{pmatrix} \boldsymbol{x}^*_{\mathbf{M}} \\ \boldsymbol{x}^*_{\mathbf{C}} \end{pmatrix} \tag{8}$$

The equation system is further constrained insofar the components of the solution vector $\boldsymbol{x}^*$ may only take values in the natural numbers including zero ($\boldsymbol{x}^* \in (\mathbb{N}^0)^{\mathrm{n}}$). Upon successful solution calculation, the solution vector contains information about how often every loop needs to iterate to provide a matching synchronization partner for every synchronization label.

To decide whether or not a valid synchronization scenario exists for a certain loop *lp*, we solve the (in)equation system once for every base vector of *lp*. Each time, we require the component value of the solution vector $\boldsymbol{x}^*$ of a different base vector of *lp* to be greater than zero to ensure the solution will involve the loop. If a solution exists, the loop may contribute to Zeno runs, otherwise it is eliminated from the result set of the analysis.

Correctness Argument. For a model to exhibit Zeno runs a synchronizing set of loops that can iterate indefinitely without delays must exist. Such an infinite iteration is periodic as the underlying automata system is finite, and we can characterize the iteration using the number of iterations of each participating loop per period. For a loop to contribute to a Zeno run thus a sound characterization involving the loop must exist. We construct sound characterizations for every unsafe loop and thus can exclude some loops upon construction failure.

5 Data-Variable Heuristics

Data variables play an important role during execution of a timed transition system in UPPAAL but have not yet been analyzed for their influence on Zeno

[2] We unify sending and receiving synchronizations by counting them positive and negative respectively. This simplification introduces false positives if a loop sends and receives on the same channel. An extension of the loop vectors to two numbers per channel could solve this problem.

runs. They may, for example, disable edges or dynamically provide upper and lower bounds for time constraints and thus modify the behavior of a model significantly. During a static analysis exact values for data variables generally can not be inferred directly. However, techniques using data-flow analysis can restrict the set of possible valuations for a data variable at certain states. Such restrictions may render possible the elimination of loops that can not iterate due to those data variable constraints and thus improve the accuracy of the Zeno run analysis even further.

In this paper we do not apply a fully-fledged data-flow analysis to UPPAAL's timed automata system. Instead, we opted for an ad-hoc approach that evaluates loop constellations and variable valuations on a case-by-case basis to eliminate impossible Zeno runs. The next two subsections present two such heuristics.

5.1 Safely Dependent Loops

Consider the timed automaton given in Fig. 3. The automaton consists of a single component that models the Fischer synchronization protocol [19]. The component uses two variables and two constants: `x` is a local clock variable, `id` is a global integer variable, `pid` is a unique, constant integer value that is greater than zero to identify the component, and `k` is a global, constant integer value greater than zero. The automaton has two loops:

1. Loop: `A` → `req` → `wait` → `cs` → `A`
2. Loop: `req` → `wait` → `req`

The first loop is a safe loop. The second loop is neither safe nor strongly non-Zeno. However, the second loop has a safe data-variable dependency: the data variable `id` is required to be zero (`id == 0`) for the loop to iterate. In addition, the loop itself sets the value of `id` to a value that does not satisfy the guard (`id = pid`) and thus the loop can not iterate on its own indefinitely. Instead, the loop depends on external influence on the data variable `id`. In this example, a valuation of `id` that satisfies the guard is only reachable if the first, safe loop is executed. Therefore, the second loop can also be considered safe because iterations of the second loop always require a safe loop iteration of the first loop ensuring time divergence. We improve the accuracy of our analysis by removing loops that have such a dependency. The dependency can be defined as follows.

Definition 5 (Safely dependent loop). *A loop lp is* safely dependent *if there exists an update u and a guard g in lp such that g is never satisfied after u is executed, regardless of the previous variable valuations, and every reachable variable valuation that satisfies g originates from an update on an edge that is part of safe loops only.*

Proposition 2. *Any run that covers a safely dependent loop lp is time-divergent.*

Proof. A run ρ that covers a safely dependent loop lp not only covers lp but also at least one loop lp' that provides a variable valuation that satisfies the guard g

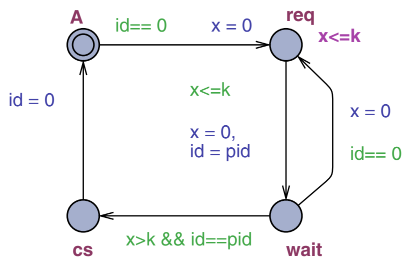


Fig. 3. Fischer protocol automaton

in lp. As all loops that provide such a valuation are safe by definition, ρ covers a safe loop lp' and thus ρ is time-divergent. □

5.2 Conflicting Loops

When specifying a model in UPPAAL the user may create loops in the system that cannot execute during model execution due to data constraints. However, those loops are still considered for Zeno run detection during static analysis. Fig. 4 shows a model with such an impossible loop. The model uses two local variables; `x` is an integer variable and `t` is a clock variable. The interesting loop is `Zero` $\rightarrow$ `One` $\rightarrow$ `Zero` because this loop can not iterate at run time as the two guards need conflicting valuations of `x` (`x == 0`, `x == 1`), but `x` is a local variable and can thus not be modified from an external source. The system is therefore free from Zeno runs. We improve the accuracy of our analysis by removing loops that have such a conflict. The conflict can be defined as follows.

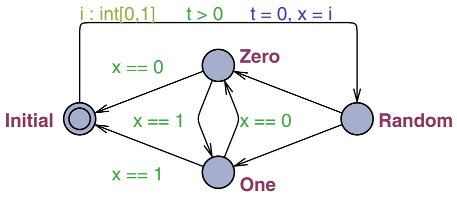


Fig. 4. Model with an impossible loop

Definition 6 (Conflicting loops). *A loop lp is* conflicting *if there exist two edges e_1 and e_2 with respective guards g_1 and g_2 involving local variables such that no variable valuation exists that satisfies g_1 and after sequential execution of the updates on a path from e_1 to e_2 also satisfies g_2.*

Proposition 3. *There is no run that covers a conflicting loop lp.*

Proof. For a conflicting loop lp there is no variable valuation that satisfies a guard in lp and, after updates have been applied, also satisfies a different, second guard in lp. Accordingly, a run can only cover one of both edges of lp but never both. Therefore, a covering run for lp does not exist. □

6 ZenoTool: Implementation and Evaluation

We implemented the analysis in a stand-alone software called ZenoTool [23] using C++. ZenoTool is a command-line program reading UPPAAL 4.0 model specification files. To determine the accuracy and efficiency of ZenoTool's analysis we analyzed 13 models that cover a broad spectrum of use cases and thus allows determining general qualities of ZenoTool.

We measured ZenoTool's run time by executing the analysis ten consecutive times and saving the fastest, the slowest, and the average run times. Valgrind's *massif* tool was used to find the peak heap memory consumption of ZenoTool.[3] All experiments were done using an Intel Core 2 Duo CPU running at 3.33GHz with 8GB of RAM on an Ubuntu 11.10 system.

In more detail, our test suite contains three example models distributed with UPPAAL, namely the *train-gate8* model, the *fischer6* model, and the *bridge* problem. Another four, the *csmacd2* and *csmacd32* models, the *fddi32* model, and the *bocdp* model, are used to benchmark UPPAAL's performance but were developed in scientific case studies [10, 15, 30]. The remaining models were derived from scientific case studies [8, 9, 12, 20, 21, 29]. Eight of the models are free from Zeno runs and the remaining five exhibit Zeno runs in one or multiple ways.

Table 1. ZenoTool analysis accuracy. The table shows the number of loops prone to Zeno runs that ZenoTool found. The Zeno run column indicates whether or not a model is free from Zeno runs (✓: free, ✗: not free). The ↯ symbol indicates a certain analysis configuration was not applicable for a model. **D** indicates usage of the safely dependent loop heuristic; **C** indicates usage of the conflicting loop heuristic. The Sync Matrix columns show the results of our approach while the Sync Groups columns show results obtained using the synchronization-group method.

Model	**Zeno Runs**	**Sync Groups**				**Sync Matrix**			
bridge.xml	✓	0	0	0	0	0	0	0	0
lipsync.xml	✓	0	0	0	0	0	0	0	0
train-gate8.xml	✓	0	↯	↯	↯	0	↯	↯	↯
fddi32.xml	✓	0	0	0	0	0	0	0	0
bocdp.xml	✓	0	0	0	0	0	0	0	0
tdma.xml	✓	2	2	2	2	2	2	2	2
gearbox.xml	✓	7	7	7	7	7	7	7	7
csmacd.xml	✗	4	4	4	4	4	4	4	4
csmacd2.xml	✗	7	7	7	7	3	3	3	3
csmacd32.xml	✗	97	97	97	97	97	97	97	97
fischer6.xml	✓	6	0	6	0	6	0	6	0
bmp.xml	✗	2	1	2	1	2	1	2	1
zeroconf.xml	✗	14	↯	↯	↯	14	↯	↯	↯
Heuristics Used			**D**	**C**	**D C**		**D**	**C**	**D C**

3 Note that additionally to the heap memory some static memory will be used, which is not accounted for here.

Table 2. ZenoTool analysis performance. The sync-group method uses no heuristics; the sync-matrix method uses both heuristics. Gómez and Bowman's results of liveness checks in UPPAAL are given for reference. The ⊥ symbol indicates calculations did not finish within reasonable time. Our values are all rounded up. The given memory consumption is the peak heap memory consumption. ZenoTool additionally needs a certain amount of static memory that is not accounted for here. The ↯ symbol again indicates that a certain analysis configuration was not applicable for a model.

	Sync groups				**Sync matrix**				**UPPAAL**	
Model	**min**	**max**	**avg**	**mem**	**min**	**max**	**avg**	**mem**	**time**	**vmem**
bridge.xml	6	10	9	587	10	11	10	671	-	-
lipsync.xml	11	11	11	792	11	13	12	881	1	53
train-gate8.xml	9	15	12	620	↯	↯	↯	↯	1496	762
fddi32.xml	77	86	80	1911	79	85	81	2011	⊥	⊥
bocdp.xml	172	183	175	1140	176	189	179	1221	⊥	⊥
tdma.xml	87	88	88	866	87	98	89	951	-	-
gearbox.xml	19	21	20	872	25	28	26	955	11644	68
csmacd.xml	10	12	11	629	8	13	10	723	5204	60
csmacd2.xml	11	11	11	633	9	15	10	721	-	-
csmacd32.xml	48	51	49	1881	95	98	96	1961	1	5
fischer6.xml	11	12	11	562	11	17	15	641	⊥	⊥
bmp.xml	11	15	12	684	13	14	13	764	1	1
zeroconf.xml	14	22	18	704	↯	↯	↯	↯	1676	315
Units	[msecs]			[kB]	[msecs]			[kB]	[secs]	[MB]

Concerning the five models that can exhibit Zeno runs all loops responsible for the occurrence of Zeno runs were found. Note that models with Zeno run approximations are not necessarily harmful but such approximations need to be carefully examined. Of the eight Zeno run free models ZenoTool proved six directly safe. For the remaining two the analysis returned a handful of false positives making inferring the absence of Zeno runs by hand easy. Thus all eight models were classified correctly with little or no manual work at all.

Comparing our approach to our re-implementation of the synchronization-group approach [14], our synchronization-matrix method yielded an improvement in accuracy only for a single model. The conflicting loops data heuristics did not find a single conflicting loop, which was not surprising since all the models are well thought-out. However, the safely dependent heuristics eliminated seven loops in two different models. In one case the heuristics managed to prove the absence of Zeno runs, which the synchronization-group method was not able to do directly. Table 1 summarizes the accuracy results.

For our performance checks we compared the synchronization-group approach without heuristics to our synchronization-matrix method using both data heuristics. This setup emphasizes that ZenoTool's performance scales well even if the work load is additionally increased because of the data heuristics. Our performance results can be seen in Table 2.

Generally, all analyses run in less than a second even for complex models. Thus, time is not a major factor when using ZenoTool. Memory consumption is also quite low. When taking a closer look at the performance results comparing the synchronization-group method to ours the run-time differences are nearly negligible. Only the *csmacd32* model shows a significant difference in run time. There, our synchronization-matrix approach including the heuristics takes nearly twice the time than the original method. This is an expected result because the (in)equation solving process in this case is not trivial.

Generally, we had expected more performance hits, but ZenoTool uses the GNU Linear Programming Kit (GLPK) to solve the constrained (in)equation system and at least our current benchmark set suggests that most of the models generate equation systems that are easy to solve for the GLPK; thus, run time does not increase significantly. Models that benefit from the synchronization matrix accuracy improvement probably will perform worse than our results show.

The performance of the data-variable heuristics depends on the accuracy of the previous analysis result set. As most experiment models returned small sets of unsafe loops the heuristics had only a small negative effect on ZenoTool's run time. Yet, models that have many unsafe loops that appear to be unsafe if one does not consider data variables might also see a drop in performance. Still ZenoTool's performance is very satisfactory.

7 Related Work

Zeno behavior is studied in many contexts in the literature. Abadi and Lamport examine it in a real-time context using variables for time [1]. Concerning timed automata, not only the detection but also the prevention of timelocks is explored. Proposals include Timed I/O Automata [11, 18], Discrete Timed Automata [13], and Timed Automata with Deadlines (TAD) [6, 7]. Büchi emptiness checks, i.e., the discovery of non-Zeno runs that satisfy an acceptance condition, are also a topic of research [17, 28]. Obviously, timed systems cannot only be modeled by timed automata [3] but in various other ways, including timed Petri nets [22] and hybrid systems [2, 16]. Progress verification methods using these modeling constructs appear in the literature as well [4, 5].

