EFFICIENT ANALYSIS AND SYNTHESIS OF COMPLEX QUANTITATIVE SYSTEMS

BY
PETER GJØL JENSEN

DISSERTATION SUBMITTED 2018

AALBORG UNIVERSITY
DENMARK

Efficient Analysis and Synthesis of Complex Quantitative Systems

Ph.D. Dissertation

Peter Gjøl Jensen

Dissertation submitted March, 2018

Dissertation submitted:	March, 2018
PhD supervisor:	Prof. Kim Guldstrand Larsen Aalborg University
Assistant PhD supervisor:	Assoc. Prof. Jiˇrí Srba Aalborg University
PhD committee:	Associate Professor Bent Thomsen (chairman) Aalborg University Professor, Dr. Jaco van de Pol University of Twente Professor Saddek Bensalem Université Grenoble Alpes
PhD Series:	Technical Faculty of IT and Design, Aalborg University
Department:	Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-178-1

Published by:
Aalborg University Press
Langagervej 2
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2018

Abstract

From toasters and to space-stations, computerized technology is pervasive in modern technology and society, therefore the need for truly correct, safe and optimal control algorithms is higher than ever. Techniques like model checking and synthesis have long promised, and to some extend delivered, correctness and optimality guarantees in limited and highly critical application areas like software for satellites, medical devices or powerplants. Common for many of the application areas is the criticality of timing; airbags, pacemakers and traffic-lights have timing constraints that should never be violated.

In this thesis, we attempt to improve the applicability of model checking and synthesis methods for timed systems by attacking three different inhibiting factors to their applicability; 1) speed of computation, 2) what can be synthesized and 3) tool integration and interaction.

To improve on the speed of computation, we attack what is called the state-space explosion problem and present alternatives for the state-space representation. We attack this problem by developing novel algorithms and datastructures for the reduction of memory and time consumption.

To increase the applicability of synthesis, we present a semi-algorithm for parameter synthesis for Timed Automata, extendable to more expressive formalisms. We also demonstrate an over/under-approximate technique for the synthesis of Metric Interval Temporal Logic specifications and show the methods feasibility on a series of examples. As a final contribution to the topic, we present a tool which encompass both ideas from the formal methods community and the machine intelligence community, providing both safe and optimal control synthesis.

In the topic of tool integration, we extend the tool UPPAAL to facilitate interoperability with other tools. We show that integration between UPPAAL and a plethora of other tools is possible via the Function Mockup Interface standard and demonstrate that UPPAAL can be used as the driving tool for a so-called co-simulation. We also present a case-study using externally defined components, such as an ARM-processor emulator, in a classical model checking context.

Resumé

Fra toastere til rumstationer har computerbaseret teknologi gennemsyret det moderne samfund, og derfor er behovet for sikre, korrekte og optimale styringsmekanismer højere end nogensinde. Teknikker, såsom modeltestning og syntese, har længe lovet, og til en hvis grad leveret korrektheds og optimalitets garantier i et begrænset omfang. Teknikkerne anvendes dog oftest kun for højkritiske domæner såsom satellitsoftware, kliniske instrumenter og kraftværk. Fællestrækket for mange af applikationsområderne er det tidskritiske aspekt. Airbags, pacemakers og trafiklys har alle tidsspecifikke krav, som aldrig bør overskrides.

I denne afhandling forsøger vi at forbedre anvendeligheden af modeltestning og syntese. Vi angriber tre forskellige begrænsende faktorer for metodernes anvendelighed: 1) Hastigheden af beregning, 2) hvad der kan syntetiseres og 3) værktøjsintegration og interaktion.

For at reducerer beregningstiden angriber vi det såkaldte tilstandsrumseksplosionproblem og præsenterer alternativer til tilstandsrummets repræsentation. Vi forsøger at tackle problemet ved at udvikle nye algoritmer der kan reducerer både beregningstid og hukommelsesforbrug.

For at forbedre anvendeligheden af syntese, præsenterer vi en semialgoritme til parameter syntese for tidsautomater, der kan udvides til formalismer med større udtrykskraft. Vi demonstrerer også en over/under approksimations teknik for syntese af Metrisk Interval Temporal Logik-speficikationer og viser, at teknikken er praktisk anvendelig på en række eksempler. Som et sidste bidrag til emnet præsenterer vi et værktøj, der omfavner ideer fra både det formelle verifikations miljø og maskinintelligens miljøet for derved at kunne syntetisere sikre og optimale styringsalgoritmer.

Indenfor værktøjsintegrationsemnet udvider vi værktøjet Uppaal således, at det kan samarbejde med andre værktøjer. Vi viser, at integration mellem Uppaal og et væld af andre værktøjer er mulig via Function Mockup Interface standarden og demonstrerer at Uppaal kan bruges som det drivende værktøj i en såkaldt co-simulering. Endvidere præsenterer vi et casestudie, der bruger externt definerede komponenter, såsom en ARM-processor emulator i en klassisk model testnings kontekst.

Contents

Preface

Behind a thesis as this, more people than just the author contribute either directly or indirectly. Several people deserve my gratitude for helping me in its creation, more than this preface gives room for.

I would like to thank Kim G. Larsen and Jiří Srba for providing me with much appreciated supervision and freedom to explore ideas of my own.

My six months in Sydney would not have been nearly as fruitful and enjoyable without the supervision and hospitality of Franck Cassez and the rest of the lunch gang of the Department of Computing at Macquarie University.

My colleagues Axel, Danny, Erik, Jakob, Mads, Pablo, Rene, Scott and Ulrik; thank you for listening to my complaints, arguments and stupid ideas. I am sure my time as a PhD-student would have been a lonesome, had it not been for you.

The path to a PhD is paved with many impressions over the years, and Alexandre David deserves thanks for inspiring me to do high performance software.

To Kim Kristensen, my early math and science teacher, I attribute part of my achievement. He introduced me to the wonders of natural science, a fascination I have kept ever since.

For many inspiring discussions around a campfire, the people of Shelterforeningen deserves thanks.

My reserve family of Gandrup, the clan of Scheel Nellemann, you have housed, wined and dined me on several occasions, for this I am grateful.

Lene Scheel Jensen and Jens Christian Gjøl Jensen, my dear parents, thank you for all the support, motivation and interest through the years. I believe that my passion for my craftsmanship, my ability critical reasoning and my work ethics are skills inherited from you, traits I admire in both of you, and virtues I will reap the fruits of for many years to come.

Lastly, Yasmin, thank you for providing me with never-ending interesting discussions, inspiration and motivation; you truly are my muse.

Peter Gjøl Jensen
Aalborg University, March 19, 2018

Part I

Introduction

Computers have become an ever-present part of ours lives and are now deeply integrated into our society. However, most computer systems are not of the kind with a screen for direct human interaction, but rather an *embedded* system controlling, supporting or communicating seamlessly with the world on our behalf. Examples of such embedded systems range from those found in modern hearing aid and dishwashers to those controlling highly *critical systems* such as nuclear power plants, trains and satellites.

Such systems need software, and in particular for critical systems, correct and safe software, in the sense that desired behavior is guaranteed without violating invariant properties. In classical software development freedom from bugs and safety requirements are validated via testing. As the correctness of both the development process and the testing phase are heavily dependent on human creativity and discipline, either phase is prone to errors.

To ensure correctness of software via *verification*, techniques such as *model checking* [49, 63, 118] have been proposed. The model checking approach accepts as input a *model* and a *specification* and can then verify, with certainty, whether or not the model satisfies the specification. It is important to note here that a model is a general term and ranges over both an exact model (for instance the program itself) and very abstract models (for instance the specification of a communication protocol).

Another approach to reduce human involvement in constructing control software is that of *synthesis* [115, 116]. As model checking, synthesis is modelcentric—however instead of checking that the specification is satisfied, the aim is to automatically construct a model that fulfills the specification. A closely related, non-constructive, sister-problem to synthesis is that of *satisfiability*. The satisfiability problem asks whether a hypothetical model can exist for which the specification is satisfied, not requiring the actual model to be constructed. Many techniques have been proposed for synthesis in various domains, among others are methods from algorithmic game theory, symbolic methods and explicit methods for reactive synthesis, parameter synthesis and machine learning.

Synthesis promises more than just correctness. Using synthesis it is also possible to prove that the system is correct even with changes to the initial parameters, or even to modify the controller in a way such that a given criteria is improved upon. This has numerous practical applications such as improving traffic-flow in a city or reducing the energy consumption of a plastic molding machine.

Common for all of the methods mentioned above is the notion of a *model*. For the correct verification and synthesis, the provided model has to include the behavior of the environment under control. While systems for light-switches are not concerned with time, physics or probabilities, one can think of many contexts in which time (airbags), physics (satellite-control) or probabilities (traffic-control) are important to correctly model the world around the

Players	Models	Methods	Tools
$\frac{1}{2}$-player	Stochastic Timed Automata [4]	Value Iteration [70, 74, 108]	Prism [97]
	Markov Chain	Statistical Model Checking [124]	Uppaal SMC [35, 36]
	Discrete Time Markov Chains [53]	Numerical methods [121]	
	Continous Time Markov Chains [12]	Monte Carlo [94, 104]	
1-player	Timed Automata (TA) [6, 7]	Difference Bound Matrices [2]	Uppaal [14]
	Priced Timed Automata (PTA) [11, 17]	Time Darts [89, 91]	Tappaal [37, 54]
	Timed Arc Petri Nets (TAPN) [75]	PTries [89]	LTSmin [22]
	Time Petri Nets [111]	Polyhedra [81]	DIVINE [13]
	Stopwatch Timed Automata [42]	Trace Abstraction Refinement [41]	Kronos [26]
	Hybrid Automata [80]	IC3 [28]	SpaceEx [73]
		CEGAR [50]	Tina [21]
			Romeo [107]
$1\frac{1}{2}$-player	Probabilistic Timed Automata [84, 98]	Value Iteration [70, 74, 108]	Prism [97]
	Markov Decision Process	Statistical Model Checking [35, 79, 124]	Fortuna [19]
	Duration Probabilistic Automata [117]	Regions [84, 98]	Modest [77]
	Priced Timed Markov Decision Processes [56]	Monte Carlo [94, 104]	Uppaal Stratego [55]
	Priced Probabilistic Time Automata [20]		
2-player	Timed Games [110]	On-the-fly algorithm [39]	Uppaal Tiga [16]
	Priced Timed Games [24]		Uppaal Stratego [55]
$2\frac{1}{2}$-games	Simple Stochastic Game [43, 44, 52]	Value itteration [46]	Prism [45]
$n\frac{1}{2}$-games	Stochastic Multi-player Games [45]	Reduction to $2\frac{1}{2}$ player games	Prism [45]

Table 1: A non-comprehensive overview of the different models, methods and tools used in with different number of players.

software. To capture these features, various *formalisms* have been proposed for modeling, each with their own trade-offs in terms of computability, applicable methods and complexity. A similar story can be told for the side of specifications. Let us now review the current state-of-the-art formalisms, methods and tools.

1 State of the Art

1.1 Models

In this thesis we will primarily focus on timed models, in this review of the models, we will thus restrict our attention to such systems. In the field of purely stochastic models, which we here call $\frac{1}{2}$-player games, Stochastic, Timed Automata [4] resides along with the timed derivatives of Markov Chains; Discrete Time Markov Chains [53] and Continuous Time Markov Chains [12].

As such models include no notion of choice, the properties one can inspect are limited to stochastic performance measures such as “What is the expected outcome of a fair dice throw?” or “How likely is it that a specific bug will be triggered in my control system?”.

If we consider 1-player games, replacing stochastics with non-deterministic choice, we obtain models such as Timed Automata (TA) [6, 7], Priced Timed Automata (PTA) [11, 17], Timed Arc Petri Nets (TAPN) [119] and Time Petri Nets (TPN) [111].

Here more interesting questions arise as the non-determinism can be interpreted in two settings; either antagonistic or protagonistic. In the antagonistic case, we can ask questions like "Can this error-state be reached?" or "Can my system livelock/deadlock?". In the protagonistic setting, we might ask to find "A Schedule satisfying the constraints of the teaching staff" or "A controller such that our robot does not tip over flowerpots". Here it is important to realize that such a constructed controller assumes that it is in total control of the environment.

Adding a stochastic adversary to the model brings us to the $1\frac{1}{2}$-player games. Here, the controlling player is able to nondeterministically choose its actions, while the adversary will have a stochastic behavior. This gives ground to $1\frac{1}{2}$-player games modeled as Probabilistic Timed Automata [84, 98], Priced Probabilistic Timed Automata, Markov Decision Process, Duration Probabilistic Automata [117] and Priced Timed Markov Decision Processes [56]. These will, given a strategy for the controller, become $\frac{1}{2}$-player games.

For the case of reactive systems, adding such a stochastic environment opens the methods up to more realistic scenarios, for instance traffic control-scenarios where one could ask the following "Assuming the cars behave with these probabilities, how do I control the traffic lights such that we most likely reduce the queuing time?" or "If I only have stochastic information on the duration of tasks, how do I plan them so that the throughput is maximized?"

In the case of 2-player games, both players are assumed to control each their set of non-deterministic actions. Here both players can have mutually exclusive goals. The synthesis problem was shown to be decidable for the class of Timed Games [110], while undecidable in general for Priced Timed Games [30].

Here we wish to synthesize a controller such that given requirements are fulfilled no matter the actions of the antagonistic opponent. Among interesting applications is the synthesis of a controller that ensures safety for an adaptive cruise-control setting [102] or to synthesize a controller ensuring a habitable environment for a pig stable [90].

The addition of a stochastic nature to the 2 players gives us $2\frac{1}{2}$ player games, where both the antagonist and the controller need to address the stochasticity in the game. For the formalism of Simple Stochastic Games [52], work has been done to obtain complexity results and investigate expressiveness of different logics [43, 44, 52]. To the best of our knowledge no similar work can be found regarding a timed extension of 2-$\frac{1}{2}$ player games.

Formalisms for n and n-$\frac{1}{2}$ player games exist and distributed controller synthesis can be seen as a special case of such a game [109]. Some of these can be reduced to, or approximated by, 1-$\frac{1}{2}$, 2 or 2-$\frac{1}{2}$ player games [45], for example using an Alternating Temporal Logic (ATL) [9], but in general we shall consider games for $n > 2$ out of the scope of this thesis.

1.2 Tools

We will here provide a short survey of tools closely related to the formalisms and logics presented in this thesis. For evaluating the purely stochastic models, tools such as Prism [97] have evolved. In the model-cheking setting of 1-player games, Uppaal [14], Tappaal [37, 54], Tina [21] and Romeo [107] exist for each their formalism, providing verification of propositions in a subset of TCTL. Games containing a stochastic adversary can have strategies synthesized by either Prism [45] or Uppaal Stratego [55, 56]. For the class of TGs Uppaal offers the Uppaal Tiga [16] extension, capable of synthesizing strategies with hard constraints. The combination of Uppaal Stratego for $1\frac{1}{2}$-player games and Uppaal Tiga for 2-player games internally in Uppaal Stratego, provides the capability of both synthesizing strategies with hard-guarantees as well as near-optimal schedulers respecting these guarantees. This was originally proposed by Bruyére et. al [32].

Prism supports n-$\frac{1}{2}$ by reduction to 2-$\frac{1}{2}$ player games [45] for, among others, ω-regular objectives using an ATL-like syntax [9].

1.3 Parameterized Models

The above mentioned models have all parameters settled a priori. An interesting family of models arise when some initial parameters are left unspecified. This leads (in the 1-player setting) to the verification-question: does there exist an initial set of parameters s.t. our model satisfies the desired specification. A special case of this question is the robustness problem for Timed Automata; is the given specification sensitive to small changes in the constants of the model [25]? However, it also gives rise to the more interesting question of parameter synthesis: find the set of parameters for which the property is satisfied (or for which the system is robust). This problem was formalized and solved for hybrid systems using the tool HyTech [82]. In recent years novel methods have been proposed, utilizing more complex strategies, such as Trace Abstraction Refinement [41], the Inverse Method [66] or for special cases such as integer parameters [92]. This notion of parametric verification and synthesis can be extended across all the above-mentioned player-settings and formalisms—however, in many cases, introducing open parameters into a model renders analysis-questions undecidable, for instance for Timed Automata [10].

1.4 Logics

Logics are used to define high-level specifications for models which are then used to either validate a model using model checking, or construct a model using synthesis. Two simple and classical logical specifications are the *safety* and *reachability* objectives. Here the goal is to either avoid an error, or to reach a desired configuration (often subjected to other requirements such as "within a given time-period" or "before you run out of power" borrowed from more expressive logics). The area of behavioral logics is heavily studied, and conventionally one considers variants one of two families of logics; the trace-based *Linear Temporal Logic* (LTL) [114] or the branching-based *Computational Tree Logic* (CTL) [48]—both sub-sets of the CTL* logic [64]. However, all three logic are strictly less expressive than μ-Calculus [96] describing a simple framework for reasoning on computational systems.

Different logics have also been developed quantifying over different aspects of the model. For probabilistic systems CTL has been extended to quantify over probabilistic measures, called Probabilistic CTL [76] and this notion has been generalized into the Weighted CTL (WCTL) [29] logic. In the setting of Timed Systems LTL has been generalized to Metric Temporal Logic (MTL) [95] while CTL have been generalized to Timed CTL [3].

1.5 Methods

Different techniques have been deployed to make the verification, synthesis or probabilistic computation faster or more precise. In the probabilistic setting, PRISM deploys value-iteration-algorithms [46, 74, 108], converging to the exact result. A different method is taken by UPPAAL SMC [35, 36], which utilizes a Monte-Carlo based method for sampling and probabilistic approximation.

In the area of timed systems, both fully symbolic, "semi-symbolic" and discrete methods have been proposed, each with their different trade-offs. These methods employ state-of-the art techniques such as Difference Bound Matrices [2], Time Darts [89, 91] or polyhedra-based representations [72, 81].

Other approaches focus on the representation of explicit points in the state-space. Here, advanced algorithmic structures, such as Binary Decision Diagrams [33], have enabled the verification of systems with astronomically sized, but finitely representable, state-spaces. Other approaches, such as PTries [88, 89] and Tree Compression [99] instead focus on different compression techniques for reducing the memory consumption.

Alternative approaches for verfication, such as Counter-Example Guided Abstraction Refinement (CEGAR) [47, 60, 83] and Trace Abstraction Refinement (TAR) [41, 78], do not explore a given model, instead they attempt to construct a "good enough" abstraction to prove or disprove a property. One can think of such an abstraction as an invariant over the system, excluding more and more infeasible behavior. For the spurious trace-based abstraction methods such as CEGAR and TAR, either a non-spurious trace is found, proving property violation is found, or the invariant eventually gets strengthened enough to disprove the property. A closely related method to CEGAR and TAR is that of IC3 [28], in which invariants are sought constructed such that the system can be proven correct.

In different domains semantical equivalence and reduction-techniques have been proposed to speed up computation; a partial-order reduction [122], L/U-extrapolation [105] and up-to-congruence techniques [23] to mention a few. Compositional analysis techniques, such as those used in UPPAAL ECDAR [57], facilitate the verification of a contract-based relation-ship between increasingly refined and decomposed systems. General techniques for reducing computation time via parallelism also exist and can be found in tools such as LTSmin [22], DIVINE [13] and parallel versions of UPPAAL SMC [35]. A generic methods for making different simulation tools interoperable (so called co-simulation) is proposed using the FMI-standard [31]. In recent work, this also encompasses UPPAAL SMC and SpaceEx [73, 120].

2 Contributions

The papers constituting the remainder of this thesis have been published in the following journals or proceedings.

(A) *PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing* was published in the proceedings of *14th International Colloquium on Theoretical Aspects of Computing (2017)* [88] and received the best paper award.
Co-Authors: Kim G. Larsen, Jiří Srba

(B) *Refinement of Trace Abstraction for Real-Time Programs* was published in the proceedings of *11th International Workshop on Reachability Problems (2017)* [41].
Co-Authors: Franck Cassez, Kim G. Larsen

(C) *Discrete and Continuous Strategies for Timed-Arc Petri Net Games* was published in *International Journal on Software Tools for Technology Transfer (2017)* [87]. The paper is an extended version of the paper *Real-Time Strategy Synthesis for Timed-Arc Petri Net Games via Discretization* published in the proceedings of *International Symposium on Model Checking Software (2016)* [86].
Co-Authors: Kim G. Larsen, Jiří Srba

(D) *Practical Controller Synthesis for $MTL_{0,\infty}$* was published in the proceedings of *The International Symposium on Model Checking Software (2017)* [106].
Co-Authors: Guangyuan Li, Axel Legay, Kim G. Larsen, Danny Bøgsted Poulsen

(E) UPPAAL STRATEGO was published in the proceedings of *Tools and Algorithms for the Construction and Analysis of Systems (2015)* [55].

Co-Authors: Alexandre David, Kim G. Larsen, Marius Mikučionis, Jakob Haahr Taankvist

(F) *Co-Simulation of Hybrid Systems with* SpaceEx *and* Uppaal was published in the proceedings of *International Modelica Conference (2015)* [120].
Co-Authors: Sergiy Bogomolov, Marius Greitschus, Kim G. Larsen, Marius Mikučionis, Thomas Strump, Stavros Tripakis

(G) *Integrating Tools: Co-Simulation in* Uppaal *using FMI-FMU* was published in the proceedings of *The 22nd International Conference on Engineering of Complex Computer Systems (2017)* [85].
Co-Authors: Kim G. Larsen, Axel Legay, Ulrik Nyman

(H) *WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with* Uppaal was published in *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday (2017)* [40].
Co-Authors: Franck Cassez, Pablo Gonzalez

The presentation is as follows; in Section 2.1 we discuss the results of papers A and B with an emphasis on the state-space representation. Section 2.2 is concerned with the synthesis problem and we here present results from papers B, C, E and D. In the last section on tool integration we discuss the papers F, G and H demonstrating different aspects of tool-composition for use with Uppaal. As the thesis has contributions to several domains within the world of model checking and synthesis, we will in this section highlight the most important and significant contributions.

2.1 State-space Representation

One of the main obstacles in the domain of model checking (and by extension synthesis) is that of the state-space-explosion problem. The problem arises, in particular, when a model is composed of multiple concurrent processes, leading to a combinatorial explosion in the number of (product) states of the model. In literature one can find numerous attempts at alleviating this problem; partial-order reductions [122], symbolic representations [33, 61, 81] or compression-algorithms [67, 99]. In this thesis we have focused on two novel, but significantly different, approaches to tackling the state-space-explosion problem.

Data structures

In Paper A we focus on improving the efficiency of the implementation of a set. Specifically our work introduces a novel data structure called Partial Trie

(PTrie) that utilizes prefix-sharing as a way of both memory-reduction and state-of-the-art comparable lookup times.

We prove correctness and termination for the operations provided for PTries and show that PTries are closed under the insert, delete and member-check operations. The main contribution of the paper can be summed up as follows.

Contribution 1
We formalize and implement a novel and generic data structure, for the representation of sets of binary strings, that compared to state-of-the-art hash-maps for use in real-life model checking reduced the memory-consumption (on average) with 60-70% while incurring a minor (5%) slowdown (on average).

While such contributions in a theoretical academic context are uninteresting, in practice, a 70% memory-reduction can significantly improve on the applicability of a method. In the context of model checking, this leads to a larger set of feasibly analyzable systems as memory is a constant factor of a computer-system while time, often, is a more abundant resource. We also note that the data structure extends beyond model checking in applicability and might see interesting use-cases in other data- and performance-intensive areas. Lastly we emphasize that the formalization provides for a sound and rigorous framework for further work into more complex operations on the data structure such as tree-walks and partial-matching, but also classical extensions of such as parallel- and distributed-versions.

Trace-based Verification

In Paper B we attack the problem from a different angle. Here we are inspired by the program verification community and study the application of the technique Trace Abstraction Refinement (TAR) [78] for real-timed systems. In particular, we study the reachability problem for Timed Automata (TA) and extensions hereof, such as Stopwatch Automata and Linear Hybrid Automata [5].

What is particular interesting, regarding the TAR-methodology, is that it attempts to prove correctness of a system using regular-languages and Hoar-triples. The main idea of TAR is not to characterize the state-space (as in conventionalmodel checking), but rather characterize impossible or unusable behavior in relation to the goal, in particular to characterize such behavior in terms of a regular language over impossible or unusable operations. If the TAR framework is able to prove that the reachability-goal lies within the impossible behavior it can answer the reachability problem. Similar if a feasible sequence of Hoar-triples can be found validating a sequence of operations over the program, a positive answer can be given.

To align the TA-based formalisms for analysis via the TAR-framework we develop an intermediary representation called a Real-Timed Program. What we argue and demonstrate in Paper B is that this formalism captures a variety of timed-formalisms and thus is widely applicable.

Contribution 2
We demonstrate the feasibility of the TAR method in the setting of timed-systems and show that the TAR based approach, in certain instances, can solve problem-instances unsolvable by current state-of-the-art methods.

Future Work

PTries have already proven useful in the model checking community [87–89] and further applications in different areas such as database-systems, filesystems and similar data-intensive areas are of interest. In that regard, as PTries are a fairly novel data-structure, classical data-structure extensions require attention; parallel- and distributed versions could be developed, and also complex operations, such as partial matching, are of interest. In particular partial matching can be useful for solving the coverability-question for Petri-nets [68, 93] or for implementing an efficient "zone"-database for the verification of Timed Automata. With more complex lookups, PTries could yield a practical performance boost for the Antichain [59] and HKC-algorithms [23] for language-inclusion checking for NFAs.

While the application of TAR for timed systems is novel, ideas and optimizations for the method have been developed in the program verification community. As such, many of these could be transferred into the timed domain, however, with extra care regarding the implicit dependencies between time-sensitive variables. Other interesting work relies on utilizing the Antichain [59], HKC [23] with simulation-relation methods [1] for speeding up the TAR method. Another uncharted direction of TAR is fully hybrid systems. While such systems can be encoded as Stopwatch Automata, which are covered directly in the work of Paper B, native support using techniques for (i)nfeasibility-computation of non-linear differential equations such as those presented by Le Coënt et al in [51, 103] can prove interesting. Such a technique will require the construction of a dedicated SAT-solver for traces over general hybrid systems, but otherwise relying on the framework of TAR. Here the key ingredient is that the TAR method only requires an interpolating solver that can 1) disprove (or prove) the feasibility of a trace and 2) construct reasonable interpolants which are neither the strongest pre nor the weakest post.

2.2 Synthesis

Synthesis has recently gained renewed attention partially due to novel algorithms, advances in the machine-intelligence community but also due to significant improvements in computational power and memory realistically attainable. In this thesis we study various aspects of the synthesis problem for timed and hybrid systems.

The synthesis problem in the model checking world conventionally attains one of two forms; either

- a partially specified model is given, and the problem is to derive strategies s.t. a set of objectives are met by filling out the unspecified parts of the model,
- or concrete model is given, however with certain parameters left unspecified, and the problem is then to provide values s.t. a set of objectives are met.

Parameter Synthesis

While the main driver for Paper B is tackling the state-space-explosion problem, our TAR based algorithm proved easily extendable to solving the parameter synthesis problem for timed- and linear-hybrid systems.

Contribution 3
We provide (to the best of our knowledge) the first adaption of the TAR method for solving the parameter set synthesis problem for timed and hybrid systems. Furthermore, we demonstrate and argue that our semi-algorithm is capable of synthesizing the exact and maximal parameter sets solving the parameter synthesis problem.

As the problem of parameter synthesis for timed-systems is undecidable in general [10], the method we propose is only a semi-algorithm. Furthermore, the method relies on the supported logic of the underlying solver which in terms implies that the algorithm is not complete in general. However, for parametric linear hybrid automata with real-valued parameters, the theory is the decidable class of First Order Logic over Linear Arithmetic using Furrier-Mutzkin [71, 112] for quantifier elimination implying that our method is a semi-algorithm for this class of systems. We demonstrate on a number of case-studies that the method can solve parameter synthesis problems not solvable by existing state-of-the-art methods.

Furthermore, we demonstrate a fully automatic characterization of the constants used in Fischer's [100] protocol ensuring mutual inclusion for a constant number of processes. We note that the the parameter set constructed is not dependent on the number of processes, thus indicating that the parameter set is universal. However, this statement was not automatically proved and methods for this feat remain further work. While we use the Z3-SAT solver [58], further work would include experimenting with different, more specialized, techniques and tools for quantifier elimination and feasibility checking.

Discrete and Continuous Time

In classical model checking for timed systems, it is well know that the answer to the *reachability* problem coincides for discrete-time semantics and continuous-time-semantics [27]. A less encouraging result is known for the *liveness* problem; here the answer in the discrete-semantics only imply the answer in the continuous-semantics. In Paper C we study similar connections in the setting of timed games [113].

Contribution 4
We provide a syntax and semantics for Timed-Arc Petri Net Games (TAPG) and prove that the answers to the synthesis problem in discrete and continuous-time do not coincide. We prove that the existence of a controller in continuous time semantics (resp. discrete time semantics) does not imply the existence of a controller in discrete time semantics (resp. continuous time semantics).

With this negative result, our work provides a positive alternative.

Contribution 5
If the controller is restricted to a sub-class of non-lazy controllers (only instantaneous actions or "wait"-actions are allowed), then the answer to the synthesis question for timed systems coincide for discrete and continuous semantics.

We use this result to implement a discrete-time strategy synthesis algorithm for TAPGs and demonstrate that the synthesis problem can be answered more efficiently for certain types of systems. This is done via an experimental evaluation comparing the state-of-the-art continuous-time synthesis tool Uppaal Tiga to our discrete-time alternative, implemented in Tappaal.

Logics

In the previously discussed work, we have been focusing on various forms of synthesis for the reachability problem the safety problem. In Paper D we study the synthesis problem in the context of a more complex logic; Metric Interval Temporal Logic (MITL) [8]. It is known that synthesis of controllers for MITL specifications on timed-systems is undecidable [62]. Nonetheless, in Paper D we study the restricted $\text{MITL}_{0,\infty}$ fragment in an attempt to assess the feasibility of synthesis regardless of its undecidability.

Contribution 6
We demonstrate the construction of a (non-deterministic) monitoring timed Büchi-automata from a $\text{MITL}_{0,\infty}$ formula and prove the construction correct. We then show that deterministic over- and under-approximations can be constructed allowing for a decidable synthesis of a controller for the over- and under-approximated problems via UPPAAL TIGA.

To demonstrate our approach, we implement the proposed approximation techniques in the tool CASAAL and combine it in a tool-chain with UPPAAL TIGA. We then apply this tool-chain on a number of case-studies, demonstrating the feasibility of the approach proposed, in particular, we compute strategies for non-trivial and periodic goals. For the presented case-studies we notice that the computed over/under-approximations often are “exact and tight”, implying exact controller synthesis.

Tool Support

In Paper E we introduce an integrated tool-chain for different synthesis methods simulation and verification on timed automata-like formalisms. The tool, namely UPPAAL STRATEGO, integrates the existing classical UPPAAL, the timed game synthesis tool UPPAAL TIGA and the statistical-modelchecking engine UPPAAL SMC with a machine-learning-based extension presented in [56]. All of these methods combined allow for the automatic construction (and verification) of, safe, near-optimal schedulers—as well as their analysis.

Contribution 7
We provide a semantic sound platform based on the UPPAAL-toolsuite for automatic synthesis, verification, evaluation and optimization of controllers for an expressive game and probabilistic extension of timed automata.

As depicted in Figure 1, UPPAAL STRATEGO is a tool that integrates work and tools previously developed in the UPPAAL family. There are two main contributions in the UPPAAL STRATEGO tool-chain; one is the seamless integration of the tools, facilitating automatic synthesis via UPPAAL TIGA and the later optimization via machine-learning. The other being the extended query- and proposition-language for specifying controllers.

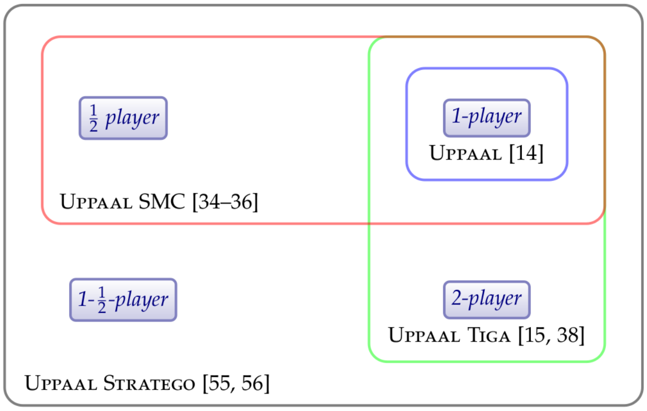


Fig. 1: The different distributions of UPPAAL embodying UPPAAL STRATEGO and the n-player problems addressed by each distribution for timed systems.

It is worth noting that this relatively recent tool-chain already has seen both academic interest for satellite- and car-control-systems [102, 123], but also industry interest with real-life case-studies and applications such as heating-control [101] and traffic-control [65].

Further Work

The adaption of the TAR method for solving 2-player games is interesting. Not only as the computations will significantly differ from the classical methods but the representation of the strategies also will. This may impact the practical applicability as TAR-based strategies can prove more "abstract" and thus more compact.

In the area of discrete-time synthesis, utilizing well-known optimizations such as partial-order reduction and Timed Darts can reduce the memory- and time-consumption further. Furthermore, while Paper C defines a subclass for which the semantics coincide for strategy-synthesis, the proof is non-constructive and thus it is not yet known how to construct a continuous-time strategy from a discrete-time strategy (and vice versa).

Investigating synthesis for stronger logics, such as $\text{MITL}_{0,\infty}$, in the context of discrete and continuous time semantics may reveal interesting classes of systems for which synthesis is decidable. The general idea of over- and under-approximation can also be pursued in synthesis for more complex systems such as hybrid automata.

Lastly all of these techniques, when matured, should be provided readily available to users. UPPAAL STRATEGO provides a solid platform for further development and some interesting challenges are still open; efficient strategy-representation, semantic-preserving export/import and certificate of correctness of the constructed controller.

2.3 Tool Integration

In the academic research in formal methods, it is often expected that a complete system description is provided in some given rigorous framework with a formal syntax and semantics. Such expectations (or assumptions) allow for strong theorems and argumentation. However, in the context of industry-sized systems this can pose limitations to applicability, but almost surely introduces redundant work and maintainability issues because the model and the implementation has to be kept in sync. An even more severe problem is the lack of homogeneity of the modeling frameworks; it is common in industry that numerous tools and frameworks are used in the construction of a product often making a joint *co-simulation* a troublesome task. To remedy this problem, among others, the Function Mockup Interface (FMI) [69] standard has been introduced. The FMI standard dictates a protocol and method for exchanging information between tools such that a joint analysis of a system can be given.

In this section we outline the contributions of the thesis in terms of easing integration of model-based-development (and synthesis) for industry using the FMI-standard—but also utilizing a direct interconnection between tools via a `C`-interface.

Co-Simulation

In Paper F and Paper G we study the co-simulation-framework proposed as part of the Function Mockup Interface standard (FMI). The FMI standard is used in industry for the exchange of (physical) models between a heterogeneous zoo of tools. While more complex modes of operations are proposed under the FMI-standard, we restrict ourselves to the simpler, co-simulation-standard. A co-simulation (in the context of the FMI standard) consists of several Function Mockup Units (FMU), composed in a system where they exchange values at given points in time, all coordinated by a so-called Master Algorithm (MA), implemented in a host simulation tool.

In Paper F we discuss the embedding of timed systems, via integration of Uppaal and SpaceEx as a FMUs, into a co-simulation environment, adjacent to FMUs native to the simulation-host-tool, Ptolemy. As illustrated in Figure 2 the proposed method considers a system composed of $A_1, \ldots, A_n$ subsystems where each subsystem is defined in a given tool (Uppaal, SpaceEx or other FMU exporting tools). The system is then composed and directed by the MA implemented in Ptolemy.

The paper demonstrates that such an integration is possible, however, it is not without problems.

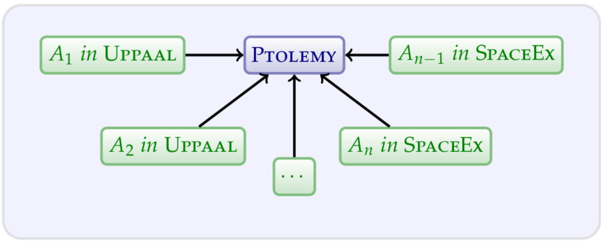


Fig. 2: Generalized idea of the FMI/FMU concept as presented in Paper F. A System composed of subsystems (green) A_1, A_2, $\ldots A_{n-1}$, A_n composed under MA (blue) in Ptolemy.

Contribution 8
We argue that the FMU-standard, for the semantic consistency of compositionality of timed-systems, has to abandon the non-zero-delay requirement. We also demonstrate that the acyclic-requirement (for avoiding algebraic loops in the simulation) conflicts with interleaving semantics for timed-systems.

What we demonstrate is that timed-systems (such as timed automata), when composed under the master-algorithm proposed by Ptolemy, exhibits only a sub-set of the behavior defined in their original compositional semantics. As such there is a strong argument for alternative MAs when timed-systems are involved.

This issue is partially solved in Paper G where we attempt to replace Ptolemy as a simulation-driver with Uppaal. This construction is made feasible by a recent extension of Uppaal, allowing for calls of external `C`-functions.

Contribution 9
We propose a framework for embedding FMUs into a Uppaal timed automata and show how a MA, dictating the semantics of the system, can itself be implemented as a timed automata. We furthermore propose a timed automata-based MA which allows for interleaving-semantics, allowing for sane composition of timed automata as FMUs.

While it is not explicitly stated in Paper G, the implementation allows for vertical composition of systems as illustrated in Figure 3. Such a compositional structure is not only practical in terms of human overview of the system but also has the potential to speed up computation as step-sizes are negotiated in a tree-like manner. As such, embedded FMU/MA compositions can negotiate more lax stepsizes.

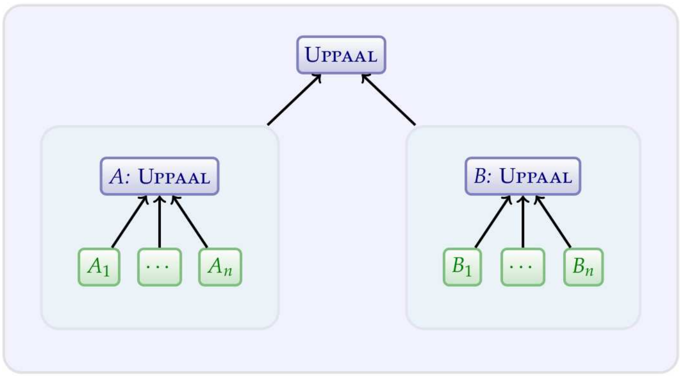


Fig. 3: MA (blue) composing the systems A and B (green) which themselves are composed of subsystems (green) $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ and a MA (blue) in UPPAAL.

Black-box verification

While the above methods for simulation are useful in practice, they do not come with hard guarantees with regard to the outcome of the analysis. However, they provide a quick way for integrating several pre-existing components into a single simulation. In Paper H we study a tool-integration in UPPAAL using the external C-linking feature recently introduced. The purpose is to offload the modeling-workload and reuse pre-existing tools for an on-the-fly unfolding of timed automata. In the paper we utilize programs for software-slicing for pre-computing an (abstract) control-flow graph (CFG). We then use an off-the-shelf ARM-processor-emulator called QEMU [18] for providing the exact semantics of each instruction of the CFG. The paper then presents a tool-chain for facilitating worst-case-execution-time analysis, gluing a UPPAAL-model containing the timing-information of the caches, with QEMU, proving a correct semantics of each operation of the CFG.

Contribution 10
We demonstrate a tool-chain composing external libraries and internal modeling in order to reduce modeling overhead and increase reuse by having the external libraries unfold the (untimed) system on-the-fly.

Further Work

While the work presented in this section only concerns itself with model checking and simulation problems, it is immediately possible to extend the given results and methods into the realm of automatic synthesis using trace-based methods. In particular the external C-functions are already available in development branch of UPPAAL STRATEGO, and thus an adaption of the

FMI/FMU framework from Paper G with Uppaal Stratego is straightforward.

However, better interoperability between methods for verification and synthesis is interesting further work; so-called co-verification. For instance, as Paper C argues, a significant difference exists between the semantic interpretations of time, hence a discrete-time optimized verification- or synthesisengine should be used for electronic controller circuits while real-time and hybrid aware engines should compute the continuous and physical part of a system. Such an idea could be implemented on top of the decompositional framework of Timed I/O Automata [57].

3 Conclusion

The work presented as part of this thesis provides a step in the direction of more efficient verification engines, practical controller synthesis and academic tools ready for integration into industrial tool-chains.

We have presented novel datastructure called PTrie that is generally applicable in a wide range of context, but in the specific setting of model checking has provided truly significant speedups. Alternative methods for state-space characterization has also been investigated, and we have successfully applied the Trace Abstraction Refinement technique to timed systems. Furthermore, we have demonstrated that this method is not only applicable to verification but with a minor extension is extendable to solve the parameter synthesis question for timed and hybrid systems.

We have proven that discrete and continuous semantics do not coincide for the safety synthesis question for Timed-Arc Petri-net Games, a result that translates directly to timed automata. To remedy the negative result, we propose a semantic restriction on the games considered. We prove that continuous and discrete-time semantics coincide for the safety-synthesis question for systems in which the controller is restricted to only untimed choices. For this class of problems, we demonstrate that an explicit-state discrete-time synthesis tool can outperform a state-of-the-art symbolic synthesis tool, Uppaal Tiga. However, we leave open the translation of a discrete (resp. continuous) time controller to a continuous (resp. discrete) time controller.

On a range of case-studies, we demonstrate that over- and under- approximations of $\text{MITL}_{0,\infty}$ is a viable strategy for the synthesis of controllers with complex objectives. An important point stressed in this work is that the approximations often are tight; the over- and under-approximation are the same, implying that the synthesis question for the given problem-instance is decidable.

In another aspect of this thesis, we study the interaction and interoperability of tools. We demonstrate both semantic issues when using the off-the-shelf protocol FMI from industry for co-simulation with timed automata. We then argue for a solution to the semantic inconsistencies, notably that using UPPAAL SMC itself as a driver for the simulation can ensure semantic integrity. At the same time we demonstrate a new way of interacting with UPPAAL SMC, namely via native C-function calls. We also demonstrate that the proposed method works well with the existing features of UPPAAL SMC and hint that such an integration is also possible for the synthesis tool UPPAAL STRATEGO.

We also utilize the external C-function calls to implement a generic framework for model checking of binary programs. We here combine UPPAAL with the ARM-emulator Qemu and program abstraction techniques from the program verification community. We show the feasibility of this approach and we argue that this greatly reduces the integration-effort needed for model checking of industrial software while at the same time reducing the room for human error.

Lastly, with STRATEGO we demonstrate tool-interaction and novel methods for synthesis, enabling synthesis of controllers for industrial systems. However, there is an effort undertaken to migrate more of the techniques and technologies developed as part of this thesis into the tool. This is an attempt to construct a platform for industry strength controller synthesis in a usable manner for engineers. Here we put an emphasis on that all parts of a system need not be modeled within the tool, but rather can be provided as a black-box for use by the tool.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar, "When simulation meets antichains: On checking language inclusion of nondeterministic finite (tree) automata," in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 158–174. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_14

[2] R. Alur, "Timed automata," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds. Springer Berlin Heidelberg, 1999, vol. 1633, pp. 8–22. [Online]. Available: http://dx.doi.org/10.1007/3-540-48683-6_3

[3] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and computation*, vol. 104, no. 1, pp. 2–34, 1993.

[4] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking for probabilistic real-time systems," in *In Automata, Languages and Programming: Proceedings of the 18th ICALP, Lecture Notes in Computer Science 510*. Springer, 1991, pp. 115–126.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems,"

in *Hybrid Systems*, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 209–229.

[6] R. Alur and D. Dill, “Automata for modeling real-time systems,” in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, M. Paterson, Ed. Springer Berlin Heidelberg, 1990, vol. 443, pp. 322–335. [Online]. Available: http://dx.doi.org/10.1007/BFb0032042

[7] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397594900108

[8] R. Alur, T. Feder, and T. A. Henzinger, “The benefits of relaxing punctuality,” *J. ACM*, vol. 43, pp. 116–146, January 1996.

[9] R. Alur, T. A. Henzinger, and O. Kupferman, “Alternating-time temporal logic,” *J. ACM*, vol. 49, no. 5, pp. 672–713, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/585265.585270

[10] R. Alur, T. A. Henzinger, and M. Y. Vardi, “Parametric real-time reasoning,” in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, ser. STOC ’93. New York, NY, USA: ACM, 1993, pp. 592–601. [Online]. Available: http://doi.acm.org/10.1145/167088.167242

[11] R. Alur, S. La Torre, and G. J. Pappas, “Optimal paths in weighted timed automata,” in *Hybrid systems: computation and control*. Springer, 2001, pp. 49–62.

[12] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, “Model-checking continuous-time markov chains,” *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 162–170, Jul. 2000. [Online]. Available: http://doi.acm.org/10.1145/343369.343402

[13] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, “DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs,” in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[14] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, “Uppaal 4.0,” in *QEST’06*, 2006, pp. 125–126.

[15] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, “Uppaal-tiga: Time for playing games!” in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. LNCS, no. 4590. Springer, 2007, pp. 121–125.

[16] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime, “Uppaal-tiga: Time for playing games!” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 121–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_14

[17] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, *Minimum-cost reachability for priced time automata*. Springer, 2001.

[18] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[19] J. Berendsen, D. Jansen, and F. Vaandrager, "Fortuna: Model checking priced probabilistic timed automata," in *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, Sept 2010, pp. 273–281.

[20] J. Berendsen, D. N. Jansen, and J.-P. Katoen, "Probably on time and within budgeton reachability in priced probabilistic timed automata," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 2006, pp. 311–322.

[21] B. Berthomieu and F. Vernadat, "Time petri nets analysis with tina," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 2006, pp. 123–124.

[22] S. Blom, J. van de Pol, and M. Weber, "Ltsmin: Distributed and symbolic reachability," in *Computer Aided Verification*. Springer, 2010, pp. 354–359.

[23] F. Bonchi and D. Pous, "Checking nfa equivalence with bisimulations up to congruence," in *ACM SIGPLAN Notices*, vol. 48, no. 1. ACM, 2013, pp. 457–468.

[24] P. Bouyer, F. Cassez, E. Fleury, and K. G. Larsen, "Optimal strategies in priced timed game automata," in *FSTTCS*, 2004.

[25] P. Bouyer, N. Markey, and O. Sankur, *Robustness in Timed Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-41036-9_1

[26] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 1998, pp. 298–302.

[27] M. Bozga, O. Maler, and S. Tripakis, "Efficient verification of timed automata using dense and discrete time semantics," in *Correct Hardware Design and Verification Methods*. Springer, 1999, pp. 125–141.

[28] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[29] T. Brihaye, V. Bruyere, and J.-F. Raskin, "Model-checking for weighted timed automata," in *FORMATS/FTRTFT*, vol. 3253. Springer, 2004, pp. 277–292.

[30] ——, "On optimal timed strategies," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2005, pp. 49–64.

[31] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-153, Aug 2013. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-153.html

[32] V. Bruyère, E. Filiot, M. Randour, and J. Raskin, "Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games," *CoRR*, vol. abs/1309.5439, 2013. [Online]. Available: http://arxiv.org/abs/1309.5439

[33] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[34] P. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, D. B. Poulsen, and A. Stainer, "Monitor-based statistical model checking for weighted metric temporal logic," in *LPAR*, 2012.

[35] P. Bulychev, A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Checking and distributing statistical model checking," in *NASA Formal Methods*. Springer, 2012, pp. 449–463.

[36] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang, "UPPAAL-SMC: statistical model checking for priced timed automata," in *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012.*, 2012, pp. 1–16. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.85.1

[37] J. Byg, K. Y. Jørgensen, and J. Srba, "TAPAAL: Editor, simulator and verifier of timed-arc Petri nets," in *Automated Technology for Verification and Analysis: 7th International Symposium*, ser. LNCS, vol. 5799. Springer, 2009, pp. 84–89. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04761-9_7

[38] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR 2005 - Concurrency Theory, 16th International Conference*, ser. Lecture Notes in Computer Science, M. Abadi and L. de Alfaro, Eds., vol. 3653. San Francisco, CA, USA: Springer, August 2005, pp. 66–80. [Online]. Available: http://dx.doi.org/10.1007/11539452_9

[39] ——, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR*, 2005.

[40] F. Cassez, P. G. de Aledo, and P. G. Jensen, *WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with UPPAAL*. Cham: Springer International Publishing, 2017, pp. 560–577. [Online]. Available: https://doi.org/10.1007/978-3-319-63121-9_28

[41] F. Cassez, P. G. Jensen, and K. G. Larsen, "Refinement of trace abstraction for real-time programs," in *International Workshop on Reachability Problems*. Springer, 2017, pp. 42–58.

[42] F. Cassez and K. Larsen, "The impressive power of stopwatches," in *CONCUR 2000—Concurrency Theory*. Springer, 2000, pp. 138–152.