8 Conclusion and Future Work

In this paper we enhanced the Zeno run detection method introduced by Gómez and Bowman by introducing a newly developed synchronization matrix. In contrast to the synchronization-group method, which only checks for available synchronization partners, our approach considers the number of necessary synchronization partners and thus improves safety propagation. In addition, we developed two data-variable heuristics. The safely dependent loop heuristics extends the concept of safety propagation to data variables assuring that variable valuations necessary for Zeno run iterations can only be attained by executing safe loops. The conflicting loop heuristics removes loops that are unable to iterate because of conflicting data-variable requirements. ZenoTool, a command-line tool implementing these analysis methods, was developed and applied to several case studies to empirically evaluate the improvements [23]. The experiment results validate the expected gain in analysis accuracy while only taking a small hit in performance. Models that benefit most from our analysis have loops that synchronize on multiple channels possibly multiple times.

In the future ZenoTool could be improved in several ways. The data-variable heuristics would benefit from a fully-fledged data-flow analysis module to broaden the applicable cases. Additionally, an interface linking UPPAAL and ZenoTool could be developed to eliminate Zeno loops that are unreachable at run time. From a practical point it is also important to make ZenoTool feature-complete as the implementation currently lacks complete support for the following 5 constructs of UPPAAL 4.0: user-defined functions, data records, scalar variables, complex expressions, and selection statements.

References

1. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
4. Bornot, S., Sifakis, J.: Relating time progress and deadlines in hybrid systems. In: Proc. of the Int. Work. on Hybrid and Real-Time Systems, pp. 286–300 (1997)
5. Bornot, S., Sifakis, J.: On the Composition of Hybrid Systems. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 49–63. Springer, Heidelberg (1998)
6. Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In: Int. Symp.: Compositionality - The Significant Difference, pp. 103–129 (1997)
7. Bowman, H.: Time and action lock freedom properties of timed automata. In: Formal Techniques for Networked and Distributed Systems, pp. 119–134 (2001)
8. Bowman, H., Faconti, G., Katoen, J.-P., Latella, D., Massink, M.: Automatic verification of a lip-synchronisation protocol using UPPAAL. Formal Aspects of Computing 10, 550–575 (1998)
9. Bowman, H., Gómez, R.: How to stop time stopping. Formal Aspects of Computing 18, 459–493 (2006)
10. Daws, C., Tripakis, S.: Model Checking of Real-Time Reachability Properties Using Abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
11. Gebremichael, B., Vaandrager, F.: Specifying urgency in timed I/O automata. In: Proc. of the 3rd IEEE Int. Conf. on Software Engineering and Formal Methods, pp. 64–73 (2005)
12. Gebremichael, B., Vaandrager, F., Zhang, M.: Analysis of the zeroconf protocol using UPPAAL. In: Proc. of the 6th ACM & IEEE Int. Conf. on Embedded Software, pp. 242–251 (2006)
13. Gómez, R., Bowman, H.: Discrete Timed Automata and MONA: Description, Specification and Verification of a Multimedia Stream. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 177–192. Springer, Heidelberg (2003)

14. Gómez, R., Bowman, H.: Efficient Detection of Zeno Runs in Timed Automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 195–210. Springer, Heidelberg (2007)
15. Havelund, K., Skou, A., Larsen, K.G., Lund, K.: Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In: Proc. of the 18th IEEE Real-Time Systems Symp., pp. 2–13 (1997)
16. Henzinger, T.A.: The theory of hybrid automata. In: LICS: Logic in Computer Science. pp. 278–292. IEEE Computer Society Press (1996)
17. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient Emptiness Check for Timed Büchi Automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)
18. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The theory of timed I/O automata, 2nd ed. Synthesis Lectures on Dist. Comp. Theory 1(1), 1–137 (2010)
19. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
20. Lindahl, M., Pettersson, P., Yi, W.: Formal Design and Analysis of a Gear Controller. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 281–297. Springer, Heidelberg (1998)
21. Lönn, H., Pettersson, P.: Formal verification of a TDMA protocol start-up mechanism. In: Proc. of the 1997 Pacific Rim Int. Symp. on Fault-Tolerant Systems, pp. 235–242 (1997)
22. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Ph.D. thesis, Cambridge, MA, USA (1974)
23. Rinast, J.: ZenoTool, A Zeno Run detection tool for UPPAAL, `http://www.sts.tu-harburg.de/research/zenotool.html`
24. Szwarcfiter, J.L., Lauer, P.E.: A search strategy for the elementary cycles of a directed graph. BIT Numerical Mathematics 16, 192–204 (1976)
25. Tarjan, R.E.: Enumeration of the elementary circuits of a directed graph. Tech. rep., Department of Computer Science, Cornell University, Ithaca, NY, USA (1972)
26. Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM 13, 722–726 (1970)
27. Tripakis, S.: Verifying Progress in Timed Systems. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
28. Tripakis, S., Yovine, S., Bouajjani, A.: Checking Timed Büchi Automata Emptiness Efficiently. Formal Methods in System Design 26, 267–292 (2005)
29. Vaandrager, F., de Groot, A.: Analysis of a biphase mark protocol with UPPAAL and PVS. Formal Aspects of Computing 18, 433–458 (2006)
30. Yovine, S.: Kronos: a verification tool for real-time systems. International Journal on Software Tools for Technology Transfer (STTT) 1, 123–133 (1997)

Frequencies in Forgetful Timed Automata

Amélie Stainer

University of Rennes 1, Rennes, France

Abstract. A quantitative semantics for infinite timed words in timed automata based on the frequency of a run is introduced in [6]. Unfortunately, most of the results are obtained only for one-clock timed automata because the techniques do not allow to deal with some phenomenon of convergence between clocks. On the other hand, the notion of forgetful cycle is introduced in [4], in the context of entropy of timed languages, and seems to detect exactly these convergences. In this paper, we investigate how the notion of forgetfulness can help to extend the computation of the set of frequencies to n-clock timed automata.

1 Introduction

Timed automata have been introduced in [1]. This model is commonly used to represent real-time systems. A timed automaton is roughly a finite automaton equipped with a finite set of continuous clocks which evolve synchronously, are used in guards and can be reset along the transitions. The usual semantics of timed automata for infinite timed words is the Büchi semantics also presented in [1]. Recently, several works propose to add quantitative aspects in verification problems, such as costs [2,5] or probabilities [9,3].

In particular, one can refine the acceptance condition by considering the proportion of time elapsed in accepting locations. A quantitative semantics for infinite timed words based on this notion of frequency has thus been introduced in [6]. Lower and upper bounds of the set of frequencies of one-clock timed automata are computed using the corner-point abstraction, a refinement of the classical region abstraction, introduced in [7]. These bounds can be used to decide the emptiness of the languages with positive frequencies and the universality for deterministic timed automata. Furthermore, the universality problem is proved to be non primitive recursive for non-deterministic timed automata with one clock and undecidable with several clocks. The techniques from [6] do not extend to timed automata with several clocks, and all counterexamples rely on some phenomenon of convergence between clocks along cycles. Beyond zenoness (when time converges along a run), other convergence phenomena between clocks were first discussed in [8]. Similarly to zenoness, these convergences correspond to behaviors that are unrealistic from an implementability point of view. A way to detect cycles with no such convergences (called forgetful cycles) has been recently introduced in [4]. This notion of forgetfulness was used to characterize timed languages with a non-degenerate entropy.

In this paper, we naturally propose to investigate how forgetfulness can be exploited to compute frequencies. First, we show that forgetfulness of a cycle in a one-clock timed automaton is equivalent to not forcing the convergence of the clock, that is the clock is reset or not bounded. Note that forgetfulness does not imply that all runs are non-Zeno. With this assumption, the set of frequencies can be exactly computed using the corner-point abstraction. Then, we show that in n-clock forgetful timed automata where time diverges necessarily along a run, the set of frequencies can also be computed thanks to the corner-point abstraction. On the one hand, the result for timed automata for which all cycles are forgetful (strong forgetfulness) is as constructive as the theorem of [6] over one-clock timed automata. On the other hand, to relax strong forgetfulness and consider timed automata whose *simple* cycles are forgetful, the proof relies on a set of canonical runs whose frequencies cover the set of all frequencies in the timed automaton.

Our contribution can also be compared with that of [7] on double priced timed automata, that is, timed automata with costs and rewards. Indeed, frequencies are a particular case of cost and reward functions. In [7], either a run of minimal ratio or an optimal family (*i.e.* ε-optimal runs for all $\varepsilon > 0$) is computed, whereas, assuming forgetfulness, the exact set of frequencies can be computed, not only the optimal ones. Our techniques might thus prove useful for double priced timed automata and maybe more generally in other contexts.

The paper is structured as follows. In the next section we introduce the model of timed automata, the quantitative semantics based on frequencies, forgetfulness and the corner-point abstraction as a tool to study frequencies. In Section 3, we propose a characterization of forgetfulness in one-clock timed automata and provide an expression for the set of frequencies for this restricted class. Last, Section 4 deals with n-clock timed automata and explains how to use forgetfulness to ensure that, when time diverges, the set of frequencies of a timed automaton and the set of ratios in its corner-point abstraction are equal.

All the details omitted, due to space constraints, in this paper, can be found in the research report [13].

2 Preliminaries

In this section, we recall the definition of timed automata with the quantitative semantics based on frequencies introduced in [6]. Then the corner-point abstraction is presented, firstly to define forgetfulness, and secondly as a tool to compute frequencies in timed automata.

2.1 Timed Automata and Frequencies

Given a finite set X of clocks, a *valuation* is a mapping $v : X \to \mathbb{R}_+$. The valuation associating 0 with all clocks is written $\overline{0}$ and $v + t$ is the valuation defined, for every clock x of X by $(v + t)(x) = v(x) + t$. For $X' \subseteq X$, $v_{[X' \leftarrow 0]}$ denotes the valuation equal to $\overline{0}$ for the clocks of X' and equal to v for the other clocks. On the other hand, a *guard* over X is a finite conjunction of constraints of the form $x \sim c$ where $x \in X$, $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. The set of guards over X is noted $G(X)$. Moreover, for a valuation v and a guard g, v *satisfies* g with the usual definition, is written $v \models g$.

Definition 1 (timed automaton). *A* timed automaton *is a tuple* $\mathcal{A} = (L, L_0, F, \Sigma, X, E)$ *such that:* L *is a finite set of locations,* $L_0 \subseteq L$ *is the set of initial locations,* $F \subseteq L$ *is the set of accepting locations,* Σ *is a finite alphabet,* X *is a finite set of clocks and* $E \subseteq L \times G(X) \times \Sigma \times 2^X \times L$ *is a finite set of edges.*

The *semantics* of a timed automaton $\mathcal{A}$ is given as a timed transition system $\mathcal{T}_\mathcal{A} = (S, S_0, S_F, (\mathbb{R}_+ \times \Sigma), \rightarrow)$ where $S = L \times \mathbb{R}_+^X$ is the set of states, $S_0 = L_0 \times \{\overline{0}\}$ is the set of initial states, $S_F = F \times \mathbb{R}_+^X$ is the set of accepting states and $\rightarrow \subseteq S \times (\mathbb{R}_+ \times \Sigma) \times S$ is the transition relation composed of all moves of the form $(\ell, v) \xrightarrow{\tau,a} (\ell', v')$ such that $\tau > 0$ and there exists an edge $(\ell, g, a, X', \ell') \in E$ with $v + \tau \models g$ and $v' = (v + \tau)_{[X' \leftarrow 0]}$.

A *run* of a timed automaton $\mathcal{A}$ is a finite or infinite sequence of moves starting in an initial state. In the sequel, unless otherwise stated, the run is assumed to be infinite. Thus, an infinite run $\rho = s_0 \xrightarrow{\tau_0,a_0} s_1 \xrightarrow{\tau_1,a_1} s_2 \xrightarrow{\tau_2,a_2} \cdots$ is said to be *Zeno* if $(\sum_{0 \leq j \leq i} \tau_j)_{i \in \mathbb{N}}$ is bounded.

Definition 2 (frequency). *Given* $\mathcal{A} = (L, L_0, F, \Sigma, X, E)$ *a timed automaton and* $\rho = (\ell_0, v_0) \xrightarrow{\tau_0,a_0} (\ell_1, v_1) \xrightarrow{\tau_1,a_1} (\ell_2, v_2) \cdots$ *an infinite run of* $\mathcal{A}$*, the* frequency *of* F *along* ρ*, denoted* $\mathsf{freq}_\mathcal{A}(\rho)$*, is defined as* $\limsup_{n \to \infty} \frac{\sum_{\{i \leq n \mid \ell_i \in F\}} \tau_i}{\sum_{i \leq n} \tau_i}$.

Note that, as in [6], the limit sup is an arbitrary choice. In the sequel, our goal is to compute the set of frequencies of the infinite runs of $\mathcal{A}$, which is written $\mathsf{Freq}(\mathcal{A})$. To do so, we sometimes distinguish $\mathsf{Freq}_Z(\mathcal{A})$ and $\mathsf{Freq}_{nZ}(\mathcal{A})$ which respectively denote the sets of frequencies of the Zeno and non-Zeno runs of $\mathcal{A}$. For example, Fig. 1 represents a timed automaton $\mathcal{A}$ with $F = \{\ell_1\}$ (accepting locations are colored in gray), such that $\mathsf{Freq}(\mathcal{A}) = \mathsf{Freq}_{nZ}(\mathcal{A}) = [0, 1[$. Indeed, there is no Zeno runs in $\mathcal{A}$ and there is an underlying constraint along the cycle which ensures that delays elapsed in the accepting location are decreasing. This implies that frequencies of an infinite run in $\mathcal{A}$ is of the form $\frac{1-\varepsilon}{\varepsilon}$ with $\varepsilon \in]0, 1]$.

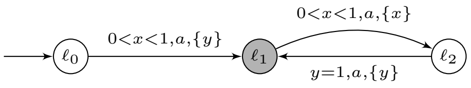


Fig. 1. A timed automaton $\mathcal{A}$ to illustrate the notion of frequency

2.2 Corner-Point Abstraction and Forgetfulness

Given the maximal constant M appearing in a timed automaton $\mathcal{A}$, the usual region abstraction forms a partition of the valuations over X, the clocks of $\mathcal{A}$.