[43] K. Chatterjee, M. Jurdziński, and T. A. Henzinger, "Quantitative stochastic parity games," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, pp. 121–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=982792.982808

[44] K. Chatterjee, M. Jurdziński, and T. Henzinger, "Simple stochastic parity games," in *Computer Science Logic*, ser. Lecture Notes in Computer Science, M. Baaz and J. Makowsky, Eds. Springer Berlin Heidelberg, 2003, vol. 2803, pp. 100–113. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45220-1_11

[45] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis, "PRISM-games: A model checker for stochastic multi-player games," in *Proc. 19th International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013, pp. 185–191.

[46] ——, "Automatic verification of competitive stochastic systems," *Formal Methods in System Design*, vol. 43, no. 1, pp. 61–92, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10703-013-0183-7

[47] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.

[48] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[49] ——, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. London, UK, UK: Springer-Verlag, 1982, pp. 52–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=648063.747438

[50] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[51] A. L. Coënt, F. De Vuyst, L. Chamoin, and L. Fribourg, "Control synthesis of nonlinear sampled switched systems using euler's method," *arXiv preprint arXiv:1704.03102*, 2017.

[52] A. Condon, "The complexity of stochastic games," *Information and Computation*, vol. 96, no. 2, pp. 203–224, 1992.

[53] C. Courcoubetis and M. Yannakakis, "Verifying temporal properties of finite-state probabilistic programs," in *Foundations of Computer Science, 1988., 29th Annual Symposium on*. IEEE, 1988, pp. 338–345.

[54] A. David, L. Jacobsen, M. Jacobsen, K. Y. Jørgensen, M. H. Møller, and J. Srba, "Tapaal 2.0: Integrated development environment for timed-arc petri nets," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 492–497.

[55] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Uppaal stratego," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds. Springer Berlin Heidelberg, 2015, vol. 9035, pp. 206–211. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_16

[56] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist, "On time with minimal expected cost!" in *ATVA*, 2014, pp. 129–145.

[57] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed i/o automata: A complete specification theory for real-time systems," in *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '10. New York, NY, USA: ACM, 2010, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1755952.1755967

[58] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[59] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin, "Antichains: A new algorithm for checking universality of finite automata," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 17–30.

[60] H. Dierks, S. Kupferschmid, and K. G. Larsen, "Automatic abstraction refinement for timed automata," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2007, pp. 114–129.

[61] D. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Automatic Verification Methods for Finite State Systems*, ser. LNCS. Springer, 1990, vol. 407, pp. 197–212. [Online]. Available: http://dx.doi.org/10.1007/3-540-52148-8_17

[62] L. Doyen, G. Geeraerts, J. Raskin, and J. Reicher, "Realizability of real-time logics," in *Proceedings of FORMATS 2009, 7th International Conference on Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, vol. 5813. Springer, 2009, pp. 133–148.

[63] E. A. Emerson and E. M. Clarke, "Characterizing correctness properties of parallel programs using fixpoints," in *Automata, Languages and Programming*, J. de Bakker and J. van Leeuwen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 169–181.

[64] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: on branching versus linear time temporal logic," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 151–178, 1986.

[65] A. B. Eriksen, C. Huang, J. Kildebogaard, H. Lahrmann, K. G. Larsen, M. Muniz, and J. H. Taankvist, "Uppaal stratego for intelligent traffic lights," in *12th ITS European Congress, European Congress and Exhibition on Intelligent Transport Systems and Services*. ERTICO-ITS Europe, 2017.

[66] Étienne André, T. Chatain, L. Fribourg, and E. Encrenaz, "An inverse method for parametric timed automata," *Electronic Notes in Theoretical Computer Science*, vol. 223, pp. 29 – 46, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066108004921

[67] S. Evangelista and J.-F. Pradat-Peyre, "Memory efficient state space storage in explicit software model checking," in *Model Checking Software: 12th International SPIN Workshop*, ser. LNCS, vol. 3639. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 43–57. [Online]. Available: http://dx.doi.org/10.1007/11537328_7

[68] A. Finkel, *The minimal coverability graph for Petri nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 210–243. [Online]. Available: https://doi.org/10.1007/3-540-56689-9_45

[69] FMI Standard Orginization, "Functional mock-up interface," http://fmi-standard.org/.

[70] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *Formal Methods for Eternal Networked Software Systems*. Springer, 2011, pp. 53–113.

[71] J. Fourier, *Analyse des travaux de l'Academie Royale des Sciences, pendant l'année 1827. Partie mathématique*, 1827.

[72] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Berlin Heidelberg, 2005, vol. 3414, pp. 258–273. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31954-2_17

[73] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 379–395. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_30

[74] S. Haddad and B. Monmege, "Reachability in mdps: Refining convergence of value iteration," in *Reachability Problems*, ser. Lecture Notes in Computer Science, J. Ouaknine, I. Potapov, and J. Worrell, Eds. Springer International Publishing, 2014, vol. 8762, pp. 125–137. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11439-2_10

[75] H.-M. Hanisch, "Analysis of place/transition nets with timed arcs and its application to batch process control," in *Application and Theory of Petri Nets 1993*. Springer, 1993, pp. 282–299.

[76] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal aspects of computing*, vol. 6, no. 5, pp. 512–535, 1994.

[77] A. Hartmanns and H. Hermanns, "A modest approach to checking probabilistic timed automata," in *Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the*. IEEE, 2009, pp. 187–196.

[78] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *SAS*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 69–85.

[79] D. Henriques, J. Martins, P. Zuliani, A. Platzer, and E. Clarke, "Statistical model checking for markov decision processes," in *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*, Sept 2012, pp. 84–93.

[80] T. A. Henzinger, *The theory of hybrid automata*. Springer, 2000.

[81] T. A. Henzinger, P.-H. Ho, and H. Wong-toi, "Hytech: A model checker for hybrid systems," *Software Tools for Technology Transfer*, vol. 1, pp. 460–463, 1997.

[82] T. A. Henzinger and H. Wong-Toi, *Using HyTech to synthesize control parameters for a steam boiler*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 265–282. [Online]. Available: https://doi.org/10.1007/BFb0027241

[83] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic cegar," in *CAV*, vol. 5123. Springer, 2008, pp. 162–175.

[84] H. E. Jensen, "Model checking probabilistic real time systems," in *Chalmers Institute of Technology*, 1996, pp. 247–261.

[85] P. G. Jensen, K. G. Larsen, A. Legay, and U. Nyman, "Integrating tools: Co-simulation in uppaal using fmi-fmu," in *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, Nov 2017, pp. 11–19.

[86] P. G. Jensen, K. G. Larsen, and J. Srba, "Real-time strategy synthesis for timed-arc Petri net games via discretization," in *Proceedings of the 23rd International Symposium on Model Checking Software (SPIN'16)*, ser. LNCS, vol. 9641. Springer, 2016, pp. 129–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-32582-8_9

[87] ——, "Discrete and continuous strategies for timed-arc petri net games," *International Journal on Software Tools for Technology Transfer*, Sep 2017. [Online]. Available: https://doi.org/10.1007/s10009-017-0473-2

[88] ——, *PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing*. Cham: Springer International Publishing, 2017, pp. 248–265. [Online]. Available: https://doi.org/10.1007/978-3-319-67729-3_15

[89] P. G. Jensen, K. G. Larsen, J. Srba, M. G. Sørensen, and J. H. Taankvist, "Memory efficient data structures for explicit verification of timed systems." in *NASA Formal Methods*, 2014, pp. 307–312.

[90] J. J. Jessen, J. I. Rasmussen, K. G. Larsen, and A. David, "Guided controller synthesis for climate controller using uppaal tiga," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, J.-F. Raskin and P. Thiagarajan, Eds. Springer Berlin Heidelberg, 2007, vol. 4763, pp. 227–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75454-1_17

[91] K. Y. Jørgensen, K. G. Larsen, and J. Srba, "Time-darts: A data structure for verification of closed timed automata," in *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012.*, 2012, pp. 141–155. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.102.13

[92] A. Jovanović, D. Lime, and O. H. Roux, *Integer Parameter Synthesis for Timed Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 401–415. [Online]. Available: https://doi.org/10.1007/978-3-642-36742-7_28

[93] R. M. Karp and R. E. Miller, "Parallel program schemata," *Journal of Computer and System Sciences*, vol. 3, no. 2, pp. 147 – 195, 1969. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000069800115

[94] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer Berlin Heidelberg, 2006, vol. 4212, pp. 282–293. [Online]. Available: http://dx.doi.org/10.1007/11871842_29

[95] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, Nov 1990. [Online]. Available: https://doi.org/10.1007/BF01995674

[96] D. Kozen, "Results on the propositional μ-calculus," *Theoretical computer science*, vol. 27, no. 3, pp. 333–354, 1983.

[97] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 2, pp. 128–142, 2004.

[98] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions," in *Formal Methods for Real-Time and Probabilistic Systems*. Springer, 1999, pp. 75–95.

[99] A. Laarman, J. van de Pol, and M. Weber, "Parallel recursive state compression for free," in *Model Checking Software: 18th International SPIN Workshop*, ser. LNCS, vol. 6823. Springer, 2011, pp. 38–56. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22306-8_4

[100] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 1–11, 1987.

[101] K. G. Larsen, M. Mikučionis, M. Muñiz, J. Srba, and J. H. Taankvist, *Online and Compositional Learning of Controllers with Application to Floor Heating*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 244–259. [Online]. Available: https://doi.org/10.1007/978-3-662-49674-9_14

[102] K. G. Larsen, M. Mikučionis, and J. H. Taankvist, *Safe and Optimal Adaptive Cruise Control*. Cham: Springer International Publishing, 2015, pp. 260–277. [Online]. Available: https://doi.org/10.1007/978-3-319-23506-6_17

[103] A. Le Coënt, J. A. dit Sandretto, A. Chapoutot, and L. Fribourg, "Control of nonlinear switched systems based on validated simulation," in *Symbolic and Numerical Methods for Reachability Analysis (SNR), 2016 International Workshop on*. IEEE, 2016, pp. 1–6.

[104] A. Legay and S. Sedwards, "Lightweight monte carlo algorithm for markov decision processes," *arXiv preprint arXiv:1310.3609*, 2013.

[105] G. Li, "Checking timed büchi automata emptiness using lu-abstractions," in *Formal Modeling and Analysis of Timed Systems: 7th International Conference*, ser. LNCS. Springer, 2009, vol. 5813, pp. 228–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04368-0_18

[106] G. Li, P. G. Jensen, K. G. Larsen, A. Legay, and D. B. Poulsen, "Practical controller synthesis for $\text{mtl}_{0,\infty}$," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: ACM, 2017, pp. 102–111. [Online]. Available: http://doi.acm.org/10.1145/3092282.3092303

[107] D. Lime, O. H. Roux, C. Seidner, and L.-M. Traonouez, "Romeo: A parametric model-checker for petri nets with stopwatches," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 54–57.

[108] O. Madani, "Polynomial value iteration algorithms for deterministic mdps," in *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 311–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=2073876.2073913

[109] P. Madhusudan and P. S. Thiagarajan, “Distributed controller synthesis for local specifications,” in *ICALP*, vol. 1. Springer, 2001, pp. 396–407.

[110] O. Maler, A. Pnueli, and J. Sifakis, “On the synthesis of discrete controllers for timed systems,” in *STACS 95*. Springer, 1995, pp. 229–242.

[111] P. M. Merlin, “A study of the recoverability of computing systems.” 1975.

[112] D. Monniaux, “A quantifier elimination algorithm for linear real arithmetic,” in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2008, pp. 243–257.

[113] A. Pnueli, E. Asarin, O. Maler, and J. Sifakis, “Controller synthesis for timed automata,” in *System Structure and Control*, Citeseer. Elsevier, 1998.

[114] A. Pnueli, “The temporal logic of programs,” in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

[115] A. Pnueli and R. Rosner, “A framework for the synthesis of reactive modules,” in *Concurrency 88: International Conference on Concurrency, Hamburg, FRG, October 18-19, 1988, Proceedings*, 1988, pp. 4–17. [Online]. Available: https://doi.org/10.1007/3-540-50403-6_28

[116] ——, “On the synthesis of a reactive module,” in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, 1989, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/75277.75293

[117] D. Poulsen and J. van Vliet, “Duration probabilistic automata,” Technical report, Aalborg University, Tech. Rep., 2011.

[118] J. P. Queille and J. Sifakis, “Specification and verification of concurrent systems in cesar,” in *International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351.

[119] C. Ramamoorthy and G. Ho, “Performance evaluation of asynchronous concurrent systems using petri nets,” *Software Engineering, IEEE Transactions on*, vol. SE-6, no. 5, pp. 440–449, Sept 1980.

[120] S. Sergiy, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikučionis, A. Podelski, T. Strump, and S. Tripakis, “Co-simulation of hybrid systems with spaceex and uppaal,” 2015, p. to appear.

[121] W. J. Stewart, *Introduction to the numerical solution of Markov chains*. Princeton University Press Princeton, 1994, vol. 41.

[122] A. Valmari, *Stubborn sets for reduced state space generation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 491–515. [Online]. Available: https://doi.org/10.1007/3-540-53863-1_36

[123] E. R. Wognsen, B. R. Haverkort, M. Jongerden, R. R. Hansen, and K. G. Larsen, *A Score Function for Optimizing the Cycle-Life of Battery-Powered Embedded Systems*. Cham: Springer International Publishing, 2015, pp. 305–320. [Online]. Available: https://doi.org/10.1007/978-3-319-22975-1_20

[124] H. L. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Computer Aided Verification*. Springer, 2002, pp. 223–235.

Part II

Papers

Paper A

PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing

Peter Gjøl Jensen, Kim Guldstrand Larsen and Jiří Srba

The paper has been published in the
Proceedings of Theoretical Aspects of Computing – ICTAC 2017, pp. 248–265,
2017.

Abstract

Sets and their efficient implementation are fundamental in all of computer science, including model checking, where sets are used as the basic data structure for storing (encodings of) states during a state-space exploration. In the quest for fast and memory efficient methods for manipulating large sets, we present a novel data structure called PTrie for storing sets of binary strings of arbitrary length. The PTrie data structure distinguishes itself by compressing the stored elements while sharing the desirable key characteristics with conventional hash-based implementations, namely fast insertion and lookup operations. We provide the theoretical foundation of PTries, prove the correctness of their operations and conduct empirical studies analysing the performance of PTries for dealing with randomly generated binary strings as well as for state-space exploration of a large collection of Petri net models from the 2016 edition of the Model Checking Contest (MCC'16). We experimentally document that with a modest overhead in running time, a truly significant space-reduction can be achieved. Lastly, we provide an efficient implementation of the PTrie data structure under the GPL version 3 license, so that the technology is made available for memory-intensive applications such as model checking tools.

1 Introduction

Formal verification techniques are being increasingly employed in many different industrial applications, including both hardware and software systems. In the hardware industry such techniques have been adopted by most of the major leading companies and a widespread adoption in the software industry is under way. Formal techniques have become essential for certain safety-critical applications for example in the avionics and aerospace industry but also in other areas—like the development of operating systems, control systems for railways and numerous other applications. The performance of the respective verification tools depends to a large extent on fast and memory efficient implementations of the underlying data structures used in the verification algorithms. This is in particular due to the state-space-explosion problem that all modern model checkers must deal with. Such tools are not only constrained by the time requirements but also by the physical limitations like the amount of memory resources of the hardware that the implementation is targeted for.

A common data structure used in model checking and many other applications is a set. We revisit the state-of-the-art implementation approaches for storing sets that offer the basic operations of inserting an element to the set, removing an element from the set and a membership check. This simple set interface is sufficient for the applications in many explicit model checkers, while the symbolic approaches may require more complex operations like intersection and union that are, however, more expensive in implementation. In order to compete with the foremost hash-based approaches for storing sets, we develop a particular tree-based representation of a set called PTrie that is optimized both for speed and memory. PTrie is designed for storing binary strings of arbitrary length but via binary encoding/decoding techniques it can be used as a general set-implementation. An early implementation of PTrie was briefly mentioned in a tool paper by Jensen et. al [1], indicating encouraging performance results. Since then the data structure was further developed, extensively tested and matured so that it became competitive with the industrial leading implementations.

Although generic data structures for sets already exist in the standard-library of `C++`, Google's `dense_hash` (and `sparse_hash`) implementations perform significantly faster (or have a smaller memory footprint) than other reasonable alternatives as documented e.g. in [2, 3]. PTrie are designed as an almost general replacement of such library implementations and yield a sensible trade off between time and space consumption by utilizing the inherent prefix-sharing whenever beneficial. The main characteristic of the structure is the partial (lazy) construction of the trie—hence the name Partial Trie (PTrie)—that is optimized for storing a large number of binary strings of varying size. At the same time the PTrie data structure utilizes the prefix-sharing of the binary strings, often resulting in significant compression of the stored data, sometime up to 70% compared to the Google's hash-based implementation . In the present paper, we formally define the syntax and semantics of PTries, give the algorithms for the interface operations, prove their correctness and provide an open-source implementation that is thoroughly tested against other approaches.

Related Work. While tries were introduced already in the 1960's [4], their primary focus was on reducing search time in large sets of text-strings. Different variants of tries have been developed during the years, such as Radix tree [5, 6] designed for storing more than single characters on edges or trie-based hashmaps for both the sequential and concurrent setting [7, 8]. Our work differs by having a very conservative approach to the expansion of the trie in order to achieve both speed and overall memory reductions. Notably, the burst tries [9] do not make use of a B-Tree-style pointer scheme and do not enforce removal of the prefix, resulting in an overhead in memory-consumption and not reduction as in PTries. The HAT-tries [7] enforce the use of hashes for elements in buckets, which is not necessary in our data structure. Moreover, neither [9] nor [7] provide a formal definition of their algorithms or the semantics, and they do not present the delete-operation (or "inverse burst"), which we provide. Also Bagwell's work on HAMT [10] is mostly using trie-structures in combination with hashes of data and comes with added memory-footprint rather than memory reduction. In our experiments, we compare the PTrie performance only with Google's densehash/sparsehash implementations as other popular trie libraries [11–13] are not competitive with Google hash libraries for the model checking application domain that relies on fast and memory efficient implementation of sets.

Various forms of trees (Red/Black trees, binary trees, heaps) are conventionally also used for implementing sets and map-like data structures but such implementations are generally regarded inferior in terms of performance [14, 15]. Binary Decision Diagrams (BDD) [16] are another efficient way of storing binary strings, however with a very high average computational cost (as documented e.g. in [1]) for the basic single-element operations such as insert and delete.

In the domain of model checking, Laarman et. al. [17] introduced a tree-style compressing data structure for multi-core model checking, a method that compresses inserted data on-the-fly by utilizing sub-string sharing between integer strings, encoded into a tree structure. A similar technique has been used by the tool DIVINE [18], leading to great memory reductions, however, at the cost of performance. While both papers demonstrate promising results, we argue that these works are orthogonal as they both rely on efficient map and set implementations. Furthermore, these methods come with a number of restrictions making them less suitable as general set and map implementations. Other model checking specific compression-techniques like *Delta*-compression [19] have been proposed but suffer from even a greater impact on running-time as well as lacking general applicability. The explicit-state model checker LoLa [20] implements a basic prefix sharing scheme for the state-compression, but has yet to provide this as a stand-alone library with accompanying benchmarks and does not include the essential performance enhancements used in PTrie.

2 Definition of PTrie

Let $\mathcal{B} = \{0, 1\}$ be a binary alphabet and let $\mathcal{B}^*$ be the set of all binary strings over $\mathcal{B}$ where ϵ is the empty string. If $w = b_1 b_2 \ldots b_n$ and $w' = b'_1 b'_2 \ldots b'_m$ then $w \circ w' = b_1 b_2 \ldots b_n b'_1 b'_2 \ldots b'_m$ is the concatenation of the two strings (we shall often write just ww' instead of $w \circ w'$). For a binary string $w = b_1 b_2 \ldots b_n$, the length of w is defined as $|w| = n$ where by definition $|\epsilon| = 0$, and we use the substring notation $w_{[i,j]}$ where $1 \leq i, j \leq n$ such that $w_{[i,j]} = b_i b_{i+1} \ldots b_j$ if $i \leq j$ and $w_{[i,j]} = \epsilon$ if $i > j$.

Let $\mathcal{B}^n$ be the set of all binary strings of length n and let $\Theta^n = \{ww' \mid w \in \mathcal{B}^*, w' \in \{\bullet\}^*, |ww'| = n\}$ be the set of all extended binary strings of length n, i.e. binary strings that can be suffixed with a sequence of wild characters $\bullet$. The semantics of an extended binary string w is the set of all binary strings it represents $[\![w]\!]$ and it is inductively defined as follows (where $b \in \mathcal{B} \cup \{\bullet\}$

and $w \in (\mathcal{B} \cup \{\bullet\})^*$).

$$\llbracket \epsilon \rrbracket = \{\epsilon\}$$

$$\llbracket b \circ w \rrbracket = \begin{cases} \{b \circ w' \mid w' \in \llbracket w \rrbracket\} & \text{if } b \in \mathcal{B} \\ \{0 \circ w', 1 \circ w' \mid w' \in \llbracket w \rrbracket\} & \text{if } b = \bullet \end{cases}$$

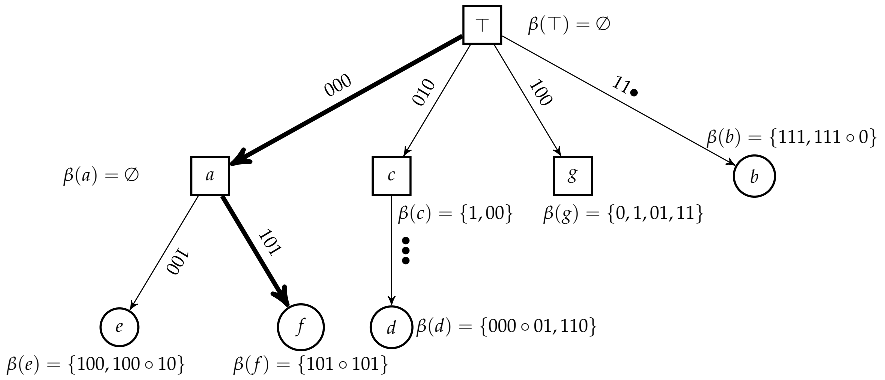
In the rest of this paper, we assume an implicitly given integer constant $\iota > 0$ called the byte size and an integer constant $\kappa \geq 2$ called the bucket size.

Definition 1 (PTrie Syntax)
A PTrie is a tuple $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ where

1. F is a finite set of forwarding vertices,
2. L is a finite set of leaf vertices such that $F \cap L = \emptyset$,
3. $E \subseteq F \times (F \cup L)$ is a finite set of edges such that $(F \cup L, E)$ is a tree,
4. $\top \in F$ is the root vertex of the tree $(F \cup L, E)$,
5. $\lambda : E \to \Theta^\iota$ is a labeling function assigning an extended binary string of length ι to each edge such that
   (a) $\llbracket \lambda(u,v) \rrbracket \cap \llbracket \lambda(u,v') \rrbracket = \emptyset$ for all $(u,v),(u,v') \in E$ where $v \neq v'$, and
   (b) $\lambda(u,v) \in \mathcal{B}^\iota$ for all $(u,v) \in E$ where $v \in F$,
6. $\beta : L \cup F \to 2^{\mathcal{B}^*}$ is a bucket function such that
   (a) $0 < |\beta(u)| \leq \kappa$ for all $u \in L$,
   (b) $|w| \geq \iota$ for all $w \in \beta(u)$ where $u \in L$,
   (c) $w_{[1,\iota]} \in \llbracket \lambda(u,v) \rrbracket$ for all $w \in \beta(v)$ where $(u,v) \in E$ and $v \in L$, and
   (d) $|w| < \iota$ for all $u \in F$ and all $w \in \beta(u)$.

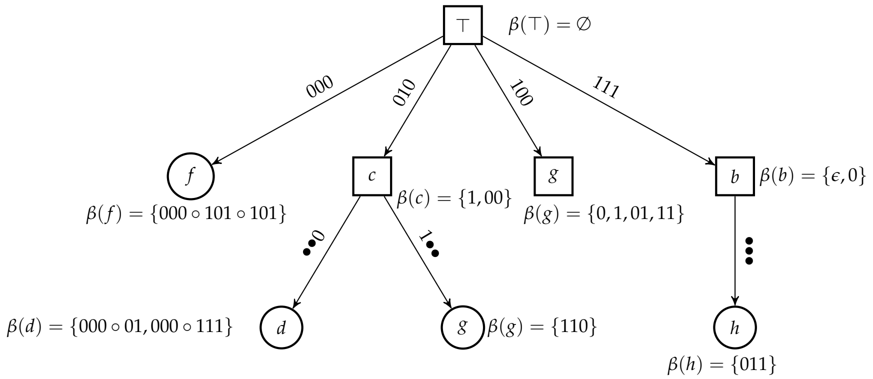
A PTrie example is given in Figure A.1a. We note particularly the difference between forwarding and leaf vertices. The bucket at a forwarding vertex contains the suffix of the string to be appended to the labels on the path from the root to the vertex (for example vertex c contains the bucket with the suffixes $\{1, 00\}$ that represent the strings $010 \circ 1$ and $010 \circ 00$). However, the bucket at a leaf vertex must first specify the concrete binary string that matches the extended binary string on its incoming edge, followed by the suffix of the string (for example the vertex b represents the strings 111 and $111 \circ 0$ as the first three bits of each string in the bucket of b must match the extended binary string $11\bullet$).

Before we introduce the main algorithms of the data structure, let us formally define the semantics of a PTrie as a set of strings that the PTrie represents.



$\top$
$\beta(\top) = \emptyset$
000
010
100
11•
$\beta(b) = \{111, 111 \circ 0\}$
$\beta(a) = \emptyset$
a
c
g
b
$\beta(c) = \{1, 00\}$
$\beta(g) = \{0, 1, 01, 11\}$
100
101
e
f
d
$\beta(d) = \{000 \circ 01, 110\}$
$\beta(e) = \{100, 100 \circ 10\}$
$\beta(f) = \{101 \circ 101\}$

(a) A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ with byte size $\iota = 3$ and maximal bucket size $\kappa = 2$ containing the binary strings $[\![\mathbb{P}]\!] = \{000 \circ 100, 000 \circ 100 \circ 10, 000 \circ 101 \circ 101, 010 \circ 1, 010 \circ 00, 010 \circ 000 \circ 01, 010 \circ 110, 100 \circ 0, 100 \circ 1, 100 \circ 01, 100 \circ 11, 111, 111 \circ 0\}$. Squares indicate forwarding vertices and circles indicate leaf-vertices. We let the labeling (λ) be implicitly indicated by the labeling on the edges. The path and suffix of the binary string $000 \circ 101 \circ 101$ is highlighted.



$\top$
$\beta(\top) = \emptyset$
000
010
100
111
f
c
g
b
$\beta(b) = \{\epsilon, 0\}$
$\beta(f) = \{000 \circ 101 \circ 101\}$
$\beta(c) = \{1, 00\}$
$\beta(g) = \{0, 1, 01, 11\}$
••0
1••
$\beta(d) = \{000 \circ 01, 000 \circ 111\}$
d
g
$\beta(g) = \{110\}$
h
$\beta(h) = \{011\}$

Fig. A.2: The PTrie from Figure A.1a after inserting $\{010 \circ 000 \circ 111, 111 \circ 011\}$ and removing $\{000 \circ 100, 000 \circ 100 \circ 10\}$.

Definition 2 (PTrie Semantics)
Let $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ be a PTrie. The semantics of $\mathbb{P}$, denoted by $[\![\mathbb{P}]\!] \subseteq \mathcal{B}^*$, is defined inductively as follows in the height of the tree so that $[\![\mathbb{P}]\!] = [\![\top]\!]$ and

$$[\![u \in L]\!] = \beta(u)$$
$$[\![u \in F]\!] = \beta(u) \cup \bigcup_{(u,v) \in E,\ v \in F} \{\lambda(u,v) \circ w \mid w \in [\![v]\!]\} \ \cup \bigcup_{(u,v) \in E,\ v \in L} [\![v]\!] \ .$$

3 Operations on PTrie

Let us assume a given PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string w. We shall now explain the algorithms for the basic set operations

- `Member`$(\mathbb{P}, w)$ for checking the existence of w in $\mathbb{P}$,
- `Insert`$(\mathbb{P}, w)$ for adding w into $\mathbb{P}$, and
- `Delete`$(\mathbb{P}, w)$ for removing w from $\mathbb{P}$.

The algorithms will use the following functions for manipulating PTries:

- `Find`$(\mathbb{P}, u, w)$ for searching from the vertex u for the binary string w,
- `Split`$(\mathbb{P}, v)$ for subdividing a vertex once its bucket size becomes larger than κ, and its inverse,
- `Merge`$(\mathbb{P}, v)$ for reducing the size of the PTrie by merging two vertices.

We also define the parent function (used by the `Split` and `Merge` algorithms) as $P : F \cup L \to F$ such that $P(v) = u$ where $u \in V$ is the unique vertex such that $(u, v) \in E$ and by agreement $P(\top) = \top$.

3.1 Member Algorithm

The algorithm for checking whether a binary string is already stored in a PTrie is presented in Algorithm 2 which is based on Algorithm 1 that searches for the presence of a binary string in a PTrie. This algorithm is also used for the insertion and deletion algorithms.

Algorithm 1 implements a search from a given vertex u following a given binary string as long as possible, until either a leaf-vertex is reached or no further match is possible and the algorithm returns the reached vertex and the suffix of the string w that could not be uniquely matched in the PTrie. This algorithm closely mimics the inductive definition of the semantics of PTrie in Definition 2.

Theorem 1
Algorithm 2 run on an input PTrie $\mathbb{P}$ and a binary string w terminates and returns tt if and only if $w \in [\![\mathbb{P}]\!]$.

Algorithm 1: `Find`$(\mathbb{P}, u, w)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$, a vertex $u \in V$ and a binary string $w$
Result: $(v, w')$ where $w'$ is a suffix of $w$ that cannot be any further matched by a (unique) path starting from $u$ and labeled with the longest possible prefix of $w$ and $v \in V$ is the vertex where this mismatch happens
begin
    if $|w| < \iota$ then
        return $(u, w)$
    $E_u = \{(u, v) \in E \mid w_{[1,\iota]} \in [\![\lambda(u, v)]\!]\}$;
    if $E_u = \emptyset$ then
        return $(u, w)$
    else
        Let $\{(u, v)\} = E_u$    // note that $|E_u| \leq 1$ due to Definition 1, case 5a
        if $v \in L$ then
            return $(v, w)$
        else
            return Find$(\mathbb{P}, v, w_{[\iota+1,|w|]})$
```

Algorithm 2: `Member`$(\mathbb{P}, w)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string $w$
Result: $tt$ if $w \in [\![\mathbb{P}]\!]$, else $ff$
begin
    $(v, w') \leftarrow$ Find$(\mathbb{P}, \top, w)$;
    if $w' \in \beta(v)$ then
        return $tt$
    else
        return $ff$
```

3.2 Insert Algorithm

Algorithm 3: `Insert`$(\mathbb{P}, w)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string $w$
Result: $\mathbb{P}'$ where $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \cup \{w\}$ and $\mathbb{P}'$ satisfies all conditions of
        Definition 1.
begin
    $(v, w') \leftarrow$ Find$(\mathbb{P}, \top, w)$;
    if $w' \in \beta(v)$ then
        return $\mathbb{P}$
    else
        if $v \in F$ then
            if $|w'| < \iota$ then
                $\beta(v) \leftarrow \beta(v) \cup \{w'\}$;
                return $(F, L, E, \top, \lambda, \beta)$
            else
                $\ell \leftarrow \underset{\ell' \in \Theta^\iota \text{ where } w'_{[1,\iota]} \in [\![\ell']\!]}{\arg\max} \begin{cases} 0 & \text{if } \exists u \in F \cup L \text{ s.t. } [\![\ell']\!] \cap [\![\lambda(v,u)]\!] \neq \emptyset \\ |[\![\ell']\!]| & \text{otherwise} \end{cases}$
                Make a fresh leaf vertex $u$;
                $L \leftarrow L \cup \{u\}$;
                $E \leftarrow E \cup \{(v, u)\}$;
                $\lambda(v, u) \leftarrow \ell$;
                $\beta(u) \leftarrow \{w'\}$;
                return $(F, L, E, \top, \lambda, \beta)$
        else
            $\beta(v) \leftarrow \beta(v) \cup \{w'\}$;
            if $|\beta(v)| \leq \kappa$ then
                return $(F, L, E, \top, \lambda, \beta)$
            else
                return Split$((F, L, E, \top, \lambda, \beta), v)$
```

We shall now focus on inserting a binary string w into a PTrie $\mathbb{P}$ as described in Algorithm 3. We start by matching the prefix of w from the root of the PTrie (line 2) to the vertex v from which we cannot follow the prefix of w any further. Either the vertex v is a forwarding vertex and if the unmatched suffix w' of w is shorter than ι, we insert it into the bucket of v at line 8 and we are done. If w' is on the other hand longer than ι, we need to create a new leaf vertex u and store w' in its bucket at line 16. The point is to label the edge (v, u) with the most general and non-conflicting label ℓ selected at line 11. In the second case where v is a leaf vertex, we add the suffix w' of w into the

Algorithm 4: `Split`$(\mathbb{P}, v)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a vertex $v \in L$ such that
      $\beta(v) > \kappa$.
Result: $\mathbb{P}'$ such that $[\![\mathbb{P}]\!] = [\![\mathbb{P}']\!]$ and $\mathbb{P}'$ satisfies all conditions of
        Definition 1
begin
    if $|[\![\lambda(P(v), v)]\!]| = 1$ then
        $F \leftarrow F \cup \{v\}$; $L \leftarrow L \setminus \{v\}$;
        $\beta(v) \leftarrow \{w_{[\iota+1,|w|]} \mid w \in \beta(v)$ and $|w| < 2\iota\}$;
        $B \leftarrow \{w_{[\iota+1,|w|]} \mid w \in \beta(v)$ and $|w| \geq 2\iota\}$;
        if $B = \emptyset$ then
            return $(F, L, E, \top, \lambda, \beta)$
        else
            Make a fresh leaf vertex $u$;
            $L \leftarrow L \cup \{u\}$; $E \leftarrow E \cup \{(v, u)\}$; $\lambda(v, u) \leftarrow \bullet^{\iota}$; $\beta(u) \leftarrow B$;
            if $|\beta(u)| \leq \kappa$ then
                return $(F, L, E, \top, \lambda, \beta)$
            else
                return Split$((F, L, E, \top, \lambda, \beta), u)$
    else
        Let $w \circ \bullet^{m} = \lambda(P(v), v)$ such that $w \in \{0,1\}^*$ and $m > 0$.
        $\ell_0 \leftarrow w0 \circ \bullet^{m-1}$; $\ell_1 \leftarrow w1 \circ \bullet^{m-1}$;
        $B_0 = \{w \in \beta(v) \mid w_{[1,\iota]} \in [\![\ell_0]\!]\}$; $B_1 = \{w \in \beta(v) \mid w_{[1,\iota]} \in [\![\ell_1]\!]\}$;
        if $B_0 \neq \emptyset$ and $B_1 \neq \emptyset$ then
            Make a fresh leaf vertex $u$;
            $L \leftarrow L \cup \{u\}$, $E \leftarrow E \cup \{(P(v), u)\}$;
            $\lambda(P(v), v) \leftarrow \ell_0$; $\lambda(P(v), u) \leftarrow \ell_1$;
            $\beta(v) \leftarrow B_0$; $\beta(u) \leftarrow B_1$;
            return $(F, L, E, \top, \lambda, \beta)$
        else
            if $B_0 \neq \emptyset$ then
                $\lambda(P(v), v) \leftarrow \ell_0$;
            else
                $\lambda(P(v), v) \leftarrow \ell_1$;
            return Split$((F, L, E, \top, \lambda, \beta), v)$
```

bucket at line 19 and should the size of the bucket exceed the maximum size κ, we call the function `Split` at line 23 to balance the PTrie.

An example of inserting two strings is given in Figure A.2. The insertion of the string $010 \circ 000$ causes the creation of the sibling g for the vertex d and splitting of the label ●●● into 0●● and 1●●. The insertion of $111 \circ 011$ implies that the leaf vertex b turns into a forwarding vertex while we create a fresh leaf vertex h and adjust the buckets accordingly.

Theorem 2
Algorithm 3 run on an input PTrie $\mathbb{P}$ and a binary string w terminates and returns a PTrie $\mathbb{P}'$ such that $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \cup \{w\}$.

3.3 Delete Algorithm

We here discuss the algorithm for removing a binary string w from a PTrie $\mathbb{P}$ as described in Algorithm 5. As with the insertion algorithm, the `Delete` algorithm may call the function `Merge` defined in Algorithm 6—a function that attempts to revert divisions previously made by the `Split` algorithm.

Initially we try to match the prefix of w to a unique path from the root of the PTrie (line 2 of `Delete`) and we let v be the vertex reached at the end of this prefix and w' be the unmatched suffix of w. If w did not exist in the PTrie, we return the unaltered PTrie at line 4. Otherwise we remove w' from the bucket of v. Either $v \in L$, and we attempt to reduce the PTrie (line 22), or we are in the more complex situation where $v \in F$. If $v \in F$ and v has no children (as illustrated by vertex g in Figure A.1a) then we can turn v into a leaf node (line 13) and attempt to reduce the size of the PTrie (line 15). However, as $\top$ has to stay in F, we return $\mathbb{P}$ if $v = \top$ (line 10). If $|\beta(v)| > \kappa$ then turning v into a leaf-node would violate condition 6a in Definition 1 and we therefore return the PTrie as it is (line 12). If $v \in F$ and v has only a single child such that this child is not a forwarding vertex, and merging v with its child will not violate condition 6a in Definition 1, then we also attempt to merge (line 18). Otherwise just return PTrie without further modifications (line 20).

An example of removing two different strings from our running example is presented in Figure A.2. The removal causes the leaf vertex e to get an empty bucket implying that it gets removed. This change in turn propagates to the vertex a that is also removed and its bucket content is merged with that of f.

Theorem 3
Algorithm 5 given a PTrie $\mathbb{P}$ and a binary string w terminates and returns a PTrie $\mathbb{P}'$ such that $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \setminus \{w\}$.

Algorithm 5: `Delete`$(\mathbb{P}, w)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string $w$
Result: $\mathbb{P}'$ where $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \setminus \{w\}$ and $\mathbb{P}'$ satisfies all conditions of
        Definition 1
begin
    $(v, w') \leftarrow$ Find$(\mathbb{P}, \top, w)$;
    if $w' \notin \beta(v)$ then
        return $\mathbb{P}$
    else
        $\beta(v) \leftarrow \beta(v) \setminus \{w'\}$;
        if $v \in F$ then
            if $v$ has no children then
                if $v = \top$ then
                    return $(F, L, E, \top, \lambda, \beta)$
                if $|\beta(v)| > \kappa$ then
                    return $(F, L, E, \top, \lambda, \beta)$
                $L \leftarrow L \cup \{v\}$; $F \leftarrow F \setminus \{v\}$;
                $\beta(v) \leftarrow \{\lambda(P(v), v) \circ w \mid w \in \beta(v)\}$;
                return Merge$((F, L, E, \top, \lambda, \beta), v)$
            else
                if $v$ has exactly one child $u$ and $u \in L$ then
                    return Merge$((F, L, E, \top, \lambda, \beta), u)$
                else
                    return $(F, L, E, \top, \lambda, \beta)$
        else
            return Merge$((F, L, E, \top, \lambda, \beta), v)$
```

Algorithm 6: `Merge`$(\mathbb{P}, v)$

```
Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a vertex $v \in L$
Result: $\mathbb{P}'$ s.t. $[\![\mathbb{P}]\!] = [\![\mathbb{P}']\!]$ and $\mathbb{P}'$ satisfies all conditions of Definition 1
begin
    if $\lambda(P(v), v) = \bullet^{\iota}$ then
        if $|\beta(v)| = 0$ and $|\beta(P(v))| > \kappa$ then
            $E \leftarrow E \setminus \{(P(v), v)\}$; $L \leftarrow L \setminus \{v\}$;
            return $(F, L, E, \top, \lambda, \beta)$
        if $P(v) = \top$ then
            return $\mathbb{P}$
        else
            $u \leftarrow P(v)$; $\ell \leftarrow \lambda(u, v)$;
            if $|\beta(v)| + |\beta(u)| \leq \kappa$ then
                $E \leftarrow (E \cup \{(P(u), v)\}) \setminus \{(P(u), u), (u, v)\}$; $F \leftarrow F \setminus \{u\}$;
                $\lambda(P(u), v) \leftarrow \ell$;
                $\beta(v) \leftarrow \{\ell \circ w \mid w \in \beta(v) \cup \beta(u)\}$;
                return Merge$((F, L, E, \top, \lambda, \beta), v)$
            else
                return $(F, L, E, \top, \lambda, \beta)$
    else
        Let $b_1 \dots b_n \bullet^m = \lambda(P(v), v)$;
        $\ell \leftarrow b_1 \dots b_{n-1} \bullet^{m+1}$;
        $V \leftarrow \{(P(v), u) \in E \mid u \neq v$ and $[\![\lambda(P(v), u)]\!] \cap [\![\ell]\!] \neq \emptyset\}$;
        if $V = \emptyset$ then
            $\lambda(P(v), v) \leftarrow \ell$;
            return Merge$((F, L, E, \top, \lambda, \beta), v)$
        else
            if $V = \{u\}$ for some $u \in L$ and $|\beta(v)| + |\beta(u)| \leq \kappa$ then
                $\lambda(P(v), v) \leftarrow \ell$;
                $\beta(v) \leftarrow \beta(v) \cup \beta(u)$;
                $E \leftarrow E \setminus \{(P(u), u)\}$; $L \leftarrow L \setminus \{u\}$;
                return Merge$((F, L, E, \top, \lambda, \beta), v)$
            else
                return $\mathbb{P}$
```

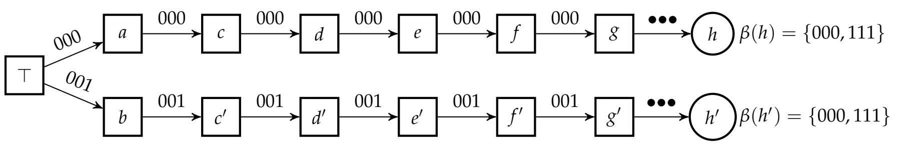


Fig. A.3: A worst-case scenario for PTries with $\iota = 3$ and $\kappa = 2$ containing 4 binary strings $\{000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000, 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 111, 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 000, 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 111\}$

4 Implementation

The PTrie interface is implemented as an open source `C++` library and it is available at `https://github.com/petergjoel/ptrie` under the GPL version 3 license. Apart from the implementation of all the basic set operations on PTries as described in this paper (implemented in `ptrie::set`), two other flavors of PTries exist: one providing unique and non-changing identifiers for inserted elements (`ptrie::stable_set`) and one providing the functionality of a map, combined with non-changing identifiers (`ptrie::map`)[1]. The source code provides further documentation and information.

Let us now settle some implementation details. We currently use the bucket size $\kappa = 64$ and the byte size $\iota = 8$, following conventions for standard byte-sizes. As modern architectures do not support addressing nor allocation of memory areas of less than a single byte, our implementation of PTries allows only the insertion of binary strings with bit-lengths that are a multiple of ι. Furthermore, to avoid frequent splits and re-merging of PTries, the `Delete` and `Merge` algorithms initiate the balancing of PTrie only once the buckets become smaller than $\frac{\kappa}{3}$, as opposed to the constant κ used in the pseudocode. The experimental evaluations point towards a slightly worse memory utilization at the exchange of less frequent re-balancing of the PTrie.

Regarding the memory for storing vertices of a PTrie, forwarding vertices are implemented as directly indexed tables with 64-bit indexes and with some additional book-keeping information they occupy 2064 bytes. Leaf vertices are, on the other hand, lightweight constructions taking up only 16 bytes. The current implementation of PTrie prefixes all inserted binary strings with their length (using two additional bytes). In our experience, such an addition generally improves the performance and reduces memory-consumption. Moreover, as we aim at making the PTries fast, the speed optimization can occasionally imply an increased memory consumption for some very specific sets of binary strings, as demonstrated in Figure A.3, where just a few

[1] Both these extension come with a smaller overhead in run-time and memory. Also, currently neither of these extensions support `Delete`.

strings create a long sequence of memory-demanding forwarding vertices. This implies that long, almost similar, binary strings which differ only at the beginning and at the end will make the PTrie perform badly in terms of memory.

Hence, depending on the specific application domain, the concrete encoding of the states into binary strings can have an effect on the PTrie performance. As a heuristic attempt to improve prefix-sharing of Petri net markings (an experiment discussed in detail in the next section), we first statically order places in the models by the number of incoming and outgoing arcs. Each such marking is then encoded according to a number of schemes in order to minimize its length. The schemes all fall in one of three categories: either only non-empty places are stored (with the least amount of bits), or a bit-vector is used to represent non-empty places in the fixed ordering of places, or we use a combination of the two previous schemes. To determine which way a marking was encoded, we prefix the encoding with a 8-bit header describing the exact encoding scheme that is employed. Details of the encoding-scheme can be found at `https://bit.ly/AlignedEncodercpp`.

5 Experimental Evaluation

We conducted two series of experiments comparing our PTrie implementation against `google::sparse_hash_set` and `google::dense_hash_set` by Google[2], generally regarded as the state-of-the-art [2, 3] space-efficient and time-efficient, respectively, implementations of sets based on hashing. We employ `jemalloc` [21] for memory allocation and `MurmurHash64A`[3] as hash-function for the hash-map implementations. In our evaluation we omit the `std::unordered_set` implementation from the standard library of `C++14` as it was consistently outperformed by the Google implementations (see [2, 3] for further benchmarks).

In the first round of experiments, we test the speed and memory requirements of insertion, deletion and lookups, simulating a workload using pseudo-random 64-bit integers (with the same seed so that the same sequence of numbers is inserted/deleted/checked in all test setups). In the second round of experiments, we modify the verification-tool `verifypn` [22][4] that is distributed as a part of the Petri net verification tool Tappaal [23, 24], and we conduct an exhaustive exploration of the full state-space of large Petri net models used at the MCC'16 competition [25]. All experiments were conducted on AMD Opteron 6376 Processors and limited to 120GB of RAM and 4 days of computation.

[2]Both available at `https://github.com/sparsehash/sparsehash`.

[3]Available at `https://github.com/aappleby/smhasher/wiki/MurmurHash2`.

[4]Available at `https://code.launchpad.net/verifypn`.

E	`ptrie`	`dense`	`sparse`	`ptrie/dense`	`ptrie/sparse`
			Insert		
28	437.2	386.0	569.1	113%	77%
29	869.0	757.1	1111.3	115%	78%
30	1749.2	1540.2	2326.7	114%	75%
31	3572.0	3081.7	4785.6	116%	75%
32	7184.6	6126.6	9963.6	117%	72%
average	2762.4	2378.3	3751.2	115%	75%
			Insert+50%Delete		
28	751.5	744.1	742.7	101%	101%
29	1516.8	1494.3	1461.9	102%	104%
30	3038.5	3032.1	2997.8	100%	101%
31	6392.3	5837.4	6150.1	110%	104%
32	13356.1	11701.0	13115.5	114%	102%
average	5011.1	4561.8	4893.6	105%	102%
			Insert+50%Member		
28	709.6	591.2	771.0	120%	92%
29	1468.4	1219.3	1583.8	120%	93%
30	2829.1	2363.0	3195.4	120%	89%
31	5839.8	4707.6	6597.3	124%	89%
32	12244.2	9473.2	13676.5	129%	90%
average	4618.2	3670.8	5164.8	123%	90%

Table A.1: Time in seconds for the simulated workload experiments

5.1 Simulated Workload

We conduct three sets of experiments called *Insert*, *Insert+50%Delete* and *Insert+50%Member*, all scaled by the number 2^E of pseudorandomly generated and inserted elements into the set implementation. In the *Insert* experiment, we iteratively insert 2^E binary numbers encoded as 64-bit unsigned integers. In the *Insert+50%Delete* and *Insert+50%Member* experiments, after each insertion, we choose with 50% probability whether to execute a `Delete` or `Member` operation, respectively. In *Insert+50%Delete*, we randomly draw for deletion an element that was previously inserted, but we do not check whether the element was already removed or not. This implies that with 33% probability it tries to remove a nonexisting element. In *Insert+50%Member*, we randomly select an element for which we do an `Member` operation, such that about one half of the existence checks are with a positive answer.

The results measuring the speed of operations are presented in Table A.1.

E	ptrie	dense	sparse	ptrie/dense	ptrie/sparse
Insert and *Insert+50%Member*					
28	2033.6	6151.7	4239.6	33%	48%
29	3197.6	12295.7	8455.9	26%	38%
30	6115.7	24583.7	16923.0	25%	36%
31	10827.6	49159.7	33908.2	22%	32%
32	37839.6	98311.7	67757.7	39%	56%
average	12002.8	38100.5	26256.9	29%	42%
Insert+50%Delete					
28	1935.8	6157.7	3032.3	31%	64%
29	3383.8	12301.6	5966.5	28%	57%
30	6960.7	24589.6	12057.8	28%	58%
31	13488.9	49165.6	24914.0	27%	54%
32	37493.6	98317.6	68195.0	38%	55%
average	12652.6	38106.4	22833.1	31%	57%

Table A.2: Memory in megabyte for the simulated workload experiments

For pure insertions, PTries are on average about 15% slower than `dense_hash` but 25% faster than `sparse_hash`. When we add deletions, PTries are about 5% slower than `dense_hash` and essentially comparable with `sparse_hash` (on average just 2% slower). In the last experiment where we add frequent queries on the presence of a string in the set, `dense_hash` becomes 23% faster but on the other hand PTries are by 10% faster than `sparse_hash`. In summary, `sparse_hash` is in general slower or equal in speed with PTrie, while `dense_hash` is the fastest of the three data structures.

However, we can see in Table A.2 a significant reduction of the memory-footprint in all of the experiments (*Insert+50%Member* is not included as its memory usage is identical with pure inserts). PTries deliver about 70% of the memory reduction compared to `dense_hash` and between 42%–57% reduction compared to `sparse_hash` (depending on whether deletions are included or not).

In conclusion, PTrie is the most memory efficient data structure that is faster or at worst equal in speed with `sparse_hash`. The fastest set implementation is `dense_hash`, however, at the cost of a large memory overhead. We remark that the drop in relative memory-reduction in the *Insert* experiment when $E = 32$ is due to the creation of a large number of forwarding vertices—this occurs with high probability for truly random strings when E is a multiple of 8.

Model	`ptrie`	`dense`	`sparse`	`ptrie/dense`	`ptrie/sparse`	10^6 states	10^6operations
a	408.7	517.8	680.5	79%	60%	42.7	486.9
b	12882.8	15888.9	19163.1	81%	67%	693.8	2151.2
c	2337.9	2839.3	3693.7	82%	63%	131.1	5553.7
d	244.6	292.3	526.6	84%	46%	113.3	863.5
e	589.2	693.6	1141.2	85%	52%	261.2	2010.6
f	69451.0	68601.0	70879.3	101%	98%	320.6	22339.6
g	16.4	16.1	20.6	102%	79%	3.0	24.9
h	318.7	312.8	389.7	102%	82%	48.9	354.4
i	5011.5	4917.2	5812.8	102%	86%	406.0	3051.2
j	69.5	67.9	78.3	102%	89%	11.5	66.8
k	25.7	20.4	21.3	126%	121%	1.7	6.7
l	41.4	32.8	33.8	126%	123%	2.8	13.2
m	439.9	345.7	647.9	127%	68%	164.4	1047.5
n	78.3	60.9	112.4	129%	70%	32.2	199.3
o	263.9	163.6	185.2	161%	143%	17.4	108.4
avg	4608.4	4482.2	5380.0	103%	86%	289.3	3195.2

Table A.3: Time in seconds for the 5 best, 5 median and 5 worst Petri net models, ordered by the performance of `ptrie` relative to `dense_hash`. Legend for the models: a=Angiogenesis-PT-05, b=PolyORBNT-PT-S05J20, c=Diffusion2D-PT-D05N010, d=SmallOperatingSystem-PT-MT0128DC0032, e=SmallOperatingSystem-PT-MT0128DC0064, f=ARMCacheCoherence-PT-none, g=TCPcondis-PT-05, h=AutoFlight-PT-01b, i=SimpleLoadBal-PT-10j=ResAllocation-PT-R020C002, k=ParamProductionCell-PT-5, l=ParamProductionCell-PT-0, m=SwimmingPool-PT-04, n=SwimmingPool-PT-03 and o=IOTPpurchase-PT-C05M04P03D02.

5.2 Real Workload by Petri Net Model Checking

In order to test the PTrie performance on a realistic scenario, we integrate PTrie as a part of a Petri net model checker. We replace the state-storage of the verification algorithm used by `verifypn` with the respective set implementations (by using an encoding of Petri net markings to binary strings as discussed in the implementation section). We then conduct an exhaustive state-space search on the P/T nets from the MCC'16 competition. To reduce the impact of auxiliary data structures used by the algorithm, we conduct the search with two different search-strategies (breadth first and depth first), and we report the minimum of the memory and time-consumption from either of these searches. We consider in total 94 Petri nets with a nontrival but feasible state-space size. More concretely, we selected all nets with more than 10^6 and less than 10^{10} reachable markings. Out of these 94 nets, PTrie-based variant completed 89 test-cases, ran out of memory on 4 models and timed out on a single instance. The `dense_hash`-based model checker completed only a subset of the test-cases solved by PTrie and exceeded the memory-bound for additional 9 nets. A similar performance was achieved by `sparse_hash` that also completed only a subset of problems solved by PTrie but exceeded the memory for 7 additional nets. In the summary tables we consider so only 80 state-space searches that were completed by all three set-implementations.

Model	ptrie	dense	sparse	ptrie/dense	ptrie/sparse	10^6 states	10^6operations
a	2815.6	16481.6	15063.5	17%	19%	435.3	2983.9
b	2817.6	16481.5	15063.6	17%	19%	432.9	2961.9
c	2855.6	16481.6	15063.6	17%	19%	432.9	2961.9
d	2883.6	16481.6	15063.6	18%	19%	435.3	2983.9
e	14707.6	65901.4	60223.4	22%	24%	1885.4	15271.5
f	16579.6	35751.6	33971.6	46%	49%	1005.9	12032.2
g	21283.5	44344.2	43515.5	48%	49%	896.3	3363.7
h	7539.6	20667.6	15373.6	37%	49%	347.6	1271.7
i	7541.6	20667.5	15375.5	37%	49%	347.6	1271.7
j	1463.6	5203.6	2965.6	28%	49%	68.2	1286.2
k	133.7	169.6	129.6	79%	103%	2.8	13.2
l	879.7	1303.6	763.6	68%	115%	17.4	108.4
m	105.7	91.6	81.6	115%	130%	1.7	6.7
n	93.6	87.5	71.6	107%	131%	1.5	5.9
o	147.7	169.6	111.6	87%	132%	2.4	9.8
avg	5150.6	13339.3	11056.9	39%	47%	289.3	3195.2

Table A.4: Memory in megabytes for the 5 best, 5 median and 5 worst Petri net models, ordered by the perforamce of PTrie relative to `sparse_hash`. Legend for the models: a=DNAwalker-PT-06track28RL, b=DNAwalker-PT-04track28LL, c=DNAwalker-PT-07track28RR, d=DNAwalker-PT-05track28LR, e=DNAwalker-PT-12ringLLLarge, f=Kanban-PT-0010, g=BridgeAndVehicles-PT-V50P50N20, h=BridgeAndVehicles-PT-V50P20N10, i=BridgeAndVehicles-PT-V50P50N10, j=AutoFlight-PT-05a, k=ParamProductionCell-PT-0, l=IOTPpurchase-PT-C05M04P03D02, m=ParamProductionCell-PT-5, n=ParamProductionCell-PT-3 and o=ParamProductionCell-PT-4.

In Table A.3 we can see that PTries are on average as fast as the fastest hash-map implementation via `dense_hash` with only a 3% overhead on average, while PTries provide significant 14% speedup compared to `sparse_hash`. There seems to be no correlation between the number of states/markings (equivalent to the number of insert operations) and the relative performance achieved. With respect to memory usage, the experiments confirm the effectiveness of PTrie as seen in Table A.4. In general we observe a significant memory footprint reduction by up to 81% compared to `sparse_hash` and on average by 53%. The reductions in the case of `dense_hash` are as expected even higher. We can notice that higher relative memory reduction occurs when we use PTries for models with a larger number of reachable states/-markings, confirming that PTries are particularly beneficial for memory demanding applications like model checking. We can observe that for some instances of prefix-sharing, PTries are particularly effective as demonstrated by the "DNAwalker"-cases (using less than 7 bytes per stored marking versus 36 for `sparse_hash`), while ineffective for the "ParamProductionCell"-cases (using more than 64 bytes per marking versus 49 for `sparse_hash`). Here we experience the situation described in Figure A.3 caused by the ordering of places in the binary encoding of markings and by the fact that there is large number of places where the number of tokens hardly ever changes during the computation.

6 Conclusion

We presented PTrie, a novel data structure for compressing sets of binary strings while providing fast operations for element addition/removal and containment checks. Compared to the state-of-the-art alternatives that either trade memory savings for time (`google::sparse_hash_set`), or focus on optimizing the speed of operations (`google::dense_hash_set`), our data structure improves the performance of `sparse_hash` both in terms of memory as well as time. Compared to `dense_hash`, we are on average 5-23% slower on random strings, while only 3% slower when storing strings coming from a real application domain, and at the same time we provide 60-70% of memory reduction.

In the future work, we plan to provide an efficient parallelization of the PTries for the use in multi-core architectures, and extend the set of basic operators with intersection, union and difference. Even though these additional operations are not necessary for explicit model checking applications, they may find other application domains and tree-based design of PTries seems to be suitable for this purpose. Finally, a research of tree-walking algorithms for PTries, facilitating complex searches through the elements of the set, are of high interest too.

Acknowledgements. We acknowledge the support from Sino-Danish Basic Research Center IDEA4CPS, the Innovation Fund Denmark center DiCyPS, and the ERC Advanced Grant LASSO. The third author is partially affiliated with FI MU in Brno.

References

[1] P. G. Jensen, K. G. Larsen, J. Srba, M. G. Sørensen, and J. H. Taankvist, "Memory efficient data structures for explicit verification of timed systems," in *NASA Formal Methods: 6th International Symposium*, ser. LNCS. Springer, 2014, vol. 8430, pp. 307–312. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06200-6_26

[2] Timonk, "Big memory, part 3.5: Google sparsehash!" https://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/, 2011, accessed: 2017-01-20. [Online]. Available: https://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/

[3] N. Welch, "Hash table benchmarks," http://incise.org/hash-table-benchmarks.html, accessed: 2017-01-20. [Online]. Available: http://incise.org/hash-table-benchmarks.html

[4] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960. [Online]. Available: http://doi.acm.org/10.1145/367390.367400

[5] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[6] G. Gwehenberger, "Anwendung einer binären verweiskettenmethode beim aufbau von listen/use of a binary tree structure for processing files," *it-Information Technology*, vol. 10, no. 1-6, pp. 223–226, 1968.

[7] N. Askitis and R. Sinha, "HAT-trie: A cache-conscious trie-based data structure for strings," in *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*. Australian Computer Society, Inc., 2007, pp. 97–105.

[8] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Acm Sigplan Notices*, vol. 47 (8). ACM, 2012, pp. 151–160.

[9] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys," *ACM Transactions on Information Systems*, vol. 20, pp. 192–223, 2002.

[10] P. Bagwell, "Ideal hash trees," *Es Grands Champs*, vol. 1195, 2001.

[11] M. Renaud, "Trie (aka. prefix tree)," https://github.com/m-renaud/trie, accessed: 2017-04-19.

[12] J. Yang, "An implementation of two-trie and tail-trie using double array," https://github.com/jianingy/libtrie, accessed: 2017-04-19.

[13] D. C. Jones, "HAT-trie implementation," https://github.com/dcjones/hat-trie, accessed: 2017-04-19.

[14] cplusplus.com, "C++ set implementation reference," http://www.cplusplus.com/reference/set/set/, accessed: 2017-01-20.

[15] ——, "C++ map implementation reference," http://www.cplusplus.com/reference/map/map/, accessed: 2017-01-20.

[16] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[17] A. Laarman, J. van de Pol, and M. Weber, "Parallel recursive state compression for free," in *Model Checking Software: 18th International SPIN Workshop*, ser. LNCS, vol. 6823. Springer, 2011, pp. 38–56. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22306-8_4

[18] P. Ročkai, V. Štill, and J. Barnat, *Techniques for Memory-Efficient Model Checking of C and C++ Code*, ser. LNCS. Cham: Springer, 2015, vol. 9276, pp. 268–282. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22969-0_19

[19] S. Evangelista and J.-F. Pradat-Peyre, "Memory efficient state space storage in explicit software model checking," in *Model Checking Software: 12th International SPIN Workshop*, ser. LNCS, vol. 3639. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 43–57. [Online]. Available: http://dx.doi.org/10.1007/11537328_7

[20] K. Wolf, "Running LoLA 2.0 in a model checking competition," *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, vol. 9930, pp. 274–285, 2016.

[21] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[22] J. Jensen, T. Nielsen, L. Oestergaard, and J. Srba, "TAPAAL and reachability analysis of P/T nets," *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, vol. 9930, pp. 307–318, 2016.

[23] A. David, L. Jacobsen, M. Jacobsen, K. Jørgensen, M. Møller, and J. Srba, "TAPAAL 2.0: Integrated development environment for timed-arc Petri nets," in *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference*, ser. LNCS, vol. 7214. Springer, 2012, pp. 492–497.

[24] J. Byg, K. Y. Jørgensen, and J. Srba, "TAPAAL: Editor, simulator and verifier of timed-arc Petri nets," in *Automated Technology for Verification and Analysis: 7th International Symposium*, ser. LNCS, vol. 5799. Springer, 2009, pp. 84–89. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04761-9_7

[25] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf, "Complete Results for the 2016 Edition of the Model Checking Contest," http://mcc.lip6.fr/2016/results.php, June 2016.

Paper B

Refinement of Trace Abstraction for Real-Time Programs

Franck Cassez, Peter Gjøl Jensen and Kim Guldstrand Larsen

The paper has been published in the
Proceedings of Reachability Problems 2017, LNCS Vol. 10506, pp. 42–58, 2017.

Abstract

Real-time programs are made of instructions that can perform assignments to discrete and real-valued variables. They are general enough to capture interesting classes of timed systems such as timed automata, stopwatch automata, time(d) Petri nets and hybrid automata. We propose a semi-algorithm using refinement of trace abstractions to solve both the reachability verification problem and the parameter synthesis problem for real-time programs. We report on the implementation of our algorithm and we show that our new method provides solutions to problems which are unsolvable by the current state-of-the-art tools.

1 Introduction

Model-checking is a widely used formal method to assist in verifying software systems. A wide range of model checking techniques and tools are available and there are numerous successful applications in the safety-critical industry and the hardware industry – in addition the approach is seeing an increasing adoption in the general software engineering community. The main limitation of this formal verification technique is the so-called *state explosion problem*. *Abstraction refinement techniques* were introduced to overcome this problem. The most well-known technique is probably the *Counter Example Guided Abstraction Refinement* (CEGAR) method pioneered by Clarke *et al.* [1]. In this method the state-space is abstracted with predicates on the concrete values of the program variables. The (counter-example guided) *refinement of trace abstraction* (TAR) method was proposed recently by Heizmann *et al.* [2, 3] and is based on abstracting the set of traces of a program rather than the set of states. These two techniques have been widely used in the context of software verification. Their effectiveness and versatility in verifying *qualitative* (or functional) properties of C programs is reflected in the most recent *Software Verification* competition results [4, 5].

Analysis of timed systems. Reasoning about *quantitative* properties of programs requires extended modeling features like real-time clocks. *Timed Automata* [6] (TA), introduced by Alur and Dill in 1989, is a very popular formalism to model real-time systems with dense-time clocks. Efficient symbolic model checking techniques for TA are implemented in the real-time model-checker UPPAAL [7]. Extending TA, e.g., with the ability to stop and resume clocks (stopwatches), leads to undecidability of the reachability problem [8, 9]. Semi-algorithms have been designed to verify *hybrid systems* (extended classes of TA) and are implemented in a number of dedicated tools [10–12]. However, a common difficulty with the analysis of quantitative properties of timed automata and extensions thereof is that ad-hoc data structures are needed for each extension and each type of problem. As a consequence, the analysis tools have special-purpose efficient algorithms and data structures suited and optimized only towards their specific problem and extension.

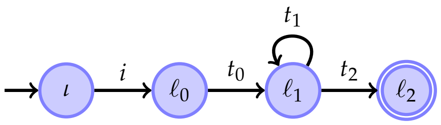
In this work we aim to provide a uniform solution to the analysis of timed systems by designing a generic semi-algorithm to analyze real-time programs which semantically captures a wide range of specification formalisms, including hybrid automata. We demonstrate that our new method provides solutions to problems which are unsolvable by the current state-of-the-art tools. We also show that our technique can be extended to solve specific problems like robustness and parameter synthesis.

Related work. The *refinement of trace abstractions* (TAR) was proposed by Heizmann *et al.* [2, 3]. It has not been extended to the verification of real-time systems. Wang *et al.* [13] proposed the use of TAR for the analysis of timed automata. However, their approach is based on the computation of the standard *zones* which comes with usual limitations: it is not applicable to extensions of TA (e.g., stopwatch automata) and can only discover predicates that are zones. Their approach has not been implemented and it is not clear whether it can outperform state-of-the-art techniques e.g., as implemented in UPPAAL. Dierks *et al.* [14] proposed a CEGAR based method for Timed Systems. To the best of our knowledge, this method got limited attention in the community.

Tools such as UPPAAL [7], SPACEEX [11], HYTECH [12], PHAVER [10], VERIFIX [15], SYMROB [16] and IMITATOR [17] all rely on special-purpose polyhedra libraries to realize their computation.

Our technique is radically different to previous approaches and leverages the power of SMT-solvers to discover non-trivial invariants for the class of hybrid automata. All the previous analysis techniques compute, reduce and check the state-space either up-front or on-the-fly, leading to the construction of significant parts of the state-space. In contrast our approach is an abstraction refinement method and the refinements are built by discovering non-trivial program invariants that are not always expressible using zones, or polyehdra. This enables us to successfully analyze (terminate) instances of non-decidable classes like stopwatch automata. A simple example is discussed in Section 2.

Our contribution. In this paper, we propose a refinement of trace abstractions (TAR) technique to solve the reachability problem and the parameter synthesis problem for real-time programs. Our approach combines an automata-theoretic framework and state-of-the-art Satisfiability Modulo Theory (SMT) techniques for discovering program invariants. We demonstrate on a number of case-studies that this new approach can compute answers to



Edge	Guard	Update	Rate
i	*True*	`x:=y:=z:=0`	`dy/dt=1`
t_0	*True*	`z:=0`	`dy/dt=0`
t_1	`x==1`	`x:=0`	`dy/dt=0`
t_2	`x-y>=1 and z<1`	-	`dy/dt=0`

Fig. B.1: Finite Automaton A_1

problems unsolvable by special-purpose tools and algorithms in their respective domain.

2 Motivating Example

The finite automaton A_1 (Fig. B.1), accepts the regular language $\mathcal{L}(A_1) = i.t_0.t_1^*.t_2$. By interpreting the labels of A_1 according to the Table in Fig. B.1, we can view it as a stopwatch automaton with 2 clocks, x and z, and one stopwatch y (the variables). Each label defines a *guard* g (a Boolean constraint on the variables), an *update* u which is a (discrete) assignment to the variables, and a *rate* (vector) r that defines the derivatives of the variables.[1] We associate with a sequence $w = a_0.a_1.\cdots.a_n \in \mathcal{L}(A_1)$, a (possibly empty) set of *timed words*, $\tau(w)$, of the form $(a_0, \delta_0).\cdots(a_n, \delta_n)$ where $\delta_i \geq 0, i \in [0..n]$. For instance, the timed words associated with $i.t_0.t_2$ are of the form $(i, \delta_0).(t_0, \delta_1).(t_2, \delta_2)$, for all $\delta_i \in \mathbb{R}_{\geq 0}, i \in \{0, 1, 2\}$ such that following constraints can be satisfied:

$$x_0 = y_0 = z_0 = \delta_0 \wedge \delta_0 \geq 0 \tag{P_0}$$

$$x_1 = x_0 + \delta_1 \wedge y_1 = y_0 \wedge z_1 = \delta_1 \wedge \delta_1 \geq 0 \tag{P_1}$$

$$x_1 - y_1 \geq 1 \wedge z_1 < 1 \wedge x_2 = x_1 + \delta_2 \wedge y_2 = y_1 \wedge z_2 = z_1 + \delta_2 \wedge \delta_2 \geq 0 \tag{P_2}$$

The initial values of the variables x, y, z (in location ι, source of edge i) are denoted x_{-1}, y_{-1}, z_{-1} and are unconstrained. Hence we assume that the initial predicate on the variables x_{-1}, y_{-1}, z_{-1} is $P_{-1} = \mathit{True}$. P_0 must be satisfied after taking i and letting time progress for $\delta_0 \geq 0$ time units, which is enforced by a constraint on the variables[2] x_0, y_0, z_0 that stand for the values of x, y, z after taking i; similarly $P_0 \wedge P_1$ must hold after $i.t_0$ and $P_0 \wedge P_1 \wedge P_2$ after $i.t_0.t_2$. Hence the set of timed words associated with $i.t_0.t_2$ is not empty iff $P_0 \wedge P_1 \wedge P_2$ is satisfiable. The *timed language*, $\mathcal{TL}(A_1)$, accepted by A_1

[1]As x and z are clocks their rate is always 1 and omitted in the Table.
[2]If x was not reset by i, we would have a constraint $x_0 = x_{-1}$, with x_{-1} unconstrained.

Sequence	PHAVER	UPPAAL
$i.t_0$	$z = x - y \wedge 0 \leq z \leq x$	$0 \leq y \leq x \wedge 0 \leq z \leq x$
$i.t_0.t_1$	$z = x - y + 1 \wedge 0 \leq x \leq z \leq x + 1$	$0 \leq z - x \leq 1 \wedge 0 \leq y$
$i.t_0.(t_1)^2$	$z = x - y + 2 \wedge 0 \leq x \leq z - 1 \leq x + 1$	$1 \leq z - x \leq 2 \wedge 0 \leq y$
$i.t_0.(t_1)^3$	$z = x - y + 3 \wedge 0 \leq x \leq z - 2 \leq x + 1$	$2 \leq z - x \leq 3 \wedge 0 \leq y$
$\dots$	$\dots$	$\dots$
$i.t_0.(t_1)^k$	$z = x - y + k \wedge 0 \leq x \leq z - k + 1 \leq x + 1$	$k - 1 \leq z - x \leq k \wedge 0 \leq y$
$\dots$	$\dots$	$\dots$

Table B.1: Symbolic representation of reachable states after a sequence of instructions. UPPAAL concludes that $\mathcal{TL}(A_1) \neq \varnothing$ due to the over-approximation using DBMs. PHAVER does not terminate.

is the set of timed words associated with all the words w accepted by A_1 i.e., $\mathcal{TL}(A_1) = \cup_{w \in \mathcal{L}(A_1)} \tau(w)$.

The *language emptiness problem* is a standard problem in Timed Automata theory and is stated as follows [6]: "given a (Timed) Automaton A, is $\mathcal{TL}(A)$ empty?". It is known that the emptiness problem is decidable for some classes of real-time programs (e.g., Timed Automata [6]), but undecidable for slightly more expressive classes (e.g., Stopwatch Automata [9]). It is usually possible to compute symbolic representations of sets of *reachable* valuations after a sequence of labels. However, to compute the set of reachable valuations we may need to explore an arbitrary and unbounded number of sequences. Hence only semi-algorithms exist to compute the set of reachable valuations. For instance, using PHAVER to compute the set of reachable valuations for A_1 does not terminate (Table B.1). To force termination, we can compute an over-approximation of the set of reachable valuations. Computing an over-approximation is sound (if we declare a timed language to be empty it is empty) but incomplete i.e., it may result in *false positives* (we declare a timed language non empty whereas it is empty). This is witnessed by the column "UPPAAL" in Table B.1 where UPPAAL over-approximates sets of valuations in the stopwatch automaton with DBMs. After $i.t_0$, the over-approximation is $0 \leq y \leq x \wedge 0 \leq z \leq x$. This over-approximation intersects the guard $x - y \geq 1 \wedge z - y < 1$ of t_2 and ℓ_2 is reachable but this is an artifact of the over-approximation.[3]

Neither UPPAAL nor PHAVER can prove that $\mathcal{TL}(A_1) = \varnothing$. The technique we introduce in this paper enables us to discover arbitrary abstractions and invariants that enable us to prove $\mathcal{TL}(A_1) = \varnothing$. Our method is a version of the *Trace Abstraction Refinement* (TAR) technique introduced in [2]. Let us demonstrate how the method works on the stopwatch automaton A_1:

- find a (untimed) word accepted by A_1. Let $w_1 = i.t_0.t_2$ be such a word.

[3]UPPAAL terminates with the result "the language may not be empty".

We check whether $\tau(w_1) = \varnothing$ by encoding the corresponding associated timed traces as described by Equations (P_0)–(P_2) and then check whether $P_0 \wedge P_1 \wedge P_2$ is satisfiable[4]. As $P_0 \wedge P_1 \wedge P_2$ is not satisfiable we have $\tau(w_1) = \varnothing$.

- from the proof that $P_0 \wedge P_1 \wedge P_2$ is not satisfiable, we can obtain an *inductive interpolant* that comprises of two predicates I_0, I_1 – one for each conjunction – over the clocks x, y, z. An example of inductive interpolant[5] is $I_0 = x \leq y$ and $I_1 = x - y \leq z$. These predicates are *invariants* of any timed word of the untimed word w_1, and can be used to annotate w_1 with pre- and post-conditions (Equation B.1), which are Hoare triples of the form $\{P\}\ a\ \{Q\}$:

$$\{True\} \quad i \quad \{I_0\} \quad t_0 \quad \{I_1\} \quad t_2 \quad \{False\} \tag{B.1}$$

$$\{True\} \quad i \quad \{I_0\} \quad t_0 \quad \mathbf{\{I_1\}} \quad \mathbf{(t_1)^*} \quad \mathbf{\{I_1\}} \quad t_2 \quad \{False\} \tag{B.2}$$

We can also prove that $\{I_1\}\ (t_1)^*\ \{I_1\}$ is a valid Hoare triple and combined with Equation B.1 this gives Equation B.2. For each word $w \in i.t_0.(t_1)^*.t_2$, $\tau(w) = \varnothing$ and as $\mathcal{L}(A_1) \subseteq i.t_0.(t_1)^*.t_2$ we can conclude that $\mathcal{TL}(A_1) = \varnothing$.

3 Real-Time Programs

Our approach is general enough and applicable to a wide range of timed systems called *real-time programs*. As an example, timed, stopwatch, hybrid automata and time Petri nets are special cases of real-time programs.

In this section we define *real-time programs*. Real-time programs define the control flow of *instructions*, just as standard imperative programs do. The instructions can update *variables* by assigning new values to them. Each instruction has a semantics and together with the control flow this precisely defines the semantics of real-time programs.

Notations. A finite automaton over an alphabet Σ is a tuple $\mathcal{A} = (Q, \iota, \Sigma, \Delta, F)$ where Q is a finite set of locations s.t. $\iota \in Q$ is the initial location, Σ is a finite alphabet of actions, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a finite transition relation, $F \subseteq Q$ is the set of *accepting* locations. A word $w = \alpha_0.\alpha_1.\cdots.\alpha_n$ is a finite sequence of letters from Σ; we let $w[i] = \alpha_i$ the i-th letter, $|w|$ be the length of w which is $n+1$. Let ϵ be the empty word and $|\epsilon| = 0$, Σ^* is the set of finite words over Σ. The *language*, $\mathcal{L}(\mathcal{A})$, accepted by $\mathcal{A}$ is defined in the usual manner as the set of words that can lead to F from ι.

[4]This can be done using an SMT-solver e.g., Z3.

[5]This is the pair returned by Z3 for $P_0 \wedge P_1 \wedge P_2$.

Let V be a finite set of real-valued variables. A *valuation* is a function $\nu : V \to \mathbb{R}$. The set of valuations is $[V \to \mathbb{R}]$. We denote by $\beta(V)$ a set of *constraints* on the variables in V. Given $\varphi \in \beta(V)$, we let $\mathit{Vars}(\varphi)$ be the set of free variables in φ. The truth value of a constraint φ given a valuation ν is denoted by $\varphi(\nu)$ and we write $\nu \models \varphi$ when $\varphi(\nu) = \mathit{True}$. We let $[\![\varphi]\!] = \{\nu \mid \nu \models \varphi\}$. An *update* of the variables in V is a binary relation $\mu \subseteq [V \to \mathbb{R}] \times [V \to \mathbb{R}]$. Given an update μ and a set of valuations $\mathcal{V}$, we let $\mu(\mathcal{V}) = \{\nu' \mid \exists \nu \in \mathcal{V} \text{ and } (\nu, \nu') \in \mu\}$. We let $\mathcal{U}(V)$ be the set of updates on the variables in V. A *rate* ρ is a function from V to $\mathbb{Q}$ (rates can be negative), i.e., an element of $\mathbb{Q}^V$. We let $\mathcal{R}(V) \subseteq \mathbb{Q}^V$ be a set of *valid* rates – that is, rates that can be written (syntactically) as a predicate on an edge. Given a valuation ν, a valid rate $\rho \in \mathbb{Q}(V)$ and a timestep $\delta \in \mathbb{R}_{\geq 0}$ the valuation $\nu + \rho \times \delta$ is defined by: $(\nu + \rho \times \delta)(v) = \nu(v) + \rho(v) \times \delta$ for $v \in V$.

Real-Time Instructions. Let $\mathcal{I} = \beta(V) \times \mathcal{U}(V) \times \mathcal{R}(V)$ be a countable set of instructions. Each $\alpha \in \mathcal{I}$ is a tuple (*guard, update, rates*) denoted by $(\gamma_\alpha, \mu_\alpha, \rho_\alpha)$. Let $\nu : V \to \mathbb{R}$ and $\nu' : V \to \mathbb{R}$ be two valuations. For each pair $(\alpha, \delta) \in \mathcal{I} \times \mathbb{R}_{\geq 0}$ we define the following transition relation:

$$\nu \xrightarrow{\alpha,\delta} \nu' \iff \begin{cases} 1. & \nu \models \gamma_\alpha (\text{guard of } \alpha \text{ is satisfied in } \nu), \\ 2. & \exists \nu'' \text{ s.t. } (\nu, \nu'') \in \mu_\alpha \text{ (discrete update allowed by } \alpha) \text{ and} \\ 3. & \nu' = \nu'' + \delta \times \rho_\alpha (\text{continuous update as defined by } \alpha). \end{cases}$$

The semantics of $\alpha \in \mathcal{I}$ is a mapping $[\![\alpha]\!] : [V \to \mathbb{R}] \to [V \to \mathbb{R}]$ that can be extended to sets of valuations as follows:

$$\begin{aligned} \nu \in [V \to \mathbb{R}], \quad [\![\alpha]\!](\nu) = & \quad \{\nu' \mid \exists \delta \geq 0, \nu \xrightarrow{\alpha,\delta} \nu'\} \\ K \subseteq [V \to \mathbb{R}], \quad [\![\alpha]\!](K) = & \quad \bigcup_{\nu \in K} [\![\alpha]\!](\nu). \end{aligned}$$

Let K be a set of valuations, $\alpha \in \mathcal{I}$ and $w \in \mathcal{I}^*$. We inductively define the *post operator Post* as follows:

$$\begin{aligned} \mathit{Post}(K, \epsilon) &= K \\ \mathit{Post}(K, \alpha.w) &= \mathit{Post}([\![\alpha]\!](K), w) \end{aligned}$$

The post operator extends to logical constraints $\varphi \in \beta(V)$ by defining it as:

$$\mathit{Post}(\varphi, w) = \mathit{Post}([\![\varphi]\!], w)$$

In the sequel, we assume that, when $\varphi \in \beta(V)$, then $[\![\alpha]\!]([\![\varphi]\!])$ is also definable as a constraint in $\beta(V)$. This inductively implies that $\mathit{Post}(\varphi, w)$ can also be expressed as a constraint in $\beta(V)$ for sequences of instructions $w \in \mathcal{I}^*$.

Timed Words and Feasible Words. A *timed word* (over alphabet $\mathcal{I}$) is a finite sequence $\sigma = (\alpha_0, \delta_0).(\alpha_1, \delta_1). \cdots .(\alpha_n, \delta_n)$ such that for each $0 \leq i \leq n$, $\delta_i \in \mathbb{R}_{\geq 0}$ and $\alpha_i \in \mathcal{I}$. The timed word σ is *feasible* if and only if there exists a set of valuations $\{\nu_0, \dots, \nu_{n+1}\} \subseteq [V \to \mathbb{R}]$ such that:

$$\nu_0 \xrightarrow{\alpha_0, \delta_0} \nu_1 \xrightarrow{\alpha_1, \delta_1} \nu_2 \cdots \nu_n \xrightarrow{\alpha_n, \delta_n} \nu_{n+1}.$$

We let $Unt(\sigma) = \alpha_0.\alpha_1. \cdots .\alpha_n$ be the *untimed* version of σ. We overload the term *feasible* as follows: an untimed word $w \in \mathcal{I}^*$ is feasible iff $w = Unt(\sigma)$ for some feasible timed word σ.

Lemma 1
An untimed word $w \in \mathcal{I}^*$ is *feasible* iff $Post(True, w) \neq False$.

Proof. The lemma follows trivially from the inductive definition of *Post*.

Real-Time Programs. The specification of a real-time program decouples the *control* (e.g., for Timed Automata, the locations) and the *data* (the clocks). A *real-time program* is a pair $P = (A_P, [\![\cdot]\!])$ where A_P is a finite automaton $A_P = (Q, \iota, I, \Delta, F)$ over the finite alphabet[6] $I \subseteq \mathcal{I}$, Δ defines the control-flow graph of the program and $[\![\cdot]\!]$ (as defined previously for $\mathcal{I}$) provides the semantics of each instruction. A timed word σ is *accepted* by P if and only if:

1. $Unt(\sigma)$ is accepted by A_P ($Unt(\sigma) \in \mathcal{L}(A_P)$) and
2. σ is feasible.

Notice that the definition of feasibility of a timed word σ is independent from the acceptance of $Unt(\sigma)$ by A_P. The *timed language, $\mathcal{TL}(P)$*, of a real-time program P is the set of timed words accepted by P, i.e., $\sigma \in \mathcal{TL}(P)$ if and only if $Unt(\sigma) \in \mathcal{L}(A_P)$ and σ is feasible.

Remark 1
We do not assume any particular values initially for the variables of a real-time program (the variables that appear in I). This is reflected by the definition of *feasibility* that only requires the existence of valuations without containing the initial one ν_0. When specifying a real-time program, initial values can be set by regular instructions. This is similar to standard programs where the first instructions can set the values of some variables.

[6] $\mathcal{I}$ can be infinite but we require the control-flow graph Δ (transition relation) of A_P to be finite.

Timed Language Emptiness Problem. The *(timed) language emptiness problem* asks the following:

Given a real-time program P, is $\mathcal{TL}(P)$ empty?

Theorem 4