In the following definition, $\lfloor t \rfloor$ and $\{t\}$ are respectively the integer part and the fractional part of the real t. The *region equivalence* $\equiv_{\mathcal{A}}$ over valuations of X is defined as follows: $v \equiv_{\mathcal{A}} v'$ if (i) for every clock $x \in X$, $v(x) \leq M$ iff $v'(x) \leq M$; (ii) for every clock $x \in X$, if $v(x) \leq M$, then $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ and $\{v(x)\} = 0$ iff $\{v'(x)\} = 0$ and (iii) for every pair of clocks $(x, y) \in X^2$ such that $v(x) \leq M$ and $v(y) \leq M$, $\{v(x)\} \leq \{v(y)\}$ iff $\{v'(x)\} \leq \{v'(y)\}$. The equivalence classes of this relation are called *regions* and $Reg_{\mathcal{A}}$ denotes the set of regions for the timed automaton $\mathcal{A}$. For each valuation v of the clocks of $\mathcal{A}$, there is a single region containing v, denoted by $R(v)$. A region R' is a *time-successor* of a region R if there exists $v \in R$ and $t \in \mathbb{R}_+$ such that $v + t \in R'$ and $R' \neq R$. The set of the time-successors of a region is naturally ordered, and the mapping $\mathit{timeSucc} : Reg_{\mathcal{A}} \to Reg_{\mathcal{A}}$ associates with any region, its first time-successor. The particular case of the region $\{\perp^X\}$ where all the clocks are larger than M is fixed as follows : $\mathit{timeSucc}(\{\perp^X\}) = \{\perp^X\}$.

Given a timed automaton, one can build a timed automaton having only region guards while preserving the set of frequencies. In fact, we need to extend the guards with constraints of the form $x - y \sim c$ where $x, y \in X$, $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$, but both models are known to be equivalent. In the sequel, timed automata are thus assumed to be split in regions and all the transitions can be fired. Moreover, in order to take into account that zero delays are not allowed in the semantics, transitions with a constraint of the form $x = 0$ are removed and transitions with a punctual constraint (of the form $x = c$) arrive directly in the time-successor (with constraint $x > c$) if x is not reset.

The corner-point abstraction is a refinement of the region abstraction, where states are formed of a region with one of its extremal points. Thus, an $\mathcal{A}$*-pointed region* (pointed region for short) is a pair (R, α) where R is a region and α an integer valuation ($\in (\mathbb{N}_{\leq M} \cup \perp)^X$, $\perp$ if the clock is not bounded in this region) belonging to the closure of R (for the usual topology), in this case, α is said to be a corner of R. The set of $\mathcal{A}$-pointed regions is written $Reg\bullet_{\mathcal{A}}$. The operations defined on the valuations of a set of clocks are extended in a natural way to the corners, with the convention that $M + 1 = \perp$ and $\perp + 1 = \perp$. Then the *timeSucc* function can be extended to pointed regions:

$$\mathit{timeSucc}(R, \alpha) = \begin{cases} (R, \alpha + 1) & \text{if } \alpha + 1 \text{ is a corner of } R \\ (\mathit{timeSucc}(R), \alpha') & \text{otherwise} \end{cases}$$

where $\forall x$, $\alpha'(x) = \alpha(x)$ if x is bounded in $\mathit{timeSucc}(R)$ and else $\alpha'(x) = \perp$.

Using this mapping, the construction of the corner-point abstraction is very similar to the usual region automaton.

Definition 3 (corner-point abstraction). *The* corner-point abstraction *of a timed automaton $\mathcal{A}$ (corner-point of $\mathcal{A}$ for short) is the finite automaton $\mathcal{A}_{cp} = (L_{cp}, L_{0,cp}, F_{cp}, \Sigma_{cp}, E_{cp})$ where $L_{cp} = L \times Reg\bullet_{\mathcal{A}}$ is the set of states, $L_{0,cp} = L_0 \times \{(\{\overline{0}\}, \overline{0})\}$ is the set of initial states, $F_{cp} = F \times Reg\bullet_{\mathcal{A}}$ is the set of accepting states, $\Sigma_{cp} = \Sigma \cup \{\varepsilon\}$, and $E_{cp} \subseteq L_{cp} \times \Sigma_{cp} \times L_{cp}$ is the finite set of edges defined as the union of discrete transitions and idling transitions:*

– discrete transitions: $(\ell, R, \alpha) \xrightarrow{a} (\ell', R', \alpha')$ *if there exists a transition* $\ell \xrightarrow{g,a,X'} \ell'$ *in* $\mathcal{A}$*, such that* $R = g$ *and* $(R', \alpha') = (R_{[X' \leftarrow 0]}, \alpha_{[X' \leftarrow 0]})$*,*
– idling transitions: $(\ell, R, \alpha) \xrightarrow{\varepsilon} (\ell, R', \alpha')$ *if* $(R', \alpha') = \mathit{timeSucc}(R, \alpha)$*.*

In particular, as a consequence of $\perp + 1 = \perp$, there is an idling loop on each state whose region is $\{\perp^X\}$. The *projection* of a (finite or infinite) run $\rho = (\ell_0, v_0) \xrightarrow{\tau_0, a_0} (\ell_1, v_1) \xrightarrow{\tau_1, a_1} \cdots$ of $\mathcal{A}$, denoted by $\mathsf{Proj}(\rho)$, is the set of runs of $\mathcal{A}_{cp}$ such that for all indices i, the i-th discrete transition goes from a state $(\ell_i, R(v_i + \tau_i), \alpha)$ to a state $(\ell_{i+1}, R(v_{i+1}), \alpha')$ and for all clocks $x \in X$, the number $\mu_i(x)$ of idling transitions of the form $(\ell, R, \alpha) \xrightarrow{\varepsilon} (\ell, R, \alpha + 1)$ since the last reset of x has to be equal to $\lfloor v_i(x) + \tau_i \rfloor$ or $\lceil v_i(x) + \tau_i \rceil$. Note that if x is bounded in a region $R(v_i + \tau_i)$, then $\mu_i(x)$ can be recovered from the associated corner α. Given $\varepsilon > 0$, we say that a (finite or infinite) run ρ *mimics up to* $\varepsilon > 0$ a (finite or infinite) run π in $\mathsf{Proj}(\rho)$ if, for all indices i, the i-th discrete transition of π goes from a state $(\ell_i, R(v_i), \alpha)$ such that, for all clock $x \in X$, if $\alpha(x) \neq \perp$ then $|v_i(x) + \tau_i - \alpha(x)| < \varepsilon$ and otherwise $|v_i(x) + \tau_i - \mu_i(x)| < \varepsilon$ (written $||v_i + \tau_i - \alpha|| < \varepsilon$ abusing notations).

In the sequel we often consider cycles of the graph of $\mathcal{A}$ (cycles of $\mathcal{A}$ for short), that is some sequences $\ell_0 \ell_1 \cdots \ell_n = \ell_0$ such that for all $0 \leq i \leq n-1$ there exists an edge from ℓ_i to ℓ_{i+1} in $\mathcal{A}$. Similarly to runs, we define the *projection of a cycle* C of $\mathcal{A}$, denoted by $\mathsf{Proj}(C)$. If C is a simple cycle with no region $\perp^X$, $\mathsf{Proj}(C)$ is the subgraph of $\mathcal{A}_{cp}$ covered by the projection of any finite run of $\mathcal{A}$ along C. If C is a simple cycle with some regions $\perp^X$, we simply add the idling loops associated with each states of the form $(\ell, \{\perp^X\}, \perp^X)$. To define the projection of a cycle C which is not simple, we first unfold the timed automaton $\mathcal{A}$ to obtain an equivalent simple cycle.

Forgetfulness was originally defined in [4] using the orbit graph. We choose here to give an alternative definition of forgetfulness based on the corner-point abstraction, which is less succinct, but will show useful for computing frequencies.

Definition 4 (forgetfulness)

– *A cycle* C *in a timed automaton is* forgetful *if* $\mathsf{Proj}(C)$ *is strongly connected;*
– *A timed automaton is* forgetful *if all its simple cycles are forgetful;*
– *A timed automaton is* strongly forgetful *if all its cycles are forgetful.*

Roughly speaking, forgetful cycles are cycles where some choices of current delays cannot impact forever on the future delays. These cycles can forget previous delays in their long term behaviors. Fig. 1 represents a timed automaton, inspired by [8], that is not forgetful. Indeed, the projection of the single cycle of this timed automaton is the subgraph with bold edges in its corner-point represented in Fig. 2, it is clearly not strongly connected. In fact, if from location ℓ_1 an a is read with x close to 0, it becomes impossible to read an a with x close to 1 in the future. More precisely, delays in ℓ_1 are smaller and smaller. Note that in Fig. 2, we did not draw the edges labelled by ε which lead to states from which no discrete transition can be fired in the future.

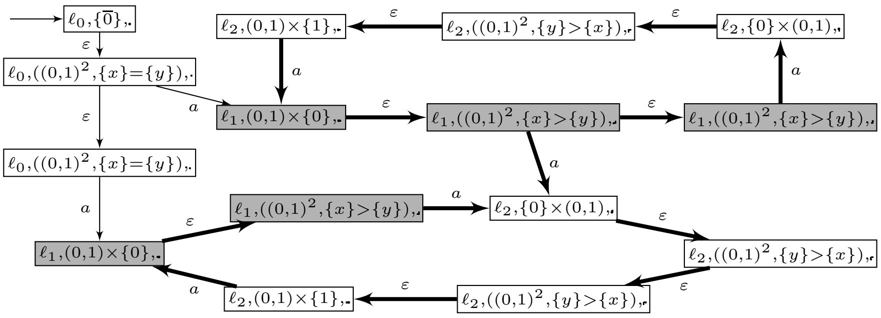


Fig. 2. Corner-point of the timed automaton from Fig. 1

We then define the notion of aperiodicity of a forgetful cycle and forgetful aperiodic timed automata.

Definition 5 (aperiodicity)

- *A forgetful cycle C in a timed automaton is* aperiodic *if for all $k \in \mathbb{N}$, the cycle obtained by the concatenation of k iterations of C is forgetful.*
- *A forgetful timed automaton is* aperiodic *if all its simple cycles are aperiodic;*

Strong forgetfulness trivially implies aperiodicity, whereas forgetfulness does not. Indeed Fig. 3 represents a timed automaton which is forgetful and periodic. The summary of its corner-point illustrates the periodicity. The cycle formed of two iterations of the simple cycle is not strongly connected, it has two distinct connected components. The projection of a forgetful cycle C in $\mathcal{A}_{cp}$ is strongly connected, then given any state s of $\mathcal{A}_{cp}$ in $\mathsf{Proj}(C)$, there are some simple cycles containing s. Intuitively, such a cycle corresponds to a number of iterations of C in $\mathcal{A}$, this is the number of non-consecutive occurrences of states sharing the same location of $\mathcal{A}$ as s. Thus, we can characterize the aperiodicity of a forgetful cycle by a notion of pseudo aperiodicity of its projection.

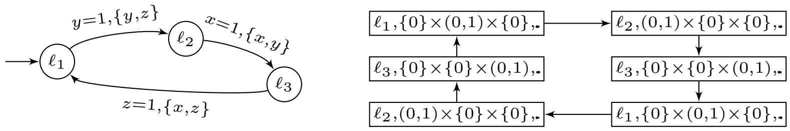


Fig. 3. A forgetful and periodic timed automaton

Proposition 1. *A forgetful cycle C is aperiodic if and only if $(*)$ the greatest common divisor, over the simple cycles D of $\mathsf{Proj}(C)$, of the numbers of iterations of C corresponding to D, is* 1.

The characterization $(*)$ of aperiodicity allows one to check it in the corner-point abstraction. The notion of aperiodicity will be a key for the relaxation of strong forgetfulness in the second part of Section 4.3.

2.3 The Corner-Point Abstraction as a Tool for Frequencies

In the corner-point, the idling transitions which do not change the current region correspond to an elapse of one time unit. In the same way as in [6], these abstract delays are used to abstract the frequencies in a timed automaton by ratios in its corner-point abstraction. To do so, the corner-point is equipped with costs and rewards as follows:

- the *reward* of a transition is 1 if it is of the form $(\ell, R, \alpha) \xrightarrow{\varepsilon} (\ell, R, \alpha')$ and 0 otherwise;
- the *cost* of a transition is 1 if the reward is 1 and the location ℓ is accepting and 0 otherwise.

In particular, the loops on the states whose region is $\{\perp^X\}$ have reward 1. Thanks to these costs and rewards, the *ratio* of an infinite run of the corner-point can be defined, similarly to the frequency in the timed automaton, as the limit sup of the ratios of the accumulated costs over the accumulated rewards. An infinite run in the corner-point is said *reward-converging* (resp. *reward-diverging*) if the accumulated reward is finite (resp. not bounded). This notion is close to zenoness of runs in a timed automaton even if some Zeno runs could be projected to reward-diverging runs in the corner-point abstraction and, the other way around, non-Zeno runs could be projected to reward-converging runs. Thus, we write $\mathsf{Rat}(\mathcal{A}_{cp})$, $\mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$ and $\mathsf{Rat}_{r-c}(\mathcal{A}_{cp})$ for the sets of the ratios of the infinite runs in $\mathcal{A}_{cp}$, the reward-diverging runs in $\mathcal{A}_{cp}$ and the reward-converging ones. We also say *reward-diverging* for a cycle of $\mathcal{A}_{cp}$ whose accumulated reward is positive.

A cycle of $\mathcal{A}_{cp}$ is said *accepting* (resp. *non-accepting*) if all its locations are accepting (resp. non-accepting) and it is said *mixed* if it has both accepting and non-accepting locations.

In the sequel, we often use the following results established in [6] and [7].

Lemma 1 ([6]). *For every run ρ in a one-clock timed automaton $\mathcal{A}$, there are two runs π and π' in $\mathsf{Proj}(\rho)$ respectively minimizing and maximizing the ratio such that:* $\mathsf{Rat}(\pi) \leq \mathsf{freq}_{\mathcal{A}}(\rho) \leq \mathsf{Rat}(\pi')$.
These runs are respectively called the contraction and the dilatation of ρ.

Lemma 2 ([6]). *Let $\{S_1, \cdots, S_k\}$ be the set of SCCs of $\mathcal{A}_{cp}$. The set $\mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$ of ratios of reward-diverging runs in $\mathcal{A}_{cp}$ is equal to $\bigcup_{S_i \in SCC}[m_i, M_i]$ where m_i and M_i are the minimal and the maximal ratios for reward-diverging cycles in S_i. Moreover, if $\mathcal{A}$ has a single clock, then $\mathsf{Freq}_{nZ}(\mathcal{A}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$.*

Lemma 3 ([7]). *Consider a transition $(\ell, R, \alpha) \rightarrow (\ell', R', \alpha')$ in $\mathcal{A}_{cp}$, take a valuation $v \in R$ such that $\delta(v) < \varepsilon$ and $|v(x) - \alpha(x)| = \mu_v(x)$ with $\mu_v(x) = \min\{|v(x) - p| \mid p \in \mathbb{N}\}$, $\nu_v(x, y) = \min\{|v(x) - v(y) - p| \mid p \in \mathbb{N}\}$ and $\delta(v) = \max(\{\mu_v(x)\} \cup \{\nu_v(x, y)\})$. There exists a valuation $v' \in R'$ such that $(\ell, v) \rightarrow (\ell', v')$ in $\mathcal{A}$, $\delta(v') < \varepsilon$ and $|v'(x) - \alpha'(x)| = \mu_{v'}(x)$.*

In particular, the latter lemma implies by induction that any run in $\mathcal{A}_{cp}$ can be mimicked in $\mathcal{A}$ up to any $\varepsilon > 0$.

3 Computation of the Set of Frequencies in a One-Clock Timed Automaton

One-clock timed automata have simpler clock behaviors than the general model. In fact, having a single clock in a timed automaton is quite close to forgetfulness in the sense that each time the clock is reset, the timed automaton forgets all the timing information. In this section, we present a new characterization of forgetfulness and we show the equivalence between forgetfulness and strong forgetfulness when there is a single clock. Last, we propose an expression for the set of frequencies of forgetful one-clock timed automata.

In a one-clock timed automaton, a reset of the clock along a cycle is linked to forgetfulness. The following lemma states the precise characterization of forgetful cycles inspired by this observation.

Proposition 2. *Let C be a cycle of a one-clock timed automaton. Then, C is forgetful if and only if the clock is reset or not bounded along C.*

In fact, Proposition 2 implies that the cycle obtained by concatenation of any sequence of forgetful cycles in a one-clock timed automaton is also forgetful. Indeed, if the clock is reset or not bounded along each cycle of the sequence, it is clearly the case for the sequence itself.

Corollary 1. *A one-clock timed automaton is forgetful iff it is strongly forgetful.*

Recall that, as illustrated in Fig. 1, Corollary 1 (as well as Proposition 2) does not hold for n-clock timed automata.

Let us now consider the set of the frequencies in a one-clock timed automaton. By Lemma 2, if there are only non-Zeno runs in a timed automaton, then the set of the frequencies equals to the set of the ratios in the corner-point. Firstly, the particular case where a timed automaton has a reward-converging cycle in its corner-point containing both accepting and non-accepting locations is easy to treat as stated in the following proposition.

Proposition 3. *Let $\mathcal{A}$ be a forgetful one-clock timed automaton. If there is a mixed reward-converging cycle in its corner-point $\mathcal{A}_{cp}$, then* $\mathsf{Freq}_Z(\mathcal{A}) =]0,1[$ *and* $\mathsf{Freq}_{nZ}(\mathcal{A}) = [0,1]$.

Now, for the general case, it is possible to consider only timed automata which do not have such cycles in their corner-point. This allows us to give a general expression for the set of frequencies of a forgetful one-clock timed automaton. For readability, let us define some notations. Given C a cycle of $\mathcal{A}$ having a reward-converging cycle in its projection, we write $p(C)$ for the set of ratios of cycle-free prefixes ending in reward-converging cycles of $\mathsf{Proj}(C)$ and $c(C)$ for the set of ratios of co-reachable reward-diverging cycles. By convention, we let $\max(\emptyset) = -1$ and $\min(\emptyset) = 2$. Then, we define $M(C) = \max(p(C) \cup c(C))$, $m(C) = \min(p(C) \cup c(C))$,

$$M(\mathcal{A}_{cp}) = \max\{M(C) \mid C \text{ acc. cycle of } \mathcal{A} \text{ with a r.-c. cycle in } \mathsf{Proj}(C)\} \text{ and}$$

$$m(\mathcal{A}_{cp}) = \min\{m(C) \mid C \text{ non-acc. cycle of } \mathcal{A} \text{ with a r.-c. cycle in } \mathsf{Proj}(C)\}.$$

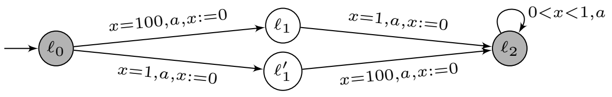


Fig. 4. A non-forgetful counterexample for Theorem 1

Theorem 1. *Let $\mathcal{A}$ be a forgetful one-clock timed automaton. If there is a mixed reward-converging cycle in $\mathcal{A}_{cp}$, then $\mathsf{Freq}_Z(\mathcal{A}) =]0,1[$ and $\mathsf{Freq}_{nZ}(\mathcal{A}) = [0,1]$. Otherwise:* $\mathsf{Freq}(\mathcal{A}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp}) \cup \big[0, M(\mathcal{A}_{cp})\big[\cup \big]m(\mathcal{A}_{cp}), 1\big]$.

Proof. The first part of Theorem 1 is established in Proposition 3, let us now assume that there is no mixed reward-converging cycles in $\mathcal{A}_{cp}$. By Lemma 2, for non-Zeno runs: $\mathsf{Freq}_{nZ}(\mathcal{A}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$. The rest of the proof is based on the following lemma dealing with plain reward-converging cycles.

Lemma 4. *Let C be a cycle of a one-clock forgetful timed automaton $\mathcal{A}$:*

- *If $\mathsf{Proj}(C)$ contains a non-accepting reward-converging cycle, then the set of frequencies of the infinite runs of $\mathcal{A}$ ending in C is $\big[0, M(C)\big[$.*
- *If $\mathsf{Proj}(C)$ contains an accepting reward-converging cycle , then the set of frequencies of the infinite runs of $\mathcal{A}$ ending in C is $\big]m(C), 1\big]$.*

Back to the proof of the second part of Theorem 1, the inclusion from right to left is straightforward from the non-Zeno case and Lemma 4.

Thanks to the equality in the non-Zeno case, the inclusion from left to right is only needed for the subset $\mathsf{Freq}_Z(\mathcal{A})$. Let thus ρ be a Zeno run. It can be projected on a reward-converging run in the corner-point. This projection necessarily ends in a strongly connected subgraph of the corner-point having zero rewards and containing only accepting locations or only non-accepting locations. We study the case where all the locations of the end are non-accepting, the other case is symmetric. By Lemma 4, the prefix of ρ corresponding to the prefix of the projection before the infinite suffix in the subgraph has a frequency smaller than $M(C)$ for a cycle C having a reward-converging projection. To conclude, the frequency of ρ is smaller than the prefix because all the locations of the suffix are non-accepting. □

Note that if the timed automaton is not forgetful, the form of the set of the frequencies can be very different from the expression given in Theorem 1. Fig. 4 gives an example of non-forgetful timed automaton such that $\mathsf{Freq}(\mathcal{A}) = \big]\frac{1}{101}, \frac{2}{102}\big[\cup \big]\frac{100}{101}, \frac{101}{102}\big[$. There is no reward-diverging run in $\mathcal{A}_{cp}$, $M(\mathcal{A}_{cp}) = -1$ because there is no accepting reward-converging cycle in $\mathcal{A}_{cp}$ and $m(\mathcal{A}_{cp}) = \frac{1}{101}$, hence the expected set of frequencies would be $]\frac{1}{101}, 1]$. The difference with forgetful timed automata is that the accumulated delays in ℓ_2 cannot diverge, therefore it is not possible to increase the frequency as much as necessary. In particular, there is no infinite run of frequency 1. More generally, this example illustrates a simple manner to obtain, for the set of frequencies, any finite union of open intervals included in $[0, 1]$.

4 Extension to n-Clock Timed Automata

There is a real gap between one-clock timed automata and n-clock timed automata. For example, in [10], the reachability problem for one-clock timed automata is proved to be NLOGSPACE-complete, whereas it becomes NP-hard with two clocks. As an other example, the language inclusion problem which is undecidable in the general case [1], becomes decidable with at most one clock [11]. In this section, we use forgetfulness and time divergence to compute the set of frequencies in n-clock timed automata. Note that these assumptions are strong but can be justified by implementability concerns.

4.1 Forgetfulness in n-Clock Timed Automata

The goal is to find some reasonable assumptions to obtain a class of timed automata whose sets of frequencies are exactly sets of ratios of their corner-point abstractions. We do not want to complexity our problem dealing with Zeno runs as we did in one-clock timed automata for which the Zeno case is already non-trivial. More precisely, we want to extend the result $\mathsf{Freq}_{nZ}(\mathcal{A}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$ of [6], from one-clock timed automata to n-clock timed automata. To do so, we first assume that timed automata are strongly non-Zeno, that is in every cycle there is one clock which is reset and lower guarded by a positive constant. This implies that there is no reward-converging run in its corner-point (strong reward-divergence [7]). For one-clock timed automata strong non-zenoness is strictly stronger than forgetfulness and implies that $\mathsf{Freq}(\mathcal{A}) = \mathsf{Freq}_{nZ}(\mathcal{A}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp})$. Unfortunately, this assumption is not sufficient for n-clock timed automata. For example, the timed automaton in Fig. 5, taken from [6], is strongly non-Zeno and such that $\mathsf{Freq}(\mathcal{A}) =]0,1] \neq \{0\} \cup \{1\} = \mathsf{Rat}(\mathcal{A}_{cp})$. In fact, this timed automaton is a typical example of forgetful timed automaton. Delays in ℓ_1 have to be larger and larger along cycles, which ensures that frequency 0 cannot be reached in $\mathcal{A}$. On the contrary, in $\mathcal{A}_{cp}$, either the accumulated reward in ℓ_1 is 0 (ratio 0) or there is one idling transition with reward 1 from a state of $\mathcal{A}_{cp}$ with location ℓ_1 and in the future, there always are such transitions in states of the form (ℓ_1, R, α) (ratio 1). Therefore, except over one-clock timed automata, forgetfulness and strong reward-divergence are not comparable.

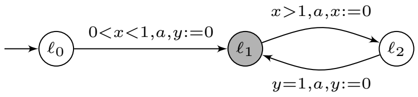

$$\begin{tikzcd}
\arrow[r] & \ell_0 \arrow[r, "0<x<1{,}a{,}y:=0"] & \ell_1 \arrow[r, bend left, "x>1{,}a{,}x:=0"] & \ell_2 \arrow[l, bend left, "y=1{,}a{,}y:=0"]
\end{tikzcd}$$

Fig. 5. A non-forgetful strongly non-Zeno timed automaton

The following theorem is the first illustration of the utility of forgetfulness to compute the set of frequencies in timed automata with several clocks.

Theorem 2. *Let $\mathcal{A}$ be a strongly non-Zeno and forgetful timed automaton. Then* $\mathsf{Freq}(\mathcal{A}) \subseteq \mathsf{Rat}(\mathcal{A}_{cp})$.

Proof (sketch). The idea is that the infinite run consisting in an infinite iteration of the cycle of minimal ratio in $\mathcal{A}_{cp}$ has a ratio smaller than the frequency of any infinite run in $\mathcal{A}$. Symmetrically, there is a run of ratio larger than the frequency of any infinite run in $\mathcal{A}$. The theorem is thus straightforward if there is a single SCC in $\mathcal{A}_{cp}$. Otherwise, forgetfulness is the key to obtain the inclusion $\mathsf{Freq}(\mathcal{A}) \subseteq \mathsf{Rat}(\mathcal{A}_{cp})$ instead of simple bounds. Indeed, given an infinite run ρ ending in an SCC of $\mathcal{A}$, by forgetfulness of the cycles, all the projections of ρ end in the same SCC of $\mathcal{A}_{cp}$. The proof can thus be done in this SCC by neglecting the prefix thanks to time divergence. □

In the sequel, we see how strongly non-zenoness and forgetfulness can be useful to obtain the other inclusion. The problem is not trivial even under these assumptions and the proof techniques could certainly be interesting in different contexts. This section allows to understand several subtleties of forgetfulness.

4.2 Techniques to Compute the Frequencies

In this section, we explain the technical aspects which allow the extension to n-clock timed automata. First, thanks to Lemma 3 we know that any infinite run in a corner-point $\mathcal{A}_{cp}$ can be mimicked up to any $\varepsilon > 0$. This lemma implies that respective lower and upper bounds of the sets of ratios and frequencies are equal, but as seen with the timed automaton in Fig. 5, $\mathsf{Freq}(\mathcal{A})$ can be very different from $\mathsf{Rat}(\mathcal{A}_{cp})$ when $\mathcal{A}$ is not forgetful. Second, the following lemma established in [12] expresses the preservation of some barycentric relations between valuations along cycles.

Lemma 5 ([12]). *Let $\mathcal{A}$ be a timed automaton and an edge (ℓ, g, a, X', ℓ') such that $(\ell, v) \to (\ell', v')$ and $(\ell, w) \to (\ell', w')$ with $R(v) = R(w)$ and $R(v') = R(w')$, then for any $\lambda \in [0,1]$ $(\ell, \lambda v + (1-\lambda)w) \to (\ell', \lambda v' + (1-\lambda)w')$.*

Naturally, this lemma can be extended to finite sequences of edges by induction.

The combination of both lemmas helps us to prove that if along a given cycle one can go from every corner to a fixed corner α, then along this cycle one can go as close to α as necessary in $\mathcal{A}$. This way of reducing the distance to corners is the key for the extension to n-clock timed automata. Indeed, when time diverges (non-zenoness), if an infinite run ρ in $\mathcal{A}$ mimics an infinite run π of $\mathcal{A}_{cp}$ up to ε converging to 0 along ρ (*i.e.* for all ε there is a suffix of ρ which mimics the corresponding suffix of π up to ϵ), then $\mathsf{freq}_{\mathcal{A}}(\rho) = \mathsf{Rat}(\pi)$.