$\mathcal{TL}(P) \neq \varnothing$ iff $\exists w \in \mathcal{L}(A_P)$ such that $Post(True, w) \not\subseteq False$.

Proof. $\mathcal{TL}(P) \neq \varnothing$ iff there exists a feasible timed word σ such that $Unt(\sigma)$ is accepted by A_P. This is equivalent to the existence of a feasible word $w \in \mathcal{L}(A_P)$, and by Lemma 1, feasibility of w is equivalent to $Post(True, w) \not\subseteq False$.

Useful Classes of Real-Time Programs. *Timed Automata* are a special case of real-time programs. The variables are called clocks. $\beta(V)$ is restricted to constraints on individual clocks or *difference constraints* generated by the grammar:

$$b_1, b_2 ::= True \mid False \mid x - y \bowtie k \mid x \bowtie k \mid b_1 \wedge b_2 \tag{B.3}$$

where $x, y \in V$, $k \in \mathbb{Q}_{\geq 0}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$[7]. We note that wlog. we omit *location invariants* as for the language emptiness problem, these can be implemented as guards. An update in $\mu \in \mathcal{U}(V)$ is defined by a set of clocks to be *reset*. Each pair $(\nu, \nu') \in \mu$ is such that $\nu'(x) = \nu(x)$ or $\nu'(x) = 0$ for each $x \in V$. The valid rates are fixed to 1, and thus $\mathcal{R}(V) = \{1\}^V$.

Stopwatch Automata can also be defined as a special case of real-time programs. As defined in [8], Stopwatch Automata are Timed Automata extended with *stopwatches* which are clocks that can be stopped. $\beta(V)$ and $\mathcal{U}(V)$ are the same as for Timed Automata but the set of valid rates is defined by the functions of the form $\mathcal{R}(V) = \{0, 1\}^V$ (the clock rates can be either 0 or 1). An example of a Stopwatch Automaton is given by the timed system $\mathcal{A}_1$ in Fig. B.1.

As there exists syntactic translations (preserving reachability) that maps hybrid automata to stopwatch automata [8], and translations that map time Petri nets [18, 19] and extensions [20, 21] thereof to timed automata, it follows that time Petri nets and hybrid automata are also special cases of real-time programs. This shows that the method we present in the next section is applicable to wide range of timed systems. What is remarkable as well, is that it is not restricted to timed systems that have a finite number of discrete states but can also accommodate infinite discrete state-spaces. For example, the automaton in Fig. B.2 has two clocks x and y and an unbounded integer variable k. Even though k is unbounded, our technique discovers the invariant $y \geq k$ at location 1 which is over a real-time clock y and the integer variable k. It allows us to prove that $\mathcal{TL}(P_2) = \varnothing$.

[7] While difference constraints are strictly disallowed in most definitions of Timed Automata, the method we propose retain its properties regardless of their presence.

4 Trace Abstraction Refinement for Real-Time Programs

In this section we propose a semi-algorithm to solve the language emptiness problem for real-time programs. The semi-algorithm is a version of the *refinement of trace abstractions* (TAR) approach [2] for timed systems.

Refinement of Trace Abstraction for Real-Time Programs. Fig. B.3 gives a precise description of the TAR semi-algorithm for real-time programs. This is the standard trace abstraction refinement semi-algorithm as introduced in [2] – we therefore omit theorems of completeness and soundness as these will be equivalent to the theorems in [2] and are proved in the exact same manner. The input to the semi-algorithm is a real-time program $P = (A_P, [\![\cdot]\!])$. An invariant of the semi-algorithm is that R is empty or contains only infeasible traces.

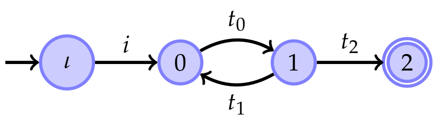
Initially the refinement R is the empty set. The semi-algorithm works as follows:

Step 1 check whether all the (untimed) traces in $\mathcal{L}(A_P)$ are in R. If this is the case, $\mathcal{TL}(P)$ is empty and the semi-algorithm terminates. Otherwise, there is a sequence $w \in \mathcal{L}(A_P) \setminus R$, goto Step 2;

Step 2 if w is feasible i.e., there is a feasible timed word σ such that $Unt(\sigma) = w$, then $\sigma \in \mathcal{TL}(P)$ and $\mathcal{TL}(P) \neq \varnothing$ and the semi-algorithm terminates. Otherwise w is not feasible, goto Step 3;

Step 3 w is infeasible and given the reason for infeasibility we can construct a finite *interpolant automaton*, $IA(w)$, that accepts w and other words that are infeasible for the same reason. How $IA(w)$ is computed is addressed in the sequel. The automaton $IA(w)$ is added to the previous refinement R and the semi-algorithm starts a new round at Step 1.

Checking Feasibility. Given a word $w \in \mathcal{I}^*$, we can check whether w is feasible by encoding the side-effects of each instruction in w, similar to a Static Single Assignment (SSA) form in programming languages.



Edge	Guard	Update
i	*True*	`x:=y:=k:=0`
t_0	$x \geq 1$	—
t_1	*True*	`x:=0; k++`
t_2	$y < k$	—

Fig. B.2: Real-time program P_2

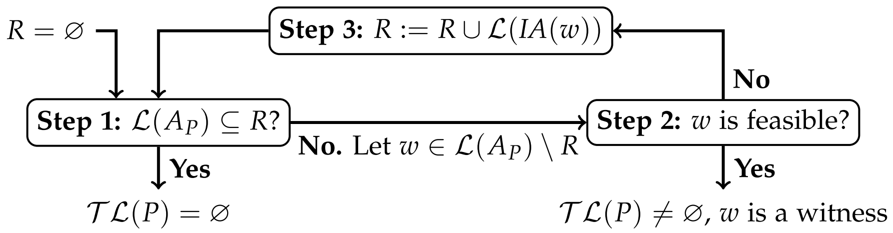

$R = \varnothing$
Step 3: $R := R \cup \mathcal{L}(IA(w))$
No
Step 1: $\mathcal{L}(A_P) \subseteq R$?
Step 2: w is feasible?
No. Let $w \in \mathcal{L}(A_P) \setminus R$
Yes
Yes
$\mathcal{TL}(P) = \varnothing$
$\mathcal{TL}(P) \neq \varnothing$, w is a witness

Fig. B.3: Trace Abstraction Refinement Semi-Algorithm for Real-Time Programs

Let us define a function for constructing such a constraint-system characterizing the feasibility of a given trace. We shall assume that constraints in $\beta(V)$ and updates in $\mathcal{U}(V)$ are syntactically defined. Let $P = (Q, q_0, \mathcal{I}, \Delta, F)$ be a real-time program and $w \in \mathcal{I}^*$ be a word over $\mathcal{I}$. Let $V^n = \{x^n, x^n_\mu \mid x \in V\} \cup \{\delta^n\}$ be a set of variables extended with an index $n \in \mathbb{N}_{\geq 0}$. For a given constraint-system $\varphi \in \beta(V)$ write $\varphi_{[V/V^n]}$ for replacing all occurrences of V with their indexed occurrence in V^n (implying that $\varphi_{[V/V^n]} \in \beta(V^n)$). We assume that the relation $\mu \in \mathcal{U}(V)$ is of SSA form, and let $\mu_{[V/(V^n,V^m)]}$ be the replacement of all occurrences of variables $x \in V$ with their indexes and sub-scripted occurrence in V^n if x is assigned to and from V^m if x is read from. As an example, $(v \leftarrow v + w)_{[V/(V^n,V^m)]} = v^n_\mu \leftarrow v^m + w^m$ where $\leftarrow$ denotes assignment. Given this we can now recursively define the function $Enc : \mathcal{I}^* \to \beta(\{V^n \mid 0 \leq n \leq |w|\})$

$$\begin{aligned} Enc(\epsilon) =& True \\ Enc(w.\alpha) =& Enc(w) \wedge \delta^n \geq 0 \wedge \varphi_{[V/V^{n-1}]} \wedge \delta^n \geq 0 \wedge \mu_{[V/(V^n_\mu, V^{n-1})]} \\ & \wedge \bigwedge_{v \in V} v^n = v^n_\mu + \rho(v) \times \delta^n \text{ where } n = |w| - 1 \text{ and } (\varphi, \mu, \rho) = \alpha \end{aligned}$$

The function $Enc : \mathcal{I}^* \to \beta(V^{\mathbb{N}_{\geq 0}})$ constructs a constraint-system characterizing exactly the feasibility of a word w:

Lemma 2
A word w is feasible i.e., $Post(True, w) \not\subseteq False$ iff $Enc(w)$ is satisfiable.

We shall frequently refer to such a constraints system $C = Enc(w)$ for some word w where $|w| = n$ as a sequence of conjunctions $P_0 \wedge \cdots \wedge P_m \wedge \cdots \wedge P_n = C$ where $P_m \in \beta(V^{m-1} \cup V^m)$ refers to the encoding of the m'th instruction, and we shall call such an element P_m a predicate.

An example of an encoding for the real-time program A_1 (Fig. B.1) is given by the predicates in Equation (P_0)–(P_2). The variables x_k, y_k, z_k denote the values of x, y, z after k steps (initially the variables can have arbitrary values). The sequence $w_1 = i.t_0.t_2$ is feasible iff $Enc(w_1) = P_0 \wedge P_1 \wedge P_2$ is satisfiable.

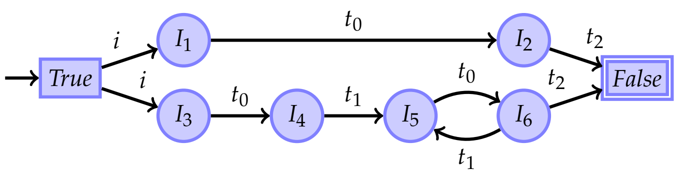


Fig. B.4: Interpolant automaton for $\mathcal{L}(IA(w_1)) \cup \mathcal{L}(IA(w2))$.

From such a sequence we can use interpolating SMT-solvers to construct a sequence of *craig-interpolants*.

Construction of Interpolant Automata. When it is determined that a trace w is infeasible, we can easily discard such a single trace and continue searching. However, the power of the TAR method is to generalize the infeasibility of a single trace w into a family (regular set) of traces. This regular set of infeasible traces is computed from the reason of infeasibility of w and is formally specified by an *interpolant automaton*, $IA(w)$. The reason for infeasibility itself has the form of an *inductive interpolant*.

Given a conjunctive formula $f = P_0 \wedge \cdots \wedge P_m$, if f is unsatisfiable, an *interpolating* SMT-solver is capable of producing inductive arguments for the unsatisfiability reason. This argument is an *inductive interpolant* $I_0, \ldots, I_{m-1}$ s.t for each $0 \leq n \leq m-1$, $I_n \wedge P_{n+1}$ implies I_{n+1} (with $I_m = False$), and for each $0 \leq n \leq m-1$, the variables in I_n appear in both P_n and P_{n+1}.

One can intuitively think of each interpolant as a *sufficient* condition for infeasibility of the post-fix of the trace and this can be represented by a sequence of Hoare triples of the form $\{P\}\ a\ \{Q\}$:

$$\{True\} \quad a_0 \quad \{I_0\} \quad a_1 \quad \{I_1\} \quad \cdots \quad \{I_{m-1}\} \quad a_m \quad \{False\}$$

Consider the real-time program P_2 of Fig. B.2 and the two infeasible untimed words $w_1 = i.t_0.t_2$ and $w_2 = i.t_0.t_1.t_0.t_2$. The Hoare triples for w_1 and w_2 are given by Equation B.4-B.5 where the predicates are: $I_1 = y \geq x \wedge (k = 0)$, $I_2 = y \geq k$, $I_3 = y \geq x \wedge k \leq 0$, $I_4 = y \geq 1 \wedge k \leq 0$, $I_5 = y \geq k + x$, $I_6 = y \geq k + 1$.

$$\{True\} \quad i \quad \{I_1\} \quad t_0 \quad \{I_2\} \quad t_2 \quad \{False\} \tag{B.4}$$

$$\{True\} \quad i \quad \{I_3\} \quad t_0 \quad \{I_4\} \quad t_1 \quad \{I_5\} \quad t_0 \quad \{I_6\} \quad t_2 \quad \{False\} \tag{B.5}$$

As can be seen in Equation B.5, the sequence contains two occurrences of t_0: this suggests that a loop occurs in the program, and this loop may be infeasible as well. Formally, because $Post(I_6, t_1) \subseteq I_5$, any trace of the form $i.t_0.t_1.(t_0.t_1.t_0)^*.t_2$ is infeasible. This enables us to construct $IA(w_2)$ as accepting the regular set of infeasible traces $i.t_0.t_1.(t_0.t_1.t_0)^*.t_2$. Overall, because w_1 is also infeasible, we obtain a refinement which is $\mathcal{L}(IA(w_1)) \cup \mathcal{L}(IA(w_2))$, Fig. B.4.

Let us formalize the interpolant-automata construction. Given the interpolants $I_0, \dots I_k$ for the constraint-system $P_0 \wedge \dots \wedge P_{k+1} = Enc(w)$ for some word w where $k = |w| - 1$ and given the automata description of our Real Time Program $\mathcal{A} = (Q, q_0, \Sigma, \Delta, F)$, then we can construct an interpolant automaton $\mathcal{A}^I = (Q^I, q_0^I, \Sigma^I, \Delta^I, F^I)$ s.t. $w \in \mathcal{L}(\mathcal{A}^I)$ and for all $w' \in \mathcal{L}(\mathcal{A}^I)$ we have that w' is infeasible. Let $Q = \{True, False, I_0, \dots, I_k\}$, $q_0 = True$, $\Sigma^I = \Sigma$, $F = \{False\}$, then we let the transition-function be the largest transition-function satisfying the following.

1. $(True, w[0], I_0) \in \Delta^I$,
2. $(I_k, w[k], False) \in \Delta^I$,
3. $(I_{n-1}, w[n-1], l_n) \in \Delta^I$ for $1 < n \leq k$, and
4. for each $1 \leq n, m \leq k$, if $I_m \subset_V I_n$ then $(I_{n-1}, w[n-1], I_m) \in \Delta^I$ where $\subset_V$ is subset-checking, modulo variable indexing.

The above conditions induce an algorithm *IA* for constructing interpolant automata from an untimed word w.

Theorem 5 (Interpolant Automata)
Let w be an infeasible word over P, then for all $w' \in \mathcal{L}(IA(w))$, w' is infeasible.

We can verify that the construction using rules 1-3 is correct as these come directly from the feasibility-check of the trace and the definition of interpolants.

The *pumping*-rule (rule 4) utilizes that if by firing some transition labeled α from some interpolant I_{n-1} gives us a "stronger" argument for infeasibility than in I_m, then surely every sequence which is infeasible from I_m is also infeasible from I_{n-1} after firing α.

Feasibility Beyond Timed Automata. Satisfiability can be checked with an SMT-solver (and decision procedures exist for useful theories). In the case of timed automata and stopwatch automata, the feasibility of a trace can be encoded as a linear program. The corresponding theory, Linear Real Arithmetic (LRA) is decidable and supported by most SMT-solvers. It is also possible to encode non-linear constraints (non-linear guards and assignments). In the latter cases, the SMT-solver may not be able to provide an answer to the SAT problem as non-linear theories are undecidable. However, we can still build on a semi-decision procedure of the SMT-solver, and if it provides an answer, get the status of a trace (feasible or not).

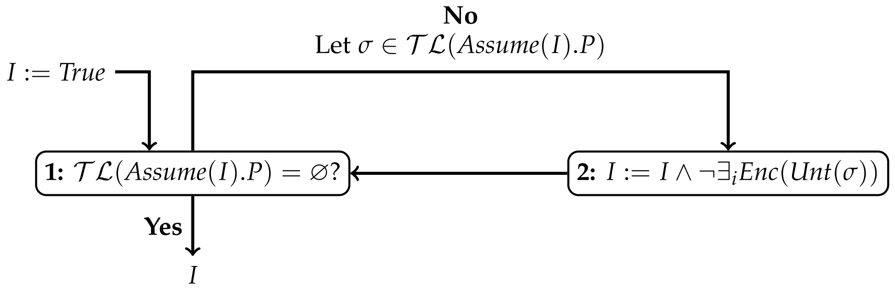

No
Let $\sigma \in \mathcal{TL}(Assume(I).P)$
$I := True$
1: $\mathcal{TL}(Assume(I).P) = \varnothing$?
2: $I := I \wedge \neg\exists_i Enc(Unt(\sigma))$
Yes
I

Fig. B.5: Semi-algorithm *SafeInit*.

5 Parameter Synthesis for Real-Time Programs

In this section we show how to use the trace abstraction refinement semi-algorithm presented in Section 4 to synthesize *good initial values* for some of the program variables. Given a real-time program P, the objective is to determine a set of *initial valuations* $I \subseteq [V \to \mathbb{R}]$ such that, when we start the program in I, P does not accept any timed word.

Given a constraint $I \in \beta(V)$, we define the associated *assume* guard-transformer for instructions that for a letter $\alpha = (\gamma, \rho, \mu)$ defines $Assume(\alpha, I) = (\gamma', \rho, \mu)$ s.t. $\gamma = \gamma \wedge I$. Let $P = (Q, \iota, \mathcal{I}, \Delta, F)$ be a real-time program. Then we can define the real-time program $Assume(I).P = (Q, \iota, \mathcal{I}, (\Delta \setminus \{(\iota, i, q_0)\}) \cup \{(\iota, Assume(i, I), q_0)\}, F)$.

Safe Initial Set Problem. The *safe initial state problem* asks the following:

> Given a real-time program P, is there $I \in \beta(V)$ s.t. $\mathcal{TL}(Assume(I).P) = \varnothing$?

Semi-Algorithm for the Safe Initial State Problem. Let $w \in \mathcal{L}(P)$. When $Enc(w)$ is satisfiable, we define the (existentially quantified) constraint:

$$\exists Vars(Enc(w)) \setminus V_{-1}.Enc(w)$$

That is, the projection of the set of solutions on the initial values of the variables. We let $\exists_i(w)$ be $\exists Vars(Enc(w)) \setminus V_{-1}.Enc(w)$ with all the free occurrences of x_{-1} replaced by x (remove index for each var). $\exists_i(w)$ is a constraint over the set of variables V (and existential quantifiers)[8].
The semi-algorithm in Fig. B.5 works as follows:

[8]Existential quantification for the theory of Linear Real Arithmetic is within the theory via Fourier–Motzkin-elimination – hence the solver only needs support for Linear Real Arithmetic for Parameter Synthesis for Stopwatch and Timed Automata.

1) initially $I = True$

2) using the semi-algorithm from Figure B.3, test if $\mathcal{TL}(Assume(I).P)$ is empty – if so P does not accept any timed word when we start from $[\![I]\!]$

3) Otherwise, there is a witness word $\sigma \in \mathcal{TL}(Assume(I).P)$, implying that $I \wedge Enc(Unt(\sigma))$ is satisfiable. We can then determine a sufficient condition $I' = \exists_i(Unt(\sigma))$ for the feasibility s.t. $(\neg I') \wedge Enc(Unt(\sigma))$ is unsatisfiable and use this to strengthen the constraint I (step 2).

If the semi-algorithm terminates, it computes exactly the set of parameters for which the system is not safe (I), captured formally by Theorem 6.

Theorem 6
If the semi-algorithm *SafeInit* terminates and outputs I, then for any $I' \in \beta(V)$, $\mathcal{TL}(Assume(I').P) = \varnothing$ if and only if $I' \subseteq I$.

$\Longrightarrow$. Let us assume by contradiction that upon termination we have the following.

$$\mathcal{TL}(Assume(I).P) \neq \varnothing$$

This violates the termination critirion of either Figure B.3 or Figure B.5.

$\Longleftarrow$. Let us assume by contradiction that upon termination there exists some $I' \neq \varnothing$ for which $I' \cap I = \varnothing$ and $\mathcal{TL}(Assume(I').P) = \varnothing$. Then let us prove inductively that no such I' can ever exist.

In the base-case in step 1, if the algorithm terminates, clearly $I' = \varnothing$ violating our requirements for the contradiction. For our contradiction to be valid, we must instead look at how we modify I in step 2. For I' to be non-empty, the quantification over parameter-values for σ must construct a larger-than-needed set of parameter value, i.e., that $I' \subseteq \neg\exists_i Enc(Unt(\sigma))$. This contradicts the definition of existential quantification. As we never over-approximate the parameter set needed for the valuation in step 2, we can conclude that I' cannot exist.

6 Experiments

We have conducted two sets of experiments, each testing the applicability of our proposed method (denoted by RTTAR) compared to state-of-the-art tools with specialized data structures and algorithms for the given setting. All experiments were conducted on AMD Opteron 6376 Processors and limited to 1 hour of computation. The RTTAR tool uses the UPPAAL parsing-library, but relies on Z3 [22] for the interpolant computation.

Verification of Timed and Stopwatch Automata. The real-time programs, P_1 of Fig. B.1 and P_2 of Fig. B.2 can be analyzed with our technique. The analysis (RTTAR algorithm, B.3) terminates in two iterations for the program P_1, a stopwatch automaton. As emphasized in the introduction, neither UPPAAL (over-approximation with DBMs) nor PHAVER can provide the correct answer to reachability problem for P_1.

To prove that location 2 is unreachable in program P_2 requires to discover an invariant that mixes integers (discrete part of the state) and clocks (continuous part). Our technique successfully discovers the program invariants I_5 and I_6 (thanks to the interpolating SMT-solver). As a result the refinement depicted in Fig. B.2 is constructed and as it contains $\mathcal{L}(A_{P_2})$ the refinement algorithm terminates and proves that 2 is not reachable. A_{P_2} can only be analyzed in UPPAAL with significant computational effort and bounded integers.

Robustness of Timed Automata. Another remarkable feature of our technique is that it can readily be used to check *robustness* of timed automata. In essence, checking robustness amounts to enlarging the guards of an TA A by an $\varepsilon > 0$. The resulting TA is A_ε. The automaton A is (safety) robust iff there is some $\varepsilon > 0$ such $\mathcal{TL}(A_\epsilon) = \varnothing$.

To address the robustness problem for a real-time program P, we use the semi-algorithm presented in Section 5 and reduce the robustness-checking problem to that of parameter synthesis. Assuming P is robust[9] i.e., there exists some $\epsilon > 0$ such that $\mathcal{TL}(A_\epsilon) = \varnothing$ and the previous process terminates we can compute the largest set of parameters for which P is robust.

As Table B.2 demonstrates, SYMROB [16] and RTTAR do not always agree on the results. Notably, since the TA `M3` contains strict guards, SYMROB is unable to compute the robustness of it. Furthermore, SYMROB over-approximates ϵ, an artifact of the so-called "loop-acceleration"-technique and the polyhedra-based algorithm. This can be observed in the modified model `M3c`, which is now analyzable by SYMROB, but differ in results compared to RTTAR. This is the same case with the model denoted `a`. We experimented with ϵ-values to confirm that `M3` is safe for all the values tested – while `a` is safe only for values tested respecting $\epsilon < \frac{1}{2}$. We can also see that our proposed method is significantly slower than SYMROB. As our tool is currently only a prototype with rudimentary state-space-reduction-techniques, this is to be expected.

Parametric Stopwatch Automata. In our last series of tests, we compare the RTTAR tool to IMITATOR [17] – the state-of-the-art parameter synthesis

[9]Proving that a system is non-robust requires proving *feasibility* of infinite traces for ever decreasing ϵ. We have developed some techniques to do so but this is outside of the scope of this paper.

Test	Time	$\epsilon <$	Time	$\epsilon <$
	SYMROB		RTTAR	
`csma_05`	0.43	1/3	68.23	1/3
`csma_06`	2.44	1/3	227.15	1/3
`csma_07`	8.15	1/3	1031.72	1/3
`fischer_04`	0.16	1/2	45.24	1/2
`fischer_05`	0.65	1/2	249.45	1/2
`fischer_06`	3.71	1/2	1550.89	1/2
`M3c`	4.34	250/3	43.10	∞
`M3`	**N/A**	**N/A**	43.07	∞
`a`	27.90	1/4	15661.14	1/2

Table B.2: Results for robustness analysis comparing RTTAR with SYMROB. Time is given in seconds. N/A indicates that SYMROB was unable to compute the robustness for the given model.

Test	IMITATOR	RTTAR
`Sched2.50.0`	201.95	1656.00
`Sched2.100.0`	225.07	656.26
A_1	DNF	0.1
`fischer_2`	DNF	0.23
`fischer_4`	DNF	40.13
`fischer_2_robust`	DNF	0.38
`fischer_4_robust`	DNF	118.11

Table B.3: Results for parameter synthesis comparing RTTAR with IMITATOR. Time is given in seconds. DNF marks that the tool did not complete the computation within an hour.

tool for reachability [10]. We shall here use the semi-algorithm is presented in Section 5 For the test-cases we use the gadget presented initially in Fig. B.1, a few of the test-cases used in [23], as well as two modified version of Fischers Protocol, shown in Fig. B.6. In the first version we replace the constants in the model with parameters. In the second version (marked by robust), we wish to compute an expression, that given an arbitrary upper and lower bound yields the robustness of the system – in the same style as the experiments presented in Section 6, but here for arbitrary guard-values.

As illustrated by Table B.3 the performance of RTTAR is slower than IMITATOR when IMITATOR is able to compute the results. On the other hand, when using IMITATOR to verify our motivating example from Fig. B.1, we observe that IMITATOR never terminates, due to the divergence of the polyhedra-computation. This is the effect illustrated in Table B.1.

[10]We compare with the `EFSynth`-algorithm in the IMITATOR tool as this yielded the lowest computation time in the two terminating instances.

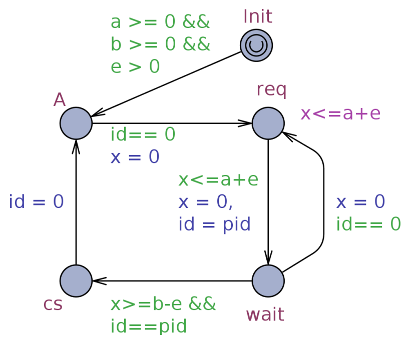


Fig. B.6: A Uppaal template for a single process in Fischers Algorithm. The variables e, a and b are parameters for ϵ, lower and upper bounds for clock-values respectively.

When trying to synthesize the parameters for Fischers algorithm, in all cases, Imitator times out and never computes a result. For both two and four processes in Fischers algorithm, our tool detects that the system is safe if and only if $a < 0 \vee b < 0 \vee b - a > 0$. Notice that $a < 0 \vee b < 0$ is a trivial constraint preventing the system from doing anything. The constraint $b - a > 0$ is the only useful one. Our technique provides a formal proof that the algorithm is correct for $b - a > 0$.

In the same manner, our technique can compute the most general constraint ensuring that Fischers algorithm is robust. The result of rttar algorithm is that the system is robust iff $\epsilon \leq 0 \vee a < 0 \vee b < 0 \vee b - a - 2\epsilon > 0$ – which for $\epsilon = 0$ (modulo the initial non-zero constraint on ϵ) reduces to the constraint-system obtained in the non-robust case.

7 Conclusion

We have proposed a version of the trace abstraction refinement approach to real-time programs. We have demonstrated that our semi-algorithm can be used to solve the reachability problem for instances which are not solvable by state-of-the-art analysis tools.

Our algorithms can handle the general class of real-time programs that comprises of classical models for real-time systems including timed automata, stopwatch automata, hybrid automata and time(d) Petri nets.

As demonstrated in Section 6, our tool is capable of solving instances of reachability problems problems, robustness, parameter synthesis, that current tools are incapable of handling.

For future work we would like to improve the scalability of the proposed method, utilizing well known techniques such as extrapolations, partial order reduction and compositional verification. Furthermore, we would like to extend our approach from reachability to more expressive temporal logics.

Acknowledgments. The research leading to these results was made possible by an external stay partially funded by Otto Mønsted Fonden.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Counterexample-Guided Abstraction Refinement*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. [Online]. Available: http://dx.doi.org/10.1007/10722167_15

[2] M. Heizmann, J. Hoenicke, and A. Podelski, *Refinement of Trace Abstraction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 69–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03237-0_7

[3] ——, "Software model checking for people who love automata," in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 36–52.

[4] D. Beyer, "Competition on software verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 504–524.

[5] F. Cassez, A. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo Marugán, "Skink: Static analysis of programs in LLVM intermediate representation (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017. Proceedings*, 2017, B - International Conferences, forthcoming.

[6] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[7] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST'06*, 2006, pp. 125–126.

[8] F. Cassez and K. G. Larsen, "The impressive power of stopwatches," in *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, ser. Lecture Notes in Computer Science, C. Palamidessi, Ed., vol. 1877. Springer, 2000, pp. 138–152. [Online]. Available: http://dx.doi.org/10.1007/3-540-44618-4_12

[9] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94 – 124, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000098915811

[10] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Berlin Heidelberg, 2005, vol. 3414, pp. 258–273. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31954-2_17

[11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 379–395. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_30

[12] T. A. Henzinger, P.-H. Ho, and H. Wong-toi, "Hytech: A model checker for hybrid systems," *Software Tools for Technology Transfer*, vol. 1, pp. 460–463, 1997.

[13] W. Wang and L. Jiao, *Trace Abstraction Refinement for Timed Automata*. Cham: Springer International Publishing, 2014, pp. 396–410. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11936-6_28

[14] H. Dierks, S. Kupferschmid, and K. G. Larsen, "Automatic abstraction refinement for timed automata," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2007, pp. 114–129.

[15] P. Kordy, R. Langerak, S. Mauw, and J. W. Polderman, *A Symbolic Algorithm for the Analysis of Robust Timed Automata*. Cham: Springer International Publishing, 2014, pp. 351–366. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06410-9_25

[16] O. Sankur, *Symbolic Quantitative Robustness Analysis of Timed Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 484–498. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_48

[17] É. André, L. Fribourg, U. Kühne, and R. Soulat, *IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 33–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32759-9_6

[18] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux, "Comparison of the expressiveness of timed automata and time petri nets," in *Formal Modeling and Analysis of Timed Systems, Third International Conference,*

FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings, ser. Lecture Notes in Computer Science, P. Pettersson and W. Yi, Eds., vol. 3829. Springer, 2005, B - International Conferences, pp. 211–225.

[19] F. Cassez and O. H. Roux, "Structural translation from time petri nets to timed automata," *Journal of Software and Systems*, vol. 79, no. 10, pp. 1456–1468, Oct. 2006.

[20] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux, "The expressive power of time Petri nets," *Theoretical Computer Science*, vol. 474, pp. 1–20, 2013.

[21] J. Byg, M. Jacobsen, L. Jacobsen, K. Jørgensen, M. Møller, and J. Srba, "TCTL-preserving translations from timed-arc Petri nets to networks of timed automata," *Theoretical Computer Science*, vol. 537, pp. 3 – 28, 2014.

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[23] É. André, G. Lipari, H. G. Nguyen, and Y. Sun, *Reachability Preservation Based Parameter Synthesis for Timed Automata*. Cham: Springer International Publishing, 2015, pp. 50–65. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17524-9_5

Paper C

Discrete and Continuous Strategies for Timed-Arc Petri Net Games

Peter Gjøl Jensen, Kim Guldstrand Larsen and Jiří Srba

The paper has been published in the
International Journal on Software Tools for Technology Transfer Vol. 10009,
pp. 1–18, 2017.
The paper is an extended version of [1] with full proofs, additional
examples, improved implementation and updated experimental results.

Automatic strategy synthesis for a given control objective can be used to generate correct-by-construction controllers of real-time reactive systems. The existing symbolic approach for continuous timed games is a computationally hard task and current tools like UPPAAL TIGA often scale poorly with the model complexity. We suggest an explicit approach for strategy synthesis in the discrete-time setting and show that even for systems with closed guards, the existence of a safety discrete-time strategy does not imply the existence of a safety continuous-time strategy and vice versa. Nevertheless, we prove that the answers to the existence of discrete-time and continuous-time safety strategies coincide on a practically motivated subclass of urgent controllers that either react immediately after receiving an environmental input or wait with the decision until a next event is triggered by the environment. We then develop an on-the-fly synthesis algorithm for discrete timed-arc Petri net games. The algorithm is implemented in our tool TAPPAALand based on the experimental evidence, we discuss the advantages of our approach compared to the symbolic continuous-time techniques.

1 Introduction

Formal methods and model checking techniques have traditionally been used to verify whether a given system model complies with its specification. However, when we consider formal (game) models where both the controller and the environment can make choices, the question now changes to finding a controller strategy such that any behaviour under such a fixed strategy complies with the given specification. The model checking approach can be used as a try-and-fail technique to check whether a given controller is correct but automatic synthesis of a correct-by-construction controller, as already proposed by Church [2, 3], is a more difficult problem as documented e.g. by the SYNTCOMP competition and SYNT workshop [4]. The area has recently seen renewed interest, partly given the rise in computational power that makes synthesis feasible. We focus on the family of timed systems, where for the model of timed automata [5] synthesis has already been proposed [6] and implemented [7, 8].

In the area of model checking, symbolic continuous-time on-the-fly methods were ensuring the success of tools such as Kronos [9], UPPAAL [10], Tina [11] and Romeo [12], utilizing the zone abstraction approach [5] via the data structure DBM [13]. These symbolic techniques were recently employed in on-the-fly algorithms [14] for synthesis of controllers for timed games [6–8]. While these methods scale well for classical reachability, the limitation of symbolic techniques is more apparent when used for liveness properties and for solving timed games. We have shown that for reachability and liveness properties, the discrete-time methods performing point-wise exploration of the state-space can prove competitive on a wide range of problems [15], in particular in combination with additional techniques as time-darts [16], constant-reducing approximation techniques [17] and memory-preserving data structures like PTrie [18, 19].

In this paper, we benefit from the recent advances in the discrete-time verification of timed systems and suggest an on-the-fly point-wise algorithm for the synthesis of timed controllers relative to safety objectives (avoiding undesirable behaviour). The algorithm is described for a novel game extension of the well-studied timed-arc Petri net formalism [20, 21] and we show that in the general setting, the existence of a controller for a safety objective in the discrete-time setting does not imply the existence of such a controller in the continuous-time setting and vice versa, not even for systems with closed guards—contrary to the fact that continuous-time and discrete-time reachability problems coincide for such timed models [22], in particular also for timed-arc Petri nets [23]. However, if we restrict ourselves to the practically relevant subclass of urgent controllers that either react immediately to the environmental events or simply wait for another occurrence of such an event, then we can use the discrete-time methods for checking the existence of a continuous-time safety controller on closed timed-arc Petri nets. The algorithm for controller synthesis is implemented in our open source tool TAPAAL [24], including the memory optimization technique via PTrie [18, 19]. The experimental data show a promising performance on a large data-set of infinite job scheduling problems and scaled instances of the disk operation scheduling problem.

Related Work. An on-the-fly algorithm for synthesizing continuous-time controllers for both safety, reachability and time-optimal reachability for time automata was proposed by Cassez et al. [8] and later implemented in the tool UPPAAL TIGA [7]. This work is based on the symbolic verification techniques invented by Alur and Dill [5] in combination with ideas on synthesis by Pnueli et. al [6] and on-the-fly dependency graph algorithms suggested by Liu and Smolka [14]. For timed games, abstraction refinement approaches have been proposed and implemented by Peter et al. [25, 26] and Finkbeiner et al. [27] as an attempt to speed up synthesis, while using the same underlying symbolic representation as UPPAAL TIGA. These abstraction refinement methods are complementary to the work presented here. Our work uses the formalism of timed-arc Petri nets that has not been studied in this context before and we rely on the methods with discrete interpretation of time as presented by Andersen et. al [15]. As an additional contribution, we implement our solution in the tool TAPAAL, utilizing memory reduction techniques by Jensen et. al [18], and compare the performance of both discrete-time and continuous-time techniques. Control synthesis and supervisory control was also studied for the family of Petri net models [28–31] but these works do not

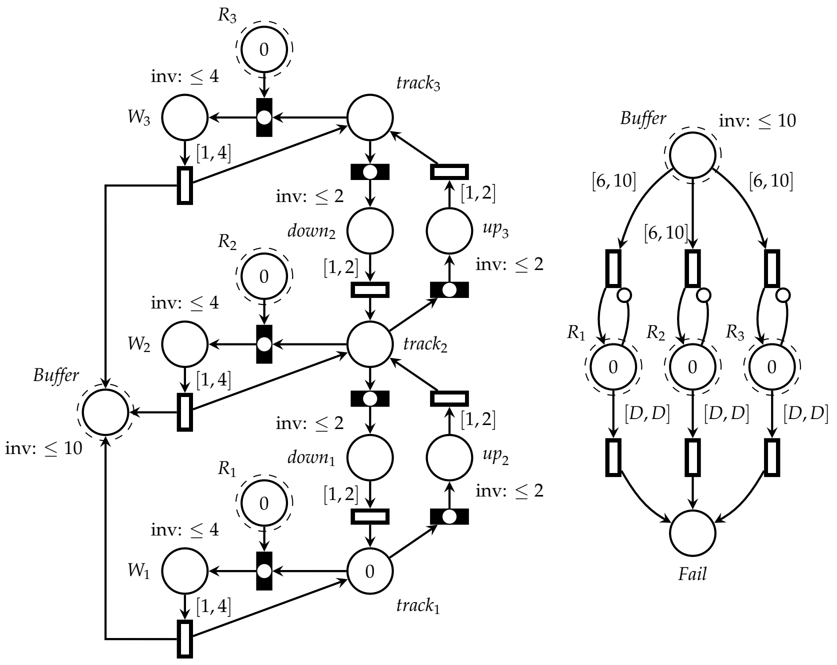


Fig. C.1: A timed-arc Petri net game model of a harddisk

consider the timing aspects.

2 Disk Operation Scheduling Example

We shall now provide an intuitive description of the timed-arc Petri net game of *disk operation scheduling* in Figure C.1, modelling the scheduler of a mechanical harddisk drive (left) and a number of read stream requests (right) that should be fulfilled within a given deadline D. The net consists of *places* drawn as circles (the dashed circle around the places R_1, R_2, R_3 and *Buffer* simply means that these places are shared between the two subnets) and *transitions* drawn as rectangles that are either filled (controllable transitions) or framed only (environmental transitions). Places can contain *tokens* (like the places R_1 to R_3 and the place $track_1$) and each token carries its own age. Initially all token ages are 0. The net also contains *arcs* from places to transitions (input arcs) or transitions to places (output arcs). The input arcs are further decorated with *time intervals* restricting the ages of tokens that can be consumed along the arc. If the time interval is missing, we assume the default $[0, \infty]$ interval not restricting the ages of tokens in any way.

In the initial *marking* (token configuration) depicted in our example, the two transitions connected by input arcs to the place $track_1$ are *enabled* and the controller can decide to *fire* either of them. As the transitions contain a white circle, they are *urgent*, meaning that time cannot pass as long at least one urgent transition is enabled. Suppose now that the controller decides to fire the transition on the left of the place $track_1$. As a result of firing the transition, the two tokens in R_1 and $track_1$ will be consumed and a new token of age 0 is produced to the place W_1. Tokens can be also transported via a pair of an input and output *transport arcs* (not depicted in our example) that will transport the token from the input to the output place while preserving its age.

In the new marking we just achieved, no transition is enabled due to the time interval $[1,4]$ on the input arc of the environmental transition connected to the place W_1. However, after one time unit passes and the token in W_1 becomes of age 1, the transition becomes enabled and the environment may decide to fire it. On the other hand, the place W_1 also contains the *age invariant* ≤ 4, requiring that the age of any token in that place may not exceed 4. Hence after age of the token reaches 4, time cannot progress anymore and the environment is forced to fire the transition, producing two fresh tokens into the places *Buffer* and $track_1$. Hence, reading the data from track 1 of the disk takes between 1ms to 4ms (depending on the actual rotation of the disk) and it is the environment that decides the actual duration of the reading operation.

The idea is that the disk has three tracks (positions of the reading head) and at each track $track_i$ the controller has the choice of either reading the data from the given track (assuming there is a reading request represented by a token in the place R_i) or move the head to one of the neighbouring tracks (such a mechanical move takes between 1ms to 2ms). The reading requests are produced by the subnet on the right where the environment decides when to generate a reading request in the interval between 6ms to 10ms. The number of tokens in the right subnet represents the parallel reading streams. The net also contains *inhibitor arcs* with a cirle-headed tip that prohibit the environmental transitions from generating a reading request on a given track if there is already one. Finally, if the reading request takes too long and the age of the token in R_i reaches the age D, the environment has the option to place a token in the place *Fail*.

The control synthesis problem asks to find a strategy for firing the controllable transitions that guarantees no failure, meaning that irrelevant of the behaviour of the environment, the place *Fail* never becomes marked (safety control objective). The existence of such a control strategy depends on the chosen value of D and the complexity of the controller synthesis problem can be scaled by adding further tracks (in the subnet of the left) or allowing for more parallel reading streams (in the subnet on the right). In what follows, we shall describe how to automatically decide in the discrete-time setting (where time can be increased only by nonnegative integer values) whether a controller strategy exists. As the controllable transitions are urgent in our example, the existence of such a discrete-time control strategy implies also the existence of a continuous-time control strategy where the environment is free to fire transitions after an arbitrary delay taken from the dense time domain—a result we formally state and prove in Section 4.

3 Definitions

Let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ and $\mathbb{N}_0^\infty = \mathbb{N}_0 \cup \{\infty\}$. Let $\mathbb{R}^{\geq 0}$ be the set of all nonnegative real numbers. A *timed transition system* (TTS) is a triple $(S, Act, \rightarrow)$ where S is the set of states, Act is the set of actions and $\rightarrow \subseteq S \times (Act \cup \mathbb{R}^{\geq 0}) \times S$ is the transition relation written as $s \xrightarrow{a} s'$ whenever $(s, a, s') \in \rightarrow$. If $a \in Act$ then we call it a *switch transition*, if $a \in \mathbb{R}^{\geq 0}$ we call it a *delay transition*. We also define the set of *well-formed closed time intervals* as $\mathcal{I} \overset{\text{def}}{=} \{[a, b] \mid a \in \mathbb{N}_0, b \in \mathbb{N}_0^\infty, a \leq b\}$ and its subset $\mathcal{I}^{\text{inv}} \overset{\text{def}}{=} \{[0, b] \mid b \in \mathbb{N}_0^\infty\}$ used in age invariants.

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

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

  - if $Type((t,p')) = Transport_j$ for some $(t,p') \in OA$ then there is exactly one $(p,t) \in IA$ such that $Type((p,t)) = Transport_j$,
  - if $Type((p,t)) = Transport_j = Type((t,p'))$ then $w((p,t)) = w((t,p'))$,

- $I : P \to \mathcal{I}^{inv}$ is a function assigning *age invariants* to places.

Remark 2
Note that for transport arcs we assume that they come in pairs (for each type $Transport_j$) and that their weights match. Also for inhibitor arcs and for input arcs to urgent transitions, we require that the guards are $[0,\infty]$. This restriction is important for some of the results presented in this paper and it also guarantees that we can use DBM-based algorithms in the tool TAPPAAL [24].

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

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

Definition 4 (Enabledness)
Assume a given TAPN $N = (P,T,T_{urg},IA,OA,g,w,Type,I)$. We say that a transition $t \in T$ is *enabled* in a marking M by the multisets of tokens

$$In = \{(p,x_p^1),(p,x_p^2),\ldots,(p,x_p^{w((p,t))}) \mid p \in {}^\bullet t\} \subseteq M$$

and

$$Out = \{(p',x_{p'}^1),(p',x_{p'}^2),\ldots,(p',x_{p'}^{w((t,p'))}) \mid p' \in t^\bullet\}$$

if

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

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

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

$$\forall (p,t) \in IA . Type((p,t)) = Inhib$$

$$\Rightarrow$$

$$|M(p)| < w((p,t))$$

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

$$\begin{gathered}\forall (p,t) \in IA . \forall (t,p') \in OA \text{ it holds that} \\ Type((p,t)) = Type((t,p')) = Transport_j \\ \Rightarrow \left(x^i_p = x^i_{p'} \wedge x^i_{p'} \in I(p')\right) \text{ for } 1 \leq i \leq w((p,t))\end{gathered}$$