Lemma 6. *Let $\mathcal{A}$ be a timed automaton and $\rho = (\ell_0, v_0) \xrightarrow{\tau_0, a_0} (\ell_1, v_1) \xrightarrow{\tau_1, a_1} \cdots \xrightarrow{\tau_{n-1}, a_{n-1}} (\ell_0, v_n)$ with $R(v_0) = R(v_n) =: r$ be a finite run of $\mathcal{A}$. If given a corner α_n of the region r and for all (ℓ_0, r, α) there is a finite run from (ℓ_0, r, α) to (ℓ_0, r, α_n) in $\mathsf{Proj}(\rho)$, then for all $\varepsilon > 0$, there exists $\rho' = (\ell_0, v_0) \xrightarrow{\tau'_0, a_0} (\ell_1, v'_1) \cdots \xrightarrow{\tau'_{n-1}, a_{n-1}} (\ell_0, v'_n)$ such that $\mathsf{Proj}(\rho') = \mathsf{Proj}(\rho)$ and $||v'_n - \alpha_n|| < \varepsilon$.*

Proof. Let us start by fixing, for all corners α, a valuation v^{ε}_{α} in r which is very close to α. Thanks to these valuations, we then define a barycentric expression for v_0. Let Ω_r the set of the corners of r. As the closure of r is the convex hull of Ω_r (for the usual topology of $\mathbb{R}^X$), there exists $(v^\varepsilon_\alpha \in r)_{\alpha \in \Omega_r}$ and $(\lambda_\alpha \in [0,1])_{\alpha\in\Omega_r}$ such that $\sum_{\alpha\in\Omega_r} \lambda_\alpha = 1$, $v_0 = \sum_{\alpha\in\Omega_r} \lambda_\alpha v^\varepsilon_\alpha$ and $||v^\varepsilon_\alpha - \alpha|| < \varepsilon$. By assumptions, there are some paths in $\mathsf{Proj}(\rho)$ going from each α to α_n. Thanks to Lemma 3, there are some finite runs from each of our valuations v^ε_α very close to the corners α to some valuations $v^\varepsilon_{\alpha,\alpha_n}$ very close to a common corner α_n. Formally, there exists $(v^\varepsilon_{\alpha,\alpha_n} \in r)_{\alpha\in\Omega_r}$ and some finite runs $(\rho_\alpha)_{\alpha\in\Omega_r}$ from $(\ell_0, v^\varepsilon_\alpha)$ to $(\ell_0, v^\varepsilon_{\alpha,\alpha_n})$ in $\mathcal{A}$ with $||v^\varepsilon_{\alpha,\alpha_n} - \alpha_n|| < \varepsilon$. Then, by Lemma 5, there is a finite run ρ' from (ℓ_0, v_0) to the state with location ℓ_0 and the valuation equal to the barycenter of the valuations $v^\varepsilon_{\alpha,\alpha_n}$ which is very close to α_n by the triangle inequality. Formally, there is a finite run ρ' from $(\ell_0, v_0 = \sum_{\alpha\in\Omega_r} \lambda_\alpha v^\varepsilon_\alpha)$ to $(\ell_0, \sum_{\alpha\in\Omega_r} \lambda_\alpha v^\varepsilon_{\alpha,\alpha_n})$. To conclude, ρ' is as needed because, by the triangle inequality, $|| \sum_{\alpha\in\Omega_r} \lambda_\alpha v^\varepsilon_{\alpha,\alpha_n} - \alpha_n|| \leq \sum_{\alpha\in\Omega_r} \lambda_\alpha ||v^\varepsilon_{\alpha,\alpha_n} - \alpha_n|| < \varepsilon$. □

To use Lemma 6, we need to find a cycle in $\mathcal{A}_{cp}$ which allows, in some sense, to synchronize all the corners of a region to a common one. Indeed, each run in $\mathcal{A}_{cp}$ corresponds to a run in $\mathcal{A}$, the existence of ρ is not a real constraint. Moreover, Lemma 6 does not depend on forgetfulness of timed automata. The following lemma illustrates how forgetfulness can help to use Lemma 6.

Lemma 7. *Let $\mathcal{A}$ be a timed automaton, X its set of clocks and a sequence $(c_i)_{1\leq i\leq K}$ with $K = 2^{|X|+1}$, of forgetful cycles containing the location ℓ of $\mathcal{A}$ such that all the cycles obtained by concatenation of the cycles of a subsequence $(c_k)_{1\leq i\leq k\leq j\leq K}$ are forgetful. Then for all pairs of corners (α, α') of the region R associated to ℓ, there is a finite run of the form $(\ell, R, \alpha) \xrightarrow{\pi_1} (\ell, R, \alpha_1) \xrightarrow{\pi_2} \cdots (\ell, R, \alpha_{K-1}) \xrightarrow{\pi_K} (\ell, R, \alpha')$ such that for all indices i, π_i corresponds to one iteration of c_i.*

Proof. Abusing notations we write $\pi \in \mathsf{Proj}(c)$ for "π corresponds to one iteration of c" Consider the subset construction with $s_0 = \{(\ell, R, \alpha)\}$ and $s_{i+1} = \{(\ell, R, \beta') \,|\, \exists(\ell, R, \beta) \in s_i, \exists\pi'_i \in \mathsf{Proj}(c_i),$ s.t. $(\ell, R, \beta) \xrightarrow{\pi'_i} (\ell, R, \beta')\}$.

First, there are at most $|X| + 1$ corners in R, hence there are at most $K = 2^{|X|+1}$ subsets of $(\ell, R, \text{all}) := \{(\ell, R, \alpha) \,|\, \alpha$ corner of $R\}$. Second, by forgetfulness of the c_i's, if $s_i = (\ell, R, \text{all})$ then for all $j > i$, $s_j = (\ell, R, \text{all})$. Third, there is no other cycles in the subset construction. Indeed, if there exists indices $i < j$ such that $s_i = s_j \neq (\ell, R, \text{all}) := \{(\ell, R, \alpha) \,|\, \alpha$ corner of $R\}$ then the cycle obtained by concatenation of cycles $c_{i+1}, \cdots, c_j$ is not forgetful, which contradicts strong forgetfulness.

As a consequence, the subset construction loops in (ℓ, R, all) forever after a cycle-free prefix whose length is thus smaller than K. Hence, there is a finite run of the form $(\ell, R, \alpha) \xrightarrow{\pi_1} (\ell, R, \alpha_1) \xrightarrow{\pi_2} \cdots (\ell, R, \alpha_{K-1}) \xrightarrow{\pi_K} (\ell, R, \alpha')$ such that for all indices i, $\pi \in \mathsf{Proj}(c_i)$. □

In the next sections we use these two lemmas to prove that our assumptions are sufficient to ensure the existence of such synchronizing cycles along infinite runs that we want to mimic in $\mathcal{A}$.

4.3 Frequencies in n-Clock Forgetful Timed Automata

We first consider the case of strongly forgetful timed automata. Thanks to Lemma 6 and by observing the consequences of forgetfulness of all the cycles in a timed automaton, we obtain a theorem which is as constructive as the corresponding result for one-clock timed automata from [6].

Theorem 3. *Let $\mathcal{A}$ be a strongly non-Zeno strongly forgetful timed automaton. Then, for every infinite run π in the corner-point of $\mathcal{A}$, there exists an infinite run ρ_π in $\mathcal{A}$ such that $\pi \in \mathsf{Proj}(\rho_\pi)$ and $\mathsf{freq}_{\mathcal{A}}(\rho_\pi) = \mathsf{Rat}(\pi)$.*

The idea is to prove, for every run π in $\mathcal{A}_{cp}$, the existence of synchronizing cycles infinitely often along π which allow to mimic it up to an ε converging to 0.

Proof. Along the infinite run π of $\mathcal{A}_{cp}$, there is a pair (ℓ, R) which appears infinitely often, possibly with different corners. Let $(\ell, R, \alpha_i)_{i\in\mathbb{N}}$ be a sequence of the occurrences of (ℓ, R) and $(\pi_i)_{i\in\mathbb{N}}$ the sequence of factors of π leading respectively from (ℓ, R, α_i) to (ℓ, R, α_{i+1}). Each π_i corresponds to a forgetful cycle c_i in $\mathcal{A}$ hence by Lemma 7, for all pairs (α, α') of corners of the region R, there is a finite run of the form $(\ell, R, \alpha) \xrightarrow{\pi'_1} (\ell, R, \alpha_1) \xrightarrow{\pi'_2} \cdots (\ell, R, \alpha_{K-1}) \xrightarrow{\pi'_K} (\ell, R, \alpha')$ with $K = 2^{|X|+1}$ and such that for all indices i, π corresponds to one iteration of c_i. In particular, this finite run belongs to the projections of exactly the same runs as $\pi_1 \cdot \pi_2 \cdots \pi_K$. As a consequence, for any finite run $\rho = (\ell, v_0) \xrightarrow{\tau_0, a_0} (\ell_1, v_1) \xrightarrow{\tau_1, a_1} \cdots \xrightarrow{\tau_{n-1}, a_{n-1}} (\ell, v_n)$ with $R(v_0) = R(v_n) = R$ and such that $\pi_0.\pi_1 \cdots \pi_K \in \mathsf{Proj}(\rho)$, for any corner β_n of the region R and for all (ℓ, R, α) there is a finite run from (ℓ, R, α) to (ℓ, R, β_n) in $\mathsf{Proj}(\rho)$. Hence, Lemma 6 can be applied to such finite runs. Then, for any ε and given ρ^i a mimicking of π until (ℓ, R, α_i), Lemma 6 ensures the existence of a extension of ρ^i to ρ^{i+K} mimicking π until (ℓ, R, α_{i+K}), such that $||v - \alpha_{i+K}|| < \varepsilon$ where v is the last valuation of ρ^{i+K}. In words, it is possible to fix some finite factors along π which allow to go as close as necessary from a corner of π along a mimicking ρ. Out of these factors, the distance to the corners of π can be preserved (Lemma 3). To conclude, these factors can be placed infinitely often to allow the convergence of the distance to the corner of π to 0, but as rarely as necessary to be neglected in the computation of the frequency. □

Theorem 3 implies that the set of ratios $\mathsf{Rat}(\mathcal{A}_{cp})$ is included in the set of frequencies $\mathsf{Freq}(\mathcal{A})$. This implies, together with Theorem 2, that if $\mathcal{A}$ is a strongly non-Zeno and strongly forgetful timed automaton, then $\mathsf{Freq}(\mathcal{A})$ is equal to $\mathsf{Rat}(\mathcal{A}_{cp})$. Strong forgetfulness is a realistic assumption from an implementability point of view, but is not satisfactory because of its difficulties to be checked. Indeed, checking if a cycle is forgetful can be done thanks to the corner-point, but there is an unbounded number of cycles in a timed automaton and we do not know any property which would allow, in general, to avoid to check them all. As a consequence, it is important to relax this assumption. We did not succeed in proving that strong forgetfulness can be relaxed in Theorem 3. Nevertheless, the inclusion $\mathsf{Rat}(\mathcal{A}_{cp}) \subseteq \mathsf{Freq}(\mathcal{A})$ still holds when strong forgetfulness is replaced by forgetfulness and aperiodicity, both of which can be checked on the corner-point abstraction.

Theorem 4. *Let $\mathcal{A}$ be a strongly non-Zeno, forgetful and aperiodic timed automaton. Then,* $\mathsf{Rat}(\mathcal{A}_{cp}) \subseteq \mathsf{Freq}(\mathcal{A})$.

Proof (sketch). The idea is to prove that, for every $\mathsf{rat} \in \mathsf{Rat}(\mathcal{A}_{cp})$, there exists an infinite run π_{rat} in $\mathcal{A}_{cp}$ of ratio rat and such that there exists a infinite run ρ_π of $\mathcal{A}$ with $\mathsf{freq}_{\mathcal{A}}(\rho_\pi) = \mathsf{Rat}(\pi_{\mathsf{rat}})$ and $\pi_{\mathsf{rat}} \in \mathsf{Proj}(\rho_\pi)$. Thanks to Lemma 2 and by reward-divergence, we have the following expression for the set of the ratios $\mathsf{Rat}(\mathcal{A}_{cp}) = \mathsf{Rat}_{r-d}(\mathcal{A}_{cp}) = \bigcup_{S_i \in SCC}[m_i, M_i]$. In fact, for all i, each value $\mathsf{rat} \in [m_i, M_i]$ is the ratio of a run π_{rat} in $\mathcal{A}_{cp}$ which alternates with the suitable proportions some cycles c_i of ratio m_i and C_i of ratio M_i in S_i. The prefix to go to c_i and the finite runs to go from a cycle to the other are neglected in the computation of the ratio by performing sufficiently many iterations at each step. Such a π_{rat} can be mimicked up to any $\varepsilon > 0$ (Lemma 3). We thus use Lemmas 7 and 6 to decrease ε. The finite runs to go from C_i to c_i are simply concatenated with $2^{|X|+1}$ iterations of c_i. The cycle c_i corresponds, in $\mathcal{A}_{cp}$ to a cycle (simple or a concatenation of a single simple cycle) $\hat{c}_i$. Aperiodicity entails that the concatenations of $\hat{c}_i$ are forgetful. Hence, Lemma 7 ensures that the finite run constituted of $2^{|X|+1}$ iterations of c_i is synchronizing and Lemma 6 that ε can decrease each time that π_{rat} goes from C_i to c_i. □

We thus obtain the following result as a corollary of Theorems 2 and 4.

Corollary 2. *Let $\mathcal{A}$ be a strongly non-Zeno, forgetful and aperiodic timed automaton. Then,* $\mathsf{Freq}(\mathcal{A}) = \mathsf{Rat}(\mathcal{A}_{cp})$.

Strong forgetfulness implies aperiodicity, hence Theorem 3 cannot help to established this equality in a more general case. However, note that Theorem 4 does not imply Theorem 3. In Theorem 3, not only the inclusion $\mathsf{Rat}(\mathcal{A}_{cp}) \subseteq \mathsf{Freq}(\mathcal{A})$ is established, but also for all infinite runs π in $\mathcal{A}_{cp}$ there exists an infinite run ρ in $\mathcal{A}$ with $\pi \in \mathsf{Proj}(\rho)$ and $\mathsf{freq}_{\mathcal{A}}(\rho) = \mathsf{Rat}(\pi)$. In Theorem 4, this is only proved for some infinite runs π of $\mathcal{A}_{cp}$.

4.4 Discussion About Assumptions

As explained above, our will to relax the strong forgetfulness is due to its difficulties to be checked. Strong forgetfulness clearly implies at once forgetfulness and aperiodicity, but a first open question is whether the other implication is true. Indeed, we did not find any example of forgetful aperiodic timed automaton which is non-strongly forgetful. We think that, either there are some one but probably with more than two clocks which is difficult to visualize, or the implication is true and proving this statement could lead to fundamental advances in the understanding of the corner-point abstraction.

An other open question is whether the hypothesis of aperiodicity in Theorem 4 can be relaxed. We use this hypothesis in the proof, but could not find counterexamples. We built some examples of periodic timed automata as in Fig. 3 , but periodic timed automata seem to be degenerated and in particular, based on punctual guards which implies bijections between runs in the timed automaton and those in its corner-point abstraction.

5 Conclusion

A quantitative semantics based on frequencies has recently been proposed for timed automata in [6]. In this paper, we used the notion of forgetfulness introduced in [4] to extend the results about frequencies in timed automata. On the one hand, thanks to forgetfulness we can compute the set of frequencies in one-clock timed automata even with Zeno behaviors, whereas only the bounds of this set was computed in [6]. On the other hand, with forgetfulness and time-divergence inspired by [7], we compute the set of frequencies in a class of n-clock timed automata, whereas techniques of [6] were not applicable. In the future, we would like to investigate more deeply the difference between forgetfulness and strong forgetfulness with the hope to extend Theorem 3. Moreover, Theorem 2 is less constructive than the equivalent result for one-clock timed automata which use notions of contraction and dilatation of a run. It would be interesting to see if forgetfulness could help to extend these constructions to n-clock timed automata. Finally, our main tool presented in Lemma 6 can be easily used for the scheduling problem in timed automata with costs and rewards studied in [7]. Thus, we can prove that in strongly non-Zeno forgetful timed automata, there is always an infinite run whose ratio is optimal. We hope that Lemma 6, which is fundamental, will be useful for a lot of problems for which the corner-point is suitable.

Acknowledgements. I am very grateful to Nathalie Bertrand for useful discussions and detailed proofreadings of this paper. Moreover, I would like to thank the reviewers for their very interesting remarks.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., La Torre, S., Pappas, G.J.: Optimal Paths in Weighted Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
3. Baier, C., Bertrand, N., Bouyer, P., Brihaye, Th., Größer, M.: Almost-sure model checking of infinite paths in one-clock timed automata. In: Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008), pp. 217–226. IEEE (2008)
4. Basset, N., Asarin, E.: Thin and Thick Timed Regular Languages. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 113–128. Springer, Heidelberg (2011)

5. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.M.T., Vaandrager, F.W.: Minimum-Cost Reachability for Priced Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
6. Bertrand, N., Bouyer, P., Brihaye, T., Stainer, A.: Emptiness and Universality Problems in Timed Automata with Positive Frequency. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 246–257. Springer, Heidelberg (2011)
7. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design 32(1), 3–23 (2008)
8. Cassez, F., Henzinger, T.A., Raskin, J.-F.: A Comparison of Control Problems for Timed and Hybrid Systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)
9. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science 282, 101–150 (2002)
10. Laroussinie, F., Markey, N., Schnoebelen, P.: Model Checking Timed Automata with One or Two Clocks. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 387–401. Springer, Heidelberg (2004)
11. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: Closing a decidability gap. In: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 54–63. IEEE (2004)
12. Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)
13. Stainer, A.: Frequencies in forgetful timed automata. Research Report 8009, INRIA, Rennes, France (July 2012), http://hal.inria.fr/hal-00714262

Mcta: Heuristics and Search for Timed Systems

Martin Wehrle[1] and Sebastian Kupferschmid[2]

[1] University of Basel, Switzerland
martin.wehrle@unibas.ch
[2] ATRiCS Advanced Traffic Solutions GmbH, Freiburg, Germany
sebastian.kupferschmid@atrics.com

Abstract. Mcta is a directed model checking tool for concurrent systems of timed automata. This paper reviews Mcta and its new developments from an implementation point of view. The new developments include both heuristics and search techniques that define the state of the art in directed model checking. In particular, Mcta features the powerful class of pattern database heuristics for efficiently finding shortest possible error traces. Furthermore, Mcta offers new search techniques based on multi-queue search algorithms. Our evaluation demonstrates that Mcta is able to significantly outperform previous versions of Mcta as well as related state-of-the-art tools like Uppaal and Uppaal/Dmc.

1 Introduction

Model checking of real-time systems is an interesting and important research issue in theory and in practice. In this context, Uppaal [2,3] is a state-of-the-art model checker for real-time systems that are modeled as timed automata [1]. Uppaal offers several approaches to successfully tackle the state explosion problem. However, to efficiently find short error traces in large concurrent systems of timed automata, additional search techniques are desired.

Mcta [19] is a tool for model checking large systems of concurrent timed automata. Mcta is optimized for *falsification*, i. e., for the efficient detection of short error traces in faulty systems. Therefore, Mcta applies the *directed model checking* approach [9]. Directed model checking is a version of model checking that applies a *distance heuristic* and a special *search algorithm* to guide the search towards error states. Distance heuristics compute a numeric value for every state s encountered during the search, reflecting an estimation of the length of a shortest trace from s to an error state. These values are used by the underlying search algorithm (e. g., the well-known A* algorithm [11,12]) to guide the search. Overall, most of the proposed distance heuristics can be computed automatically based the description of the input system. Therefore, directed model checking is a fully automatic approach as well. For the special setting when *admissible* distance heuristics are applied (i. e., distance heuristics that are guaranteed to never overestimate the real error distance), directed model checking allows for *optimal* search, i. e., in this case, directed model checking computes *shortest possible* error traces with the A* search algorithm. This is desirable because shorter error traces allow one to better understand the reason for the bug.

In this paper, we review MCTA and its new developments from an implementation point of view. In particular, we provide an overview of MCTA's lightweight and flexible architecture. This architecture is tailored to engineering an efficient model checker based on heuristic search methods. The current version of MCTA (MCTA-2012.05 or MCTA-2012 for short) supports both optimal and suboptimal search methods. In the setting of optimal search, MCTA-2012 features a powerful admissible *pattern database heuristic*. To get a feeling of the power of MCTA-2012's heuristic search methods in an optimal search setting, we provide a snapshot of MCTA-2012's performance in Table 1. The problems D_1–D_6 stem from an industrial real-time case study (see Sec. 6 for details). A dash indicates that the corresponding tool exceeded the memory limit of 4 GByte. We observe that MCTA-2012 shows superior performance.

Table 1. Snapshot of MCTA-2012's performance in an optimal search setting. The table provides the best runtime in seconds for MCTA-2012, for MCTA-0.1 (corresponding to the predecessor of MCTA-2012 that has been released in 2008 [19]), for UPPAAL/DMC, and for UPPAAL-4.0.13.

Instance	MCTA-2012	MCTA-0.1	UPPAAL/DMC	UPPAAL-4.0.13
D_1	10.2	81.2	84.7	90.5
D_2	12.2	433.4	255.3	539.0
D_3	12.3	487.0	255.6	548.4
D_4	13.9	288.0	256.7	476.4
D_5	60.1	–	–	–
D_6	66.4	–	–	–

Furthermore, in the setting of suboptimal search, MCTA now features search algorithms that extend classical directed model checking by applying a *multi-queue* approach using *several* open queues instead of only one. This approach can be applied with arbitrary distance heuristics.

MCTA is written in C++ and Python. It is released under the GPL and can be obtained from the website `http://mcta.informatik.uni-freiburg.de/`. The website particularly provides a binary of MCTA, the source code, relevant benchmark problems, and related papers. Subsequently, when we want to distinguish between the new and the earlier version of MCTA, the new version is called MCTA-2012, whereas the earlier version that corresponds to the last tool paper [19] is called MCTA-0.1 (as also indicated on the website).

The remainder of the paper is organized as follows. In Sec. 2, we give the preliminaries that are needed for this work. In Sec. 3, we present MCTA's basic architecture, based on which the newly developed components and their implementation are described in detail in Sec. 4 and Sec. 5. Furthermore, an experimental evaluation is given in Sec. 6. Finally, we conclude the paper in Sec. 7 and give an overview of next development steps.

2 Preliminaries

In this section, we introduce the preliminaries that are needed for this paper. In Sec. 2.1, we give a brief introduction to the timed automata formalism. In Sec. 2.2, we describe the classical directed model checking approach that MCTA is based on.

2.1 Timed Automata

We consider a class of timed automata that is extended with bounded integer variables. A timed automaton $\mathcal{A}$ consists of a finite set of *locations* and a set of *edges* that connect (some of) $\mathcal{A}$'s locations. Every location features a *clock invariant* represented as a conjunction of clock constraints $x \prec n$ for a clock x and an integer $n \in \mathbb{N}$, where $\prec \in \{<, \leq\}$. Furthermore, edges are annotated with *guards*, *effects* and a *synchronization label* from a global synchronization alphabet Σ. The *guard* of an edge consists of a clock and an integer guard, consisting of clock and integer constraints, respectively. The *effect* of an edge consists of a list of clocks (to be reset) and a list of integer assignments. A *(parallel) system* of timed automata is defined as a set $\mathcal{M} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ of timed automata.

The operational semantics of a system $\mathcal{M}$ of timed automata is defined as follows. As the explicit size of $\mathcal{M}$'s state space is infinite, we use a symbolic representation of the state space that is sound and complete. This representation is based on *zones*. In this setting, a *global state* consists of a discrete part and a symbolic part. It is defined as a tuple $s = \langle L, V, Z \rangle$, where L is a function that evaluates for every automaton in $\mathcal{M}$ the current location in s and V is a function that evaluates for every integer variable the current value in s. L and V define the discrete part of s. Furthermore, Z is the *zone* of s, i. e., a conjunction of clock constraints that describes the possible values of the clock variables in s. Z defines the symbolic part of s. We define a *transition* in $\mathcal{M}$ either as a set of one edge that has a special internal void label (asynchronous communication), or as a set of two edges from different automata with the same synchronization label from Σ. Guards and effects of transitions are defined accordingly. A transition t is *applicable* in a state s if the location, integer and clock guards of t are satisfied in s. In this case, the *successor state* $t[s]$ of s is defined as the state s where the locations and the integer values are first changed according to the effect of t, and the zone of $t[s]$ is defined as an update of the zone of s according to the clock guard and the clock resets of t. Finally, the resulting zone of $t[s]$ is maximized while preserving consistency with the location invariants of the destination locations of t. The resulting state space of $\mathcal{M}$ is called the *zone graph* of $\mathcal{M}$.

2.2 Directed Model Checking

In general, depending on the distance heuristic and the search algorithm, directed model checking influences the order in which the state space is traversed. For a system of timed automata $\mathcal{M}$, directed model checking is performed on the zone graph of $\mathcal{M}$. The basic model checking algorithm of MCTA is shown in Fig. 1.

```
 1 function dmc($\mathcal{M}$, $h$, $\varphi$):
 2     open = empty priority queue
 3     closed = $\emptyset$
 4     open.insert($s_0$, priority($h$, $s_0$))
 5     while open $\neq \emptyset$ do:
 6         $s$ = open.getMinimum()
 7         if $s \models \varphi$ then:
 8             generateErrorTrace($s$)
 9         closed = closed $\cup \{s\}$
10         for each transition $t$ of $\mathcal{M}$ that is applicable in $s$ do:
11             if $t[s] \notin$ closed then:
12                 open.insert($t[s]$, priority($h$, $t[s]$))
13     return True
```

Fig. 1. MCTA's basic directed model checking algorithm

For a given system $\mathcal{M}$, a distance heuristic h, and an error property φ (i. e., a negated invariant property), MCTA performs a reachability algorithm on the zone graph of $\mathcal{M}$. Therefore, MCTA maintains a priority queue *open* that contains encountered states for which the successor states have not yet been computed, and a *closed* list that contains the *explored* states, i. e., the states for which the successor states have already been computed. Starting with the initial state s_0, MCTA iteratively computes successor states and evaluates them with a *priority* function. For all encountered states, the priority function computes a *priority value* which is determined by the distance heuristic h and the applied search algorithm. According to the priority value, MCTA iteratively removes a best state s from *open* and checks if s is an error state (line 6–8). If this is the case, an error trace in generated by back-tracing from s (therefore, MCTA additionally stores information in the states about how they have been reached). If s is not an error state, s is stored in *closed*, and the successors of s are computed, evaluated and inserted into *open* if they are not already explored.

At this point, it is important to note that the algorithm in Fig. 1 should be read on a conceptual, rather than on an implementation level. For example, on an implementation level, the closed list is a special kind of hash table (rather than a set) that supports a certain inclusion test for states. We will come back to these points in Sec. 3, specifically for a discussion on MCTA's data structures, distance heuristics and search algorithms.

3 Mcta's Architecture and Features

In this section, we give an overview of MCTA's overall architecture and MCTA's features. Therefore, in Sec. 3.1, we present a high-level overview of the modules

Mcta consists of. In Sec. 3.2 and Sec. 3.3, we specifically describe Mcta's distance heuristics and search algorithms.

3.1 Mcta's Basic Architecture

Mcta consists of the modules *parser*, *system*, *search*, and *heuristics*. The input of Mcta consists of a file that contains a description of the timed automata system, and a file that contains the property to check. The property to check is a CTL formula of the form $\exists\Diamond\varphi$, where φ is a conjunction of constraints that speaks about automata and variables (i. e., φ describes the error states). Currently, Mcta supports a part of Uppaal's input language.