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

$$\forall (t,p') \in OA) . \ Type((t,p')) = Normal$$

$$\Rightarrow$$

$$x^i_{p'} = 0 \text{ for } 1 \leq i \leq w((t,p')).$$

A TAPN N defines a TTS $T(N) \overset{\text{def}}{=} (\mathcal{M}(N), T, \rightarrow)$ where states are the markings and the transitions are as follows.

- If $t \in T$ is enabled in a marking M by the multisets of tokens In and Out then t can *fire* and produce the marking $M' = (M \setminus In) \uplus Out$ where $\uplus$ is the multiset sum operator and $\setminus$ is the multiset difference operator; we write $M \xrightarrow{t} M'$ for this switch transition.
- A time *delay* $d \in \mathbb{R}^{\geq 0}$ is allowed in M if
  - $(x + d) \in I(p)$ for all $p \in P$ and all $x \in M(p)$, and
  - if $M \xrightarrow{t} M'$ for some $t \in T_{urg}$ then $d = 0$.

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

Let $\rightarrow \overset{\text{def}}{=} \bigcup_{t \in T} \xrightarrow{t} \cup \bigcup_{d \in \mathbb{R}^{\geq 0}} \xrightarrow{d}$. By $M \xrightarrow{d,t} M'$ we denote that there is a marking M'' s.t. $M \xrightarrow{d} M'' \xrightarrow{t} M'$.

The semantics defined above in terms of timed transition systems is called the *continuous-time semantics*. If we restrict the possible delay transitions to take values only from nonnegative integers and the markings to be of the form $M : P \longrightarrow \mathcal{B}(\mathbb{N}_0)$, we call it the *discrete-time semantics*.

An example of a TAPN modeling an office fridge can be seen in Figure C.2. Initially, the *Fridge* contains two tokens, representing two boxes of yogurt. If a sudden *Hunger* occurs—which happens every 6 to 24 hours—the yogurts are moved to the *Eat* place. As the arcs moving the yogurts are diamond tipped transport-arcs (with weight two), the ingredients retain their age when moved. At the *Eat* place, we can now either put an uneaten yogurt back to the fridge (by firing the middle transition and hence preserving its age) or eat it and replace it with a new product, resetting the age of the (now replaced) yogurt—this occurs when firing the top transition. Notice that both transitions are urgent, denoted by the rectangle with an empty inner circle. This implies that we need to choose immediately whether or not we consume any of the yogurts. When replacing the yogurt, we also place a token in the *Watching* place. As the new yogurt is precious to us, we will for the next 12 hours be watching the fridge, inhibiting anyone to steal the yogurt—here modeled by an circle-tipped inhibitor arc. After exactly 12 hours, the token in *Watching* is forced to disappear as a combination of the guards *[12,12]* and the invariant $[0, 12]$ abbreviated as inv: $\leq$ 12. If any yogurt reaches an age between 36 to 42 hours, and we are not *Watching* the fridge, then someone may *Steal* a yogurt. At the same time, if any of the yogurts gets more than three days old, it may be sent to the *Bin*.

As this describes a plain TAPN, it is always nondeterministically decided what happens when, so it is surely possible that a yogurt gets stolen or is placed to the bin. However, there are some transitions that we are clearly in control of and others that are controlled by the environment. This brings us to the definition of a two player game.

3.1 Timed-Arc Petri Net Game

We extend the TAPN model into the game setting by partitioning the set of its transitions into the controllable and uncontrollable ones.

Definition 5 (Timed-Arc Petri Net Game)
A Timed-Arc Petri Net Game (TAPG) is a TAPN with its set of transitions T partitioned into the set of transitions T_{ctrl} owned by the controller and the set T_{env} owned by the environment.

Let us transform our fridge model from Figure C.2 into a TAPG, depicted in Figure C.3. In the game, we are able to define which transitions are con-

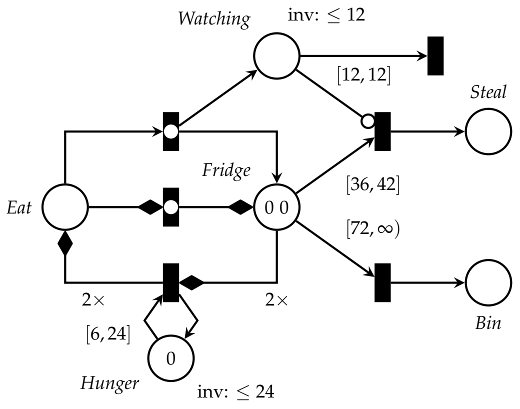


Fig. C.2: A TAPN model of the office fridge

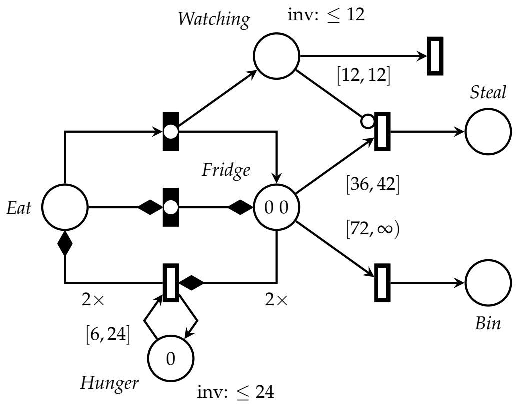


Fig. C.3: A TAPG model of the office fridge

trollable (denoted by solid rectangles, like the two urgent transitions in our figure) and which are not controllable and its firing is determined by the environment (denoted by frame rectangles). Hence we are not in control of when we get hungry or whether other person will steal or throw out our food.

Let G be a fixed TAPG. Recall that $\mathcal{M}(G)$ is the set of all markings over the net G. A *controller strategy* for the game G is a function

$$\sigma : \mathcal{M}(G) \to \mathcal{M}(G) \cup \{wait\}$$

from markings to markings or the special symbol *wait* such that

- if $\sigma(M) = wait$ then either M can delay forever ($M \xrightarrow{d}$ for all $d \in \mathbb{R}^{\geq 0}$), or there is $d \in \mathbb{R}^{\geq 0}$ where $M \xrightarrow{d} M'$ and for all $d'' \in \mathbb{R}^{\geq 0}$ and all $t \in T_{ctrl}$ we have that whenever $M' \xrightarrow{d''} M''$ then $M'' \overset{t}{\not\rightarrow}$, and

- if $\sigma(M) = M'$ then there is a $d \in \mathbb{R}^{\geq 0}$ and there is a $t \in T_{ctrl}$ such that $M \xrightarrow{d,t} M'$.

Intuitively, a controller can in a given marking M either decide to wait indefinitely (assuming that it is not forced by age invariants or urgency to perform some controllable transition) or it can suggest a delay followed by a controllable transition firing. The environment can in the marking M also propose to wait (unless this is not possible due to age invariants or urgency) or suggest a delay followed by firing of an uncontrollable transition. If both the controller and environment propose transition firing, then the one preceding with a shorter delay takes place. In the case where both the controller and the environment propose the same delay followed by a transition firing, then any of these two firings can (nondeterministically) happen.

This intuition is formalized in the notion of *plays* following a fixed controller strategy that summarize all possible executions for any possible environment.

Let $\pi = M_1 M_2 \ldots M_n \ldots \in \mathcal{M}(G)^\omega$ be an arbitrary finite or infinite sequence of markings over G and let M be a marking. We define the concatenation of M with π as $M \circ \pi = M M_1 \ldots M_n \ldots$ and extend it to the sets of sequences $\Pi \subseteq \mathcal{M}(G)^\omega$ so that $M \circ \Pi = \{M \circ \pi \mid \pi \in \Pi\}$.

Definition 6 (Plays According to the Strategy σ)
Let G be a TAPG, M a marking on G and σ a controller strategy for G. We define a function $\mathbb{P}_\sigma : \mathcal{M}(G) \rightarrow 2^{\mathcal{M}(G)^\omega}$ returning for a given marking M the set of all possible plays starting from M under the strategy σ.

- If $\sigma(M) = \mathit{wait}$ then $\mathbb{P}_\sigma(M) = \bigcup\{M \circ \mathbb{P}_\sigma(M') \mid d \in \mathbb{R}^{\geq 0},\ t \in T_{env},\ M \xrightarrow{d,t} M'\} \cup X$ where $X = \{M\}$ if $M \xrightarrow{d}$ for all $d \in \mathbb{R}^{\geq 0}$, or if there is $d' \in \mathbb{R}^{\geq 0}$ such that $M \xrightarrow{d'} M'$ and $M' \overset{d''}{\not\rightarrow}$ for any $d'' > 0$ and $M' \overset{t}{\not\rightarrow}$ for any $t \in T_{env}$, otherwise $X = \emptyset$.

- If $\sigma(M) \neq \mathit{wait}$ then according to the definition of controller strategy we have $M \xrightarrow{d,t} \sigma(M)$ and we define $\mathbb{P}_\sigma(M) = \bigcup\{M \circ \mathbb{P}_\sigma(\sigma(M))\} \cup \bigcup\{M \circ \mathbb{P}_\sigma(M') \mid d' \leq d, t' \in T_{env}, M \xrightarrow{d',t'} M'\}$.

The first case says that the plays from the marking M where the controller wants to wait consist either of the marking M followed by any play from a marking M' that can be reached by the environment from M after some delay and firing a transition from T_{env}, or a finite sequence finishing with the marking M if it is the case that M can delay forever, or we can reach a deadlock where no further delay is possible and no transition can fire.

The second case where the controller suggests a transition firing after some delay, contains M concatenated with all possible plays from $\sigma(M)$ and from $\sigma(M')$ for any M' that can be reached by the environment before or at the same time the controller suggests to perform its move.

We can now define the safety objectives for TAPGs. A safety objective is a Boolean expression over arithmetic predicates which observe the number of tokens in the different places of the net. Let φ be so a Boolean combination of predicates of the form $e \bowtie e$ where $e ::= p \mid n \mid e + e \mid e - e \mid e * e$ and where $p \in P$, $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ and $n \in \mathbb{N}_0$. The semantics of φ in a marking M is given in the natural way, assuming that p stands for $|M(p)|$ (the number of tokens in the place p). We write $M \models \varphi$ if φ evaluates in the marking M to true. We can now state the safety synthesis problem.

Definition 7 (Safety Synthesis Problem)
Given a marked TAPG G with the initial marking M_0 and a safety objective φ, decide if there is a controller strategy σ such that

$$\forall \pi \in \mathbb{P}_\sigma(M_0). \forall M \in \pi. M \models \varphi \, . \tag{C.1}$$

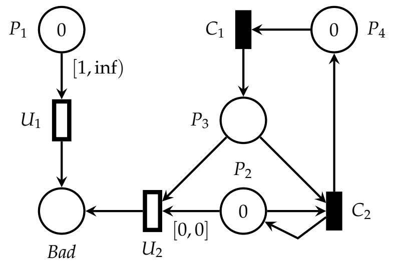
If Equation (C.1) holds then we say that σ is a *winning controller strategy* for the objective φ.

As an example, we might want to find a strategy for managing our yogurts in the office, modelled in Figure C.3. Formally, we can state our goal of not having our food stolen nor thrown out as the safety objective $\mathit{Steal} = 0 \land \mathit{Bin} = 0$. One can verify that this can be achieved by e.g. always consuming both yogurts every time hunger strikes. Another safe controller strategy is to return a yogurt into the fridge as long as it is strictly less than 12 hours old, otherwise to eat it.

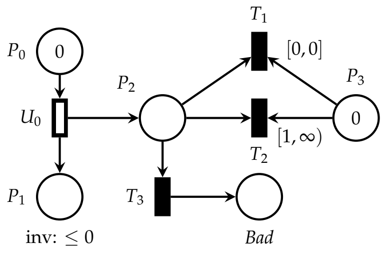
As another example in Figure C.1, we may wish to synthesize, for a given deadline D, a controller for the safety objective $\mathit{Fail} = 0$, hence yielding a controller that can serve three parallel streams with the maximal latency of D. The synthesis problem for this scenario is considerably more complex and with shall return to this problem in our experiments.

4 Synthesis in Continuous and Discrete Time

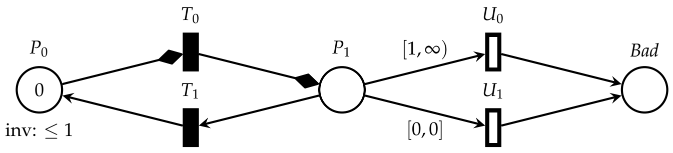
It is known that for classical TAPNs with closed intervals, the continuous and discrete-time semantics coincide up to reachability [23], which is what safety synthesis reduces to if the set of controllable transitions is empty. Contrary to this, Figures C.4a and C.4b show that this does not hold in general for safety strategies.



(a) A TAPG where *Bad* ≤ 0 can be guaranteed by the controller under the continuous-time semantics (by exploiting Zeno behaviour) but not under the discrete-time semantics.



(b) A TAPG where *Bad* ≤ 0 can be guaranteed by the controller under the discrete-time semantics but not under the continuous-time semantics.



(c) A TAPG where *Bad* ≤ 0 can be guaranteed by the controller under the continuous-time semantics (without exploiting Zeno behaviour) but not under the discrete-time semantics.

Fig. C.4: Difference between continuous and discrete-time semantics

For the game in Figure C.4a, there exists a strategy for the controller and the safety objective $Bad \leq 0$ but only in continuous-time semantics as the controller has to keep the age of the token in place P_1 strictly below 1, otherwise the environment can mark the place *Bad* by firing U_1. If the controller instead fires transition C_1 without waiting, U_2 becomes enabled and the environment can again break safety. Hence it is impossible to find a discrete-time strategy as even the smallest possible discrete delay of 1 time unit will enable U_1. However, if the controller waits an infinitesimal amount (in the continuous semantics) and fires C_1, then U_2 will not be enabled as the token in P_2 aged slightly. The controller can now fire C_2 and repeat this strategy over and over in order to keep the token in P_1 from ever reaching the age of 1.

The counter example described before relies on Zeno behaviour, however, this is not needed if we use transport arcs which do not reset the age of tokens (depicted by arrows with diamond-headed tips), as demonstrated in Figure C.4c. Here the only strategy for the controller to avoid marking the place *Bad* is to delay some fraction and then fire T_0. Any possible integer delay (1 or 0) will enable the environment to fire U_0 or U_1 before the controller gets to fire T_1. Hence we get the following proposition.

Proposition 1
There is a TAPG and a safety objective where the controller has a winning strategy in the continuous-time semantics but not in the discrete-time semantics.

Figure C.4b shows, on the other hand, that a safety strategy guaranteeing $Bad \leq 0$ exists only in the discrete-time semantics but not in the continuous-time semantics. Here the environment can mark the place *Bad* by initially delaying 0.5 and then firing U_0. This will produce a token in P_1 which restricts the time from progressing further and thus forces the controller to fire T_3 as this is the only enabled transition. On the other hand, in the discrete-time semantics the environment can either fire U_0 immediately, but then T_1 will be enabled, or it can wait (a minimum of one time unit), which in turn enables T_2. Hence the controller can in both cases avoid the firing of T_3 in the discrete-time semantics. This implies the following proposition.

Proposition 2
There is a TAPG and a safety objective where the controller has a winning strategy in the discrete-time semantics but not in the continuous-time semantics.

This indeed means that the continuous and discrete-time semantics are incomparable and it makes sense to consider both of them, depending on the concrete application domain and whether we consider discretized or continuous time. Nevertheless, there is a practically relevant subclass of the problem where we consider only urgent controllers and where the two semantics coincide. This class contains, for example, all digital circuit-controllers supervising a real-time environment and other applications such as worst-case-optimal controller synthesis for duration probabilistic automata [32].

We say that a given TAPG is with an *urgent controller* if all controllable transitions are urgent, formally $T_{ctrl} \subseteq T_{urg}$. For example the game net in Figure C.3 is with urgent controller as the two controllable transitions are both urgent. We can now state our main result of this section, showing that the discrete and continuous semantics coincide for the subclass timed-arc Petri net games with urgent controllers.

Theorem 7
Let G be a TAPG with urgent controller and let φ be a safety objective. There is a winning controller strategy for G and φ in the discrete-time semantics iff there is a winning controller strategy for G and φ in the continuous-time semantics.

4.1 Proof of Theorem 7

The rest of this section is now devoted to proving Theorem 7. Clearly, if the urgent controller has a winning strategy while the environment is allowed to make real-time delays, it also has a winning strategy if the environment is only allowed to perform discrete-time delays. We prove the opposite implication by showing that any universal evidence for nonexistence of a controller winning strategy in the continous semantics can be translated into such an evidence in discrete semantics (under the restriction that we consider only urgent controllers). We start by defining a witness for the fact that the controller does not have a winning strategy. A witness can allow for noninteger delays of the environmental moves and the main development of the proof is based on showing that such a witness can be transformed into another one with integer delays only. We do so by translating a witness into a system of linear constraints consisting only of difference constraints which guarantee that if the system has a real solution then it also has an integer solution.

Let us first define a witness for the nonexistence of a controller strategy for a given TAPG G with urgent controller. The intuition of the witness is to provide a strategy for the environment such that it considers all possible choices of the controller. Thus, for the environment choices there are only singleton continuations (if any), and for controller choices there is a multitude of possible continuations.

Definition 8
A *witness* is a function $\gamma : \mathcal{M}(G) \to 2^{\mathcal{M}(G)}$ for a marked TAPG G with urgent controller that for every marking M defines the next possible markings for the environment to consider. The function γ must satisfy the following conditions for every $M \in \mathcal{M}(G)$.

- If $M \not\models \varphi$ then $\gamma(M) = \emptyset$.
- Else if there is no $d \in \mathbb{R}^{\geq 0}$ and no $t \in T$ such that $M \xrightarrow{d} M' \xrightarrow{t}$ then $\gamma(M) = \emptyset$.
- Else if for all $t \in T_{ctrl}$ it holds that $M \overset{t}{\not\to}$ then $\gamma(M) = \{M'\}$ where for some $d \in \mathbb{R}^{\geq 0}$ and some $t \in T_{env}$ it holds that $M \xrightarrow{d,t} M'$.
- Else
  - either $\gamma(M) = \{M'\}$ such that $M \xrightarrow{t} M'$ for some $t \in T_{env}$, or
  - $\gamma(M) = \{M' \mid M \xrightarrow{t} M', t \in T_{ctrl}\}$ provided that there is at least one $t \in T_{ctrl}$ such that $M \xrightarrow{t}$.

Let us note that as the controller is urgent, the cases above enumerate all next markings to consider once we fix some environmental strategy but still consider all possible controller moves. Let $\pi \in \mathcal{M}(G)^+$ and by $last(\pi)$ we denote the last marking in the nonempty sequence of markings π. We now define $\Gamma_0 = \{M_0\}$ containing the initial marking and $\Gamma_k = \{\pi \circ M \mid \pi \in \Gamma_{k-1},\ M \in \gamma(last(\pi))\} \cup \{\pi \mid \pi \in \Gamma_{k-1},\ \gamma(last(\pi)) = \emptyset\}$ such that Γ_k describes all possible plays of length at most k under a fixed environmental strategy.

Observe now that a witness γ disproves the existence of any controller winning strategy for the objective φ iff there is $k \in \mathbb{N}$ such that $\Gamma_k = \Gamma_{k+1}$ and for all $\pi \in \Gamma_k$ we have $last(\pi) \not\models \varphi$ showing that every branch of the tree eventually breaks φ. We call such a witness a *counter witness* and denote Γ_k as $\mathbb{P}(\gamma)$. In what follows, whenever the witness function γ for a given marking M returns a set of next markings, we implicitly assume that we know what time delay and transition was fired (according to the definition of γ) in order to reach each individual marking from $\gamma(M)$.

Given a counter witness disproving the existence of any controller winning strategy, we shall now construct a linear constraint system describing a family of witnesses while preserving the plays in $\mathbb{P}(\gamma)$ and alternating only the delays during each play. The goal is to prove that the delays can be altered into integer delays while still preserving the counter witness. The technique of encoding a net computation via a linear constraint system is an adaption of the one used by Mateo et al. [23]. The notation from [23] for describing the linear programs corresponding to a given trace in the system is reused but generalized from a single trace to trees.

Let G be a marked TAPG with urgent controller, and γ be a counter witness for the safety objective φ. A *counter witness tree* is a graph where all plays from $\mathbb{P}(\gamma)$ are organized such that they share prefixes and the edges are labelled by a real-time delay and a transition that was fired after the delay in

$$\begin{tikzcd}
 & & & M_{0000} \arrow[r, dashed, "1.6{,} \mathit{Fail}"] & M_{00000} \not\models \varphi \\
 & M_{00} \arrow[r, dashed, "1.4{,} \mathit{track}_2"] & M_{000} \arrow[ur, "up_3"] \arrow[r, "W_2"] \arrow[dr, "down_1"'] & M_{0001} \arrow[r, dashed, "1.6{,} \mathit{Fail}"] & M_{00010} \not\models \varphi \\
M_0 \arrow[ur, "up_2"] \arrow[dr, "W_1"'] & & & M_{0002} \arrow[r, dashed, "1.6{,} \mathit{Fail}"] & M_{00020} \not\models \varphi \\
 & M_{01} \arrow[r, dashed, "3.0{,} \mathit{Fail}"] & M_{010} \not\models \varphi & &
\end{tikzcd}$$

Fig. C.5: The tree representation of a counter witness for $\varphi = \mathit{Fail} \leq 0$ for the example in Figure 1 where $D = 3$. The labels of the edges indicate which place will receive a new token by delaying and firing the corresponding transition. Dashed edges indicate the choices of the environment, solid edges are the controller choices (with default delay 0).

order to reach the next marking in the play (according to γ). An example of a counter witness tree is given in Figure C.5. Here the solid edges correspond to the controller choices (with the implicit delay 0 as we are restricted to an urgent controller) and the dashed edges are the environmental choices where arbitrary real-time delays are possible.

Remark 3
Strictly speaking, organizing the plays so that they share prefixes, may result in the fact that two different plays, after their prefixes diverged, can still both converge later on the same marking M. Hence this situation will not give us a tree structure as the node M will have more than one parent. This is undesirable, so we will implicitly assume that once plays in the game started to differ after a common prefix, any possible markings that are afterwards shared among such plays will appear in the witness tree is several copies (one for each such branch containing a shared marking).

The idea is now to create a constraint system from the counter witness where the concrete delays are replaced by variables, and then solve the resulting constraint system and argue that there is an integer solution to the system. Let us first construct a table Θ, assisting us in creating this constraint system and reflecting the classical firing rule of P/T nets (disregarding the timing constraints for a moment).

The table Θ that serves as a documentation for the counter witness γ and it is given in the form of a matrix with m rows (representing tokens) where m is the maximum number of tokens in any marking in $\mathbb{P}(\gamma)$ and n columns (representing markings in the counter witness tree) where n is the total number of nodes in the counter witness tree. Here we let M and M' to range over specific marking in the counter witness tree (columns) and y range over the tokens in the markings (rows). As each column of the table represents a single marking where not necessarily all m tokens are used, we mark each field $\Theta_{y,M}$ with either $\perp$ (unused token) or the pair (p, f)

where $p \in P$ represents the location of the token and $f \in \{0, \bullet\}$ is a flag signalling whether the age of the given token was reset to 0 or left unchanged (represented by the value $\bullet$). We let $\Theta^{place}_{y,M}$ and $\Theta^{flag}_{y,M}$ to denote the elements p and f of the pair of $\Theta_{y,M}$, respectively.

We can now define the notion of a valid table.

Definition 9 (Valid table for a witness γ)
Given a counter witness γ with its corresponding witness tree, a table Θ is *valid* if the following conditions are satisfied.

a) For the initial marking $M_0 = \{(p_1, x_1), (p_2, x_2), \ldots, (p_k, x_k)\}$, it holds that the first column of Θ with index M_0 is given by $\Theta_{y,M_0} \stackrel{\text{def}}{=} (p_y, x_y)$ if $1 \le y \le k$, and $\Theta_{y,M_0} \stackrel{\text{def}}{=} \perp$ if $k < y \le m$.

b) For each column M in the table where $M \models \varphi$ and an edge $M \xrightarrow{d,t} M'$ in the witness tree, there are two sets *Consume*, *Produce* $\subseteq \{1, 2, \ldots, m\}$ of token indices and a bijection $\mathcal{U} : \{0, \ldots, m\} \to \{0, \ldots, m\}$ such that

- *Consume* is giving the (indices of) tokens consumed when the transition t was fired in the marking M after the delay of d time units in order to reach the marking M' and where it holds that for all $p \in {}^\bullet t$ we have $w(p,t) = |\{y \in Consume \mid \Theta^{place}_{y,M} = p\}|$, and for all $p \in P \setminus {}^\bullet t$ we have $\{y \in Consume \mid \Theta^{place}_{y,M} = p\} = \emptyset$,
- *Produce* is giving the (indices of) tokens produced in the column M' by firing the transition t and where it holds that for all $p \in t^\bullet$ we have $w(t,p) = |\{y \in Produce \mid \Theta^{place}_{y,M'} = p\}|$ and for all $p \in P \setminus t^\bullet$ we have $\{y \in Produce \mid \Theta^{place}_{y,M'} = p\} = \emptyset$, and
- the bijection $\mathcal{U} : \{1, \ldots, m\} \to \{1, \ldots, m\}$ maps the indices of column M to those in the column M' such that
  1. if $|Consume| \le |Produce|$ and $y \in Consume$ then $\mathcal{U}(y) \in Produce$,
  2. if $|Consume| \ge |Produce|$ and $\mathcal{U}(y) \in Produce$ then $y \in Consume$,
  3. if $y \in Consume$ and $Type((\Theta^{place}_{y,M}, t)) = Transport_j = Type((t, p'))$ then $\mathcal{U}(y) = y$ and $\Theta^{place}_{y,M'} = p'$,
  4. if $y \in \{1, \ldots, m\} \setminus Consume$ and $\Theta_{y,M} \neq \perp$ then $\mathcal{U}(y) = y$ and $\Theta^{place}_{y,M'} = \Theta^{place}_{y,M}$,
  5. if $y \in \{1, \ldots, m\} \setminus Consume$ and $\Theta^{place}_{y,M} = \perp$ then either $\mathcal{U}(y) \in Produce$, or $\mathcal{U}(y) = y$ and $\Theta^{place}_{y,M'} = \perp$, and
  6. if $\Theta_{\mathcal{U}(y),M'} = \perp$ then $y \in Consume$ or $\Theta^{place}_{y,M'} = \perp$.

	M_0	M_{00}	M_{000}	M_{0000}	M_{00000}
1	$(R_1, 0)$	$(R_1, \bullet)$	$(R_1, \bullet)$	$(R_1, \bullet)$	$(\mathit{Fail}, 0)$
2	$(R_2, 0)$	$(R_2, \bullet)$	$(R_2, \bullet)$	$(R_2, \bullet)$	$(R_2, \bullet)$
3	$(R_3, 0)$	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$
4	$(\mathit{track}_1, 0)$	$(\mathit{up}_2, 0)$	$(\mathit{track}_2, 0)$	$(\mathit{up}_3, 0)$	$(\mathit{up}_3, \bullet)$

	M_{0001}	M_{00010}	M_{0002}	M_{00020}	M_{01}	M_{010}
1	$(R_1, \bullet)$	$(\mathit{Fail}, 0)$	$(R_1, \bullet)$	$(\mathit{Fail}, 0)$	$(W_1, 0)$	$(W_1, \bullet)$
2	$(W_2, 0)$	$(W_2, \bullet)$	$(R_2, \bullet)$	$(R_2, \bullet)$	$(R_2, \bullet)$	$(\mathit{Fail}, 0)$
3	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$	$(R_3, \bullet)$
4	$\perp$	$\perp$	$(\mathit{down}_1, 0)$	$(\mathit{down}_1, \bullet)$	$\perp$	$\perp$

Table C.1: A valid table for the counter witness tree in Figure C.5. The rows track the placement of the four tokens we have in the game (here $\perp$ means that a token is not present in the net). Otherwise each cell indicates where is the token located (first coordinate) and whether its age did not change compared to the previous marking (the value $\bullet$ in the second component) or if it was reset to the age 0 (again indicated in the second component).

- and for the column M' in the table holds:
  - if $\mathit{Type}((p,t)) = \mathit{Inhib}$ for some $p \in P$ then $|\{y \in \{1,\dots,m\} \mid \Theta^{place}_{y,M} = p\}| < w(p,t)$
  - for all $y \in \mathit{Produce}$, if $\mathit{Type}((t, \Theta^{place}_{y,M'})) = \mathit{Normal}$ then $T^{flag}_{y,M'} = 0$ else $T^{flag}_{y,M'} = \bullet$, and
  - if $y \notin \mathit{Produce}$ and $\Theta_{y,M'} \neq \perp$ then $T^{flag}_{y,M'} = \bullet$.

We can now transform the counter witness tree from Figure C.5, into such a table, as presented in Table C.1.

By following the indices, and the columns given by the table, one can reconstruct an untimed version of the tree given in Figure C.5, verifying that the table encodes a valid tree of traces in the classical untimed semantics of Petri nets.

Each column of the table with index M now defines the corresponding untimed marking $u(M)$ as follows.

Definition 10 (Untimed marking for column M)
Let M be a column index in a valid table. We define the untimed marking $u(M) \overset{\text{def}}{=} \{\Theta^{place}_{y,M} \in P \mid 1 \leq y \leq m\}$ as a multiset of all places where a token is present in the column M of the table.

From the construction, we can verify the validity of the following lemma.

Lemma 3 (Untimed consistency of a valid table)
Let Θ be a valid table and M and M' two of its column indices such that $M \xrightarrow{d,t} M'$. Then $u(M) \xrightarrow{t} u(M')$ in the classical (untimed) Petri net semantics.

While the construction so far preserves the movement of tokens, it does not encode the restrictions of token ages. We continue by encoding these timing constraints imposed by guards, age invariants and urgency.

We introduce first some notation. Let $M \xrightarrow{d,t} M'$ be an edge in the witness tree. By e_M we denote the global *execution time* of the transition t, yielding the total time elapsed since the computation started from the initial marking until the transition t was fired. Note that if $t \in T_{env}$ then from M there is a unique outgoing edge in the witness tree and if $t \in T_{ctrl}$ then there can be several outgoing edges but all delays on such edges are 0 as we deal with urgent controller. Hence the global execution time of firing t can be associated to the marking M. Now we can define a shorthand $age(y, M)$ for the age of the token given by the row y in a valid table for γ, just at the moment of firing the transition t.

Definition 11 (Token-age expression)
Let Θ be a valid table for γ and let M be its column index. We define $age(y, M)$, where $1 \leq y \leq m$, as the expression

$$\text{“}\ e_M - e_{M_{j-1}}\ \text{”}$$

such that in the witness tree for γ with the branch

$$M_0 M_1 \dots M_{j-1} M_j M_{j+1} \dots M_i \dots M_k \in \mathbb{P}(\gamma)$$

where $M = M_i$ and it is the case that $\Theta^{flag}_{y,M_j} = 0$ and $\Theta^{flag}_{y,M_{j+1}} = \Theta^{flag}_{y,M_{j+2}} = \dots = \Theta^{flag}_{y,M_i} = \bullet$. By convention M_{-1} is replaced with 0, such that i.e. $age(y, M_0) = e_{M_0}$.

We can now construct a system of inequalities from a valid table Θ.

Definition 12 (Constraint system)
Let Θ be a valid table for a counter witness γ. The constraint system $\mathcal{C}$ for Θ is the set of inequations over the variables e_M where M is a column index of Θ and $\mathcal{C}$ is constructed as follows. For each two column indices M and M' with an edge $M \xrightarrow{d,t} M'$ in the witness tree for γ, we

- add to $\mathcal{C}$ the constraint $e_M \leq e_{M'}$, and
- if $\Theta^{place}_{y,M} = p$ and $y \in Consume$[1] and $(p, t) \in IA$ and $g((p, t)) = [\ell, u]$, we add $\ell \leq age(y, M)$ and if $u \neq \infty$ also $age(y, M) \leq u$ to $\mathcal{C}$, and
- if M' enables some urgent transition then we add $e_{M'} - e_M = 0$ to $\mathcal{C}$.

[1]The set *Consume* for the edge $M \xrightarrow{d,t} M'$ was fixed in Definition 9, part b).

If the initial marking enables some urgent transition then we also add the constraint $e_{M_0} = 0$. Finally, we add the inequalities for age invariants such that for all $y \in \{1, \ldots, m\}$ and all column indices M:

- if $\Theta^{place}_{y,M} = p$ and $I(p) = [0, u]$ where $u \in \mathbb{N}_0$, we add the inequality $age(y, M) \leq u$ to $\mathcal{C}$.

Given our witness from Figure C.5, translated into a valid table in Table C.1, we can now construct the constraint system as follows. In order to simplify the notation, we shall write e.g. e_{010} instead of $e_{M_{010}}$.

- First, we add the inequalities for preserving the ordering of transition in the witness tree:

$$e_0 \leq e_{00},\ e_{00} \leq e_{000},\ e_{000} \leq e_{0000},\ e_{0000} \leq e_{00000}$$

$$e_{000} \leq e_{0001},\ e_{0001} \leq e_{00010}$$

$$e_{000} \leq e_{0002},\ e_{0002} \leq e_{00020}$$

$$e_0 \leq e_{01},\ e_{01} \leq e_{010}\ .$$

- For the nontrivial age invariants, we add

$$age(2, M_{0001}) \leq 4,\ age(2, M_{00010}) \leq 4$$

$$age(1, M_{01}) \leq 4,\ age(1, M_{010}) \leq 4$$

$$age(4, M_{00}) \leq 2,\ age(4, M_{0000}) \leq 2$$

$$age(4, M_{00000}) \leq 2,\ age(4, M_{0002}) \leq 2$$

$$age(4, M_{00020}) \leq 2$$

  that expand to

$$e_{0001} - e_{000} \leq 4,\ e_{00010} - e_{000} \leq 4$$

$$e_{01} - e_0 \leq 4,\ e_{010} - e_0 \leq 4$$

$$e_{00} - e_0 \leq 2,\ e_{0000} - e_{000} \leq 2$$

$$e_{00000} - e_{000} \leq 2,\ e_{0002} - e_{000} \leq 2$$

$$e_{00020} - e_{0002} \leq 2\ .$$

- For urgency, we add the constraints

$$e_0 = 0,\ e_{000} - e_{00} = 0\ .$$

- Finally for the guards on input arcs, we add the constraints

$$1 \leq age(4, M_{00}) \leq 2,\ 3 \leq age(1, M_{0000}) \leq 3$$
$$3 \leq age(1, M_{0001}) \leq 3,\ 3 \leq age(1, M_{0002}) \leq 3$$
$$3 \leq age(2, M_{01}) \leq 3$$

that expand to

$$1 \leq e_{00} - e_0 \leq 2,\ 3 \leq e_{0000} \leq 3$$
$$3 \leq e_{0001} \leq 3,\ 3 \leq e_{0002} \leq 3$$
$$3 \leq e_{01} \leq 3\ .$$

One can verify that the original global real-time delays from Figure C.5 form a solution of the constructed constraint system: $e_0 = 0$, $e_{00} = 1.4$, $e_{000} = 1.4$, $e_{0000} = 3.0$, $e_{00000} = 3.0$, $e_{0001} = 3.0$, $e_{00010} = 3.0$, $e_{0002} = 3.0$, $e_{00020} = 3.0$, $e_{01} = 3.0$, $e_{010} = 3.0$ Moreover, the constructed equation system also has an integer solution (this is not only a coincidence), e.g. $e_0 = 0$, $e_{00} = 2$, $e_{000} = 2$, $e_{0000} = 3$, $e_{00000} = 3$, $e_{0001} = 3$, $e_{00010} = 3$, $e_{0002} = 3$, $e_{00020} = 3$, $e_{01} = 3$, $e_{010} = 3$ and such a solution also forms a counter witness in our TAPG with urgent controller.

Lemma 4
Let γ be a counter witness for a TAPG G and the safety objective φ.

a) There is a valid table Θ for γ and the corresponding constraint system $\mathcal{C}$ has a solution.

b) Let e_M where M ranges over the columns of Θ be another solution of $\mathcal{C}$. Then for any play $M_0 M_1 M_2 \dots M_k \in \mathbb{P}(\gamma)$ such that $M_0 \overset{d_0,t_0}{\rightarrow} M_1 \overset{d_1,t_1}{\rightarrow} M_2 \overset{d_2,t_2}{\rightarrow} \dots M_k$, the same sequence of transitions $t_0, t_1, t_2, \dots, t_{k-1}$ can be fired from M_0 with the following delays: $M_0 \overset{e_{M_0},t_0}{\rightarrow} M'_1 \overset{e_{M_1}-e_{M_0},t_1}{\rightarrow} M'_2 \overset{e_{M_2}-e_{M_1},t_2}{\rightarrow} \dots M'_k$ such that $M_i \models \varphi$ iff $M'_i \models \varphi$.

c) The constraint system $\mathcal{C}$ has an integer solution.

Proof. a) From the requirements for a valid table, Lemma 3 and the construction of the constraint system, we can by case analysis verify that if we define $e_{M_i} = d_0 + d_1 + \dots + d_i$ in the play $M_0 M_1 M_2 \dots M_i \dots M_k \in \mathbb{P}(\gamma)$ where $M_0 \overset{d_0,t_0}{\rightarrow} M_1 \overset{d_1,t_1}{\rightarrow} M_2 \overset{d_2,t_2}{\rightarrow} \dots M_i \overset{d_i,t_i}{\rightarrow} \dots M_k$, then this forms a solution for the system $\mathcal{C}$. Hence by considering the timing given in the counter witness tree, we have a solution for the constraint system.

b) On the other hand, we notice that the system $\mathcal{C}$ contains all the timing constraints that are necessary for successfully executing all the transitions in the counter witness tree and for producing valid plays. Then simply computing the relative delays before a transition firing can be done by subtracting from the global execution time when the transition is fired the global execution time of the transition that we fired right before. Clearly, as both M_i and M'_i where achieved by firing the same sequence of transitions with just different delays, they have the same token distribution and hence $M_i \models \varphi$ iff $M'_i \models \varphi$.

c) As the constraint system $\mathcal{C}$ uses only difference constraints (difference of at most two variables is compared with a constant), it falls within the special subset of linear programming problems with totally unimodular matrices [33]. For this specific subclass, solving the constraint-system reduces to a shortest-path problem with integer weights only. This reduction implies that an integer solution of such a system exists [34, 35], provided that the system is solvable, which it is by part a) of the lemma.

From Lemma 4 we have that if it is possible to construct a counter witness for the existence of a controller strategy in the continuous-time semantics, then we can translate such a counter witness into a counter witness with integer delays only. This concludes the proof of Theorem 7.

5 Discrete-Time Algorithm for Synthesis

We shall now define the discrete-time algorithm for synthesising controller strategies for TAPGs. As the state-space of a TAPG is infinite in several aspects (the number of tokens in reachable markings can be unbounded and even for bounded nets the ages of tokens can be arbitrarily large), the question of deciding the existence of a controller strategy is in general undecidable (already the classical reachability is undecidable [36] for TAPNs).

We address undecidability by fixing a constant k, bounding the number of tokens in any marking reached by the controller strategy. This means that instead of checking the safety objective φ, we verify instead the safety objective $\varphi_k = \varphi \wedge k \geq \sum_{p \in P} p$ that at the same time ensures that the total number of tokens is at most k. This will, together with the extrapolation technique below, guarantee the termination of the algorithm. We note that in case the net is bounded, there is always some constants k for which checking the property φ_k is equivalent to the original safety property φ and hence the analysis is both sound and complete in this case.

5.1 Extrapolation of TAPGs

We shall now recall a few results from [15] that allow us to make finite abstractions of bounded nets (in the discrete-time semantics). The theorems and lemmas in the rest of this section also hold for continuous-time semantics, however, the finiteness of the extrapolated state-space is not guaranteed in this case.

Let $G = (P, T, T_{env}, T_{ctrl}, T_{urg}, IA, OA, g, w, Type, I)$ be a TAPG. In [15] the authors provide an algorithm for computing a function $C_{max} : P \to (\mathbb{N}_0 \cup \{-1\})$ returning for each place $p \in P$ the maximum constant associated to this place, meaning that the ages of tokens in place p that are strictly greater than $C_{max}(p)$ are irrelevant. The function $C_{max}(p)$ for a given place p is computed by essentially taking the maximum constant appearing in any outgoing arc from p and in the place invariant of p, where a special care has to be taken for places with outgoing transport arcs (details are discussed in [15]). In particular, places where $C_{max}(p) = -1$ are the so-called *untimed* places where the age of tokens is not relevant at all, implying that all the intervals on their outgoing arcs are $[0, \infty]$.

Let M be a marking of G. We split it into two markings $M_>$ and $M_\leq$ where $M_>(p) = \{x \in M(p) \mid x > C_{max}(p)\}$ and $M_\leq(p) = \{x \in M(p) \mid x \leq C_{max}(p)\}$ for all places $p \in P$. Clearly, $M = M_> \uplus M_\leq$.

We say that two markings M and M' in the net G are equivalent, written $M \equiv M'$, if $M_\leq = M'_\leq$ and for all $p \in P$ we have $|M_>(p)| = |M'_>(p)|$. This means that for tokens with ages below the maximum constants M and M' agree and also on the number of tokens above the maximum constant. Let us here introduce the notion of timed bisimilarity (we refer e.g. to [37] for more information).

Definition 13 (Timed Bisimulation)
A binary relation $\mathcal{R}$ over the markings is a timed bisimulation if for any two markings such that $M\mathcal{R}\hat{M}$ we have

- if $M \xrightarrow{d} M'$ then $\hat{M} \xrightarrow{d} \hat{M}'$ such that $M'\mathcal{R}\hat{M}'$,
- if $\hat{M} \xrightarrow{d} \hat{M}'$ then $M \xrightarrow{d} M'$ such that $M'\mathcal{R}\hat{M}'$,
- if $M \xrightarrow{t} M'$ then $\hat{M} \xrightarrow{t} \hat{M}'$ such that $M'\mathcal{R}\hat{M}'$, and
- if $\hat{M} \xrightarrow{t} \hat{M}'$ then $M \xrightarrow{t} M'$ such that $M'\mathcal{R}\hat{M}'$.

This means that delays and transition firings on one side can be matched by exactly the same delays and transition firings on the other side and vice versa.

We can now see that the above defined relation $\equiv$ is an equivalence relation and it is also a timed bisimulation.

Theorem 8 ([15])
The relation $\equiv$ is a timed bisimulation.

We can now define a function computing canonical representatives for each equivalence class of $\equiv$.

Definition 14 (Cut)
Let M be a marking. We define its canonical marking $cut(M)$ by $cut(M)(p) = M_{\leq}(p) \uplus \{ \underbrace{C_{max}(p)+1, \ldots, C_{max}(p)+1}_{|M_{>}(p)| \text{ times}} \}$.

The idea of cut is to utilize our knowledge from Theorem 8 that for any two markings which are timed bisimilar, it is sufficient to only do computations on one of them. The *cut* function takes this one step further and defines, for a group of bisimilar markings, a single canonical representative M capable of exhibiting the same behaviour as any marking M' where $M = cut(M')$.

Lemma 5 ([15])
Let M, M_1 and M_2 be markings. Then (i) $M \equiv cut(M)$, and (ii) $M_1 \equiv M_2$ if and only if $cut(M_1) = cut(M_2)$.

Note that as our safety objective φ deals only with the number of tokens in places but not with their actual ages, we get that $M \models \varphi$ if and only if $cut(M) \models \varphi$ for any marking M.

5.2 The Algorithm

After having introduced the extrapolation function *cut* and our enforcement of the k-bound, we can now design an algorithm for computing a controller strategy σ, provided such a strategy exists.

Algorithm 7 describes a discrete-time method to check if there is a controller strategy or not. It is centered around four data structures: *Waiting* set for storing markings to be explored, *Losing* set that contains marking where such a strategy does not exist, *Depend* function for maintaining the set of dependencies to be reinserted to the waiting list whenever a marking is declared as losing, and *Processed* set for already processed markings. All markings in the algorithm are always considered modulo the *cut* extrapolation. The algorithm performs a forward search by repeatedly selecting a marking M from *Waiting* and if it can determine that the controller cannot win from this marking, then M gets inserted into the set *Losing* while the dependencies of M are put to the set *Waiting* in order to backward propagate this information. If the initial marking is ever inserted to the set *Losing*, we can terminate and announce that a controller strategy does not exist. If this is not the case and there are no more markings in the set *Waiting*, then we terminate with success. In this case, it is also easy to construct the controller strategy by making choices so that the set *Losing* is avoided.

Algorithm 7: Safety Synthesis Algorithm