The *parser* module of Mcta uses Uppaal's timed automata parser library (UTAP), which is released under the LGPL and freely available at `http://www.uppaal.org/`. After parsing the input, Mcta generates an internal representation of the input system and the property. The corresponding algorithms and data structures to build this representation are part of the *system* module. The system representation is used by the *search* module which performs a search on the zone graph of this representation using a distance heuristic and a search algorithm (according to the algorithm given in Fig. 1). Mcta offers several kinds of distance heuristics and search algorithms (see Sec. 3.2 and Sec. 3.3 for an overview). In our setting, a distance heuristic is a function $h : \mathcal{S} \to \mathbb{N} \cup \{\infty\}$ that returns for each state of $\mathcal{S}$ an estimation of its error distance. The distance heuristics are implemented in the *heuristics* module. The overall architecture of Mcta is depicted in Fig. 2.

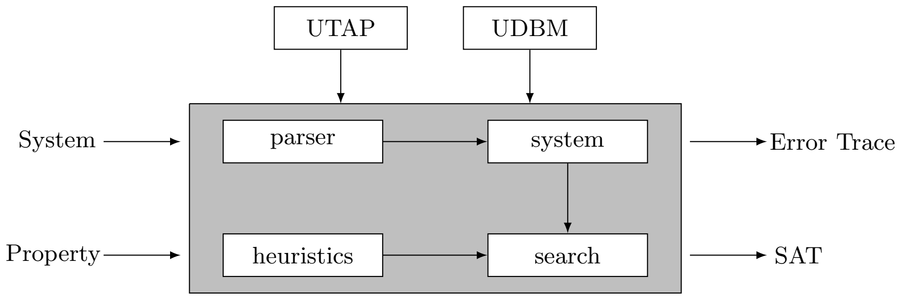


Fig. 2. Mcta's basic architecture

The *search* module is central to Mcta. It consists of the *search engine*, which implements the global while loop of Fig. 1, and uses dedicated data structures for *states*, for the *open* queue and for the *closed* list. For the internal representation of zones, Mcta uses Uppaal's difference bound matrices library (UDBM), which is released under the GPL and freely available at `http://www.uppaal.org/`. The *open* queue and the *closed* list are special kinds of hash tables. Overall, the design of the search engine is lightweight, which is supposed to simplify the implementation of new search algorithms. Furthermore, the interface to the heuristics module is intended to simplify the implementation of new distance heuristics.

3.2 The Heuristics Module

The heuristics module of MCTA-2012 features several distance heuristics to guide the search. To estimate the error distance of a state s, the distance heuristics compute an abstract error trace $\pi^{\#}$ that starts in an abstraction of s, and use the length of $\pi^{\#}$ as the estimation for the length of a concrete error trace from s. We give a short description of the different approaches in the following.

1. The d^U and d^L distance heuristics [8] are based on the local graph distances of the automata of the input system. Synchronization, integer variables and clock variables are ignored.
2. The h^L and h^U distance heuristics [17] are based on the *monotonicity abstraction*, which abstracts the original semantics of the system. The monotonicity abstraction assumes that variables are set-valued and, once they obtain a value, keep this value forever. The sets that contain the collected values grow monotonically over transition application, hence the name of the abstraction. The h^L and h^U distance heuristics compute abstract error traces based on this abstraction.
3. A pattern database heuristic based on downward pattern refinement [18]. We do not go into detail here but refer the reader to Sec. 4.

Compared to the earlier version MCTA-0.1, the h^U heuristic and the pattern database heuristic based on downward pattern refinement are new developments.

3.3 The Search Module

In this section, we focus on a description of MCTA's search algorithms that are the essential part of the *search* module. The search algorithms make use of the estimated error distances provided by the distance heuristic.

1. The standard greedy search algorithm [22] and A* search algorithm [11,12], including the uninformed search algorithms depth-first and breadth-first search.
2. The notion of *useless transitions* provides an approach to evaluate transitions (rather than just evaluating states) [26,25]. Transitions t are called *useless* in a state s if no shortest error trace starts in s with t. This criterion is approximated such that it can be computed efficiently. For this approach, the current version of MCTA maintains two open queues q_0 and q_1, where q_1 maintains states that are reached by a useless transition, and q_0 maintains the remaining states. The q_1 queue is accessed only if q_0 is empty.
3. Iterative-context bounding [20] is an approach that stems from the area of software model checking. In our setting, it corresponds to an iterative deepening search algorithm that prefers states that are reached with low number of *context switches*, i. e., with a low number of transition applications of different automata.

4. Context-enhanced directed model checking [24] is a further technique to additionally prioritize transitions during directed model checking. Similar to the iterative context bounding algorithm, it gives preference to states that are reached by a transition that interferes with previously applied transitions. In contrast, context switches are defined and exploited in a different way.

In comparison to Mcta-0.1, the implementation of the iterative context-bounding approach and the implementation of context-enhanced directed model checking are new developments. Both of these algorithms are based on multiple open queues. We will describe their implementation in Sec. 5.

4 Mcta's Pattern Database Heuristics

In this section, we describe Mcta's implementation for *pattern database (PDB) heuristics* in general, and the implementation of an extended version of downward pattern refinement in particular. We assume the reader is roughly familiar with pattern databases, and only give a short introduction. Pattern database heuristics are a class of admissible distance heuristics that come from the area of Artificial Intelligence [4,7]. For an input system $\mathcal{M}$ and a subset $\mathcal{P}$ of the system components of $\mathcal{M}$ (the so-called *pattern*), a pattern database PDB is a data structure that contains the abstract states of $\mathcal{M}|_{\mathcal{P}}$, where $\mathcal{M}|_{\mathcal{P}}$ denotes the projection abstraction of $\mathcal{M}$ that is obtained by abstracting away all systems components that are not contained in $\mathcal{P}$. Furthermore, for all abstract states in the PDB, the corresponding abstract error distance is stored. The PDB is computed once *prior* to directed model checking. *During* directed model checking, the PDB is used as a distance heuristic $h^{\mathcal{P}}$ by mapping every encountered concrete state s to a corresponding abstract state $s^{\#}$. The distance value $h^{\mathcal{P}}(s)$ of s is defined as the corresponding abstract error distance of $s^{\#}$.

4.1 Mcta's Architecture for Pattern Databases

Assuming that a pattern $\mathcal{P}$ is given, we present Mcta's framework for the computation of pattern databases (see Sec. 4.2 how Mcta computes a suitable pattern). Mcta performs three steps to compute the pattern database for $\mathcal{P}$: Abstracting the system, then computing the entire abstract state space $S^{\#}$, and finally, computing the abstract error distances for the abstract states in $S^{\#}$.

Abstracting the System. First, for the given input system $\mathcal{M}$ and the pattern $\mathcal{P}$, Mcta computes a *projection abstraction* of $\mathcal{M}$ based on $\mathcal{P}$ by abstracting away all system components that do not occur in $\mathcal{P}$. Therefore, Mcta applies the *abstractor* tool which also comes with the current Mcta-2012 release. The abstractor tool works as follows. For integer and clock variables v to be abstracted, the abstractor removes v from the guards and effects of the transitions of $\mathcal{M}$. If there is an edge with an effect such that the new value of v depends on a variable in $\mathcal{P}$, then v is abstracted away, too. Moreover, for an automaton $\mathcal{A}$ to be abstracted, the abstractor replaces $\mathcal{A}$ with a new automaton $\mathcal{A}'$ that only consists of one location. Moreover, $\mathcal{A}'$ contains loop edges for all edges of $\mathcal{A}$ where the guard is abstracted, but the effects and synchronization labels are kept. Doing so, we obtain an overapproximation $\mathcal{M}|_{\mathcal{P}}$ of the original system $\mathcal{M}$.

Computing the Abstract State Space. Second, for the obtained abstraction $\mathcal{M}|_{\mathcal{P}}$ of the original system $\mathcal{M}$, the entire reachable state space of $\mathcal{M}|_{\mathcal{P}}$ is computed in a forward manner and dumped into a file. For the traversal of the abstract state space, an extended version of the original search engine is used. This extended version specifically takes into account that if a state s that is already in the closed list is encountered again (i. e., on a different trace), the (new) transition that led to s is stored additionally. The abstract states and transitions are stored in a serialized form. Moreover, abstract error states are stored with a special error flag. Overall, we end up with a file that contains all the abstract states (where abstract error states are specifically indicated) together with all the abstract transitions.

Computing the Abstract Error Distances. Finally, based on the file that contains the abstract state space, we apply the external tool PDBGEN to generate the final pattern database. PDBGEN comes with the current MCTA-2012 release and computes the abstract error distances for a given abstract state space. This is done in a backwards manner via a version of Dijkstra's algorithm. More precisely, PDBGEN starts by assigning the error distance zero to all the abstract error states, and by assigning infinity to all the other states. In the following, PDBGEN iteratively checks the predecessor states and updates the distance value if it is reached more cheaply than before. The output is a file that contains the serialized abstract states together with their abstract error distances. This is the pattern database which can be fed into MCTA to be used as a distance heuristic. We finally remark that, doing this 3-step process to compute the PDB, we avoid the expensive regression operation on the zone graph.

4.2 Running Mcta with Extended Downward Pattern Refinement

To compute the pattern, MCTA uses the external tool MCTA-PDB that is implemented in Python. MCTA-PDB acts as a wrapper around the pattern generator PDBGEN and MCTA. More precisely, MCTA-PDB first generates the underlying pattern. The pattern is generated with an algorithm based on downward pattern refinement [18]. In addition to the originally proposed h^{dpr} distance heuristic, MCTA-2012 applies explicit search in intermediate abstractions to deal with clock variables more explicitly. Doing so leads to a more-fine grained approach to select clocks than proposed in the original paper (where *all* clock variables have been selected for the pattern by default). For the resulting pattern, the above described 3-step process is performed. Finally, MCTA-PDB calls MCTA with the resulting pattern database. To apply MCTA with the pattern database heuristic based on downward pattern refinement, select the following command line parameters.

```
mcta-pdb --dprc --astar SYSTEM PROPERTY
```

5 Multi-Queue Search Algorithms

In this section, we describe Mcta's implementation for *multi-queue search algorithms*. After giving a brief conceptual description in Sec. 5.1, we present Mcta's general framework for this approach in Sec. 5.2. In the subsequent sections, we specifically describe the implementation of two multi-queue search algorithms from the literature, namely iterative context bounding [20] in Sec. 5.3, and context-enhanced directed model checking [24] in Sec. 5.4.

5.1 The General Approach

In the setting where not only a distance heuristic, but also an additional quality measure to guide the search is available, there is the question of how to exploit this additional information. In such cases, a popular approach is to maintain multiple open queues instead of only one. Within this approach, states are pushed into different open queues according to the additional quality measure, and ordered in this queue according to the original distance heuristic. The "best" state to explore next is then defined as the "best" state according to the distance heuristic in the "best" open queue according to the additional quality measure. For example, multi-queue approaches have been successfully applied in the area of AI planning for *combining* distance heuristics [23] (i. e., in this case, the additional quality measure is another distance heuristic), or for additionally evaluating *transitions* rather than only evaluating states [14,13,25]. Furthermore, in the area of model checking, similar approaches have been proposed to evaluate transitions based on iterative context bounding [20], interference contexts [24], and the notion of *useless* transitions [26].

For the rest of this section, we assume a setting where a distance heuristic (to evaluate states) *and* a technique to evaluate transitions is available. The idea is to exploit this additional information by preferably exploring states that are estimated to be near to an error state (which corresponds to low distance values as before) *and* that are reached by a transition that is estimated to guide the search properly towards an error state. More precisely, the evaluation of transitions determines the open queue in which the resulting successor state is maintained, and (as in the classical approach) the distance heuristic determines the ordering of the states in the queues. Formally, the priority function from the algorithm in Fig. 1 becomes a function with domain $\mathbb{N} \times \mathbb{N}$, i. e., it does no longer only assign a natural number to states s, but additionally a natural number for the transition that led to s to determine in which open queue s is maintained.

5.2 Mcta's Architecture for Multi-Queue Search Algorithms

The high-level architecture of Mcta to maintain several open queues is best described by the following template functions. They show how an extended open queue that internally consists of multiple open queues is accessed to get and insert states. The algorithmic design is rather straight forward and depicted in Fig. 3.

```
1 function insert($s$, $h$):
2     $k$ = evaluate predecessor transition $t$ of $s$
3     $q_k$.insert($s$, $h(s)$)

1 function getMinimum():
2     determine open queue $q_k$ to access
3     return $q_k$.getMinimum()
```

Fig. 3. Multi-queue accessing functions

In the above algorithm, we assume that an upper bound on the number of queues can be computed (see Sec. 5.3 and Sec. 5.4 how this is done for the individual approaches). In contrast to the classical approach in Fig. 1, the *insert* function computes a natural number k to determine the quality of the transition that led to the state s that is inserted. This number in turn determines the index of the queue in which s is inserted. Furthermore, *getMinimum()* returns the best state of the open queue that is accessed next; note that it depends on the applied search algorithm *how* this queue is actually determined.

5.3 Implementation of Iterative Context Bounding

Iterative context bounding (ICB) has been proposed for the purpose of testing multithreaded programs [20]. Roughly speaking, ICB performs an iterative deepening search with the objective to minimize the number of *context switches*, i. e., the number of execution points on a trace where the scheduler forces the active thread to change.

MCTA implements this approach by considering threads as automata. Therefore, a context switch occurs if two consecutive transitions on a trace belong to two different automata. ICB can be combined with arbitrary distance heuristics as well as uninformed search (where the latter corresponds to the original approach). MCTA applies a special search engine that maintains two open queues q_0 and q_1 (i. e., $k \in \{0, 1\}$ in Fig. 3). The *insert* function is implemented as follows. For a state s and an applicable transition t in s, the successor state is inserted into q_0 if the edge(s) of t and the edge(s) of the predecessor transition of s belong to the same automata. The *getMinimum()* function is implemented by always accessing q_0 until q_0 gets empty. If this is the case, then q_0 and q_1 are exchanged, reflecting that the number of context switches has been increased by one. In case q_1 is empty, too, we report that no error state is reachable.

Running Mcta with Iterative Context Bounding. To run MCTA with iterative context bounding, use the `--icb=1` flag when MCTA is called. We remark that MCTA supports additional options to define a context, but we do not go into detail here. A short description of these parameters is given when MCTA is called with the `--help` option.

5.4 Implementation of Context-Enhanced Directed Model Checking

Context-enhanced directed model checking is a further multi-queue search approach [24]. In contrast to iterative context-bounding, contexts are essentially defined based on *interference* of transitions, where transitions t and t' *interfere* if t writes a variable that is read by t', or t' writes a variable that is read by t, or t and t' write a common variable. Moreover, during the search, more than two open queues are maintained in general. The approach is based on preferably exploring states that are reached by transitions that interfere with previously applied transitions. More precisely, states are preferably explored if they are reached with a transition with low *interference distance* to the previously applied transition, where the interference distance of t and t' is defined as the smallest $k \in \mathbb{N}$, $k \geq 1$, such that there are transitions $t_1, \ldots, t_k$ with the property that t interferes with t_1, t_1 interferes with t_2, ..., and t_k interferes with t'.

In Mcta, context-enhanced directed model checking is implemented as follows. First, for every transition t in the system, the interference distance of t is computed to every other transition in the system. This is a all pairs-shortest-path problem for which we apply the *Floyd Warshall* algorithm [10]. The resulting interference distances are stored in a 2-dimensional vector. The maximal interference distance N for two transitions defines the number of open queues (obviously, N is defined because systems have a finite number of transitions). More precisely, we maintain a global open queue Q that consists of a vector of open queues $q_0, \ldots, q_N$. The *insert* function in Fig. 3 is defined as follows. For a state s' that is supposed to be inserted into Q, Mcta determines the interference distance k of the predecessor transition t' of s' and the predecessor transition of the predecessor state of s', where s' is inserted into queue q_k. In the special case that the effect of t' satisfies a constraint of the property that is subject to model checking, s' is inserted in q_0. The *getMinimum()* function determines the smallest non-empty queue in Q to get the next state.

Running Mcta with Iterative Context Bounding. To run Mcta with the context-enhanced directed model checking approach, use the `--ce` flag when Mcta is called. We remark that Mcta supports additional options for this setting, but we do not go into detail here. A short description of these parameters is given when Mcta is called with the `--help` option.

6 Mcta's Performance

In this section, we present an experimental evaluation of Mcta-2012 on large and challenging real-time benchmarks. Specifically, some of these benchmarks stem from industrial real-time case studies. To evaluate the performance of Mcta-2012, a bug has been inserted in all of them.

The case study "Single-Tracked Line Segment" [15] (the problem instances $C_1, \ldots, C_9$ and $D_1, \ldots, D_9$) models a distributed real-time controller for a segments of tracks where trams share a piece of track. The distributed controller is supposed to ensure that never two trams that drive in opposite directions are simultaneously given permission to enter the shared piece of track. The controller was modeled in terms of PLC automata [6], which is an automata-like notation for real-time programs. With the tool Moby/RT [21], the PLC automata system has been transformed into abstractions of its semantics in terms of timed automata. For the evaluation of Mcta-2012, we chose the property that never both directions are given permission to enter the shared segment simultaneously.

As a further set of benchmarks, we used a case study called "Mutual Exclusion" (problem instances $M_1, \ldots, M_4$ and $N_1, \ldots, N_4$). As suggested by the name, in this case study, mutual exclusion has to be established for real-time systems. It is based on a protocol that is described by Dierks [5]. We refer the reader to the website of Mcta for a more detailed description. All of these benchmarks can also be obtained from the Mcta website.

The experiments have been performed on an AMD Opteron Processor 6174 with 2.2 GHz system and 4 GByte of memory. We compare Mcta-2012 in an optimal search setting with the best technique described in the last tool paper [19] (corresponding to Mcta-0.1). We also provide results for the tools Uppaal-4.0.13 and Uppaal/Dmc [16]. Uppaal provides an efficient implementation of breadth-first search, whereas the other tools apply the directed model checking approach. Note that in this paper, we do not compare to Uppaal's randomized depth first search (rdfs) because rdfs is not guaranteed to find *shortest* possible error traces (see the earlier tool paper of Mcta [19] for a comparison of suboptimal search techniques including Uppaal's rdfs). We used the options that lead to the best experimental results for each tool. In particular, for Mcta-2012, we used extended downward pattern refinement as described in Sec. 4.2. The results are given in Table 2. For Mcta-12 and Uppaal/Dmc, the best search options to find shortest error traces are PDB approaches; therefore, for these tools, the pure search time in the concrete state space is reported additionally.

The results clearly indicate that Mcta-2012 mostly outperforms the other directed model checking tools on these problems. Moreover, we observe that these problems are large and complex because even Uppaal, which provides a very efficient implementation of breadth-first search, cannot solve all of them. Furthermore, we observe that the preprocessing of Mcta-2012 often takes most of the overall model checking time. However, the preprocessing time mostly pays off, specifically compared to the uninformed search provided by Uppaal, but also compared to the other directed model checking tools. For the M instances, the pure search time of Mcta-2012 is still comparable to the search time of most of the other tools. Moreover, in these instances, we observe that the overall number of explored states as well as the number of explored states per second is lower for Uppaal than for Mcta-2012. Although we do not know the exact reason, we suppose that this is the case because Uppaal uses a more efficient representation of the zone graph. We finally remark that we have also successfully *verified* correct systems with Mcta-2012. This is possible because admissible heuristics h can be used as a pruning method: If $h(s) = \infty$ for a state s, then the real error distance of s is infinity as well, and hence, s can safely be pruned (recall

Table 2. Results with the A* search algorithm. Abbreviations: "Mcta-12": Mcta-2012, "Mcta-08": Mcta-0.1, "U/dmc": Uppaal/Dmc, "runtime": overall runtime including any preprocessing in seconds, "explored states": number of explored concrete states, dashes indicate out of memory (> 4 GByte). For Mcta-12 and U/dmc that rely on PDBs, the pure search time in the concrete (i. e., time without preprocessing) is reported in parenthesis.

	runtime in seconds				explored states				trace
Inst.	Mcta-12	Mcta-08	U/dmc	Uppaal	Mcta-12	Mcta-08	U/dmc	Uppaal	length
M_1	2.2 (0.3)	0.6	3.0 (0.2)	0.5	29029	41455	190	14290	47
M_2	2.9 (0.9)	2.6	3.2 (0.2)	2.1	99528	164856	4417	51485	50
M_3	3.7 (1.7)	3.0	3.4 (0.4)	2.2	165336	189820	11006	52987	50
M_4	8.2 (6.2)	13.5	4.0 (1.0)	8.8	549999	724030	41359	186435	53
N_1	2.6 (0.1)	2.7	18.0 (0.4)	3.8	3606	93951	345	28196	49
N_2	3.1 (0.6)	14.7	12.1 (0.5)	17.1	26791	438394	3811	100078	52
N_3	4.2 (1.7)	19.1	14.7 (4.5)	17.5	70439	547174	59062	102124	52
N_4	13.0 (10.4)	95.3	34.3 (27.8)	76.4	388076	2317206	341928	370459	55
C_1	1.3 (0.1)	0.2	0.8 (0.1)	0.2	98	12458	130	21008	54
C_2	1.4 (0.1)	0.7	1.1 (0.7)	0.5	98	32751	89813	55544	54
C_3	1.4 (0.1)	0.8	0.8 (0.0)	0.6	98	37126	197	74791	54
C_4	1.4 (0.1)	7.5	0.9 (0.1)	6.0	312	301818	1140	553265	55
C_5	1.5 (0.1)	60.9	1.0 (0.1)	53.1	1178	2174789	7530	3977279	56
C_6	1.5 (0.1)	605.6	1.1 (0.3)	514.3	2619	20551913	39436	33526538	56
C_7	1.6 (0.1)	–	1.7 (0.8)	–	4247	–	149993	–	56
C_8	1.6 (0.2)	–	1.7 (0.9)	–	5416	–	158361	–	56
C_9	1.7 (0.2)	–	1.7 (0.8)	–	13675	–	127895	–	57
D_1	10.2 (0.3)	81.2	84.7 (65.0)	90.5	2789	1443874	4610240	4048866	78
D_2	12.2 (0.4)	433.4	255.3 (5.4)	539.0	5086	6931937	4223	21478364	79
D_3	12.3 (0.4)	487.0	255.6 (5.4)	548.4	5161	7900038	2993	21553760	79
D_4	13.9 (0.3)	288.0	256.7 (5.4)	476.4	1023	4660652	2031	18487819	79
D_5	60.1 (6.4)	–	–	–	122204	–	–	–	102
D_6	66.4 (10.5)	–	–	–	426571	–	–	–	103
D_7	67.1 (7.9)	–	–	–	180132	–	–	–	104
D_8	68.3 (6.2)	–	–	–	28285	–	–	–	104
D_9	71.4 (6.3)	–	–	–	12186	–	–	–	105

that admissible heuristics never overestimate the real error distance). For more details, including experimental results, we refer the reader to the literature [18].

7 Conclusion

In this paper, we have reviewed Mcta and its new developments from an implementation point of view. The new developments include heuristics and search techniques for both optimal and suboptimal search. We have observed that Mcta-2012 is very useful in efficiently finding shortest possible error traces in faulty systems. For the future, we specifically aim at developing new admissible distance heuristics. A main issue for research will be to effectively find the sweet spot of the trade-off to be as accurate as possible on the one hand, and as cheap to compute as possible on the other hand.

Acknowledgments Parts of this work have been done while the authors have been employed by the University of Freiburg, Germany. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Behrmann, G., Bengtsson, J.E., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL Implementation Secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Culberson, J.C., Schaeffer, J.: Pattern databases. Comp. Int. 14(3), 318–334 (1998)
5. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. Formal Aspects of Computing 16(2), 104–120 (2004)
6. Dierks, H.: Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Habilitation thesis, University of Oldenburg, Germany (2005)
7. Edelkamp, S.: Planning with pattern databases. In: Proc. ECP, pp. 13–24 (2001)
8. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2), 247–267 (2004)
9. Edelkamp, S., Schuppan, V., Bošnački, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on Directed Model Checking. In: Peled, D.A., Wooldridge, M.J. (eds.) MoChArt 2008. LNCS, vol. 5348, pp. 65–89. Springer, Heidelberg (2009)
10. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM 5(6), 345 (1962)
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Systems Science and Cybernetics 4(2), 100–107 (1968)
12. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to a formal basis for the heuristic determination of minimum cost paths. SIGART Newsletter 37, 28–29 (1972)
13. Helmert, M.: The Fast Downward planning system. JAIR 26, 191–246 (2006)
14. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. JAIR 14, 253–302 (2001)
15. Krieg-Brückner, B., Peleska, J., Olderog, E.R., Baer, A.: The UniForM workbench, a universal development environment for formal methods. In: Proc. MF, pp. 1186–1205. Springer (1999)
16. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: Uppaal/DMC – Abstraction-Based Heuristics for Directed Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)
17. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI Planning Heuristic for Directed Model Checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)

18. Kupferschmid, S., Wehrle, M.: Abstractions and Pattern Databases: The Quest for Succinctness and Accuracy. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 276–290. Springer, Heidelberg (2011)
19. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster Than Uppaal? In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 552–555. Springer, Heidelberg (2008)
20. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proc. PLDI, pp. 446–455. ACM Press (2007)
21. Olderog, E.R., Dierks, H.: Moby/RT: A tool for specification and verification of real-time systems. J. UCS 9(2), 88–105 (2003)
22. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley (1984)
23. Röger, G., Helmert, M.: The more, the merrier: Combining heuristic estimators for satisficing planning. In: Proc. ICAPS, pp. 246–249. AAAI Press (2010)
24. Wehrle, M., Kupferschmid, S.: Context-Enhanced Directed Model Checking. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 88–105. Springer, Heidelberg (2010)
25. Wehrle, M., Kupferschmid, S., Podelski, A.: Useless actions are useful. In: Proc. ICAPS, pp. 388–395. AAAI Press (2008)
26. Wehrle, M., Kupferschmid, S., Podelski, A.: Transition-Based Directed Model Checking. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 186–200. Springer, Heidelberg (2009)

Author Index

Akshay, S. 11
Asarin, Eugene 27

Basset, Nicolas 27
Basten, Twan 1
Béal, Marie-Pierre 27
Bertrand, Nathalie 43

Carter, Rebekah 59
Chilton, Chris 75

Dalsgaard, Andreas E. 91
Degorre, Aldric 27
Dimitrova, Rayna 107

Ellen, Christian 123

Feo-Arenis, Sergio 155
Finkbeiner, Bernd 107, 204
Fränzle, Martin 123

Gerwinn, Sebastian 123
Ghasemieh, Hamed 139
Gribaudo, Marco 139

Haverkort, Boudewijn 139
Hélouët, Loïc 11
Hendriks, Martijn 1
Herrera, Christian 155

Jard, Claude 11
Junttila, Tommi 171

Kindermann, Roland 171
Kupferschmid, Sebastian 252
Kwiatkowska, Marta 75

Laarman, Alfons 91
Larsen, Kim G. 7, 91
Lime, Didier 11

Muñiz, Marco 155, 188

Navarro-López, Eva M. 59
Niemelä, Ilkka 171

Olesen, Mads Chr. 91

Perrin, Dominique 27
Peter, Hans-Jörg 204
Podelski, Andreas 155, 188

Remke, Anne 139
Rinast, Jonas 220
Roux, Olivier H. 11

Schewe, Sven 43
Schupp, Sibylle 220
Somers, Lou 1
Stainer, Amélie 236

Trčka, Nikola 1

van de Pol, Jaco 91

Wang, Xu 75
Wehrle, Martin 252
Westphal, Bernd 155, 188