```
Input: A TAPG $G = (P, T, T_{env}, T_{ctrl}, T_{urg}, IA, OA, g, w, Type, I)$, initial
       marking $M_0$, a safety objective $\varphi$, a bound $k$.
Output: $tt$ if there exists a controller strategy ensuring $\varphi$ from $M_0$ and
        not exceeding $k$ tokens in any intermediate marking, $ff$
        otherwise
begin
    $Waiting := Losing := Processed := \emptyset$; $\varphi_k = \varphi \wedge k \geq \sum_{p \in P} p$;
    $M \leftarrow cut(M_0)$; $Depend[M] \leftarrow \emptyset$;
    if $M \not\models \varphi_k$ then
        $Losing \leftarrow \{M\}$
    else
        $Waiting \leftarrow \{M\}$
    while $Waiting \neq \emptyset \wedge cut(M_0) \notin Losing$ do
        $M \leftarrow pop(Waiting)$;
        $Succs_{env} := \{cut(M') \mid t \in T_{env}, M \xrightarrow{t} M'\}$;
        $Succs_{ctrl} := \{cut(M') \mid t \in T_{ctrl}, M \xrightarrow{t} M'\}$;
        $Succs_{delay} := \begin{cases} \emptyset & \text{if } M \overset{1}{\not\to} \\ \{cut(M')\} & \text{if } M \xrightarrow{1} M' \end{cases}$
        if $\exists M' \in Succs_{env}$ s.t. $M' \not\models \varphi_k \vee M' \in Losing$ then
            $Losing \leftarrow Losing \cup \{M\}$;
            $Waiting \leftarrow (Waiting \cup Depend[M]) \setminus Losing$;
        else
            if $Succs_{ctrl} \cup Succs_{delay} \neq \emptyset \wedge \forall M' \in Succs_{ctrl} \cup Succs_{delay}.$
             $M' \not\models \varphi_k \vee M' \in Losing$ then
                $Losing \leftarrow Losing \cup \{M\}$;
                $Waiting \leftarrow (Waiting \cup Depend[M]) \setminus Losing$;
            else
                if $M \notin Processed$ then
                    foreach $M' \in (Succs_{ctrl} \cup Succs_{env} \cup Succs_{delay})$ do
                        if $M' \notin Losing \wedge M' \models \varphi_k$ then
                            $Depend[M'] \leftarrow Depend[M'] \cup \{M\}$;
                            $Waiting \leftarrow Waiting \cup \{M'\}$;
        $Processed \leftarrow Processed \cup \{M\}$;
    return $tt$ if $cut(M_0) \notin Losing$, else $ff$
```

To prove that our algorithm is correct, we prove termination, soundness, and completeness.

Lemma 6 (Termination)
Algorithm 7 terminates.

Proof. Let us first argue that the sets *Waiting*, *Losing* and *Processed* can contain only an a priori bounded number of markings. To see this, we observe that markings added to *Processed* and *Losing* are only those that were previously removed from *Waiting*. Therefore it is sufficient to show that the number of different markings in *Waiting* is bounded. From the definition of φ_k at line 2, we can see that a marking satisfies φ_k only if is has at most k tokens and due to the test at line 23, only such markings can be inserted to *Waiting*. For the given number k, we notice that there are only finitely many extrapolated (by the function *cut*) markings with at most k tokens. Hence the set *Waiting* can contain only a bounded number of different extrapolated markings.

Unless the test at line 4 succeeds and we immediately terminate due to the check at line 8, the following invariant will hold during the execution of the main while-loop at lines 8 to 26: $\mathit{Waiting} \cap \mathit{Losing} = \emptyset$. This is due to the fact that at every line where we add markings to *Waiting* (lines 15, 19 and 25), we are guaranteed that such markings are not in the set *Losing*, and we only add a new marking to *Losing* (lines 14 and 18) when the marking was just popped from *Waiting* at line 9. Similarly, it is easy to observe that during the execution of the algorithm, the sets *Losing* and *Processed* are only growing (there are no lines that ever remove elements from these sets).

In each iteration of the while-loop, we can observe that the size of the set *Waiting* is either decreased by popping an element at line 9, or if new elements are added to *Waiting* then either the cardinality of *Losing* or *Processed* increases by one. We show that by analysing the lines where new elements are possibly added to *Waiting*. This can happen at lines 15 and 19 but at the previous lines 14 and 18, respectively, the marking M got inserted into *Losing*. Because this M was just popped from *Waiting* at line 9 and due to the previously introduced fact that $\mathit{Waiting} \cap \mathit{Losing} = \emptyset$, the cardinality of *Losing* is so increased. Another place where a new marking can be added to *Waiting* is at line 25. However, in this case we know that $M \notin \mathit{Processed}$ due to the test at line 21, and this implies that by executing line 26, the cardinality of the set *Processed* increased.

In summary, in each iteration of the while-loop either the size of *Waiting* decreases and if not then either the cardinality of *Losing* or *Processed* increases. As these sets are a priori bounded, we know that eventually the set *Waiting* becomes empty and the algorithm terminates (unless the algorithm already terminated due to the successful check $\mathit{cut}(M_0) \in \mathit{Losing}$ at line 8).

Having proved the termination, we now continue by proving soundness.

Lemma 7 (Soundness)
If Algorithm 7 returns *ff* then there is no controller winning strategy from M_0 for the safety objective $\varphi_k = \varphi \wedge k \geq \sum_{p \in P} p$.

Proof. We prove the lemma by establishing the invariant claiming that there is no controller winning strategy for any marking ever inserted into the set *Losing*. By observing that the algorithm returns *ff* only if $cut(M_0) \in Losing$ and the fact that by Theorem 8 the marking $cut(M_0)$ is losing if and only if M_0 is losing (note that our safety logic only queries the number of tokens in places but not their actual ages), this will conclude the proof of this lemma.

The invariant clearly holds before the while-loop is entered as *Losing* is empty. Assume now that the set *Losing* contains markings for which the controller has no strategy to satisfy φ_k and that we add a new marking M to the set *Losing*. We want to argue that the controller does not have a winning strategy from the marking M. A new marking M can be added only at lines 14 or 18.

- If the marking M was added to *Losing* at line 14 then surely, by the test at line 13, there is an environmental transition $M \xrightarrow{t} M'$ with $t \in T_{env}$ such that $cut(M') \not\models \varphi_k$ (and hence also $M' \not\models \varphi_k$) or $cut(M') \in Losing$. Hence, clearly the controlled cannot have a winning strategy from M as the environment can force the computation to the marking M' that either breaks the safety formula φ_k or $cut(M')$ belongs to *Losing* that contains only markings from which controller cannot win and as $cut(M')$ is losing for the controller, so is the marking M' by using Theorem 8.
- If the marking M was added to *Losing* at line 18 then the environment can let the controller to act from M, implying that the controller has to either fire some $t \in T_{ctrl}$ such that $M \xrightarrow{t} M'$ or perform a unit delay such that $M \xrightarrow{1} M'$ and in both situations the controller cannot win from M' as in the previous case because either $cut(M') \not\models \varphi_k$ or $cut(M') \in Losing$.

The soundness of the approach is hence introduced.

Finally, we can present the completeness lemma.

Lemma 8 (Completeness)
If Algorithm 7 returns *tt* then the controller has a winning strategy from M_0 for the safety objective $\varphi_k = \varphi \wedge k \geq \sum_{p \in P} p$.

Proof. Assume that Algorithm 7 returns *tt*. We shall define a winning strategy for the controller starting from the initial marking M_0. Note that as we consider here discrete-time semantics, it is enough for each marking M visited during a play to determine whether the controller's strategy should suggest

(i) to fire some controllable transition without any delay, (ii) to perform a delay of one time unit, or (iii) to do nothing (allowed only if none of the options (i) and (ii) are possible). Such a strategy can, in a straightforward way, be extended to the controller strategy as defined in Section 3.1 that proposes from a marking M to delay d time units followed by firing of a controllable transition, or to wait for ever. In what follows, we shall define a winning strategy for the controller from a marking M by defining it for the marking $cut(M)$. By Theorem 8 such a strategy from $cut(M)$ is a valid winning strategy also for the marking M.

The intuition behind the controller strategy is to make sure that any play includes only markings M such that $cut(M) \in Processed \setminus Losing$ (in the rest of this proof we consider the sets *Processed* and *Losing* after the termination of the algorithm). By examining that the set *Processed* contains only markings that were previously in *Waiting* and that any marking inserted into *Waiting* at line 25 must satisfy the proposition φ_k because of the check at line 23 (the markings inserted to *Waiting* at lines 15 and 19 were in the set *Waiting* earlier), we can see that all markings in $Processed \setminus Losing$ satisfy the formula φ_k. Hence staying in $Processed \setminus Losing$ is a safe controller strategy.

Let us so assume a marking $M \in Processed \setminus Losing$. We shall now determine whether the controller should propose to fire some controllable transition enabled in M, to delay in M for one time unit, or to do nothing. There are three cases to consider.

- If $M \xrightarrow{1} M'$ such that $cut(M') \in Processed \setminus Losing$ then the controller will propose to delay for one time unit. Clearly, there cannot be any $t \in T_{env}$ with $M \xrightarrow{t} M''$ where $M'' \not\models \varphi_k$ or $cut(M'') \in Losing$ as otherwise this would be detected at line 13 and it would imply that $M \in Losing$ (due to the back-propagation of this fact at line 15). This contradicts our assumption that $M \in Processed \setminus Losing$ and hence the controller's choice to delay one time unit is safe here.

- If $M \xrightarrow{1} M'$ but $cut(M') \notin Processed \setminus Losing$ then as $cut(M')$ was surely processes due to the classical forward search implemented at lines 21 to 25, this can only be possible if $cut(M') \in Losing$. As the fact that a marking is added to *Losing* is back-propagated at lines 15 and 19, and because $M \notin Losing$, we know due to the test at line 17 there is at least one $t \in T_{ctrl}$ such that $M \xrightarrow{t} M''$ such that $M'' \models \varphi_k$ and $cut(M'') \notin Losing$. Should this not be the case, then M would end up in the set *Losing* at line 18. This contradicts our initial assumption that $M \in Processed \setminus Losing$. The controller can in this case propose to fire one of such transitions t discussed above and the resulting marking will belong to $Processed \setminus Losing$. Clearly, as in the previous case, any environmental transition enabled in M must also lead to a marking from *Processed* $\setminus$ *Losing*.

- If $M \overset{1}{\not\to}$ then as before any environmental transition from M leads to a marking from *Processed* $\setminus$ *Losing* and if some controller transition is enabled at M then at least one such transition will bring us, by arguments already given above, to *Processed* $\setminus$ *Losing*.

We have so defined a controller strategy that will visit only markings from *Processed* $\setminus$ *Losing* and hence it is a winning controller strategy for the objective φ_k.

We are now ready to state the correctness of our algorithm that follows from Lemmas 6, 7 and 8.

Theorem 9 (Correctness)
Algorithm 7 terminates and returns *tt* if and only if there is a controller strategy for the safety objective $\varphi_k = \varphi \wedge k \geq \sum_{p \in P} p$.

Clearly, if the input Petri net game is k-bounded (there is no reachable marking with more than k tokens) then a marking satisfies φ_k if and only if it satisfies φ and hence our algorithm decides the existence of the controller winning strategy for the safety objective φ (in the discrete-time semantics). For unbounded nets, the existence of such a controller winning strategy is undecidable (already the reachability problem is undecidable for timed-arc Petri nets under discrete-time semantics [36]) but our algorithm can possibly find a controller winning strategy for the objective φ that visits only markings with a bounded number of tokens even for games on unbounded timed-arc Petri nets. There can though still be other controller winning strategies that may require an unbounded number of tokens in the visited markings. Such strategies will not be discovered by our algorithm.

6 Experiments

The discrete-time controller synthesis algorithm was implemented in the tool TAPPAAL [24] and we evaluate the performance of the implementation by comparing it to UPPAAL TIGA [7] version 0.18, the state-of-the-art continuous-time model checker for timed games. The experiments were run on AMD Opteron 6376 processor limited to using 19 GB of RAM[2] and with one hour timeout (denoted by ⌚).

Compared to our experiments presented at [1], the performance of TAPPAALimproved as we now use a more efficient PTrie [18] implementation that is both faster and has a smaller memory foot-print than the one used in [1].

[2]UPPAAL TIGA only exists in a 32 bit version, but for none of the tests the 4GB limit was exceeded for UPPAAL TIGA.

1 Stream	$D = 133$	$D = 173$	$D = 213$	$D = 253$	$D = 293$	$D = 333$	$D = 373$
Tracks	70	90	110	130	150	170	190
TAPPAAL	16.86s	36.84s	69.55s	119.68s	182.60s	271.82s	376.48s
UPPAAL	36.41s	76.63s	193.37s	351.17s	509.46s	1022.83s	1604.04s
2 Streams	$D = 19$	$D = 27$	$D = 35$	$D = 43$	$D = 51$	$D = 59$	$D = 67$
Tracks	6	8	10	12	14	16	18
TAPPAAL	1.17s	5.61s	19.13s	49.23s	114.23s	225.38s	426.99s
UPPAAL	19.11s	93.46s	436.15s	1675.85s	3328.66s	🕒	🕒
3 Streams	$D = 17$	$D = 21$	$D = 25$	$D = 29$	$D = 35$	$D = 39$	$D = 43$
Tracks	3	4	5	6	7	8	9
TAPPAAL	1.30s	8.5s	38.16s	129.27s	454.08s	1153.65s	🕒
UPPAAL	885.56s	🕒	🕒	🕒	🕒	🕒	🕒

Table C.2: Time in seconds to find a controller strategy for the disk operation scheduling for the smallest D where such a strategy exists.

6.1 Disk Operation Scheduling

In the disk operation scheduling model presented in Section 2 we scale the problem by changing the number of tracks and the number of simultaneous read streams. An equivalent model using the timed automata formalism was created for UPPAAL TIGA. We then ask whether a controller exists respecting a fixed deadline D for all requests. For each instance of the problem, we report the computation time for the smallest deadline D such that it is possible to synthesize a controller. Notice that the disk operating scheduling game net has an urgent controller, hence the discrete and continuous-time semantics coincide.

The results in Table C.2 show that our algorithm scales considerably better than TiGa (that suffers from the large fragmentation of zone federations) as the number of tracks increases (by which we scale the size of the problem) and it is significantly better when we add more read streams (by which we scale the concurrency and consequently also the number of timed tokens/-clocks).

6.2 Infinite Job Shop Scheduling

In our second experiment, infinite job shop scheduling, we consider the duration probabilistic automata [38]. Kempf et al. [32] showed that "non-lazy" schedulers are sufficient to guarantee optimality in this class of automata. Here non-lazy means that the controller only chooses what to schedule at the moment when a running task has just finished (the time of this event is determined by the environment). We here consider a variant of this problem that should guarantee an infinite (cyclic) scheduling in which processes—while competing for resources—must meet their deadlines. The countdown of a process is started when its first task is initiated and the process deadline is

2 Processes/7-13 tokens

Max Age	10 Tasks		12 Tasks		14 Tasks		16 Tasks		18 Tasks	
5	(100)	**54s**	**(100)**	**118s**	**(100)**	**238s**	**(100)**	**464s**	**(100)**	**661s**
$D \leq 144$	(100)	100s	(98)	413s	(85)	1201s	(35)	🕒	(18)	🕒
10	**(100)**	270s	**(100)**	**699s**	**(98)**	**1281s**	**(87)**	**2370s**	**(28)**	🕒
$D \leq 288$	(96)	**221s**	(69)	1443s	(43)	🕒	(16)	🕒	(1)	🕒
15	**(100)**	852s	**(85)**	2043s	(28)	🕒	**(15)**	🕒	**(5)**	🕒
$D \leq 432$	(87)	**315s**	(60)	**1960s**	(19)	🕒	(8)	🕒	(0)	🕒
20	(84)	1982s	(23)	🕒	(14)	🕒	(4)	🕒	**(2)**	🕒
$D \leq 576$	**(90)**	**554s**	**(66)**	**2914s**	**(34)**	🕒	(4)	🕒	(1)	🕒

3 Processes/10-19 tokens

Max Age	2 Tasks		3 Tasks		4 Tasks		5 Tasks		6 Tasks	
5	**(100)**	**2s**	**(100)**	**33s**	**(100)**	**295s**	**(71)**	**1375s**	**(42)**	🕒
$D \leq 57$	(99)	16s	(69)	1827s	(4)	🕒	(0)	🕒	(0)	🕒
10	**(100)**	**14s**	**(99)**	**328s**	**(50)**	**3538s**	**(20)**	🕒	**(8)**	🕒
$D \leq 114$	(98)	32s	(52)	3338s	(6)	🕒	(0)	🕒	(0)	🕒
15	**(100)**	44s	**(73)**	**1052s**	**(32)**	🕒	**(6)**	🕒	**(1)**	🕒
$D \leq 171$	(98)	**27s**	(50)	🕒	(1)	🕒	(0)	🕒	(0)	🕒

4 Processes/13-25 tokens

Max Age	2 Tasks		3 Tasks		4 Tasks		5 Tasks		6 Tasks	
5	**(95)**	**178s**	**(35)**	🕒	**(9)**	🕒	**(1)**	🕒	(0)	🕒
$D \leq 66$	(3)	🕒	(0)	🕒	(0)	🕒	(0)	🕒	(0)	🕒
10	**(62)**	**1805s**	**(12)**	🕒	**(3)**	🕒	(0)	🕒	(0)	🕒
$D \leq 132$	(0)	🕒	(0)	🕒	(0)	🕒	(0)	🕒	(0)	🕒

Table C.3: Results for infinite scheduling of DPAs. The first row in each age-instance is TAPPAAL, the second line is UPPAAL TIGA. The format is (X) Ys where X the number of solved instances (within 3600 seconds) out of 100 and Y is the median time needed to solve the problem. The largest possible constant for each row is given as an upper bound of the deadline D.

	2 Yogurts/9 tokens		3 Yogurts/13 tokens	
Scale	Uppaal	Tappaal	Uppaal	Tappaal
1/6	1.10s	0.22s	🕒	95.44s
1/3	1.11s	5.64s	🕒	OOM
1/2	1.12s	42.68s	🕒	OOM
2/3	1.15s	231.29s	🕒	OOM
5/6	1.11s	656.10s	🕒	OOM
1/1	1.11s	OOM	🕒	OOM

Table C.4: Results for the office fridge example from Figure C.3 where constants are scaled by the given factor. We limit the number of tokens in the net to 9 or 13. Time is given in seconds, OOM signifies that the tool exceeded the memory-limitation (19 GB) and 🕒 indicates that more than one hour of computation time was used.

met if the process is able to execute its last task within the deadline. After such a completed cycle, the process starts from its initial configuration and the deadline-clock is restarted. The task of the controller is to find a schedule such that all processes always meet their deadline. The problem can be modelled using urgent controller, in which case the discrete and continuous-time semantics coincide.

The problem is scaled by the number of parallel processes, number of tasks in each processes and the size of constants used in guards (except the deadline *D* that contains a considerably larger constant). For each set of scaling parameters, we generated 100 random instances of the problem and report on the number of cases where the tool answered the synthesis problem (within one hour deadline) and if more than 50 instances were solved, we also compute the median of the running time.

The comparison with Uppaal Tiga in Table C.3 shows a trend similar to the previous experiment. Our algorithm scales nicely as we increase the number of tasks as well as the number of processes. This is due to the fact that the zone fragmentation in TiGa increases with the number of parallel components and more distinct guards. When scaling the size of constants, the performance of the discrete-time method gets worse and eventually Uppaal Tiga can solve more instances.

6.3 Office Fridge Example

As the last experiment, we return to our motivating example from Figure C.3. In this experiment, we scale all constants in the model by the factors of $\frac{1}{6}$, $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{5}{6}$ and 1. We also scale the number of yogurts from 2 to 3—this also changes the weight on the transport-arc from *Fridge* to *Eat* to 3.

As illustrated in Table C.4, our algorithm is sensitive to the size of the constants. This is expected as the algorithm uses an explicit exploration of the discrete state-space. We observe that eventually our algorithm runs out of memory—in particular with the exact values as provided in Figure C.3. Compared to UPPAAL TIGA, it is apparent that the symbolic approach does not suffer from scaling the sizes of constants, however, it exceeds the one hour timeout for the case of 3 yogurts while we can still solve this problem for the scaling factor $\frac{1}{6}$.

7 Conclusion

We introduced timed-arc Petri net games and showed that for urgent controllers, the discrete and continuous-time semantics coincide. The presented discrete-time method for solving timed-arc Petri net games scales considerably better with the growing size of problems, compared to the existing symbolic methods. On the other hand, symbolic methods scale better with the size of the constants used in the model. In the future work, we may try to compensate for this drawback by using approximate techniques that "shrink" the constants to reasonable ranges while still providing conclusive answers in many cases, as demonstrated for pure reachability queries in [17]. Another future work includes the study of different synthesis objectives, as well as the generation of continuous-time strategies from discrete-time analysis techniques on the subclass of urgent controllers.

Acknowledgments. The research leading to these results has received funding from the project DiCyPS funded by the Innovation Fund Denmark, the Sino Danish Research Center IDEA4CPS and the ERC Advanced Grant LASSO. The third author is partially affiliated with FI MU, Brno, Czech Republic.

References

[1] P. G. Jensen, K. G. Larsen, and J. Srba, "Real-time strategy synthesis for timed-arc Petri net games via discretization," in *Proceedings of the 23rd International Symposium on Model Checking Software (SPIN'16)*, ser. LNCS, vol. 9641. Springer, 2016, pp. 129–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-32582-8_9

[2] A. Church, "Logic, arithmetic, and automata," in *Proc. Internat. Congr. Mathematicians (Stockholm, 1962)*. Djursholm: Inst. Mittag-Leffler, 1963, pp. 23–35.

[3] ——, "Application of recursive arithmetic to the problem of circuit synthesis," *Journal of Symbolic Logic*, vol. 28, no. 4, pp. 289–290, 1963.

[4] S. Jacobs, R. Bloem, R. Brenguier, R. Könighofer, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker, "The second reactive synthesis competition (SYNTCOMP 2015)," in *Proceedings of the Fourth Workshop on Synthesis (SYNT'15)*, ser. EPTCS, vol. 202, 2016, pp. 27–57. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.202.4

[5] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[6] A. Pnueli, E. Asarin, O. Maler, and J. Sifakis, "Controller synthesis for timed automata," in *System Structure and Control*, Citeseer. Elsevier, 1998.

[7] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime, "Uppaal-tiga: Time for playing games!" in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 121–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_14

[8] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *IN CONCUR 05, LNCS 3653*. Springer, 2005, pp. 66–80.

[9] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *CAV*, 1998, pp. 546–550.

[10] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST'06*, 2006, pp. 125–126.

[11] B. Berthomieu and F. Vernadat, "Time Petri nets analysis with TINA," in *Third International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society, 2006, pp. 123–124.

[12] G. Gardey, D. Lime, M. Magnin, and O. Roux, "Romeo: A tool for analyzing time Petri nets," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds. Springer, 2005, vol. 3576, pp. 261–272.

[13] D. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Automatic Verification Methods for Finite State Systems*, ser. LNCS. Springer, 1990, vol. 407, pp. 197–212. [Online]. Available: http://dx.doi.org/10.1007/3-540-52148-8_17

[14] X. Liu and S. A. Smolka, "Simple linear-time algorithms for minimal fixed points (extended abstract)," in *Proceedings of the 25th International*

Colloquium on Automata, Languages and Programming, ser. ICALP '98. London, UK, UK: Springer-Verlag, 1998, pp. 53–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=646252.686017

[15] M. Andersen, H. Larsen, J. Srba, M. Sørensen, and J. Taankvist, "Verification of liveness properties on closed timed-arc Petri nets," in *MEMICS'12*, ser. LNCS, vol. 7721. Springer, 2013, pp. 69–81.

[16] K. Jørgensen, K. G. Larsen, and J. Srba, "Time-darts: A data structure for verification of closed timed automata," in *Proceedings Seventh Conference on Systems Software Verification*, ser. EPTCS, vol. 102. Open Publishing Association, 2012, pp. 141–155.

[17] S. Birch, T. Jacobsen, J. Jensen, C. Moesgaard, N. Samuelsen, and J. Srba, "Interval abstraction refinement for model checking of timed-arc Petri nets," in *Proceedings of the 12th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'14)*, ser. LNCS, vol. 8711. Springer-Verlag, 2014, pp. 237–251.

[18] P. G. Jensen, K. G. Larsen, J. Srba, M. G. Sørensen, and J. H. Taankvist, "Memory efficient data structures for explicit verification of timed systems," in *NASA Formal Methods: 6th International Symposium*, ser. LNCS. Springer, 2014, vol. 8430, pp. 307–312. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06200-6_26

[19] P. G. Jensen, K. G. Larsen, and J. Srba, "PTrie: Data structure for compressing and storing sets via prefix sharing," in *Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC'17)*, ser. LNCS, vol. 10580. Springer, 2017, p. 18, to appear.

[20] T. Bolognesi, F. Lucidi, and S. Trigila, "From Timed Petri Nets to Timed LOTOS," in *Proc. of PSTV'90*. North-Holland, 1990, pp. 395–408.

[21] H. Hanisch, "Analysis of Place/Transition Nets with Timed Arcs and its Application to Batch Process Control," in *Proc. of Application and Theory of Petri Nets*, ser. LNCS, vol. 691. Springer, 1993, pp. 282–299.

[22] M. Bozga, O. Maler, and S. Tripakis, "Efficient verification of timed automata using dense and discrete time semantics," in *Correct Hardware Design and Verification Methods*. Springer, 1999, pp. 125–141.

[23] J. Mateo, J. Srba, and M. Sørensen, "Soundness of timed-arc workflow nets in discrete and continuous-time semantics," *Fundamenta Informaticae*, vol. 140, no. 1, pp. 89–121, 2015.

[24] A. David, L. Jacobsen, M. Jacobsen, K. Jørgensen, M. Møller, and J. Srba, "TAPAAL 2.0: Integrated development environment for timed-arc Petri

nets," in *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference*, ser. LNCS, vol. 7214. Springer, 2012, pp. 492–497.

[25] H. Peter, "Component-based abstraction refinement for timed controller synthesis," in *2013 IEEE 34th Real-Time Systems Symposium*, IEEE. IEEE Computer Society, 2009, pp. 364–374.

[26] H.-J. Peter, R. Ehlers, and R. Mattmüller, "Synthia: Verification and synthesis for timed automata." in *CAV*. Springer, 2011, pp. 649–655.

[27] B. Finkbeiner and H.-J. Peter, "Template-based controller synthesis for timed systems," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 392–406.

[28] B. Finkbeiner, "Bounded synthesis for Petri games," in *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*, ser. LNCS. Springer, 2015, vol. 9360, pp. 223–237.

[29] B. Finkbeiner and E. Olderog, "Petri games: Synthesis of distributed systems with causal memory," in *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification*, ser. EPTCS, vol. 161, 2014, pp. 217–230. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.161.19

[30] J. Raskin, M. Samuelides, and L. Begin, "Petri games are monotone but difficult to decide," Université Libre De Bruxelles, Technical Report, 2003.

[31] Q. Zhou, M. Wang, and S. Dutta, "Generation of optimal control policy for flexible manufacturing cells: A Petri net approach," *The International Journal of Advanced Manufacturing Technology*, vol. 10, no. 1, pp. 59–65, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01184279

[32] J.-F. Kempf, M. Bozga, and O. Maler, "As soon as probable: Optimal scheduling under stochastic uncertainty," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, N. Piterman and S. Smolka, Eds. Springer Berlin Heidelberg, 2013, vol. 7795, pp. 385–400. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_27

[33] J. Cong, B. Liu, and Z. Zhang, "Scheduling with soft constraints," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, Nov 2009, pp. 47–54.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms. third edition." 2009.

[35] A. Hoffman and J. Kruskal, "Integral boundary points of convex polyhedra, in Linear Inequalities and Related Systems (H. Kuhn and A. Tucker, Eds.)," *Annals of Maths. Study*, vol. 38, pp. 223–246, 1956.

[36] V. Ruiz, F. C. Gomez, and D. de Frutos Escrig, "On non-decidability of reachability for timed-arc Petri nets," in *Proceedings of the 8th International Workshop on Petri Net and Performance Models (PNPM'99)*, 1999, pp. 188–196.

[37] K. G. Larsen and Y. Wang, "Time-abstracted bisimulation: Implicit specifications and decidability," *Information and Computation*, vol. 134, no. 2, pp. 75 – 101, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540197926237

[38] O. Maler, K. G. Larsen, and B. Krogh, "On zone-based analysis of duration probabilistic automata," in *Proceedings of the 12th International Workshop on Verification of Infinite-State Systems*, 9 2010.

Paper D

Practical Controller Synthesis for MTL$_{0,\infty}$

Guangyuan Li, Peter Gjøl Jensen, Kim Guldstrand Larsen,
Axel Legay and Danny Bøgsted Poulsen

The paper has been published in the
Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, pp. 102–111, 2017.

Abstract

Metric Temporal Logic $MTL_{0,\infty}$ is a timed extension of linear temporal logic, LTL, with time intervals whose left endpoints are zero or whose right endpoints are infinity. Whereas the satisfiability and modelchecking problems for $MTL_{0,\infty}$ are both decidable, we note that the controller synthesis problem for $MTL_{0,\infty}$ is unfortunately undecidable. As a remedy of this we propose an approximate method to the synthesis problem, which we demonstrate to be adequate and scalable to practical examples. We define a method for converting $MTL_{0,\infty}$ formulas into (nondeterministic) Timed Game Büchi Automata and furthermore show how to construct determinized over- and underapproximation of a such. For the proposed method, we present a toolchain seamlessly integrating the needed components for practical $MTL_{0,\infty}$ synthesis. Lastly we demonstrate on a number of case-studies the applicability and scalability of the proposed method.

1 Introduction

Automatic controller synthesis offers the promise of a disruptive technology for developing correct-by-construction control software. In short, controller synthesis is concerned with the algorithmic construction of a control strategy, that will ensure a given behavioural specification to be satisfied regardless of the input provided by an environment. This problem was first stated in a discrete time setting by Church in 1962 in [1] and then theoretically solved for various specification formalisms in [2] and later works [3–8].

The synthesis problem is computationally harder for linear time logics than the satisfiability and model checking problems, and was for this reason considered intractable for a long time. Until recently, the intractability of proposed methods stemmed from the determinization of Büchi automata, which is a computationally hard problem. However, the synthesis problem has recently gained in practical performance due to the development of the so-called Safraless synthesis algorithms [9] that avoid the Büchi determinization phase. For real-time specification, the Metric Interval Temporal Logic (MITL [10]) is a logic that has proven its usefulness for speciciations [11] and thus a logic adequate for synthesing controllers. Unfortunately, the synthesis problem is known to be undecidable [12] for general MITL - but restricting the formulas to certain sub-classes the synthesis problem is rendered decidable [3, 13]. Overall, the main challenge in the real-time setting is that the Safraless approach is not always applicable as determinization is not possible in general. Allowing only upper or lower bounds on all until operators gives a sub-class of MITL called $MTL_{0,\infty}$. Although the satisfiability and model checking problems for $MTL_{0,\infty}$ are both decidable, the controller synthesis problem is still rendered undecidable – this follows trivially from the work on Event Clock Logic by Doyen et. al [12], as we will show later in this paper. However, it is still possible to synthesise controllers for some $\text{MTL}_{0,\infty}$ formulas by use of an approximate technique – as we present here.

The main obstacle for synthesising a controller for a $\text{MTL}_{0,\infty}$ objective is the construction of a Deterministic Timed Büchi Automaton equivalent to the objective. Unfortunately, in a previous work [14], we already argued that the sub-class with only upper bounds ($MTL_{\leq a}$) is non-deterministic in the sense that for some formulas no Deterministic Timed Büchi Automaton exists. In that work, we showed how to construct over- and under-approximating automata for a given specification. The construction was implemented in the tool Casaal and used for monitoring purposes. Furthermore, experimental results witnessed that the approximations were often exact and when not exact, at least tight. The often "exact and tight" propertys of our previous work gave hope that a similar construction could be made for the full class of $\text{MTL}_{0,\infty}$ formulas and used for controller synthesis. The idea is to parallel compose the automaton into the model of the environment and obtain a Timed Game with Büchi Objectives - a tool like Uppaal Tiga [15] can then be used to synthesise the controller. For the cases where an deterministic and exact Büchi automaton does not exist for the objective, the under-approximation may be used instead to construct a safe controller. For the purpose of synthesis for the over-approximation is mainly to verify the non-existence of a controller i.e. if you cannot synthesise a controller for the over-approximation you cannot synthesise one for the original objective.

In the current paper we show how to construct under- and overapproximation for $\text{MTL}_{0,\infty}$ objectives and we extend Casaal for this new construction. Experiments show that in many cases the approximations are in fact exact. Our main contribution is the approach for synthesising controllers for $\text{MTL}_{0,\infty}$ objectives, but along the way we also develop the – to our knowledge – first exact translation from $\text{MTL}_{0,\infty}$ to (non-deterministic) Timed Büchi Automata. That particular construction is since modified – using techniques developed in a previous work [14] – to obtain the final under-/over-approximating deterministic Timed Büchi Automata. Another contribution of the paper is a tool chain that seamlessly integrate Casaal and Uppaal Tiga [15] to form a practical way of synthesising controllers for $\text{MTL}_{0,\infty}$ objectives. We also demonstrate the applicability of our method on a number of case-studies, showing that the synthesis of controllers for $\text{MTL}_{0,\infty}$ objectives is feasible within a reasonable computation time for non-trivial formulas and reasonable model-sizes. Our experiments demonstrate that the over- and underapproximation is often exact, supporting our claim of an "exact and tight" property. In short, our contributions are

- a full and exact translation of $\text{MTL}_{0,\infty}$ objectives into (non-deterministic) Timed Büchi Automata,

- an automatic construction of deterministic over- and underapproximations, implemented in Casaal,
- seamless integration between Uppaal Tiga and Casaal in a single toolchain for synthesis, and
- a demonstration of the approach on a number of case-studies.

The paper is structured in the following way: in Section 2, we introduce timed games and $\mathrm{MTL}_{0,\infty}$. Section 3, proposes the translation from $\mathrm{MTL}_{0,\infty}$ to (non-deterministic) Timed Büchi Automata. Section 4 presents the tool chain and demonstrates the applicability and efficiency of the tool chain through a number of practical examples.

Related Work The continuous semantics and the pointwise semantics are two commonly adopted semantics for MITL. Rajeev Alur et al. in [10] proposed a procedure for translating MITL (under continuous semantics) into timed Büchi automata, this procedure has never been implemented in practice. Oded Maler et al. [16] proposed a procedure to translate MITL(under continuous semantics) into temporal testers (not timed Büchi automata), their procedure also has not been implemented. Marc Geilen [17] has implemented a procedure to translate bounded $\mathrm{MTL}_{0,\infty}$ to timed automata, the semantics he used is also the continuous semantics. As for pointwise semantics, in previous papers [14, 18], we have provided a constructions and a tool component (Casaal) for translating the safety fragment and co-safety of $\mathrm{MTL}_{0,\infty}$ into timed automata. In a recent paper [19], Thomas Brihaye et al. proposed a technique to translate MITL into Timed Büchi Automata through alternating timed automata. Their approach is based on a new (interval) semantics, where clock valuations are not real values, but intervals with real endpoints.

2 Timed Games and $\mathrm{MTL}_{0,\infty}$

Let us introduce the main formalism and definitions used throughout the text. A timed word ω over a finite set of actions Σ is an infinite sequence of time points and actions $\omega = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$, where for every i we have $a_i \in \Sigma$, $t_i \in \mathbb{R}_{\geq 0}$ and $t_{i+1} \geq t_i$. A timed word $\omega = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ is called non-Zeno if the sequence $\{t_i\}_{i\in\mathbb{N}}$ is unbounded.

Let X be a finite set of real-valued variables called clocks. A clock bound over X has the form $x \sim n$ or $x - y \sim n$, where $x, y \in X$, $\sim \in \{<, \leq, \geq, >\}$ and $n \in \mathbb{Z}_{\geq 0}$. We denote the set of all possible clock bounds over X by $\mathcal{B}(X)$, and let $\Theta(X)$ be the set of all Boolean formulas over $\mathcal{B}(X)$ (including conjunctions and disjunctions). A valuation over X is an element of $\mathbb{R}^X_{\geq 0}$, i.e. it is a function $v : X \to \mathbb{R}_{\geq 0}$. We let $\overline{0}$ be the valuation that assigns 0 to any clock from X. For a given valuation v, clock set $Y \subseteq X$ and real number

$\delta \in \mathbb{R}_{\geq 0}$ we let $v + \delta$ to be the valuation such that $(v + \delta)(x) = v(x) + \delta$ for every clock $x \in X$; and $v[Y]$ is the valuation where $v[Y](x) = 0$ if $x \in Y$ and $v[Y](x) = v(x)$ otherwise.

Definition 15
A Timed Büchi Automaton (TBA) over actions Σ is a tuple $\mathcal{A} = (L, \ell_0, X, F, E)$, where L is a finite set of locations, ℓ_0 is the initial location, X is a finite set of clocks, $F \subseteq L$ is a set of accepting locations, and $E \subseteq L \times \Sigma \times \Theta(X) \times 2^X \times L$ is a set of edges.

The semantics of a TBA $\mathcal{A}$ is defined by a Labeled Transition System (LTS) $(S, s_0, \rightarrow)$. The set of states $S = L \times \mathbb{R}^X_{\geq 0}$ of a TBA consists of pairs of locations and valuations over X. The initial state s_0 is $(l_0, \overline{0})$. There exists a *delay* transition $(l, v_1) \xrightarrow{\delta} (l, v_2)$, iff $\delta \in \mathbb{R}_{\geq 0}$ and $v_2 = v_1 + \delta$. There exists a *discrete* transition $(l_1, v_1) \xrightarrow{a} (l_2, v_2)$ if there exists an edge $e = (l_1, a, g, Y, l_2)$ such that $v_1 \models g$ and $v_2 = v_1[Y]$. In the latter case we say that an edge e is *enabled* in the state (l_1, v_1). A TBA is deterministic if any state (l, v) has at most one successor for any action $a \in \Sigma$.

A run ρ of a TBA $\mathcal{A}$ is an infinite sequence of alternating delay and discrete transitions $\rho = (l_0, \overline{0}) \xrightarrow{\delta_1} (l_1, v_1) \xrightarrow{a_1} (l_2, v_2) \xrightarrow{\delta_2} (l_3, v_3) \xrightarrow{a_2} (l_4, v_4) \xrightarrow{\delta_3} \ldots$. We say$\rho$ is accepting if $l_i \in F$ for infinitely many i. For $i \in \mathbb{N}$ we denote by ρ_i the finite prefix of ρ upto (l_i, v_i). We denote by $\mathit{Exec}(\mathcal{A})$ ($\mathit{Exec}^f(\mathcal{A})$) the set of all (finite) runs of $\mathcal{A}$. A timed word $\omega = (\delta_1, a_1)(\delta_1 + \delta_2, a_2)(\delta_1 + \delta_2 + \delta_3, a_3) \ldots$ is accepted by a TBA $\mathcal{A}$ if there exists an accepting run ρ for which ω is the corresponding timed word. We use $\mathcal{L}(\mathcal{A})$ to denote the set of all non-Zeno timed words accepted by $\mathcal{A}$. An ordinary Timed Automaton (TA) with final locations may be represented as a Timed Büchi Automaton by making all final locations terminal (looping) and accepting.

Definition 16
A Timed Game with Büchi conditions (TGB) over disjoint sets of controllable and uncontrollable actions, Σ_c and Σ_u, is a Timed Büchi Automaton $\mathcal{G} = (L, l_0, X, F, E)$ over $\Sigma_c \cup \Sigma_u$. A Timed Game (TG) is a TGB where all locations are accepting.

A strategy for a TGB $\mathcal{G}$ is a mapping σ, which given a finite run ρ describes how the run may proceed according to a controller. Formally $\sigma : \mathit{Exec}^f(\mathcal{G}) \rightarrow \Sigma_c \cup \{\lambda\}$, where λ indicates a delay action. A strategy σ is only allowed to suggest actions allowed by the TGB and thus, given a finite run ρ ending in a state (l, v), (1) if $\sigma(\rho) = a \in \Sigma_c$, then there must exist a transition $(l, v) \xrightarrow{a} (l', v')$ and (2) if $\sigma(\rho) = \lambda$, there must exist a positive delay $\delta \in \mathbb{R}_{>}$ such that $(l, v) \xrightarrow{\delta} (l', v')$.

Given a strategy σ, we say that an infinite run $\rho = (l_0, \overline{0}) \xrightarrow{\delta_1} (l_1, v_1) \xrightarrow{a_1} (l_2, v_2) \xrightarrow{\delta_2} (l_3, v_3) \xrightarrow{a_2} (l_4, v_4) \xrightarrow{\delta_3} \ldots$ is consistent with σ if for any $i \in \mathbb{N}$ either

$(l_i, v_i) \xrightarrow{a_i} (l_{i+1}, v_{i+1})$ and $(a_i \in \Sigma_u) \vee ((a_i \in \Sigma_c) \wedge a_i = \sigma(\rho_i))$, or $a_i = \delta \in \mathbb{R}_{>0}$ and $\sigma(\rho_i \xrightarrow{\delta'}) = \lambda$ whenever $\delta' < \delta$. We denote by *Outcome*$(\mathcal{G}, \sigma)$ all runs that are consistent with σ, and denote by $\mathcal{L}(\mathcal{G}, \sigma)$ the corresponding set of timed words.

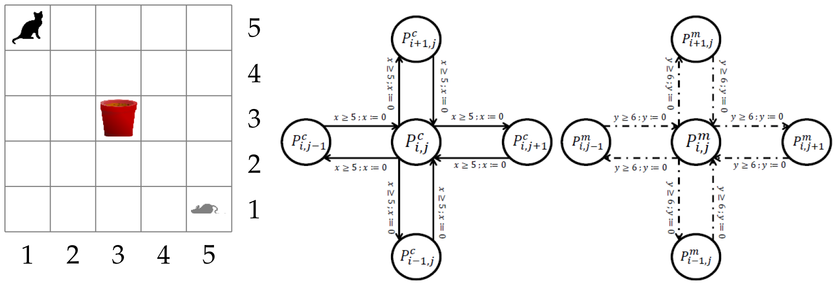
Given a TGB $\mathcal{G}$, we say that a strategy σ is winning if whenever $\rho \in$ *Outcome*$(\mathcal{G}, \sigma)$, then ρ is accepting. Given a TG $\mathcal{G}$ and a set of timed words $\mathcal{L}$, we say that a strategy σ is winning with respect to the objective $\mathcal{L}$ if $\mathcal{L}(\mathcal{G}, \sigma) \subseteq \mathcal{L}$. When $\mathcal{L}$ is expressed using a deterministic TBA, the following easily obtained result is crucial for the method we develop in the following sections:

Theorem 10
Let $\mathcal{G}$ be a TG and $\mathcal{A}$ a determinstic TBA. Then $\mathcal{G}$ has a winning strategy with respect to $\mathcal{L}(\mathcal{A})$ if and only if the TGB $\mathcal{G} \otimes \mathcal{A}$ has a winning strategy[1].

The emptiness problem for TBA is known to be PSPACE-complete [20] and the existence of winning strategies for TGB is EXPTIME-complete [21]. Moreover, for the synthesis problem, memoryless strategies are sufficient. The tools UPPAAL and UPPAAL TIGA provide efficient on-the-fly exploration of a finite *symbolic reachability graph*, where the nodes are *symbolic states*. A symbolic state S is a pair (l, Z), where l is a location and Z is a so-called *zone* being the set of valuations satisfying a given clock constraint $g \in \mathcal{B}(X)$. In particular, a winning strategy σ produced by UPPAAL TIGA for a given TGB is represented using zones. More precisely, for each location l, the representation R_σ gives a finite set of pairs $R_\sigma(l) = \{(Z_1, a_1), \ldots, (Z_n, a_n)\}$, where $a_i \in \Sigma_c \cup \{\lambda\}$ with $Z_i \cap Z_j = \emptyset$ if $i \neq j$. Given a state (l, v) the value of the strategy σ is simply a if $v \in Z$ with $(Z, a) \in R_\sigma(l)$.

Example 2.1
Consider the game in Fig. D.1, where a *Cat* chases a *Mouse* on a 5×5 grid. Initially the *Cat* and the *Mouse* are in positions $(1,5)$ and $(5,1)$, respectively. During the chase, they may both repeatedly move to any legal neighbouring position (note that position $(3,3)$ is illegal as there is already a flower-pot). Formally, the chasing game is modelled as a TG, being the product of a TG component for the *Cat* (controller) and a TG component for the *Mouse* (environment). For both the *Cat* and the *Mouse*, there is a minimum time-seperation between two consequitive moves, being 5 and 6 respectively. A simplest objective of the game is for the *Cat* to catch the *Mouse*, i.e. to bring the Timed Game into a product-state $(P^c_{i,j}, P^m_{i,j})$ for some legal position (i, j). More advanced objectives could be to ensure that the *Cat* will repeatedly catch the *Mouse*, and to do so within a maximum

[1] $\mathcal{G} \otimes \mathcal{A}$ is the TGB obtained as a synchronous product of $\mathcal{G}$ and $\mathcal{A}$, where accepting states are determined by the $\mathcal{A}$ component.



(a) The *Arena** **(b)** The *Cat* **(c)** The *Mouse*

Fig. D.1: The *Arena* for Chasing Game (a) and snippets for the TG components for the *Cat* (b) and for the *Mouse* (c). Edges between P^c nodes are controllable (full) and labeled with the guard $x \geq 5$ and the reset $x := 0$. Edges between P^m nodes are uncontrollable (dashed) and labeled with the guard $y \geq 6$ and the reset $y := 0$.

time-bound, say 40. In addition the *Cat* might for some reason want to repeatedly return to its initial position with some (minimum or maximum) time-seperation.

2.1 Metric Temporal Logic MTL$_{0,\infty}$

Applying Theorem 10 it suffices to express the objectives of a TG as deterministic TBAs in order to enable controller synthesis. However, often it will be far easier and significantly less error-prone to express objectives in a suitable temporal logic, e.g. MTL$_{0,\infty}$.

Definition 17
An MTL$_{0,\infty}$ formula φ over actions Σ is defined by the grammar

$$\varphi ::= true \,|\, a \,|\, \neg\varphi \,|\, \varphi_1 \wedge \varphi_2 \,|\, \bigcirc\varphi \,|\, \varphi_1 \mathsf{U}_{\sim d}\, \varphi_2, \,|\, \varphi_1 \widehat{\mathsf{U}}_{\sim d}\, \varphi_2$$

where $a \in \Sigma$, $\sim \in \{<, \leq, \geq, >\}$ and $d \in \mathbb{N}$.

The common abbreviations are: $false = \neg true$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 = (\neg\varphi_1) \vee \varphi_2$, $\varphi_1 \mathsf{R}_{\sim d}\, \varphi_2 = \neg(\neg\varphi_1 \mathsf{U}_{\sim d} \neg\varphi_2)$,$\varphi_1 \widehat{\mathsf{R}}_{\sim d}\, \varphi_2 = \neg(\neg\varphi_1 \widehat{\mathsf{U}}_{\sim d} \neg\varphi_2)$, $\Diamond_{\sim d}\, \varphi = true\, \mathsf{U}_{\sim d}\, \varphi$, $\widehat{\Diamond}_{\sim d}\, \varphi = true\, \widehat{\mathsf{U}}_{\sim d}\, \varphi$, $\Box_{\sim d}\, \varphi = false\, \mathsf{R}_{\sim d}\, \varphi$ and $\widehat{\Box}_{\sim d}\, \varphi = false\, \widehat{\mathsf{R}}_{\sim d}\, \varphi$.

The semantics of MTL$_{0,\infty}$ is defined over *infinite* timed words. Let w^i be the i-th suffix of the timed word w. For a given infinite timed word $w = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ and an MTL$_{0,\infty}$ formula φ, the satisfaction relation $w^i \models \varphi$ is defined inductively:

1. $w^i \models true$

2. $w^i \models a$ iff $a_i = a$
3. $w^i \models \neg\varphi$ iff $w^i \not\models \varphi$
4. $w^i \models \bigcirc\varphi$ iff $w^{i+1} \models \varphi$
5. $w^i \models \varphi_1 \wedge \varphi_2$ iff $w^i \models \varphi_1$ and $w^i \models \varphi_2$
6. $w^i \models \varphi_1 \mathsf{U}_{\sim d}\, \varphi_2$ where $\sim \in \{<, \leq, \geq, >\}$ iff there exists j such that $j \geq i$, $w^j \models \varphi_2$, $t_j - t_i \sim d$, and $w^k \models \varphi_1$ for all k with $i \leq k < j$
7. $w^i \models \varphi_1 \widehat{\mathsf{U}}_{\sim d}\, \varphi_2$ where $\sim \in \{<, \leq, \geq, >\}$ iff there exists j such that $j > i$, $w^j \models \varphi_2$, $t_j - t_i \sim d$, and $w^k \models \varphi_1$ for all k with $i < k < j$

An infinite timed word w satisfies an $\mathrm{MTL}_{0,\infty}$-formula φ iff $w^1 \models \varphi$. The language $\mathcal{L}(\varphi)$ of φ is the set of all infinite non-Zeno timed words that satisfy φ.

In [22], Doyen et al. proved that the controller synthesis problem for ECL (Event Clock logic) is undecidable. It is trival to check that all the future temporal operator in ECL can be defined in $\mathrm{MTL}_{0,\infty}$(for instance, $\rhd_{[a,b]}\varphi$ can be defined as $(\widehat{\Box}_{<a} \neg\varphi) \wedge \widehat{\Diamond}_{\leq b}\varphi)$. So the future fragment of ECL is a subset of $\mathrm{MTL}_{0,\infty}$. In [22] some past time temporal operators, e.g. $\ominus$(the last-time) and $\lhd_{=0}$ (the last occurrence), are used to encode the configurations and the infinite space-bounded runs for lossy 3-counter machines. We find that these past time formulas can be replaced by some future time formulas: for instance, $\Box(\mathcal{Q} \rightarrow (\ominus tick \wedge \lhd_{=0} tick))$ can be replaced by $\Box(\bigcirc\mathcal{Q} \rightarrow (tick \wedge \rhd_{=0}\mathcal{Q}))$, and $\Box(c \rightarrow (\lhd_{=0}\mathcal{AB}))$ can be replaced by $\Box(\bigcirc c \rightarrow (\mathcal{AB} \wedge \lhd_{=0} c))$. Thus the controller synthesis problem for the future fragment of ECL is also undecidable, and so is $\mathrm{MTL}_{0,\infty}$. We summarise the above reasoning with the following theorem.

Theorem 11
The $\mathrm{MTL}_{0,\infty}$ controller synthesis problem is undecidable.

Still, interesting properties exists for which we want to synthesise controllers.

Example 2.2
Reconsidering Example 2.1, we may formulate the first objective as $\Diamond Catch$, where $Catch = \vee_{i,j}(P^c_{i,j} \wedge P^m_{i,j})$. Repeated, and timed-bounded repeated catching may be expressed as the formulas $\Box\Diamond Catch$ and $\Box\Diamond_{\leq 40} Catch$. Finally, we may conjoin the formula $\Box\Diamond_{\geq 200} P^c_{1,5}$, which expresses that the *Cat* always revisits its initial position after at least 200 time-units.

In the following sections we present a procedure for translating $\mathrm{MTL}_{0,\infty}$ into a Timed Büchi Automaton. The translation goes by first translating the

$\mathrm{MTL}_{0,\infty}$ formula into a Transition-based Timed Büchi automaton (TTBA) and subsequently using the degeneralization algorithm proposed in [23] to translate the TTBA into an equivalent TBA.

Definition 18
A Transition-based Timed Büchi automaton (TTBA) over actions Σ is a tuple $\mathcal{A} = (L, l_0, X, F, E)$, where L is the set of locations, l_0 is the initial location, X is a finite set of clocks, F is a finite set of accepting conditions, and $E \subseteq L \times \Sigma \times \Theta(X) \times 2^F \times 2^X \times L$ is a set of edges.

The set of states (including initial state $s_0 = (l_0, \overline{0})$) and the set of delay transitions of a TTBA are defined as for a TBA. For a TTBA there exists a *discrete* transition $(l_1, v_1) \xrightarrow{a,F_1} (l_2, v_2)$ if there exists an edge (l_1, a, g, F_1, Y, l_2) in the TTBA such that $v_1 \models g$ and $v_2 = v_1[Y]$.

A run of a TTBA is an infinite sequence of alternating delay and discrete transitions $s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{a_1,F_1} s_2 \xrightarrow{\delta_2} s_3 \xrightarrow{a_2,F_2} s_4 \xrightarrow{\delta_3} \ldots$.

A timed word $w = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ over Σ is accepted by a TTBA $\mathcal{A}$ iff there exists states $s_0, s_1, s_2, \ldots$ where s_0 is the initial state of $\mathcal{A}$ such that $s_0 \xrightarrow{t_1} s_1 \xrightarrow{a_1,F_1} s_2 \xrightarrow{t_2-t_1} s_3 \xrightarrow{a_2,F_2} s_4 \xrightarrow{t_3-t_2} \ldots$ is a run of $\mathcal{A}$, and for each $f \in F$, there are infinitely many i where $f \in F_i$. We denote by $\mathcal{L}(\mathcal{A})$ the set of all non-Zeno timed words that are accepted by $\mathcal{A}$.

3 From $\mathrm{MTL}_{0,\infty}$ to Timed Büchi Automata

In this section, we first present the translation of an $\mathrm{MTL}_{0,\infty}$ into a TTBA, the translation goes through four phases. First we construct a closure of a formula in Section 3.1, giving the information needed for constructing extended formulas. In Section 3.2 we continue by constructing extended formulas containing book-keeping information for the time-constrained operators – such as monitoring clocks (and their resets). Next, we show how to transform a formula (via a normal-form of the formula) into a TTBA. Finally one can derive deterministic over- and underapproximations for such a TTBA, based on the classical subset-construction from NFA to DFA – this construction is only subtly different than the one presented by Bulychev [14], we thus refrain from repeating it.

In the rest of this section, we assume that φ is an $\mathrm{MTL}_{0,\infty}$ formula over Σ and has been transformed into positive normal form, where the negation operator ($\neg$) is not allowed ($\neg$*true* is replaced by *false*, $\neg a$ is replaced by $\bigvee_{b \in \Sigma \setminus \{a\}} b$ when a is an action in Σ), and additionally the syntax is extended with the *release* operator R being the dual of U. Without loss of generality, we also assume that all temporal operators occurring in φ are included in $\{\mathsf{U}_{\leq d}, \mathsf{R}_{\leq d}, \mathsf{U}_{\geq d}, \mathsf{R}_{\geq d}\}$.

3.1 Closures & Extended Formulas

We use $\mathsf{Sub}(\varphi)$ to denote all the sub-formulas of φ. For each $\varphi_1 \mathsf{U}_{\leq d}\, \varphi_2 \in \mathsf{Sub}(\varphi)$, we assign a clock $x_{(\varphi_1 \mathsf{U}_{\leq d}\, \varphi_2)}$. Intuitively these clocks are used by the resulting TTBA to determine the time progression since starting to evaluate whether $(\varphi_1 \mathsf{U}_{\leq d}\, \varphi_2)$ is satisfied. We let $X_{U\leq} = \{x_{(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)} \mid \varphi_1 \mathsf{U}_{\leq d}\, \varphi_2 \in \mathsf{Sub}(\varphi)\}$ be the set of all $\mathsf{U}_{\leq d}$-clocks and let $X_{U\geq}$, $X_{R\leq}$ and $X_{R\geq}$ be sets of clocks defined in a similar way. For untimed modalities, $\varphi_1 \mathsf{U} \varphi_2$ and $\varphi_1 \mathsf{R} \varphi_2$, we do not assign clocks and thus assume $d > 0$ when we write $\mathsf{U}_{\geq d}$ or $\mathsf{R}_{\geq d}$ in this section. For a clock bound $x \sim$d, where $\sim \in \{\leq, \geq\}$ we write $\overline{x \sim d}$ for the negated constraint e.g. $\overline{x \leq d} = x > d$.

The set of *basic formulas* for φ, written as $\mathsf{BF}(\varphi)$, is a finite set defined by the following rules:

1. If $\bigcirc \varphi_1 \in \mathsf{Sub}(\varphi)$, then $\varphi_1 \in \mathsf{BF}(\varphi)$
2. If $\varphi_1 \mathsf{U} \varphi_2 \in \mathsf{Sub}(\varphi)$ or $\varphi_1 \mathsf{U}_{\geq d}\, \varphi_2 \in \mathsf{Sub}(\varphi)$, then $\varphi_1 \mathsf{U} \varphi_2 \in \mathsf{BF}(\varphi)$
3. If $\varphi_1 \mathsf{R} \varphi_2 \in \mathsf{Sub}(\varphi)$ or $\varphi_1 \mathsf{R}_{\geq d}\, \varphi_2 \in \mathsf{Sub}(\varphi)$, then $\varphi_1 \mathsf{R} \varphi_2 \in \mathsf{BF}(\varphi)$
4. If $\varphi_1 \mathsf{U}_{\sim d}\, \varphi_2 \in \mathsf{Sub}(\varphi)$ where $\sim \in \{\leq, \geq\}$ and x is the clock assigned to $\varphi_1 \mathsf{U}_{\sim d}\, \varphi_2$, then
   $\varphi_1 \mathsf{U}_{\sim d - x}\, \varphi_2$, $x \sim d$, $\overline{x \sim d} \in \mathsf{BF}(\varphi)$.
5. If $\varphi_1 \mathsf{R}_{\sim d}\, \varphi_2 \in \mathsf{Sub}(\varphi)$ where $\sim \in \{\leq, \geq\}$ and x is the clock assigned to $\varphi_1 \mathsf{R}_{\sim d}\, \varphi_2$, then
   $\varphi_1 \mathsf{R}_{\sim d - x}\, \varphi_2$, $x \sim d$, $\overline{x \sim d} \in \mathsf{BF}(\varphi)$.

Informally, $\varphi_1 \mathsf{U}_{\leq d - x} \varphi_2$ encodes that the TTBA has started evaluating $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ in a previous state (s) and therefore from the current state (s') the formula $\varphi_1 \mathsf{U}_{\leq d'} \varphi_2$ should be satisfied where $d' = d - v(x)$ and $v(x)$ is the distance in time between s and s'. Similarly interpretation exists for $\mathsf{U}_{\leq d-x}$, $\mathsf{U}_{\leq d-x}$, $\mathsf{R}_{\leq d-x}$ and $\mathsf{R}_{\geq d-x}$. A formal definition is in Definition 19

As a conjunction of formulas can be represented as a set of formulas, we will use $2^{\mathsf{BF}(\varphi)}$ for both the powerset of $\mathsf{BF}(\varphi)$ and the set of all conjunctive formulas over $\mathsf{BF}(\varphi)$. Notice that because a conjunction with zero conjuncts is true then *true* $\in 2^{\mathsf{BF}(\varphi)}$. The closure of φ, denoted $\mathsf{CL}(\varphi)$, is the set of all positive Boolean combinations (i.e., without negation) over $\mathsf{BF}(\varphi)$. $\mathsf{CL}(\varphi)$ will form the set of non-initial locations for the deterministic TTBAs we construct for φ. Obviously, $\mathsf{CL}(\varphi)$ has only finitely many different non-equivalent formulas.

As information preserved in the closure and the basic formulas is not sufficient for the construction of the TTBA, we here introduce the notion of extended formula. Initially, for a given clock x we define the function $\mathsf{rst}(x)$ (and $\mathsf{unch}(x)$) for assigning clock-resets (and non-resets) of the clocks that track the temporal progress of the timed operators $\mathsf{U}_{\geq}$, $\mathsf{R}_{\leq}$ (and $\mathsf{U}_{<}$,$\mathsf{R}_{\geq}$). These functions are later used for constructing the TTBA and capture "For the validity of the formula, when starting to evaluate a time-constrained operator $\mathsf{U}_{\geq}$ or $\mathsf{R}_{\leq}$ ($\mathsf{U}_{\leq}$ or $\mathsf{R}_{\geq}$), if $\mathsf{rst}(x)$ ($\mathsf{unch}(x)$) then x must be (must not be) reset". We here note that $\mathsf{rst}(x)$ only if $x \in X_{\mathsf{U}_{\geq}} \cup X_{\mathsf{R}_{\leq}}$, and symmetrically $\mathsf{unch}(x)$ only if $x \in X_{\mathsf{U}_{\leq}} \cup X_{\mathsf{R}_{\geq}}$. One can observe the application of rst (unch) in the definition of the function β in the rules 11 and 13 (10 and 16).

Let $F_\varphi = \{\psi \in \mathsf{Sub}(\varphi) |$ if there exists ψ_1 such that $\psi_1 \mathsf{U} \psi \in \mathsf{Sub}(\varphi)$or $\psi_1 \mathsf{U}_{\geq d} \psi \in \mathsf{Sub}(\varphi)$ for some $d\}$. For each $\psi \in F_\varphi$ we introduce a boolean variable a_ψ that indicates ψ is assumed to be false at the present state and define $\{a_\psi \mid \psi \in F_\varphi\}$ as the set of all such boolean variables for subformulas of φ. We shall later use F_φ to construct the acceptance condition for the TTBA in Section 3.2.

Now we define $\mathsf{Ext}(\varphi)$, the set of extended formulas for φ, as the smallest set satisfying the following rules:

1. $\mathsf{Sub}(\varphi) \subseteq \mathsf{Ext}(\varphi)$
2. $\{a_\psi \mid \psi \in F_\varphi\} \subseteq \mathsf{Ext}(\varphi)$
3. If $\phi \in \mathsf{CL}(\varphi)$, then ϕ, $\bigcirc\phi \in \mathsf{Ext}(\varphi)$
4. If $x \in X_{\mathsf{U}_{\leq}}$ or $x \in X_{\mathsf{R}_{\geq}}$, then $\mathsf{unch}(x) \in \mathsf{Ext}(\varphi)$
5. If $x \in X_{\mathsf{U}_{\geq}}$ or $x \in X_{\mathsf{R}_{\leq}}$, then $\mathsf{rst}(x) \in \mathsf{Ext}(\varphi)$
6. If Φ_1, $\Phi_2 \in \mathsf{Ext}(\varphi)$, then $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2 \in \mathsf{Ext}(\varphi)$

$\mathsf{Ext}(\varphi)$ includes all the formulas needed to construct a TTBA for φ. Extended formulas can be interpreted using extended timed words. An *extended timed word* $\omega = (t_1, a_1, v_1)(t_2, a_2, v_2)(t_3, a_3, v_3)\ldots$ is a sequence where $w = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ is a timed word over Σ, and for every $i \in \mathbb{N}$, v_i is a clock valuation over X=$X_{U\leq} \cup X_{U\geq} \cup X_{R\leq} \cup X_{R\geq}$ such that for all $x \in X$, either $v_{i+1}(x) = v_i(x) + t_{i+1} - t_i$ or $v_{i+1}(x) = t_{i+1} - t_i$.

The semantics for extended formulas is naturally induced by the semantics of $\mathrm{MTL}_{0,\infty}$ formulas:

Definition 19
Let $\omega = (t_1, a_1, v_1)(t_2, a_2, v_2)(t_3, a_3, v_3)\ldots$ be an extended timed word and $\Phi \in \mathrm{Ext}(\varphi)$. The satisfaction relation $\omega^i \models_e \Phi$ is inductively defined as follows:

1. $\omega^i \models_e x \sim d$ iff $v_i(x) \sim d$, where $\sim \in \{<, \leq, >, \geq\}$
2. $\omega^i \models_e a_\psi$ iff $w^i \not\models \psi$
3. $\omega^i \models_e \mathsf{rst}(x)$ iff $v_{i+1}(x) = t_{i+1} - t_i$

4. $\omega^i \models_e \mathsf{unch}(x)$ iff $v_{i+1}(x) = v_i(x) + t_{i+1} - t_i$
5. $\omega^i \models_e \phi$ iff $w^i \models \phi$, if $\phi \in \mathsf{Sub}(\varphi)$
6. $\omega^i \models_e \varphi_1 \mathsf{U}_{\sim d-x} \varphi_2$ iff there exists j such that $j \geq i$, $w^j \models \varphi_2$, $t_j - t_i \sim d - v_i(x)$, and $w^k \models \varphi_1$ for all k with $i \leq k < j$, where $\sim \in \{\leq, \geq\}$.
7. $\omega^i \models_e \varphi_1 \mathsf{R}_{\sim d-x} \varphi_2$ iff for all $j \geq i$ such that $t_j - t_i \sim d - v_i(x)$, either $w^j \models \varphi_2$ or there exists k with $i \leq k < j$ and $w^k \models \varphi_1$, where $\sim \in \{\leq, \geq\}$.
8. $\omega^i \models_e \Phi_1 \wedge \Phi_2$ iff $\omega^i \models_e \Phi_1$ and $\omega^i \models_e \Phi_2$
9. $\omega^i \models_e \Phi_1 \vee \Phi_2$ iff $\omega^i \models_e \Phi_1$ or $\omega^i \models_e \Phi_2$
10. $\omega^i \models_e \bigcirc \Phi$ iff $\omega^{i+1} \models_e \Phi$

ω^i is a *model* of Φ if $\omega^i \models_e \Phi$ and two extended formulas are said to be *equivalent* if they have exactly the same models.

3.2 Constructing non-deterministic automata

Let us now construct a TTBA $\mathbb{A}_\varphi = (L, l_0, X, F, E)$ for which $\mathcal{L}(\mathbb{A}_\varphi) = \mathcal{L}(\varphi)$. The intuition of the elements of $\mathbb{A}$ is

- $L = \{\varphi\} \cup 2^{\mathsf{BF}(\varphi)}$, indicating that "in location $\ell \in L$ the future must satisfy ℓ",
- $\ell_0 = \varphi$, as the entire proposition is initially assumed satisfied,
- $X = X_{\mathsf{U}_\geq} \cup X_{\mathsf{R}_\leq} \cup X_{\mathsf{U}_\leq} \cup X_{\mathsf{R}_\geq}$ is the set of monitoring clocks,
- $F = F_\varphi$ is the set of accepting locations which must be visited infinitly often, and
- $E = (L \times \Sigma \times \Theta(X) \times 2^F \times 2^X \times L)$ is the transition relation where a single edge $(\psi, \alpha, g, F', \lambda, \psi')$ captures the inductive argument "when α is observed, ψ is true only if ψ' is true in the future given that g is satisfied and the clocks in λ are reset – implying that formulas in the set F' are false".

Edges of the TTBA are found by rewriting each $\psi \in \{\varphi\} \cup 2^{\mathsf{BF}(\varphi)}$ into a formula of $\mathsf{Ext}(\psi)$ that tells what action should be performed by the next transition, what clocks should be reset and what are the future obligations. This is similar to our work in [14] but in the current paper the rewriting also tells what subset of F_φ are assumed to be false. This difference is crucial since [14] did not consider Büchi acceptance conditions. The rewriting is done by the β function, capturing the condition for the input formula to be satisfied at the "current point in time" while using the next-operator to specify "under what condition is the next observation valid, what should be satisfied after the next observation, and what monitoring clocks should be reset" – a intuition is utilized when construction a normal-form over the rewritten formula. We inductively define β as

1. $\beta(\mathit{true}) = \mathit{true}$
2. $\beta(\mathit{false}) = \mathit{false}$
3. $\beta(\bigcirc\varphi_1) = \bigcirc(\varphi_1)$
4. $\beta(\varphi_1) = \varphi_1$, if φ_1 is an action or a clock bound
5. $\beta(\varphi_1 \wedge \varphi_2) = \beta(\varphi_1) \wedge \beta(\varphi_2)$
6. $\beta(\varphi_1 \vee \varphi_2) = \beta(\varphi_1) \vee \beta(\varphi_2)$
7. $\beta(\varphi_1 \mathsf{U} \varphi_2) = \beta(\varphi_2) \vee (\beta(\varphi_1) \wedge a_{\varphi_2} \wedge \bigcirc(\varphi_1 \mathsf{U} \varphi_2))$
8. $\beta(\varphi_1 \,\mathsf{R}\, \varphi_2)$= $\beta(\varphi_2) \wedge (\beta(\varphi_1) \vee \bigcirc(\varphi_1 \,\mathsf{R}\, \varphi_2))$
9. $\beta(\varphi_1 \,\mathsf{U}_{\leq d}\, \varphi_2)$ = $\beta(\varphi_2) \vee (\beta(\varphi_1) \wedge$ $\bigcirc((x \leq d) \wedge (\varphi_1 \,\mathsf{U}_{\leq d-x}\, \varphi_2)))$, where x is the clock assigned to $\varphi_1 \,\mathsf{U}_{\leq d}\, \varphi_2$
10. $\beta(\varphi_1 \,\mathsf{U}_{\leq d-x}\, \varphi_2)$ = $\beta(\varphi_2) \vee (\beta(\varphi_1) \wedge \mathsf{unch}(x) \wedge$ $\bigcirc((x \leq d) \wedge (\varphi_1 \, U_{\leq d-x}\, \varphi_2)))$
11. $\beta(\varphi_1 \,\mathsf{U}_{\geq d}\, \varphi_2)$ = $\beta(\varphi_1) \wedge (\beta(\varphi_2) \vee a_{\varphi_2}) \wedge \mathsf{rst}(x) \wedge$ $\bigcirc((\varphi_1 \,\mathsf{U}_{\geq d-x}\, \varphi_2) \vee ((x \geq d) \wedge (\varphi_1 \,\mathsf{U}\, \varphi_2)))$, where x is the clock assigned to $\varphi_1 \,\mathsf{U}_{\geq d}\, \varphi_2$
12. $\beta(\varphi_1 \,\mathsf{U}_{\geq d-x}\, \varphi_2)$ = $\beta(\varphi_1) \wedge (\beta(\varphi_2) \vee a_{\varphi_2}) \wedge$ $\bigcirc((\varphi_1 \mathsf{U}_{\geq d-x}\, \varphi_2) \vee ((x \geq d) \wedge (\varphi_1 \mathsf{U}\, \varphi_2)))$
13. $\beta(\varphi_1 \,\mathsf{R}_{\leq d}\, \varphi_2) = \beta(\varphi_2) \wedge (\beta(\varphi_1) \vee \mathsf{rst}(x) \wedge$ $\bigcirc((\varphi_1 \,\mathsf{R}_{\leq d-x}\, \varphi_2) \vee (x > d)))$, where x is the clock assigned to $\varphi_1 \,\mathsf{R}_{\leq d}\, \varphi_2$
14. $\beta(\varphi_1 \,\mathsf{R}_{\leq d-x}\, \varphi_2) = \beta(\varphi_2) \wedge (\beta(\varphi_1) \vee$ $\bigcirc((\varphi_1 \,\mathsf{R}_{\leq d-x}\, \varphi_2) \vee (x > d)))$
15. $\beta(\varphi_1 \,\mathsf{R}_{\geq d}\, \varphi_2)$ = $\beta(\varphi_1) \vee$ $\bigcirc(((x < d) \wedge (\varphi_1 \,\mathsf{R}_{\geq d-x}\, \varphi_2)) \vee (\varphi_1 \,\mathsf{R}\, \varphi_2))$, where x is the clock assigned to $\varphi_1 \mathsf{R}_{\geq d}\, \varphi_2$
16. $\beta(\varphi_1 \,\mathsf{R}_{\geq d-x}\, \varphi_2)$ = $\beta(\varphi_1) \vee (\mathsf{unch}(x) \wedge$ $\bigcirc(((x < d) \wedge (\varphi_1 \,\mathsf{R}_{\geq d-x}\, \varphi_2)) \vee (\varphi_1 \mathsf{R} \varphi_2)))$

As an example, let us briefly discuss rules 9 and 10. Here the transformation of rule 9 states that either φ_2 is true already or φ_1 is true, at the next observation the temporal constraint $x \leq d$ must be respected, and ϕ_1 must be true until ϕ_2 is true under the restriction that d is deducted by the amount monitored by x. The transformation of rule 10 is similar to the above, however, we also require that the monitoring clock x is not reset as the clock is vital for tracking the validity of the entire formula. The remaining rules 12-16 follow a similar pattern.

From the rules defining β, we note that the rules are constructed in such a way that alternative futures are separated by disjunction and each alternative is "guarded" by a necessary condition. For instance, let $\varphi = \alpha_1 \mathsf{U} \alpha_2$ for $\alpha_1, \alpha_2 \in \Sigma$, then for φ to be satisfied, either the next observation is α_2, and φ is satisfied, or the next observation is α_1, in which case, α_2 must not be observed – and the next observation must recursively satisfy φ again. As we will now generalize, this implies that our TTBA must have a transition from φ to φ, given that α_1 is observed and not α_2 – as well as a transition from φ to an accepting state under the condition that α_2 is observed.

From the definition of β we can see that $\beta(\psi)$ is an extended formula in $\mathsf{Ext}(\varphi)$. From the semantics given in Section 2.1 for $\mathrm{MTL}_{0,\infty}$, we know that $(\bigvee_{a \in \Sigma} a) \equiv \mathit{true}$ and for any $a, b \in \Sigma$, if $a \neq b$, then $a \wedge b \equiv \mathit{false}$. Using these facts and that $\bigcirc$ distributes over disjunction and conjunction, we can show by induction that $\beta(\psi)$ can be transformed equivalently into a disjunction of the following form:

$$\bigvee_{j=1}^{k} \left(a_j \wedge g_j \wedge A_j^a \wedge \mathsf{rst}(X_j) \wedge \mathsf{unch}(Y_j) \wedge \bigcirc(\psi_j)\right)$$

where for every j between 1 and k: $a_j \in \Sigma$ is an action, g_j is a conjunction of clock bounds, A_j is a subset of F, X_j is a subset of $X_{\mathsf{U}\geq} \cup X_{\mathsf{R}\leq}$, Y_j is a subset of $X_{\mathsf{U}\leq} \cup X_{\mathsf{R}\geq}$, $\psi_j \in 2^{\mathsf{BF}(\varphi)}$, $\mathsf{rst}(X_j)$ is the abbreviation of $\bigwedge_{x \in X_j} \mathsf{rst}(x)$, $\mathsf{unch}(Y_j)$ is the abbreviation of $\bigwedge_{x \in Y_j} \mathsf{unch}(x)$ and A_j^a is the abbreviation of $\bigwedge_{f \in A_j} a_f$.

We call each $a_j \wedge g_j \wedge A_j^a \wedge \mathsf{rst}(X_j) \wedge \mathsf{unch}(Y_j) \wedge \bigcirc(\psi_j)$ a basic conjunction of $\beta(\psi)$. From each basic conjunction $a_j \wedge g_j \wedge A_j^a \wedge \mathsf{rst}(X_j) \wedge \mathsf{unch}(Y_j) \wedge \bigcirc(\psi_j)$ of $\beta(\psi)$ we define the transitions from ψ to ψ_j by

$$(\psi, a_j, g_j, F_j, \lambda, \psi_j) \in E \quad \mathit{iff} \quad F_j = F \setminus A_j \text{ and } X_j \subseteq \lambda \subseteq (X \setminus Y_j)$$

Theorem 12
Let φ be an $\mathrm{MTL}_{0,\infty}$formula over Σ, and let $\mathbb{A}_\varphi$ be the TTBA built according to the procedure given above. Then $\mathcal{L}(\mathbb{A}_\varphi) = \mathcal{L}(\varphi)$.

Given a basic conjunction $a \wedge g \wedge A^a \wedge rst(X_1) \wedge \mathsf{unch}(Y_1) \wedge \bigcirc(\psi_1)$, its sub-formula $\mathsf{rst}(X_1) \wedge \mathsf{unch}(Y_1)$ tells us that the clocks in X_1 should be reset and the clocks in Y_1 should not be reset. It does not tell us what to do with the remaining clocks. In the construction so far we thus enumerate all the possible situations for clocks in $X \setminus (X_1 \cup Y_1)$, and hence get a non-deterministic choice as to what clocks to reset for a basic conjunction. However, we will see that this particular non-determinism can be avoided as there exists a best choice, which is to reset all clocks in $(X_{\mathsf{U}\leq} \cup X_{\mathsf{R}\geq}) \setminus Y_1$ and keep all clocks in $(X_{\mathsf{U}\geq} \cup X_{\mathsf{R}\leq}) \setminus X_1$ unchanged. The intuition of this choice is that each clock $x \in (X_{\mathsf{U}\leq} \cup X_{\mathsf{R}\geq})$ should be reset to zero unless $\mathsf{unch}(x)$ is asked to be true, and each clock $x \in (X_{\mathsf{U}\geq} \cup X_{\mathsf{R}\leq})$ should not be reset unless $\mathsf{rst}(x)$ is asked to be true. Using this approach, for a given basic conjunction $a \wedge g \wedge A^a \wedge rst(X_1) \wedge \mathsf{unch}(Y_1) \wedge \bigcirc(\psi_1)$ of $\beta(\psi)$, the transition $(\psi, a, g, F \setminus A, \lambda, \psi_1)$ with $\lambda = (X_1 \cup ((X_{U\leq} \cup X_{R\geq}) \setminus Y_1))$ will be the unique transition from ψ to ψ_1.

Theorem 13
Let φ be a $\mathrm{MTL}_{0,\infty}$ formula over Σ and $\mathcal{A}$ be the TTBA with best-choice-clock-resets above , then $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\mathbb{A}_\varphi)$.

Example 3.1
Let $\Sigma = \{p, !p\}$ and $f = \Box(p \rightarrow \bigcirc(!p\, \mathsf{U}_{\geq 10}\, p))$, then X = $X_{\mathsf{U}\geq}$ = $\{x\}$, $X_{\mathsf{U}\leq}$ = $X_{\mathsf{R}\geq}$= $X_{\mathsf{R}\leq}$ = $\emptyset$, F=$\{p\}$, and

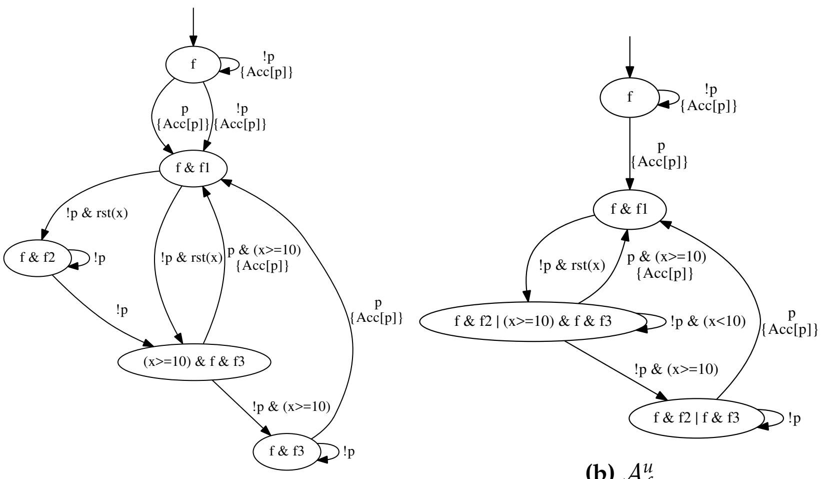
$$\begin{aligned} \beta(f) &= \beta(p \rightarrow \bigcirc(!p\mathsf{U}_{\geq 10}p)) \wedge \bigcirc(f) \\ &= (!p \vee \bigcirc(!p\mathsf{U}_{\geq 10}p)) \wedge \bigcirc(f) \\ &= (!p \wedge \bigcirc(f)) \vee (p \wedge \bigcirc(f \wedge f_1)) \vee (!p \wedge \bigcirc(f \wedge f_1)), \end{aligned}$$

where $f_1 =!p\, \mathsf{U}_{\geq 10}\, p$.

By the construction of $\mathbb{A}_f$, f has 6 outgoing transitions: (f, $!p$, true, $\{\}$, $\{p\}$, f), (f, $!p$, true, $\{x\}$, $\{p\}$, f), (f, p, true, $\{\}$, $\{p\}$, $f \wedge f_1$), (f, p, true, $\{x\}$, $\{p\}$, $f \wedge f_1$), (f, $!p$, true, $\{\}$, $\{p\}$, $f \wedge f_1$), (f, $!p$, true, $\{x\}$, $\{p\}$, $f \wedge f_1$).

By theorem 13, the following three can be removed: (f, $!p$, true, $\{x\}$, $\{p\}$, f), (f, p, true, $\{x\}$, $\{p\}$, $f \wedge f_1$), (f, $!p$, true, $\{x\}$, $\{p\}$, $f \wedge f_1$). The other three will remain. Similarly we can compute the outgoing transitions for $f \wedge f_1$, etc.

We observe that the structure of the disjunctive normal form gives the sufficient conditions for generating the under and over-approximations by applying the method discussed by Bulychev [14]. Fig.D.2(a) shows us the reduced TTBA $\mathcal{A}_f$ and Fig.D.2(b) shows us the determinized under-approximation.



(a) $\mathcal{A}_f$

(b) $\mathcal{A}^u_f$

Fig. D.2: The resulting automata for $f = \Box(p \rightarrow \bigcirc(!p\,\mathsf{U}_{\geq 10}\,p))$, where f_1 $=!p\,\mathsf{U}_{\geq 10}\,p$, f_2 = $!p\,\mathsf{U}_{\geq 10-x}\,p$, and f_3 = $!p\,\mathsf{U}\,p$.

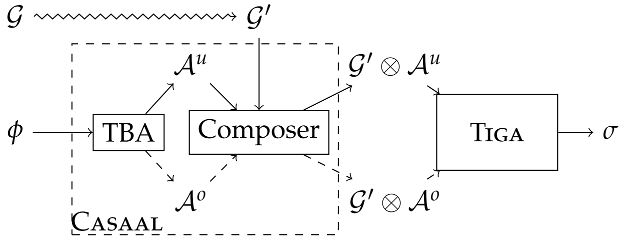


Fig. D.3: The tool chain workflow. The squiggly arrow indicate a manually performed step. The dashed arrow indicated the symmetric flow for the over approximation.

4 Tool Chain

The conversions of $\mathrm{MTL}_{0,\infty}$ to TBA has been implemented in the tool component Casaal- a stand alone tool that was first described in [14]. In Figure D.3 we illustrate the work flow of using Casaal in combination with Uppaal Tiga. The starting-point of the workflow is a standard Uppaal Tiga TG $\mathcal{G}$ together with an $\mathrm{MTL}_{0,\infty}$ property ϕ. The TG $\mathcal{G}$ is manually instrumented into $\mathcal{G}'$ to make the propositions in ϕ visible for the constructed monitor. Casaal then takes ϕ and constructs under- and over-approximating deterministic TBA $\mathcal{A}^u$ and $\mathcal{A}^o$. Furthermore, Casaal constructs a new combined TGB $\mathcal{G}' \oplus \mathcal{A}^u$ that is then passed on to Uppaal Tiga that will attempt to construct a winning strategy for the given property.

4.1 Experimental Evaluation

Cat and Mouse Example

As our first evaluation example we consider the *Cat* and *Mouse* game from Example 2.1. As objectives we consider the properties in Table D.1. The properties are choosen to span the expressive power of $\mathrm{MTL}_{0,\infty}$ covering both safety, liveness and mixtures. Table D.1 reports for each property the size of the generated TBA in terms of number of edges and locations. It furthermore reports the time and memory consumed by Casaal for constructing the TBA. For Uppaal Tiga we measure the time and memory used for synthesising a strategy. Finally the number of zones making up the strategy is reported as a means of quantifying the size of the strategy. We note that for all except one of the considered properties the tool chain provides exact answers as to whether a strategy exists or not[2].

Train-Gate Example

As our second example we consider the classic and scalable Uppaal Train-Gate example used for illustrating verification using Uppaal. Here the challenge is to automatically synthesise correct-by-construction control strategies with respect to various objectives using our tool chain. In the example a number of trains has to pass over a common bridge, while the control strategy to be synthesised will take different actions to ensure safety and a variation of (bounded) liveness objectives.

A train is initially in Safe location and may approach (uncontrollably) at any moment. When the train is approaching it will alert the controller, which in turn should be synthezised to take appropriate actions to ensure the objective. In particular, while approaching a train can only be stopped within 10 time units of its signal of approach. Once stopped, a train may at any point be restarted and granted access to the bridge by the controller – all depending on the given objective. We attempted to synthesise strategies for the controller for various parameterized formulas (see Table D.2).

For the properties proposed in Table D.2, we observe in Table D.3 that the under- and over-approximation yield the same TBA, hence the approximation is exact. We note that for all but ϕ_1 the formulas are tractable for Casaal with a running time of less than a minute. Still, it is interesting to see how the size of the TBA increases quite rapidly for ϕ_1 when adding an extra train. For ϕ_2, ϕ_3 and ϕ_4, even though the generated TBA are equivalent in size,

[2]All of the experiments reported were done using Casaal version 3.0 and a development snapshot of Uppaal Tiga running on an Intel Core i7-4578U 3.0 GHz processor. Uppaal Tiga was run using the following options: `-search-order 0 -backwards-order 0 -priority-order 2`. Both tools were limited to 4GB of memory and individually allowed to compute for up to 2 hours.

CASAAL				UPPAAL TIGA			
		$\lvert L \rvert$	$\lvert E \rvert$	$\exists\sigma$	Time	Mem	#Zones
-	ψ_1	5	15	*false*	1.10	19	N/A
-	ψ_2	4	12	*true*	21.69	25	11417
-	ψ_3	3	9	*true*	32.38	27	13022
-	ψ_4	5	18	*true*	35.77	29	12996
-	ψ_5	9	45	*false*	130.90	45	N/A
-	ψ_6	9	45	*true*	151.30	45	28851
-	ψ_7	9	45	*true*	150.44	44	28958
-	ψ_8	4	15	*true*	27.78	24	16676
-	ψ_9	3	9	*true*	4.17	27	8813
-	ψ_{10}	4	15	*false*	61.78	39	N/A
-	ψ_{11}	4	15	*true*	49.15	37	18380
U	ψ_{12}	7	32	*true*	11.79	50	111277
O	ψ_{12}	8	38	*true*	1.08	34	30812
U	ψ_{13}	6	43	*false*	0.00	0	N/A
O	ψ_{13}	8	61	*true*	1.09	32	24427

Table D.1: Experimental results for the *Cat* and *Mouse* model from Example D.1. Time is given in seconds and memory in megabytes. We omit the resource consumption of CASAAL as these are negligible for the given formulas. Formulas marked with a dash yield equivalent TBAs for the under- and over-approximation, while *U* and *O* signify the under- and over-approximation respectively. Formulas, with description, can be found in Table D.6.

ϕ_0	No collisions and all trains will cross after approaching $(\Box \neg collision) \wedge \bigwedge_{i=1}^{n} (\Box (Appr[i] \implies \Diamond Cross[i]))$
ϕ_1	No collisions and all trains will cross after 10 time units after approaching. $(\Box \neg collision) \wedge \bigwedge_{i=1}^{n} ((\Box (Appr[i] \implies \Diamond_{\geq 10} Cross[i])))$
ϕ_2	No collisions and all trains will cross before 10 ten time units after approaching. $(\Box \neg collision) \wedge \bigwedge_{i=1}^{n} ((\Box (Appr[i] \implies \Diamond_{\leq 10} Cross[i])))$
ϕ_3	No collisions and all trains will cross before 30 time units after approaching. $(\Box \neg collision) \wedge \bigwedge_{i=1}^{n} ((\Box (Appr[i] \implies \Diamond_{\leq 30} Cross[i])))$
ϕ_4	No collisions and all trains will cross before 50 time units after approaching. $(\Box \neg collision) \wedge \bigwedge_{i=1}^{n} ((\Box (Appr[i] \implies \Diamond_{\leq 50} Cross[i])))$

Table D.2: Specifications for the Train-Gate example.

the time for synthesis used by Uppaal Tiga differs quite a lot. We here observe that ϕ_2 through ϕ_4 are structurally the same formula and differ only in the bounds provided. Thus the TGBs constructed by Casaal will also be structurally equivalent – however, for Uppaal Tiga the difference in the provided bounds, and hence the clock-guards of the constructed TGB, will result in different intersections of zones. As a result, we can observe an increasing zone-fragmentation from ϕ_2 through ϕ_4.

Chinese Juggler In our third case-study, we consider the synthesis problem for the scalable Chinese Juggler. The juggler is tasked with keeping n plates balancing on sticks. If a plate has been balancing for more than s time units, it can turn unstable. If a disk is unstable, after u time-units it can fall to the ground and shatter. At the same time, the juggler can only stabilize the disks at a certain pace, leaving him to choose which plates to stabilize for achieving his goal. A classical control problem is to ensure that no disk breaks. We instantiate the disks with $s_1 = 8, s_2 = 8, s_3 = 20, s_4 = 20$ and $u_1 = 6, u_2 = 3, u_3 = 10, u_4 = 3$ for the 0-4 and syntehsise controllers for the properties shown in Table D.5 for $1 \leq n \leq 4$.

We observe in Table D.4 that for all the specifications for the Chinese Juggler, the constructed TTBA is exact. We also note that the resource-consumption of Casaal is negligible. For the untimed specification ϕ_0 we can see that the constructed TTBA is small, resulting in good scalability. However, ϕ_2 and ϕ_3 in particular, show an exponential growth in the number of transitions, leading to an explosion of the number of zones needed to represent the strategy, ultimately leading to poor performance for the largest

	Casaal					Uppaal Tiga			
	#Trains	Time	Mem	#Loc	#Edges	$\exists\sigma$	Time	Mem	#Zones
ϕ_0	1	0.04	35	3	6	*true*	0.01	9	11
ϕ_1	1	0.04	35	5	17	*true*	0.01	9	21
ϕ_2	1	0.04	35	3	8	*true*	0.02	9	6
ϕ_3	1	0.04	35	3	8	*true*	0.02	9	12
ϕ_4	1	0.04	35	3	8	*true*	0.01	9	12
ϕ_0	2	0.04	35	7	30	*true*	0.01	9	152
ϕ_1	2	0.08	35	31	309	*true*	0.15	12	804
ϕ_2	2	0.03	35	5	40	*false*	0.09	10	N/A
ϕ_3	2	0.03	35	5	40	*false*	0.28	10	N/A
ϕ_4	2	0.03	35	5	40	*true*	0.20	10	132
ϕ_0	3	0.04	35	17	112	*true*	0.52	15	1556
ϕ_1	3	0.72	67	174	3719	*true*	58.59	216	24211
ϕ_2	3	0.04	36	9	170	*false*	16.77	56	N/A
ϕ_3	3	0.04	36	9	170	*false*	44.69	77	N/A
ϕ_4	3	0.04	36	9	170	*true*	53.70	81	2738
ϕ_0	4	0.05	37	41	360	*true*	26.95	141	14479
ϕ_1	4	10.48	365	893	37256	??	TO		
ϕ_2	4	0.13	40	17	664	??	TO		
ϕ_3	4	0.08	40	17	664	??	TO		
ϕ_4	4	0.08	40	17	664	??	TO		
ϕ_0	5	0.17	40	97	1056	*true*	2740.56	3192	132371
ϕ_1	5	174.15	3175	4358	336505	??	TO		
ϕ_2	5	0.23	56	33	2462	??	TO		
ϕ_3	5	0.21	56	33	2462	??	TO		
ϕ_4	5	0.20	56	33	2462	??	TO		

Table D.3: Experimental results for the Train-Gate controller synthesis. Time is given in seconds and memory in megabytes.

Casaal				Uppaal Tiga			
	n	$\|L\|$	$\|E\|$	$\exists\sigma$	Time	Mem	#Zones
ψ_1	1	2	4	*true*	0.01	10	11
ψ_2	1	3	6	*true*	0.01	10	13
ψ_3	1	4	13	*true*	0.01	10	22
ψ_1	2	6	24	*true*	0.03	10	119
ψ_2	2	5	36	*true*	0.26	10	179
ψ_3	2	30	279	*true*	2.24	10	903
ψ_1	3	16	96	*true*	4.26	18	1498
ψ_2	3	9	162	*true*	504.31	116	2997
ψ_3	3	173	3546	??	TO		
ψ_1	4	40	320	*true*	790.78	333	26300
ψ_2	4	17	648	??	TO		
ψ_3	4	892	36364	??	TO		

Table D.4: Experimental results for the Chinese-Juggler controller synthesis. Time is given in seconds, memory in megabytes and n gives the number of plates being juggled.

ϕ_0	If a plate turns unstable, it eventually becomes stable $\bigwedge_{i=1}^{n}(\Box(unstable[i] \implies \Diamond stable[i]))$
ϕ_1	If a plate turns unstable, it becomes stable within 5 time-units $\bigwedge_{i=1}^{n}(\Box(unstable[i] \implies \Diamond_{\leq 5} stable[i]))$
ϕ_2	If a plate turns unstable, it becomes stable after 5 time-units $\bigwedge_{i=1}^{n}(\Box(unstable[i] \implies \Diamond_{\geq 5} stable[i]))$

Table D.5: Specifications for the Chinese-Juggler example.

instance.

5 Conclusions

In this paper we have significantly extended the practical scope of automatic controller synthesis for real-time systems. In particular, our method supports synthesis for all objectives expressed in $\mathrm{MTL}_{0,\infty}$, a sublogic of MTL containing LTL and rich enough to express a wide variety of safety, liveness and bounded liveness properties. In general the synthesis problem for $\mathrm{MTL}_{0,\infty}$ is undecidable. We overcome this obstacle by a new algorithm implemented Casaal converting $\mathrm{MTL}_{0,\infty}$ into under-approximating Timed Büchi Automata. Combined with Uppaal Tiga supporting synthesis for Timed Games with Büchi conditions we obtain a complete tool chain. In our experimental evaluation we demonstrated that for complex $\mathrm{MTL}_{0,\infty}$ we predominantly obtain “exact and tight” approximations, supporting our initial claim. Furthermore, we showed on a number of scalable examples that synthesis for MTL$_{0,\infty}$ objectives is feasible using our tool-chain.

Future work includes to refine our determinstic under- and overapproximation construction by using the breakpoint technique [24] for Büchi determinization.

Acknowledgements. This work is partially supported by the National Science Foundation of China Nos. NNFC 61472406 and NNFC 61532019, the BBMF through the ASSUME project BBMF 01IS15031J, the Sino-Danish Center IDEA4CPS, the ERC Advanced Grant LASSO, the Innovation Fund Center DiCyPS, the DFF Research Project ASAP and ANR contract Malthy.

References

[1] A. Church, “Logic, Arithmetic, Automata,” in *Proc. International Mathematical Congress*, 1962.

[2] J. R. Buchi and L. H. Landweber, “Solving Sequential Conditions by Finite-State Strategies,” *Transactions of the American Mathematical Society*, vol. 138, pp. 295–311, 1969. [Online]. Available: http://dx.doi.org/10.2307/1994916

[3] P. Bouyer, L. Bozzelli, and F. Chevalier, “Controller synthesis for MTL specifications,” in *In Proc. 17th International Conference on Concurrency Theory (CONCUR’06*, 2006.

[4] E. A. Emerson and E. M. Clarke, “Using branching time temporal logic to synthesize synchronization skeletons,” *Sci. Comput. Program.*, vol. 2, no. 3, pp. 241–266, 1982.

[5] O. Kupferman and M. Y. Vardi, “μ-calculus synthesis,” in *MFCS*, 2000, pp. 497–507.

[6] Z. Manna and P. Wolper, “Synthesis of communicating processes from temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 1, pp. 68–93, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/357233.357237

[7] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, NY, USA: ACM, 1989, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/75277.75293

ψ_1	Whenever the mouse is caught it, it should not caught again before after 10 time units and it should be caught infinitely often. $(\square(\mathit{Catch} \implies \bigcirc(\neg \mathit{Catch} \mathsf{U}_{\geq 10} \mathit{Catch}))) \wedge \square \diamond \mathit{Catch}$
ψ_2	Catch the mouse within 100 time units $\diamond_{\leq 100} \mathit{Catch}$
ψ_3	Wheneever the cat is at its initial position; then the mouse is caught within 100 time units $\square(\mathit{Initial} \implies \diamond_{\leq 100} \mathit{Catch})$
ψ_4	Cat should be at its initial place within 10 time units and whenever its at initial position, it should catch the mouse within 100 time units. $\diamond_{\leq 10} \wedge \psi_3$
ψ_5	Cat should be at its initial place after 10 time units and whenever its at initial position, it should catch the mouse within 100 time units. $\diamond_{\geq 10} \wedge \psi_3$
ψ_6	Cat should be at its initial place after 10 time units and whenever its at initial position, it should catch the mouse within 110 time units. $\diamond_{\geq 10} \wedge \square(\mathit{Initial} \implies \diamond_{\leq 110} \mathit{Catch})$
ψ_7	Cat should always return to its initial position before 200 time units; and always catch the mouse within 110 time units after visiting initial $\square \diamond_{\leq 200} \wedge \square(\mathit{Initial} \implies \diamond_{\leq 110} \mathit{Catch})$
ψ_8	Cat should always return to its initial position; and always catch the mouse within 110 time units after visiting initial $\square \diamond \wedge \square(\mathit{Initial} \implies \diamond_{\leq 110} \mathit{Catch})$
ψ_9	After catching the mouse, the cat should return to its initial state within 40 time units. $\square(\mathit{Catch} \implies \diamond_{\leq 40} \mathit{Initial})$
ψ_{10}	When the cat is at its initial position, it should catch the mouse within 100 units and after catching the mouse it should return to its initial position within 40 time units. $(\square \mathit{Initial} \implies \diamond_{\leq 100} \mathit{Catch}) \wedge (\square(\mathit{Catch}) \implies \diamond_{\leq 40} \mathit{Initial})$
ψ_{11}	When the cat is at its initial position, it should catch the mouse within 110 units and after catching the mouse it should return to its initial position within 40 time units. $(\square \mathit{Initial} \implies \diamond_{\leq 110} \mathit{Catch}) \wedge (\square(\mathit{Catch}) \implies \diamond_{\leq 40} \mathit{Initial})$
ψ_{12}	Eventually, within 200 time units, the lazy cat moves away from the initial position and plays with the mouse for 50 time units. Within 100 units of moving from the initial position, the cat moves back to the initial position. $\Diamond_{\geq 200}(\neg \mathit{Initial} \wedge (\square_{\leq 50} \neg \mathit{Catch} \implies \mathit{Catch}) \wedge (\Diamond_{\leq 100} \mathit{Initial}))$
ψ_{13}	It always holds that within 200 time units, the lazy cat moves away from the initial position and plays with the mouse for 50 time units. Within 100 units of moving from the initial position, the cat moves back to the initial position. $\square(\Diamond_{\geq 200}(\neg \mathit{Initial} \wedge (\square_{\leq 50} \neg \mathit{Catch} \implies \mathit{Catch})) \wedge (\Diamond_{\leq 100} \mathit{Initial}))$

Table D.6: Specifications for the *Cat* and *Mouse* example.

[8] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0325013

[9] O. Kupferman, N. Piterman, and M. Vardi, "Safraless compositional synthesis," in *18th Conference on Computer Aided Verification*, 2006, pp. 31–44.

[10] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *J. ACM*, vol. 43, pp. 116–146, January 1996.

[11] R. Alur, "Formal verification of hybrid systems," in *Proceedings of the ninth ACM international conference on Embedded software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 273–278. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038685

[12] L. Doyen, G. Geeraerts, J. Raskin, and J. Reicher, "Realizability of real-time logics," in *Proceedings of FORMATS 2009, 7th International Conference on Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, vol. 5813. Springer, 2009, pp. 133–148.

[13] O. Maler, D. Nickovic, and A. Pnueli, "On synthesizing controllers from bounded-response properties," in *CAV*, 2007, pp. 95–107.

[14] P. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, D. B. Poulsen, and A. Stainer, "Monitor-based statistical model checking for weighted metric temporal logic," in *LPAR*, 2012.

[15] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Uppaal-tiga: Time for playing games!" in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. LNCS, no. 4590. Springer, 2007, pp. 121–125.

[16] O. Maler, D. Nickovic, and A. Pnueli, "From mitl to timed automata," in *FORMATS'06*. Springer, 2006, pp. 274–289.

[17] M. Geilen, "An improved on-the-fly tableau construction for a real-time temporal logic," in *In International Conference on Computer Aided Verification*. Springer, 2003, pp. 276–290.

[18] P. Bulychev, A. David, K. G. Larsen, and G. Li, "Efficient controller synthesis for a fragment of $\mathrm{MTL}_{0,\infty}$," *Acta Informatica*, vol. 51, no. 3-4, pp. 165–192, 2014.

[19] T. Brihaye, M. Estiévenart, and G. Geeraerts, "On mitl and alternating timed automata over infinite words," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2014, pp. 69–84.

[20] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[21] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *STACS*, 1995, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/3-540-59042-0_76

[22] L. Doyen, G. Geeraerts, J.-F. Raskin, and J. Reichert, "Realizability of real-time logics," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2009, pp. 133–148.

[23] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of ltl formulae to büchi automata," in *Formal Techniques for Networked and Distributed Sytems—FORTE 2002*. Springer, 2002, pp. 308–326.

[24] A. Morgenstern, K. Schneider, and S. Lamberti, "Generating deterministic ω-automata for most ltl formulas by the breakpoint construction." in *MBMV*, 2008, pp. 119–128.

Paper E

Uppaal Stratego

Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen,
Marius Mikučionis and Jakob Haahr Taankvist

The paper has been published in the
Proceedings of Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, LNCS Vol. 9035, pp. 206–211, 2015.

Abstract

UPPAAL STRATEGO *is a novel tool which facilitates generation, optimization, comparison as well as consequence and performance exploration of strategies for stochastic priced timed games in a user-friendly manner. The tool allows for efficient and flexible "strategy-space" exploration before adaptation in a final implementation by maintaining strategies as first class objects in the model checking query language. The paper describes the strategies and their properties, construction and transformation algorithms and a typical tool usage scenario.*

1 Introduction

Model checking may be used to verify that a proposed controller prevents an environment from causing dangerous situations while, at the same time, operating in a desirable manner. This approach has been successfully pursued in the setting of systems modeled as finite-state automata, timed automata, and probabilistic automata of various types with nuSMV [1], FDR [2], UPPAAL [3] and PRISM [4] as prime examples of model checking tools supporting the above mentioned formalisms. Most recently the simulation-based method of *statistical* model checking has been introduced in UPPAAL SMC [5], allowing for highly scaleable analysis of *fully* stochastic Sriced Timed Automata with respect to a wide range of performance properties. For instance, expected waiting-time and cost, and time-bounded and cost reachability probabilities, may be estimated (and tested) with an arbitrary precision and high degree of confidence. Combined with the symbolic model checking of UPPAAL this enables an adequate analysis of mixed critical systems, where certain (safety) properties must hold with absolute certainty, whereas for other quantitative (performance) properties a reasonably good estimation may suffice, see e.g. [6].

Rather than verifying a *proposed* controller, synthesis – when possible – allows an algorithmic construction of a controller which is guaranteed to ensure that the resulting systems will satisfy the desired correctness properties. The extension of controller synthesis to timed and hybrid games started in the 90s with the seminal work of Pnueli et al. [7, 8] on controller synthesis for timed games where the synthesis problem was proven decidable by a symbolic dynamic programming technique. In UPPAAL TIGA [9, 10] an efficient on-the-fly algorithm for synthesis of reachability and safety objectives for timed games has been implemented, with a number of successful industrial applications having been made including zone-based climate control for pig-stables [11] and controllers for hydraulic pumps with 60% improvement in energy-consumption compared with industrial practice at the time [12, 13].

However, once a strategy has been synthesized for a given objective no

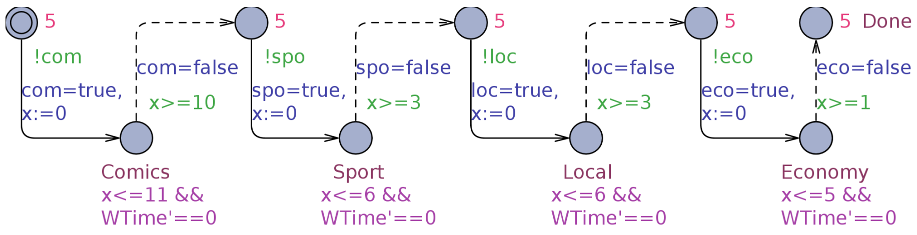


Fig. E.1: Uppaal Stratego template of a single person reading a newspaper.

further analysis has been supported so far. In particular it has not been possible to make a deeper examination of a synthesized strategy in terms of other additional properties that may or may not hold under the strategy. Neither has it been possible to optimize a synthesized non-deterministic safety strategy with respect to desired performance measures. Both of these issues have been addressed by the authors in recent work [14, 15], and in this paper we present the tool Uppaal Stratego which combines these techniques to generate, optimize, compare and explore consequences and performance of strategies synthesized for stochastic priced timed games in a user-friendly manner. In particular, the tool allows for efficient and flexible "strategy-space" exploration before adaptation in a final implementation.

Uppaal Stratego[1] integrates Uppaal and the two branches Uppaal SMC [5] (statistical model checking), Uppaal Tiga [10] (synthesis for timed games) and the method proposed in [15] (synthesis of near-optimal schedulers) into one tool suite. Uppaal Stratego comes with an extended query language where strategies are first class objects that may be constructed, compared, optimized and used when performing (statistical) model checking of a game under the constraints of a given synthesized strategy.

Consider the jobshop scheduling problem shown in Fig. E.1 which models a number of persons sharing a newspaper. Each task process reads a section of the paper, whereas only one person can read a particular section at a time. Each reader wants to read the newspaper in different orders, and the stochastic environment chooses how long it takes to read each section. This makes the problem a problem of finding a strategy, rather than finding a static scheduler as in the classical jobshop scheduling problem.

Figure E.1 shows a stochastic priced timed game (SPTG) which models one person reading the newspaper. The circles are locations and the arrows are transitions. The solid arrows are transitions controlled by the controller and the dashed are transitions controlled by the stochastic environment. The model reflects the reading of the four sections in the preferred order (here comics, sport, local and economy) for the preferred amount of time. In the top locations the person is waiting for the next section to become available; here four Boolean variables are used to ensure mutex on the reading of a sec-

[1]Uppaal Stratego is available at `http://people.cs.aau.dk/~marius/stratego/`.

tion. In the bottom locations, the person is reading the particular section for a duration given by a uniform distribution on the given interval, e.g. [10,11] for our person's reading of sport. The stopwatch `WTime` is only running in the waiting locations thus effectively measuring the accumulated time when the person is waiting to read. Given a complete model with several persons constantly competing for the sections, we are interested in synthesizing strategies for several multi-objectives, e.g. synthesize a strategy ensuring that all persons have completed reading within 100 minutes, and then minimize the expected waiting time for our preferred person.

2 Games, Automata and Properties

Using the features of Uppaal Stratego we can analyze the SPTG in Fig. E.1. Internally, Uppaal Stratego has different models and representations of strategies, an overview of these and their relations are given in Fig. E.2. The model seen in Fig. E.1 is a SPTG, as `WTime` is a cost function or price with location dependent rate (here 0 or 1), and we assume that environment takes transitions according to a uniform distribution over time.

As shown in Fig. E.2 we can abstract a SPTG into a timed game (TGA). This abstraction is obtained simply by ignoring the prices and stochasticity in the model. Note that since prices are observers, this abstraction does not affect the possible behavior of the model, but merely forgets the likelihood and cost of various behaviors. The abstraction maps a $1\frac{1}{2}$-player game, where the opponent is stochastic into a 2-player game with an antagonistic opponent.

Given a TGA ($\mathcal{G}$) we can use Uppaal Tiga to synthesize a strategy σ (either deterministic or non-deterministic). This strategy can, when put in parallel with the TGA, $\mathcal{G}|\sigma$, be model checked in the same way as usual in Uppaal. We can also use the strategy in a SPTG $\mathcal{P}$, and obtain $\mathcal{P}|\sigma$. Under a strategy it is possible to do statistical model checking (estimation of probability and cost, and comparison), which enables us to observe the behavior and performance of the strategy when we assume that the environment is purely stochastic. This also allows us to use to use prices under σ, even though they were not

$$\begin{tikzcd}
\text{Timed Game} & & \text{Strategy} & \text{(Statistical) Model Checking Timed Automata} \\
\mathcal{G} \arrow[rr, "\text{Synthesis Uppaal Tiga}", "\phi"'] & & \sigma \arrow[r, dashed] \arrow[dl, dashed] & \mathcal{G}|\sigma \\
\mathcal{P} \arrow[u, "\text{Abstraction}"] \arrow[r, dashed] & \mathcal{P}|\sigma \arrow[r, "\phi", "\text{Synthesis}"'] & \sigma^{\circ} \arrow[r, dashed] & \mathcal{P}|\sigma^{\circ} \\
\text{Stochastic Priced Timed Game} & & \text{Strategy} & \text{Statistical Model Checking Stochastic Priced Timed Automata}
\end{tikzcd}$$

Fig. E.2: Overview of models and their relations. The lines show different actions. The dashed lines show that we use the object.

considered in the synthesis of σ. From both $\mathcal{P}$ and $\mathcal{P}|\sigma$ learning is possible using the method proposed in [15]. The learning algorithm uses a simulation based method for learning near-optimal strategies for a given price metric. If σ is the most permissive strategy guaranteeing some goal, then the learning algorithm can optimize under this strategy, and we will get a strategy σ° which is near-optimal but still has the guarantees of σ. As the last step we can construct $\mathcal{P}|\sigma^\circ$, which we can then do statistical model checking on.

3 Strategies

In Uppaal Stratego we operate three different kinds of strategies, all memoryless. *Non-deterministic strategies* are strategies which give a *set* of actions in each state, with the most permissive strategy – when it exists – offering the largest set of choices. In the case of timed games, most permissive strategies exist for safety and time-bounded reachability objectives. *Deterministic strategies* give *one* action in each state. *Stochastic strategies* give a *distribution* over the set of actions in each state. Fig. E.3 shows how strategies are generated and used. For generating strategies, we can use Uppaal Tiga or the method proposed in [15] on SPTGs. Uppaal Tiga generates (most permissive) *non-deterministic* or *deterministic* strategies. The method proposed in [15] generates strategies which are deterministic. A strategy generated with Uppaal Stratego can undergo different investigations: model checking, statistical model checking and learning. Learning consume non-deterministic strategies (potentially multiple actions per state) and may produce a deterministic one by selecting a single action for each state, such that the final deterministic strategy is optimized towards some goal. Figure E.3 shows that currently it is possible to model check only under symbolically synthesized strategies (as opposed to optimized ones) as symbolic model checking requires the strategy to be represented entirely in terms of zones (constraint systems over clock values and their differences). Statistical model checking can only be done under stochastic strategies. All deterministic strategies can be thought of as stochastic by assigning a probability of 1 to the *one* choice. To evaluate non-deterministic strategies statistically we applying a stochastic uniform

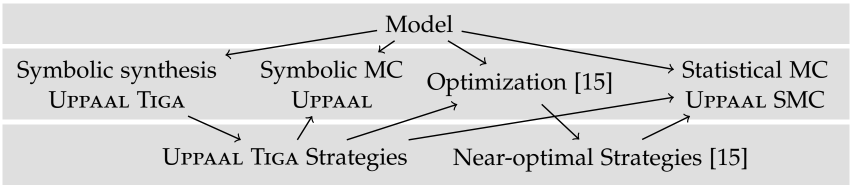
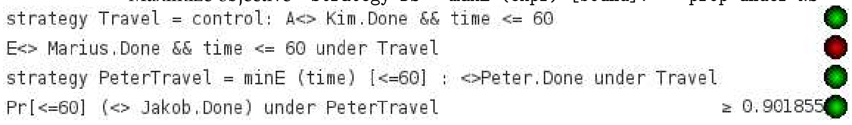


Fig. E.3: Overview of algorithms and data structures in Uppaal Stratego.

Table E.1: Types of queries.

UPPAAL	Safety	`A[] prop under NS`
	Liveness	`A<> prop under NS`
TIGA	Guarantee objective	`strategy NS = control: A<> prop`
	Guarantee objective	`strategy NS = control: A[] prop`
SMC	Evaluation	`Pr[bound](<> prop) under SS`
	Expected	`value E[bound;int](min: prop) under SS`
	Simulations	`simulate int [bound]{expr1,expr2} under SS`
[15]	Minimize objective	`strategy DS = minE (expr) [bound]: <> prop under NS`
	Maximize objective	`strategy DS = maxE (expr) [bound]: <> prop under NS`



```
strategy Travel = control: A<> Kim.Done && time <= 60
E<> Marius.Done && time <= 60 under Travel
strategy PeterTravel = minE (time) [<=60] : <>Peter.Done under Travel
Pr[<=60] (<> Jakob.Done) under PeterTravel                   ≥ 0.901855
```

Fig. E.4: UPPAAL STRATEGO queries and results for the model in Fig. E.1.

distribution over the non-deterministic choices.

4 Query Language

We let strategies become first class citizens by introducing strategy assignment `strategy S =` and strategy usage `under S` where `S` is an identifier. These are applied to the queries already used in UPPAAL, UPPAAL TIGA and UPPAAL SMC as well as those proposed in [15]. An overview of these queries is given in Table E.1. Notice that we changed the syntax of the queries presented in [15]. Recall the example with the four authors sharing a newspaper as presented in Fig. E.1. We compute a strategy for Kim to reach his plane within one hour on line 1 in Fig. E.4. Respecting this, we find that Marius cannot join, as the query on line 2 is not satisfied. Instead, we optimize that Peter joins in on line 3 (`[<=60]` is a bound on how long the simulations we learn from used can be). Finally, line 4 estimates that Jakob is done with probability $\geq$0.9 under Peter's optimizations.

References

[1] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, ser. CAV'02. London, UK: Springer-Verlag, 2002, pp. 359–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=647771.734431

[2] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3 — a modern model checker for CSP," in *Tools and Algorithms for the Con-*

struction and Analysis of Systems, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413, 2014, pp. 187–201.

[3] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST'06*, 2006, pp. 125–126.

[4] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[5] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang, "UPPAAL-SMC: statistical model checking for priced timed automata," in *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012.*, 2012, pp. 1–16. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.85.1

[6] A. David, K. G. Larsen, A. Legay, and M. Mikučionis, "Schedulability of Herschel-Planck revisited using statistical model checking," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, 2012, pp. 293–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34032-1_28

[7] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," in *STACS'95*, ser. Lecture Notes in Computer Science, E. Mayr and C. Puech, Eds. Springer Berlin Heidelberg, 1995, vol. 900, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/3-540-59042-0_76

[8] E. Asarin, O. Maler, and A. Pnueli, "Symbolic controller synthesis for discrete and timed systems," in *Hybrid Systems II*, ser. Lecture Notes in Computer Science, P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, Eds. Springer Berlin Heidelberg, 1995, vol. 999, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/3-540-60472-3_1

[9] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR 2005 - Concurrency Theory, 16th International Conference*, ser. Lecture Notes in Computer Science, M. Abadi and L. de Alfaro, Eds., vol. 3653. San Francisco, CA, USA: Springer, August 2005, pp. 66–80. [Online]. Available: http://dx.doi.org/10.1007/11539452_9

[10] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, “Uppaal-Tiga: Time for playing games!” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 121–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_14

[11] J. J. Jessen, J. I. Rasmussen, K. G. Larsen, and A. David, “Guided controller synthesis for climate controller using uppaal tiga,” in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, J.-F. Raskin and P. Thiagarajan, Eds. Springer Berlin Heidelberg, 2007, vol. 4763, pp. 227–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75454-1_17

[12] H. Zhao, N. Zhan, D. Kapur, and K. G. Larsen, “A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example,” 2012.

[13] F. Cassez, J. J. Jessen, K. G. Larsen, J. Raskin, and P. Reynier, “Automatic synthesis of robust and optimal controllers - an industrial case study,” in *Hybrid Systems: Computation and Control, 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings*, ser. Lecture Notes in Computer Science, R. Majumdar and P. Tabuada, Eds., vol. 5469. Springer, 2009, pp. 90–104. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00602-9_7

[14] A. David, H. Fang, K. G. Larsen, and Z. Zhang, “Verification and performance evaluation of timed game strategies,” in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, A. Legay and M. Bozga, Eds. Springer International Publishing, 2014, vol. 8711, pp. 100–114. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10512-3_8

[15] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist, “On time with minimal expected cost!” in *ATVA*, 2014, pp. 129–145.

Paper F

Co-Simulation of Hybrid Systems with SpaceEx and Uppaal

Sergiy Bogomolov, Marius Greitschus, Peter G. Jensen, Kim G. Larsen, Marius Mikucionis, Thomas Strump, Stavros Tripakis

The paper has been published in the
Proceedings of the 11th International Modelica Conference, pp. 159–169, 2015.

Abstract

The Functional Mock-up Interface (FMI) is an industry standard which enables co-simulation of complex heterogeneous systems using multiple simulation engines. In this paper, we show how to use FMI in order to co-simulate hybrid systems modeled in the model checkers SpaceEx *and* Uppaal. *We show how FMI components can be automatically generated from* SpaceEx *and* Uppaal *models. We also validate the co-simulation approach by comparing the simulations of a room heating benchmark in two cases: first, when a single model is simulated in* SpaceEx*; and second, when the model is split in two submodels, and co-simulated using* SpaceEx *and* Uppaal. *Finally, we perform a measurement experiment on a composite model to show a potential for statistical model checking using stochastic co-simulations.*

1 Introduction

Despite advances in model checking and other formal verification techniques, simulation remains the workhorse for system analysis. A plethora of simulation tools are available today, from academia as well as from industry. These tools support a large variety of modeling languages, targeted at different types of systems from various disciplines (e.g., mechanical, electrical, digital, continuous or discrete, or mixes thereof). Unfortunately, these tools can rarely interoperate. This is a problem because modern cyber-physical systems are highly complex and multidisciplinary, requiring specialized modeling languages and tools from several domains.

The *Functional Mock-up Interface*[1] (FMI) is a standard developed to address this problem. FMI defines an XML schema for describing simulation components and a C API that these components must implement. The components are called *functional mock-up units*, or FMUs. An FMU is typically generated automatically *(exported)* from some simulation tool, and corresponds to a (sub-)model designed in that tool. The sub-models/FMUs are then *imported* into a *host* simulator. The host commands the simulation by calling the API methods of the FMUs, thus effectively achieving integration of the original simulation environments. FMI supports two integration modes: (a) *model exchange*, where the host simulator is handles the numerical integration; and (b) *co-simulation*, where each FMU implements its own numerical integration mechanism (or any other internal mechanism to advance its state in time). Because each mode imposes its own requirements on FMUs (for instance, in model exchange, the FMUs must provide the host with information such as state derivatives, which are not necessary for co-simulation) the FMI APIs for the two modes are different.

[1]See `https://www.fmi-standard.org/` for more details.

In this paper, we use FMI in order to connect two state-of-the-art modeling and verification tools for cyber-physical systems: SpaceEx [1] and Uppaal [2]. SpaceEx is a tool for modeling and verifying *hybrid systems* [3]. Uppaal is primarily a model-checker for *timed automata* [4], however, it also supports statistical model checking of hybrid systems [5].

Our goal is to integrate these two tools for co-simulation. That is, we want to be able to: (a) build a sub-model of the system (e.g., the model of the *plant* under control) in SpaceEx; (b) build another sub-model (e.g., the *controller*) in Uppaal; (c) automatically generate an FMU for each sub-model; (d) import the FMUs, connect and co-simulate them in a host environment.

The motivations for connecting SpaceEx and Uppaal in this manner are numerous. First, although both SpaceEx and Uppaal support simulation of hybrid systems, each tool offers its own modeling language, which is not compatible with that of the other tool. Translating from one language to the other is limited to common features supported by the tools. For example, even though the frameworks CIF [6, 7] and HSIF [8] solve the complexity problem of one format translation to another by performing at most two translations, the approach still suffers from the fact that Uppaal features like committed locations and C-like function code are not supported in SpaceEx and Uppaal has limited support for ODEs. Moreover, by using co-simulation, we are able to take advantage not just of the specific strengths of the language of each tool, but also of their native simulation engines, since each FMU is internally running essentially a "copy" of the simulation algorithm of the original tool.

As host environment we use the tool ptolemy[2]. ptolemy is a modeling and simulation environment for heterogenous systems [9]. Recently, support has been implemented in ptolemy for using it as a host environment for co-simulation based on FMI. FMUs (developed by other tools) can be imported into ptolemy, connected using ptolemy's graphical user interface, and co-simulated using an implementation of the co-simulation algorithm described by [10]. This algorithm has desirable properties, such as *determinacy*, namely, the fact that the results of the simulation are independent of arbitrary factors such as names of the FMUs, order of creation, or order of evaluation in the diagram.

The contributions of this paper are the following:

1. We show how FMUs can be generated automatically from models of hybrid and timed automata built in SpaceEx and Uppaal. There are several subtleties involved in this, as hybrid and timed automata are models designed primarily with verification in mind, whereas FMI is designed for simulation and therefore imposes certain properties on FMUs, such as determinism.

[2]See `http://ptolemy.eecs.berkeley.edu/`.

2. We report on the implementation and case studies. In particular, we apply our co-simulation framework to a room heating benchmark [11].

3. We validate the co-simulation algorithm proposed by [10] by comparing the results of the case study in two settings: (a) when the case study is modeled and simulated in a single tool, and (b) when the various components of the case study are modeled in two tools and co-simulated using our framework. We show that our co-simulation framework computes the same simulation trajectories as the setting (b) provided that the maximum simulation step size of co-simulation is sufficiently small.

4. We demonstrate how stochastic simulations can be included into the composite model with hybrid systems and applied a simple statistical measurement to show the potential for statistical model checking using FMI co-simulations.

The rest of the paper is organized as follows. In Sec. 2, we introduce the necessary background on FMI for this work. Afterwards, we present our translation of SpaceEx and Uppaal models into FMUs in Sec. 3. This is followed by the case study in Sec. 4. We discuss related work in Sec. 5. Finally, we conclude the paper in Sec. 6.

2 Background on FMI

Conceptually an FMU can be seen as a (timed) state machine. This machine has a set of input variables (or *ports*), a set of output variables, and a set of internal states. The machine interacts with its environment only by means of a clearly defined set of *interface methods*. These methods are specified in the FMI standard. For the purposes of this paper, and following the formalization presented by [10], the key interface methods of FMI (for co-simulation) are:

- A method to initialize the state of the FMU. If S is the set of states of the FMU, then $\texttt{init} \in S$.

- A method `set` to set a given input variable to a certain value. The signature of `set` is $\texttt{set} : S \times U \times \mathbb{V} \to S$, where U is the set of input variables of the FMU, and $\mathbb{V}$ is the set of all possible values (for simplicity we ignore typing and use a single universe $\mathbb{V}$ of values for all variables). Given state s, input variable $u \in U$, and value $v \in \mathbb{V}$, $\texttt{set}(s, u, v)$ returns the new state obtained after setting u to v.

- A method `get` which returns the value of a given output variable. Its signature is $\texttt{get} : S \times Y \to \mathbb{V}$, where Y is the set of output variables of the FMU. Given state s and output variable $y \in Y$, $\texttt{get}(s, y)$ returns the value of y in s.

- A method doStep which advances the state of the machine in time. Its signature is doStep : $S \times \mathbb{R}_{\geq 0} \rightarrow S \times \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers. The behavior of doStep is explained below.

As said above, an FMU is essentially a state machine: the get method corresponds to the output function of the machine, while the doStep method corresponds to the transition function. The difference is that doStep takes as input a *time step* $h \in \mathbb{R}_{\geq 0}$: in that sense, an FMU is a timed state machine.

The behavior of doStep is as follows. Given state $s \in S$, and time step $h \in \mathbb{R}_{\geq 0}$, a call to doStep$(s, h)$ is interpreted as the co-simulation algorithm "asking" the FMU to perform a simulation step of length h. For a number of reasons, including numerical integration issues, the FMU may "accept" or "reject" this request. If it rejects, it means that it was not able to advance time by h (but may have been able to advance time by a smaller delay $h' < h$). Formally, doStep(s, h) returns a pair (s', h') where $s' \in S$ is a state and $h' \in \mathbb{R}_{\geq 0}$ is a time step, such that:

- either $h' = h$, which is interpreted as F having *accepted* h, and having moved to a new state s';

- or $0 \leq h' < h$, which is interpreted as F having *rejected* h, but having made partial progress up to h', and having reached a new state s'.

It is worth noting that FMUs are *deterministic* machines, in the sense that for a given sequence of inputs (i.e., a sequence of input values and time steps), the sequence of states and outputs that the machine produces is unique. This is because there is a unique initial state init $\in S$, and set, get, doStep are all *total functions*. Moreover, the fact that these functions are total implies that the machine is able to accept any input at any time, therefore, it is implicitly *input-enabled*.

We also rely on zero-time steps in a sense of allowing doStep(s, h) calls with $h = 0$ (despite that version 2.0 of the FMI standard forbids this), because they are essential for modeling discrete transitions like instantaneous mode switches in hybrid automata models.

In addition to the above, each FMU comes with a set of *input-output dependencies*, $D \subseteq U \times Y$. D specifies for each output variable which input variables it depends upon (if any): $(u, y) \in D$ means that output variable y depends on input variable u. This information is used to ensure that a network of FMUs has no cyclic dependencies, and also to determine the order in which all network values are computed during a simulation step [10].

FMI specifies the methods that every FMU must implement, but it does *not* specify the co-simulation algorithm (also called a *master algorithm*). In fact, devising such an algorithm with good properties is not a trivial problem, and has been the topic of previous work [10]. In that work, two co-simulation algorithms were proposed and proved to have desirable properties, such as termination of a simulation step, and *determinacy*. The determinacy property says that the results of a simulation do not depend on the order in which the algorithm chooses to call `doStep` over a set of FMUs. This ensures that the simulation results are well-defined and are not influenced by arbitrary factors such as FMU names, order of creation, geometrical position in the diagram of a graphical model, etc., as is often the case with simulation tools.

In a nutshell, the co-simulation method proposed by [10] relies on the following principle. First, the co-simulation algorithm chooses a default time step, $h_{\max}$, called the *maximum step size*. Second, the algorithm saves the state of each FMU in the model (FMI specifies methods for an FMU to export and import its state, although these are optional). Assuming there are n FMUs, $F_1, ..., F_n$, the algorithm maintains n states, $s_1, ..., s_n$. Third, the algorithm calls F_i.$\texttt{doStep}(s_i, h_{\max})$ on each FMU F_i, and collects the returned time steps $h'_1, ..., h'_n$. There are two cases: either all FMUs accepted the proposed time step, i.e., $h'_1 = h'_2 = \cdots = h'_n = h_{\max}$, in which case this simulation step is over, and the algorithm proceeds to the next one; or at least one FMU F_i rejected $h_{\max}$, i.e., $h'_i < h_{\max}$ for some i. In the latter case, the algorithm computes the minimum of $h'_1, ..., h'_n$, $h_{\min} = \min\{h'_1, ..., h'_n\}$, restores the saved state of each FMU, and tries again with new step size $h_{\min}$.

Assuming that the FMUs satisfy the reasonable "monotonicity" property that if they were able to advance time by h'_i then they are also able to advance time by any smaller step, and by the fact that $h_{\min}$ is smaller than all h'_i, the second attempt is guaranteed to succeed. That is, $h_{\min}$ will be accepted by all FMUs. As a result, at most after two attempts, a co-simulation step is successful, and the algorithm proceeds with the next step, repeating the same procedure as above.

The FMI standard sets out a framework where FMUs share the notion of time and exchange variable values via input-output ports: outputs from one FMU are mapped as inputs to other FMU(s) and so on. The output port values are said to be owned and controlled by the emitting FMU, whereas the inputs are computed and provided by another (outputting) FMU. The framework foresees that before producing an output an FMU may first need some input values and thus input-output dependency information is introduced. Overall the I/O port connectivity graph derived from the model of interconnected FMUs, together with the local I/O dependencies of each individual FMU, result in a global I/O dependency graph for the entire model [10].

Time and I/O values are synchronized by the co-simulation algorithm: the time is agreed by repeatedly consulting each FMU and the I/O values are propagated according to dependencies. The co-simulation algorithm assumes that each FMU provides a static dependency list of its ports before simulation starts, and that the resulting global I/O dependency graph is acyclic, and therefore there exists a schedule for computing the value of every input port before the value of a dependent output port is requested [10].

3 Translating Models into FMUs

The behavior of individual FMUs is provided by the model-checker's simulation engines based on the guidelines described by [12]. In particular, the report distinguishes continuous and discrete dynamics. The continuous behavior is modeled by differential equations over continuous variables whose values can be shared among FMUs by the means of port connections. The output ports of an FMU are mapped to the owned/controlled variables which are read and written to, whereas input ports map to read-only variables within the FMU.

The discrete behavior is modeled by discrete transitions in the timed/hybrid automata control flow structure. The discrete transitions are designed to be executed with micro-steps of zero delay. Transitions can also be decorated with event labels and each tool supports its own kind(s) of synchronizing compositions internally and therefore the discrete transition synchronization is also handled individually within the tools. [12] provides the means of discrete transition synchronization by allocating two special port variables: one for incoming (input) synchronization and one for outgoing (output) synchronization. The domain of discrete input (output) ports coincides with the set of input (output) labels plus a special value *absent* which denotes no synchronization or an internal discrete transition.

3.1 Uppaal

Uppaal uses timed automata models [4], extended with discrete variables over structured types to describe behaviors of a timed system. In timed automata, the continuous dynamics is controlled by real-valued clock variables (with derivatives set to one) and discrete states complemented with integer variables – both of which are candidates for exchange via FMU input-output ports. Statistical model checking (SMC) extensions [5, 13] allow a finer control of the clock derivatives by means of ordinary differential equations, moreover the discrete transitions are stochastic where the execution is determinized by probability distributions over time and over branching edges. The stochastic semantics of a parallel composition is similar to the FMI co-simulation algorithm [10]: the way the minimum delay is negotiated and thus the timed composition within the FMI framework is straightforward, and task is to find a systematic way of handling discrete synchronizations. Uppaal also supports the maximal progress or ASAP semantics on edges labeled with urgent channels.

Uppaal supports the notion of discrete I/O synchronization natively by means of input and output channel labels. Thus, its discrete input and output transitions can be mapped directly to the input/output port variables of an FMU that is dedicated to transfer the synchronization label name. Nonetheless, we distinguish the following kinds of transitions: internal (transitions without I/O channel synchronization or internally synchronized transitions for which channels are not marked as input or output), input transitions (labeled by an input synchronization where the channel name is marked as an FMU input), and output transitions (labeled by an output synchronization where channel is marked as an FMU output). The marked outputs are controlled by the UPPAAL simulation and are executed asynchronously irrespective of whether the receiving FMU is ready to synchronize. Meanwhile, the input transitions are executed only when there is a corresponding input label set on a discrete input port. At most one (internal, input or output) transition is allowed at a time, hence fine-grained simulation control can be achieved by the co-simulation algorithm.

UPPAAL FMUs do not introduce I/O dependencies between continuous variables because the models do not use algebraic expressions to compute variable values. Instead of algebraic expressions the automata use discrete transitions to update the variable values. However, only one discrete transition is allowed at a time, therefore all discrete outputs have dependencies on the inputs dedicated to synchronization labels which restrict the selection of a particular discrete transition and hence specific variable update.

3.2 SpaceEx

SPACEEX [1] uses hybrid automata to describe system behavior where the continuous variable derivatives are constrained by differential equations. The continuous variables are candidates for input and output exchange via FMU ports. The discrete transitions of hybrid automata can be decorated with labels. Synchronization may involve multiple participating processes, but there is no notion of input and output – all processes are equal contributors, therefore the simulator needs to implement the input/output semantics required by FMI. We use a special label naming notation to mark input and output labels (see Fig. F.5). The transitions with input labels are only executed when the discrete input variable of FMU is set to the corresponding label name. Meanwhile, the transitions with an output label are controlled by SPACEEX' simulation, and are executed asynchronously by setting the discrete output variable with the label name irrespectively of whether the receiving FMU can synchronize with it. We ensure the SPACEEX FMU determinism by enforcing the must-semantics of discrete transitions in a hybrid automaton. In other words, a discrete transition is taken as soon as its guard is enabled. Finally, we resolve the non-determinism between input, output, and internal transitions in the following way: input transitions have priority over output transitions and output transitions are preferred over the internal ones.

Both UPPAAL and SPACEEX translations simulate the source models as they are without intermediate transformations, except of the following additions:

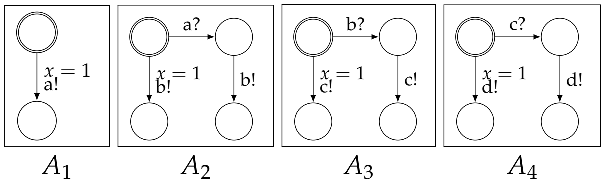


Fig. F.1: An example of four timed automata chain.

1) input enabledness is ensured by broadcast channels in Uppaal modeling and asynchronous I/O is implemented for SpaceEx synchronization labels, 2) for determinization SpaceEx uses maximal progress whereas Uppaal uses stochastic semantics with a possibility of urgent channels for maximal progress.

3.3 Discussion on Co-Simulation Semantics

In this section, we discuss the co-simulation semantics and contrast it to those typically used by a model checking tool. In particular, we demonstrate by example how the FMI co-simulation algorithm resolves input/output dependencies and contrast it with execution analysed in a model checker. Our goal is to offer insights in the differences of the two semantics.

Consider a system model shown in Fig. F.1 which consists of four timed automata composed in parallel. Labels of the form $a!$ denote sending output a, whereas $a?$ denotes receiving an input a. The variable x is a *clock* measuring time starting from zero. The constraint $x = 1$ is a *guard* which allows the corresponding transition of the automaton to occur only if the guard is satisfied, i.e., in this case only when x equals 1. The automata synchronize in a chain: the first can output a to the second one, the second one can output b to the third one and so on.

In principle, the system can be loaded into an FMI model in any combination: individually (one automaton per FMU) or collectively (multiple automata per FMU), but before an FMU can be loaded into an FMI model, it must declare its input/output dependencies. According to [10] each automaton should expose an input/output variable which will contain the synchronization label value. Automaton A_1 in the example above will have only an output variable, which may have values {*a, absent*}. Automaton A_2 will have an input variable ranging over {*a, absent*} and an output variable ranging over {*b, absent*}, and so on. The special value *absent* denotes that currently there is no synchronization. Timed automata must declare a dependency between its input and output label variable in order to avoid simultaneous input and output synchronizations.

In addition, it is assumed that each FMU is *input-enabled*, meaning that it can handle (i.e., it is able to receive) any declared input at any time. If a component is not input-enabled and an input synchronization is triggered then simulation is aborted, to avoid such situation we allow only broadcast channels, which do not block the sender process and receiver may simply ignore the synchronization if has no receiving edge.

Suppose the automata from Fig. F.1 are loaded within separate FMUs and connected according to synchronization labels. That is, the output of $FMU(A_1)$ is connected to the input of $FMU(A_2)$, the output of $FMU(A_2)$ is connected to the input of $FMU(A_3)$, and so on. The co-simulation algorithm would detect that it has to fulfill inputs values for the $FMU(A_4)$, $FMU(A_3)$, and $FMU(A_2)$ in order to proceed, therefore the input/output value propagation will have to start with $FMU(A_1)$ and then proceed to the $FMU(A_2)$ etc.. Once the values of all input and output variables are propagated, the algorithm proceeds with advancing each FMU in time by calling $doStep()$. It is this dynamic behavior in time which interests us in this example.

In particular, observe that $A_{2,3,4}$ automata are non-deterministic in the sense that, according to Uppaal semantics, at time $x = 1$ an automaton can either delay, or take an outputting transition, or synchronize on inputs. For instance, at time $x = 1$, A_2 can either emit b, or receive a (which will be available in this case, because it is sent by A_1 at exactly that time), or let the time pass. In timed automata semantics, all these options are possible at the individual component level. Moreover, not only individual components can be non-deterministic, but their composition is non-deterministic as well, based on so-called *interleaving semantics*. This means that when multiple automata are enabled at a given time, the choice of which one to execute is arbitrary. Non-determinism is a useful abstraction and thus model reduction technique in verification and model checking. The same is true when these tools are used for simulation, i.e. different simulations in Uppaal may yield different results.

In FMI, the situation is very different, as all FMUs are treated as deterministic components, and their composition, ensured by the co-simulation algorithm, is guaranteed to yield deterministic results as well. Interestingly, in this example, if all automata decide to output at time $x = 1$, some of them will succeed outputting in parallel, while others will be preempted by incoming inputs. In particular, the master algorithm will request $FMU(A_1)$ to produce its output, and thus $FMU(A_2)$ will be busy handling an input and will not be producing output at that time. Since $FMU(A_2)$ is not sending anything, then $FMU(A_3)$ will be free to produce an output and hence preempt $FMU(A_4)$.

As witnessed from above, such FMI system selects a particular sequence of steps (which is expected) but is not able to simulate all possible execution orders as in original semantics even if we allow FMUs to determinize their actions by themselves, which means that FMI simulations are selecting a particular subset of all possible behaviors and some behaviors may not be reproducible in FMI. Also FMI simulations may contain parallel synchronizations (e.g. actions $A_1 \overset{a}{\rightarrowtail} A_2$ and $A_3 \overset{c}{\rightarrowtail} A_4$ at the same computation step) which are possible only in several steps in timed automata semantics (action a and only then action c within zero-time), hence the intermediate state between a and c actions might not be accessible in FMI without very fine grained control over individual $doStep()$ calls in one zero-time computation step. However, the successor state of such parallel executions can be matched with a state after multiple transitions in the given automata semantics, hence the FMI simulation states in between system computation steps are included in the original semantics, albeit definite proof requires more formal insight to examine all scenarios.

4 Case Study

We have implemented the FMI standard in the Uppaal [2] and SpaceEx [1] model checkers by providing model export to FMU[3]. In this section, we present and evaluate the performance of the resulting FMI framework on a case study inspired by the well-known room heating benchmark originally proposed by [11]. Our model consists of a room with a heater (Fig. F.2a) and a controller (Fig. F.2b) which regulates the heater behavior. We model the room and the controller as a SpaceEx and Uppaal FMU, respectively (see Fig. F.3). Our bang-bang controller turns the heater on and off as soon as some temperature thresholds T_{low} and T_{high} have been reached. The as-soon-as-possible behavior is enforced by using urgent channels which effectively make the controller deterministic. The room temperature T evolves according to the following differential equation:

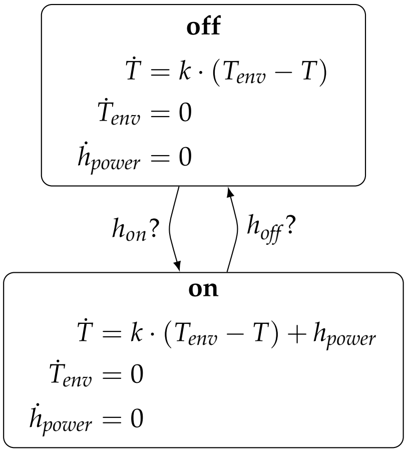
$$\begin{aligned} \dot{T} &= k \cdot (T_{env} - t) + h_{power} \\ \dot{T}_{env} &= 0 \\ \dot{h}_{power} &= 0 \end{aligned}$$

In other words, the room temperature depends linearly on the difference between the current room temperature T and outside temperature T_{env}. We assume the outside temperature T_{env} and heater power h_{power} to be constant. The constant k defines the heat exchange rate between the room and outside environment. If the heater is off, the heater power is set to zero.

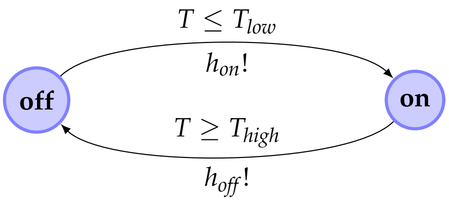
4.1 Evaluation

We evaluate our FMU framework by comparing simulation trajectories of the FMUs with the ones produced by a SpaceEx model consisting of both the

[3]A package containing the benchmarks is available for download at `http://swt.informatik.uni-freiburg.de/tool/spaceex/co-simulation`.



(a) Room component modelled in SPACEEX. The component switches between "on" and "off" modes. The temperature variable T is exported as output and synchronization labels h_{on} and h_{off} as inputs.



(b) Controller in UPPAAL uses urgent channels to ensure as-soon-as-possible transition trigger. Temperature T is an input and labels h_{on} and h_{off} are outputs.

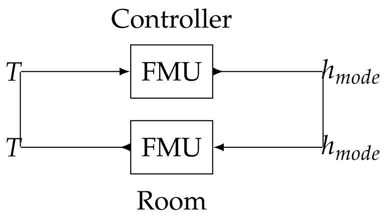
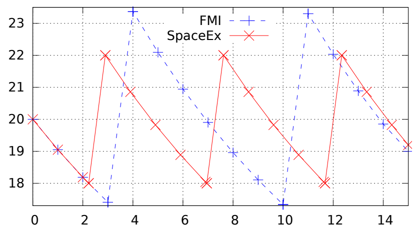
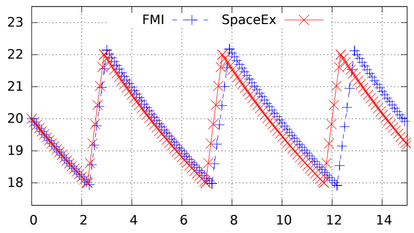


Fig. F.3: SPACEEX and UPPAAL FMUs connected using the room temperature T and heater mode h_{mode}.

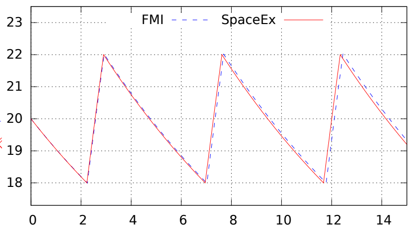
controller and room components. We consider three different simulation step values: 1 (see Fig. F.4a), 0.1 (see Fig. F.4b) and 0.01 (see Fig. F.4c). Considering the simulations, we observe that the FMU trajectories *overshoot* the controller constraints in the sense that the controller exhibits a delayed reaction when the room temperature crosses the temperature thresholds. The behavior is justified by the fact that the method call `doStep` for every FMU relies only on the *local* information about the state evolution when making decisions, e.g., the controller FMU does not have any information about the room temperature evolution beyond the value which can be provided when the method `doStep` is called. Therefore, the controller FMU detects that the guard is enabled only a *simulation iteration later* after this event has already happened. We observe that the impact of the overshooting can be made arbitrary small



(a) Maximum step size 1.



(b) Maximum step size 0.1.



(c) Maximum step size 0.01.

Fig. F.4: Simulation trajectories: each red × is a data point reported by SpaceEx, and blue + reported by the co-simulation.

by choosing a small enough simulation step (see Fig. F.4c vs. Fig. F.4a and Fig. F.4b).

We note that the overshooting problem is *inherent* to the considered master algorithm and can be circumvented by incorporating additional cross-component knowledge into the master algorithm. Overall, our experiments validate that on this case study our co-simulation framework based on SpaceEx and Uppaal provides equivalent simulation results compared to the setting where all components are modelled in one tool.

4.2 Supervisory Control Example

In this section, we show how supervisory control systems similar to the benchmarks presented by [11] can be modeled using the FMI paradigm. Compared to Section 4.1, we consider a model of the building with two rooms sharing a common wall and a heater. In this setting, the room temperature is influenced by both the outside temperature and heat transfer between the rooms. Figure F.5 shows a hybrid automaton from SpaceEx modeling the room temperature dynamics. The difference from the previous example here is an extra term $(Tother - t) * 0.2$ denoting a contribution from another room. Another room is modeled analogously except that it responds to *heater2_on* and *heater2_off* signals instead of *heater1_on* and *heater1_off*.

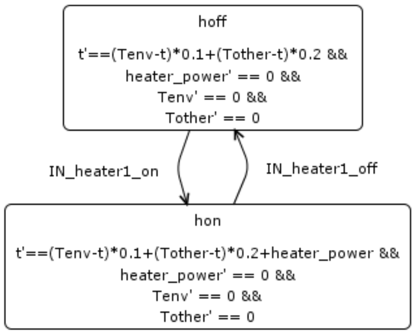


Fig. F.5: Hybrid automaton for a heated room connected to another room. Inputs are temperatures *Tenv*, *Tother* and labels *IN_heater1_on* and *IN_heater1_off*, while output is temperature *t*. We use the prefix *IN* to mark input labels.

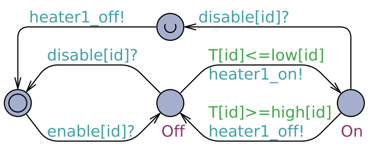
Our controller consists of two parts: local bang-bang controller and a supervisor shown in Fig. F.6. In order to model the transitions of the heaters between the rooms, we assume that the controllers can be turned on/off by the supervising controller. Therefore, the local controller has an extra mode besides *On* and *Off* which stands for the controller being currently deactivated. The supervising controller has two kinds of stochastic behavior: it can pick any pair of rooms (one recipient and another donor) to transfer the heater, and it can choose the timing of transfer. When a pair of rooms is selected (by choosing concrete room identifiers for *rec* and *donor* variables) the donor is disabled by moving from location *decide* to location *move* and the recipient is enabled by going from *move* to *idle*. The supervisor may stay in location *idle* arbitrary long, but the exact duration is decided by an exponential probability distribution of rate 1 which means the duration of 1/1 time units on average. Similarly the supervisor may stay in *decide* and *move* but the duration will be 1/10000 on average, i.e. denoting that the heater is moved rather quickly.

Figure F.7 shows the overall component connectivity diagram where the supervisor is reading temperatures from each room and controls the local movable heater controllers. The movable heaters then may either turn on the heat in their room or let them cool off giving the heat to outside. The individual heated rooms are then connected to the outside temperature and to each other denoting the heat exchange. The splitter FMUs are repeaters needed to connect multiple components to the same signal.

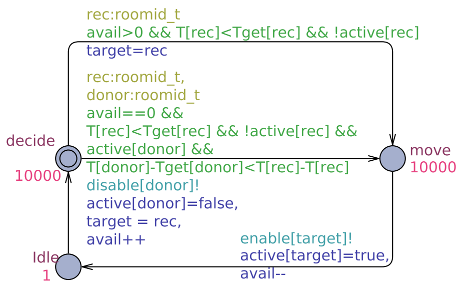
In the following, we discuss the behavior of the resulting composed model. Figure F.8 shows the temperature dynamics in each room. In particular, the plot shows that in the beginning the temperature drops until the supervisor detects a room temperate below $Tget = 17^\circ$, then around 6 time units a heater raises the temperature in room 1. The local controller keeps rising the temperature until it goes over 22° bound at around 7.5 time units. Notice that the temperature in room 2 also rises due to heat exchange between the rooms. Around 10 time units the supervisor decides to hand over the heater to room 2. At 14 time units the heater is switched back to room 1 and so on. We can conclude that even though the temperature drops well below 18° overall it seems that the controllers manage to sustain the temperature at the similar level without loosing control (without dropping to outside temperature level).

4.3 Stochastic Simulations and SMC

The following is a demonstration of statistical model checking (SMC) using the FMI framework. We show how the performance of two stochastic controllers simulated by Uppaal can be compared using SMC approach together with the heated room simulation provided by SpaceEx. Figure F.9 shows two controllers: (a) reacting within 1 time unit to 18.0° and 22.0° temperature bounds and (b) reacting within 2 time units to 19.0° and 21.0° temperature bounds. The channels used in these controllers are not urgent and therefore the delay between temperature detection and heater activation is decided stochastically based on uniform distribution over the allowed delay by invariants, i.e. the concrete delay will be chosen from $[0, 1]$ for the first controller and from $[0, 2]$ for the second one. The *On* and *Off* locations do not have any invariant and therefore in principle the process may stay there forever. In such cases Uppaal uses an exponential (Poisson) probability distribution to decide a particular time delay and hence asks to provide a rate of the exponential. The higher the exponential rate, the shorter the delays,



(a) Local bang-bang controller which can be moved (disabled). The inscribed U means urgent location where time delay is not allowed. The inputs are temperature variable *T[id]* and labels *enable[id]* and *disable[id]*, while outputs are labels *heater1_on* and *heater1_off*.



(b) Supervising controller moves the heaters between rooms by reading inputs on *T[i]* and sending outputs on labels *enable[i]* and *disable[i]* where *i* is the room index.

Fig. F.6: Two layers of Uppaal controllers.

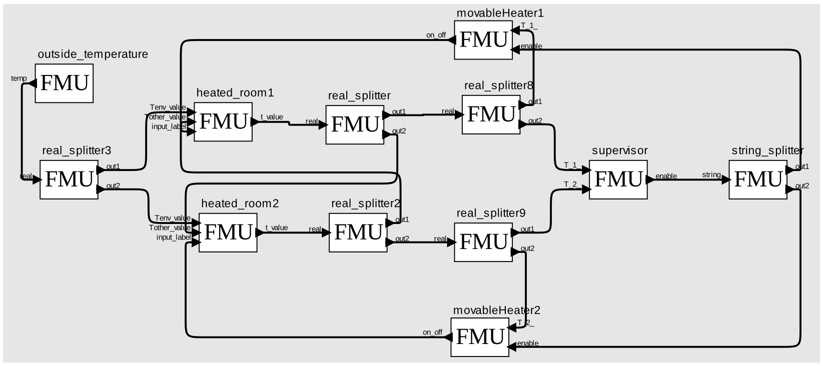


Fig. F.7: ptolemy diagram for supervisory control of two heated rooms.

hence we can provide a high rate to ensure that the detecting transition is fired arbitrary quickly.

In our setup, we would like to know which controller is better at keeping the room temperature within 18.0° and 22.0° bounds. In order to answer this question we setup two FMI models for each controller with an equal room, run 100 simulations with 100 time units in length and 0.05 granularity, compute the amount of time spent outside the temperature range for each simulation and then compute the confidence intervals for both models. Table F.1 shows a summary of amounts of time during which the temperature was either below or above the range. The estimated time duration use confidence interval (CI) notation which means that if we repeat the measurement experiment then the real mean (which is unknown) will fall into the interval with a probability of 95%. The results show that the second controller was

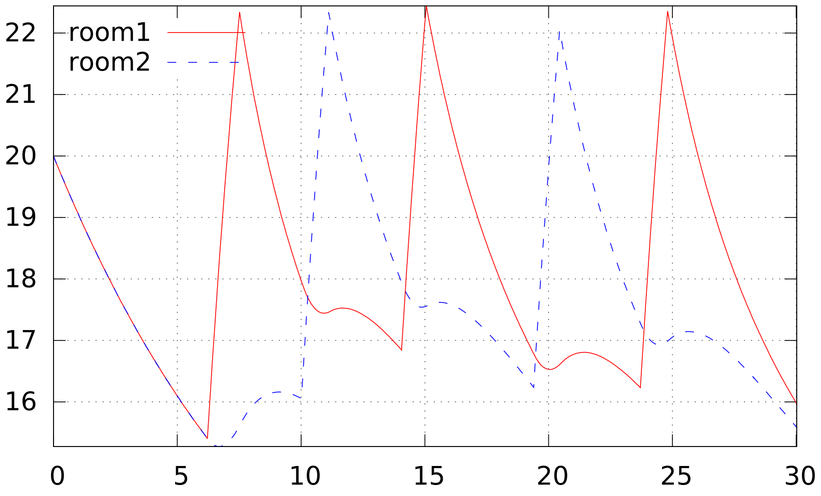
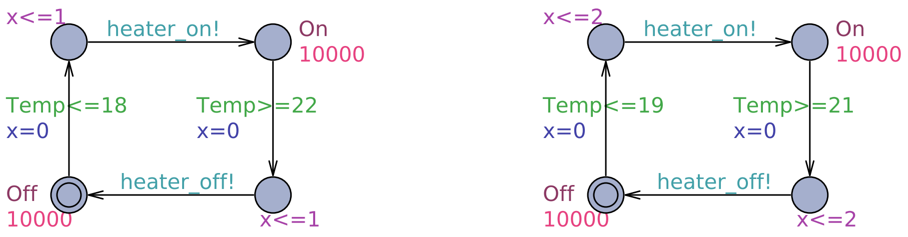


Fig. F.8: Temperature trajectories for each of the rooms composed with stochastic supervising controller.



(a) Wide range and fast. **(b)** Narrow range and slow.

Fig. F.9: Stochastic controllers use regular (non-urgent) channels, therefore timings are stochastic: delays are distributed uniformly (when clock invariant is used) and exponentially (in locations *On* and *Off*).

more successful at maintaining the lower bound of the temperature, but was more overshooting beyond the upper bound. In total, the first controller kept the temperature in good range longer by 8.57 time units on average, which is much larger than confidence interval, hence the first controller is better.

Table F.1: Time with temperature outside the range (95% CI).

Controller	Time below	Time above	Total
Wide and fast	7.56 ± 0.20	32.69 ± 3.36	40.26 ± 0.59
Narrow and slow	2.40 ± 0.19	46.43 ± 0.82	48.83 ± 0.79

5 Related Work

The FMI standard and corresponding documentation are constantly evolving, as new versions of the standard are developed. The web site[4] also contains a list of tools supporting FMI. Descriptions of FMI can also be found in the academic literature [14].

Discussions about the limitations of FMI can be found in the works by [10, 15]. [10] also formalize the main methods of FMI (`get`, `set`, `doStep`) by establishing a *contract* (pre-/post-conditions) for each method and propose a *master algorithm* (i.e., a co-simulation algorithm). Furthermore, the authors proves its termination, determinacy, and other properties. However, the paper does not discuss how FMUs can be created. A different, master-slave based, co-simulation approach is proposed by [16], but formal properties such as determinacy are not discussed in this work.

[15] defines a suite of test models that should be supported by a hybrid co-simulation environment, giving a mathematical model of an ideal behav-

[4]`https://www.fmi-standard.org/`

ior, plus a discussion of practical implementation considerations. Furthermore, the paper describes a set of basic modeling components in the spirit of ptolemy actors (constant, gain, adder, integrator, etc.). Finally, the authors provide a kind of denotational description for each component (input and output signals), but no encoding into FMUs is discussed.

The FMU generation problem for various formalisms is discussed by [12]. This work only refers to a generic model of timed machines which does not include the particularities of UPPAAL's timed automata. In addition, hybrid automata are not considered in this work.

Recently, the co-simulation algorithm presented by [10] has been implemented in the open-source ptolemy tool. As mentioned above, we use this framework in order to import FMUs into ptolemy and co-simulate them. However, this framework does not address the FMU generation problem.

[17] present a plugin for Rhapsody for generating FMUs from Statechart SysML blocks. They provide high level guidelines for how to generate Statechart FMUs, but do not provide a formalization. [18] also discuss how to encode statechart models, described in MechatronicUML.

Co-simulation is one, but not the only approach to solve the tool interoperability problem. A further attempt to solve this problem is the *Hybrid Systems Interchange Format* (HSIF), designed with the goal of being "a sort of 'maximum common denominator' among all hybrid system modeling environments" [19]. HSIF aims therefore at defining a "maximal syntax" where all the syntax of different languages could be translated into. It could be seen as a type of XML schema for hybrid systems. HSIF is primarily aimed at enabling model *translation* between different hybrid system tools. [20] present the tool Hyst which provides an automatic source-to-source model translation between a number of up-to-date hybrid model checkers. In their approach, [20] do not use any intermediate format like HSIF. Both model translation-based approaches outlined above provide support only of the *common subset* of the tool features. Our co-simulation framework does not limit the model designer to use the "maximal syntax" among all the tools because every FMU takes care of its features independently.

6 Conclusions

We have shown how two state-of-the-art modeling and verification tools for hybrid systems, SPACEEX and UPPAAL, can be integrated using the FMI co-simulation standard. The result is a powerful framework which allows users to build submodels separately in each of the two tools (as well as in other tools potentially), then generate individual FMUs for each submodel, and then combine all the FMUs into a single model, which can be co-simulated within the ptolemy FMI implementation. We demonstrated the feasibility of our framework by comparing the co-simulation results of a simple two-FMU model to the simulation results that are obtained when we model and simulate the entire system in a single tool. By empirical evaluation on case studies we found that, provided time steps are small enough, individual components can ensure timely reactions to continuous signals, and the simulations can be made arbitrary close to self-contained model simulation. In addition to individual tool export to FMUs, we showed how the non-deterministic models can be determinized using stochastic semantics and included into FMI co-simulation. We also provided an example how statistical model checking can be performed using numerous FMI simulations which is an essential feature evaluating stochastic behavior. The integration of model-checkers into co-simulation frameworks provides further possibilities of analyzing early design models like conformance monitoring by checking that a simulation trace of a refined (e.g. hybrid) model is included in a more a abstract (e.g. timed automata) specification. We envision our work being a further step towards integrating tools developed in the formal methods community into the industrial system design and modeling workflow of cyber-physical systems.

7 Acknowledgments

We are grateful to Christopher Brooks, Fabio Cremona, and Edward Lee from UC Berkeley, for their work on the ptolemy framework. This work was partly supported by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), 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), by EU FET project SENSATION, the Sino-Danish Center IDEA4CPS and the Center DiCyPS of the Danish Innovation Foundation, by Academy of Finland, by the National Science Foundation (awards #1329759 and #1139138), and by the Industrial Cyber-Physical Systems Research Center (iCyPhy) supported by IBM and United Technologies Corporations.

References

[1] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable Verification of Hybrid Systems," in *23rd International Conference on Computer Aided Verification (CAV)*, ser. LNCS, S. Q. Ganesh Gopalakrishnan, Ed. Springer, 2011.

[2] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[4] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[5] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang, "Statistical model checking for networks of priced timed automata," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, U. Fahrenberg and S. Tripakis, Eds. Springer Berlin Heidelberg, 2011, vol. 6919, pp. 80–96.

[6] D. N. Agut, D. A. van Beek, and J. Rooda, "Syntax and semantics of the compositional interchange format for hybrid systems," *The Journal of Logic and Algebraic Programming*, vol. 82, no. 1, pp. 1 – 52, 2013.

[7] H. Beohar, D. E. N. Agut, D. A. van Beek, and P. J. L. Cuijpers, "Hierarchical states in the compositional interchange format," in *Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010.*, ser. EPTCS, L. Aceto and P. Sobocinski, Eds., vol. 32, 2010, pp. 42–56.

[8] A. Pinto, L. P. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli, "Interchange format for hybrid systems: Abstract semantics," in *Hybrid Systems: Computation and Control*, ser. LNCS, J. P. Hespanha and A. Tiwari, Eds. Springer Berlin Heidelberg, 2006, vol. 3927, pp. 491–506.

[9] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[10] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, S. Tripakis, M. Wetter, and M. Masin, "Determinate Composition of FMUs for Co-Simulation," in *13th ACM & IEEE International Conference on Embedded Software (EMSOFT'13)*, 2013.

[11] A. Fehnker and F. Ivancic, "Benchmarks for hybrid systems verification," in *In Hybrid Systems: Computation and Control (HSCC)*. Springer, 2004, pp. 326–341.

[12] S. Tripakis, "Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation – SAMOS XV*, 2015.

[13] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," *International Journal on Software Tools for Technology Transfer*, 2015.

[14] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *8th International Modelica Conference*. Dresden, Germany: Modelica Association, Mar. 2011.

[15] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for Hybrid Cosimulation Standards," in *Hybrid Systems: Computation and Control (HSCC)*, 2015.

[16] J. Bastian, C. Clauß, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *8th International Modelica Conference*, 2011.

[17] Y. A. Feldman, L. Greenberg, and E. Palachi, "Simulating Rhapsody SysML Blocks in Hybrid Models with FMI," in *10th Modelica Conference*, 2014, pp. 43–52.

[18] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner, "Generating Functional Mockup Units from Software Specifications," in *9th Modelica Conference*, 2012, pp. 765–774.

[19] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone, "Interchange formats for hybrid systems: review and proposal," in *Hybrid Systems: Computation and Control*, ser. HSCC. Springer, 2005.

[20] S. Bak, S. Bogomolov, and T. T. Johnson, "HYST: a source transformation and translation tool for hybrid automaton models," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC, Seattle, WA, USA, April 14-16, 2015*. ACM, 2015, pp. 128–133.

Paper G

Integrating Tools: Co-Simulation in Uppaal using FMI-FMU

Ulrik Nyman, Peter Gjøl Jensen, Kim Guldstrand Larsen and Axel Legay

The paper has been published in the
Proceedings of 22nd International Conference on Engineering of Complex Computer Systems, pp. 11–19, 2017.

Abstract

While standalone tools for verification and modeling have proven useful, their chosen formalism and description-language can at times be restrictive. We demonstrate how to use Uppaal SMC *to analyze controller systems consisting of Function Mockup Units (FMU) modeled in other tools, such as Matlab and Modelica. Apart from supporting FMI-FMU modules the newly added* `C` *interface can call any external function. The only requirement for sound analysis is statelessness and determinism of the external function. We demonstrate the expressive power by implementing the FMI-FMU master algorithm as a timed automata, interfacing with external, non-native and non-trivial Function Mockup Units (FMU). We also model two components in* Uppaal SMC *exporting one of them as an FMU while keeping the other as a native component. Furthermore we demonstrate the first simulation environment for the Function Mockup Units, capable of checking bounded MITL properties.*

1 Introduction

Model-checking tools often come with support for a specific formalism, imposing restrictions on the behavior of the modeled system. Such formalisms serve the purpose of giving the system a *semantics* such that the behaviour of the system carries a sound and consistent meaning. In particular without a semantics, investigating the accuracy or probabilistic behavior of a system is nonsensical. At the same time, a variety of formalisms have emerged in different domains, each well suited for a specific task (eg. digital, mechanical or thermodynamic modeling), but incapable of *co-simulating* – that is, to obtain joint results. To remedy this, the *Function Mockup Interface*-standard (FMI) [1] was proposed to enable domain-specific modeling tools to be used side by side when encapsulated in a *Function Mockup Unit* (FMU). In particular, the FMI-standard defines a protocol for how values *can* be communicated between FMUs by standardizing the interface-description and giving a common `C`-api.

However, while the FMI standard only specifies how values *can* be communicated, it purposely does not specify the details of how they *will* be communicated, calling for the development of so-called *Master Algorithms* (MA) for coordinating the interaction between several FMUs.

As previously demonstrated [2], the MA has an impact on the semantics of the overall system, and is thus of great importance to the overall meaning of the measures obtained. We demonstrate that our implementation of the MA will, given enough simulations, eventually explore all possible executions. This ensures that we will also discover cases where the ordering of events has a great impact on the outcome.

We will be implementing our approach as an extension of the statistical model checking tool UPPAAL SMC [3], thus ensuring that we have a formally defined semantics. We show that meaningful probabilistic measures can be obtained, and in particular that we can statistically verify MITL (Metric Interval Temporal Logic) properties. The work presented here extends beyond applications within the FMI-standard – we propose and implement an extension of UPPAAL that allows for calling arbitrary C-libraries during the statistical simulation. This effectively opens up UPPAAL SMC to a great number of applications – and we will argue that under statelessness of the external library, that doing so is semantically sound. By statelessness, we do not mean that the components cannot change their state, but that all relevant state information should be communicated through the FMI-FMU interface such that it is controlled by the MA. This is also required in order to retain the semantics in future settings such as classical model checking, as demonstrated in [4]. External FMUs can contain stochastic behavior, but they should not contain unresolved non-determinism, as such non-determinism would not have a meaningful interpretation when simulating the system.

Notice that for FMUs we embrace the argument of Broman et. al. [5] – that it is sane to expect that components only can give a maximum delay and must accept all smaller delays.

1.1 Interleaving Semantics

While the work of Bogolomov et. al [2] showcased UPPAAL as an FMU alongside SpaceEx with ptolemy as MA, it also exposed inconsistencies in the semantics of timed automata when recomposed via the MA as opposed to internally in UPPAAL. Let us recall the example provided by Bogolomov et. al; consider the four TAs presented in Figure G.1. If the four TAs communicate in a pipeline pattern s.t. A1 outputs to A2, A2 to A3 and A3 to A4, then only a single, deterministic trace can occur. Given that all TAs start in their initial location (marked by double-circle) and with clock-value x1=x2=x3=x4=0 nothing will happen until a single unit of time has elapsed ($x1 == 1$, $x2 == 1$, $x3 == 1$ and $x4 == 1$). After exactly one time unit (enforced by the invariants $x1 <= 1$, $x2 <= 1$, $x3 <= 1$ and $x4 <= 1$), all the automata are able to output – however, as TAs synchronize on channels (a, b, c, d, output marked by ! and input by ?), it becomes important in which order the automata synchronize. Notice here that even though all the synchronizations happen at the same instance of time, it is important to track the “happened before” relationship. Keeping this in mind, we can observe that only a single deterministic trace of the system can occur given the pipeline communication-pattern, namely that an $a!$ always will be observed followed by $c!$, implying that on all traces, eventually the product-state A1.A && A2.AB && A3.C && A4.CD is reached. However, one can easily verify using UPPAAL that this is not the case. In fact, all possible permutations with at least one winner is possible, as UPPAAL im-

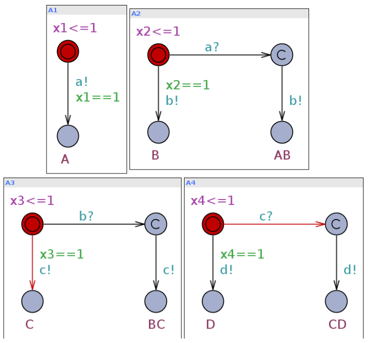


Fig. G.1: Four Uppaal timed automata exemplifying the different semantics of different Master Algorithms. An intuitive explanation of the syntax of Uppaal timed automata can be found in Section 5.2.

plements interleaving semantics – which most definitions of NTA (Networks of Timed Automata) require.

We argue that such behavior is also important in real-life models, for instance due to a too coarse granularity on the observation of time. In particular in a probabilistic context, such under-approximation of the behavior of the system leads to erroneous probability estimates. While we do not address channel-based synchronization between FMUs in this paper, we note that similar erroneous behavior can be achieved using only integer-valued variables. For the specifics of channel based synchronization between FMUs, we refer the reader to [2].

Another known problem in modeling and simulation is that of *zero-crossing*; namely that it is impossible in a simulation setting to detect the exact time when a certain value crosses a zero threshold. As mentioned in [6] this is also a problem in the context of FMI-FMU models. This is a problem that we are aware of, but that our current solution does not try to address.

1.2 Contributions

We demonstrate the ease with which already existing system models from Modelica can be exported as FMUs and used for verifying statistical properties of the model in Uppaal SMC. In particular, the ability to specify a MA within a sound semantical framework facilitating probabilistic and temporal reasoning is a strength of the methods proposed in this paper. These features are essential to modeling real-world scenarios where behavior is inherently uncertain and time-dependent – such scenarios include signal noise, human interaction and general natural phenomena. As we base our approach on top of a tool that already supports the notions of time and probabilities, we automatically gain the analytical capabilities of this tool. While we only demonstrate the Statistical Model Checking features of Uppaal SMC in this paper, our proposed method of embedding FMUs as timed automata also enables classical model checking, controller synthesis and controller learning on composed models using the more complex features of Uppaal, Uppaal Tiga and Uppaal Stratego under some reasonable restrictions on the FMUs.

The contributions of the paper can be summarized as follows:

- Implementation of direct call of FMI-FMU modules from inside a statistical model checking tool that supports time and probabilities.
- Discussion on different semantics for FMI-FMU implementations.
- Flexible modeling of the master algorithm as a timed automaton template.
- Statistical model checking of bounded reachability of temporal logics (MITL).

1.3 Related Work

FMI-FMU has its origin in the European research project MODELISAR. It was created specifically for integrating a wide variety of modeling tools. Industry tools that support the FMI-FMU standard include: MATLAB/Simulink, MapleSim and AUTOSAR Simulation [1].

In the following we try to cover recent and relevant related work that take a practical approach to co-simulating industrial systems.

Pazold et. al. [7] compare different approaches for simulating a HVAC system for a complete building. They conclude that a weak coupling using a co-simulation strategy using sub-models exported as FMUs is a reasonable approach. A case-study by Pedersen et. al. [8] describes how to integrate a special purpose maritime embedded system into an FMI-FMU co-simulation setting at the company MAN Diesel & Turbo.

We also find that it is relevant to look at related work that considers the semantics of the complete co-simulation system and Master Algorithm (MA).

In [9] Guermazi et al. provide a framework for co-simulating the UML models specified in Papyrus (the open-source UML/SysML modeler of the Eclipse foundation) in an FMI-FMU context. The work is build on top of the formal semantics foundation of UML (fUML [10])

In [5] Broman et al. argue for an number of sanity conditions for a MA. In this work we deviate by 1. allowing the order of FMI/FMU-definition determine the outcome of the simulation and 2. not requiring an order on the input/output-relation. However, both properties can be ensured by modifying the proposed timed automaton template for a MA. At the same time, Broman et al. propose the `getMaxStepSize`-extension of the FMI-FMU standard to enable step-size negotiation between FMUs.

In [2] Bogomolov et al. argue for allowing Zero-delay step-sizes in the `doStep`-procedure to enable timed automata style synchronizations across FMUs – a value which is otherwise strictly disallowed by the FMI-FMU standard. This paper also discusses using Hybrid- and timed automata as FMUs and the related semantic difficulties and utilizes the step-negotiation strategy introduced in [5].

In [6] Cremona et al. introduce the concept of step revision, similar to that presented in [5], as a method for ensuring the accuracy of modeling a mix of continuous-time and discrete-event systems in an attempt to address the *Zero-crossing* problem.

Statistical model checking of Priced Timed Automata and Stochastic Hybrid Systems was introduced into the UPPAAL tool-chain by Bulychev et al. in [11, 12]. In their work they present methods and algorithms for obtaining various statistical measures over stochastic systems using expressive logics [3, 13]. While the UPPAAL SMC tool has been used in a number of case-studies ([14–18]) it previously did not facilitate co-simulation as master-algorithm using the FMI-FMU standard.

The PLASMA statistical model checking tool [19] supports statistical analysis over a System of Systems composed of FMUs [20]. However, their work does not provide a formal semantics of the composed system or individual components, nor does it allow `C`-functions to be embedded directly in the model. Furthermore, the integration of FMUs in `C` allows for analysis, synthesis and learning using the more complex features of classical UPPAAL [21] and UPPAAL STRATEGO [3] – features that are out of the scope of this paper.

2 Semantics

We shall here describe the semantics of Timed Automata (TA), Network Of Timed Automata (NTA), Stochastic Timed Automata (STA) and Function Mockup Units (FMUs). We will then discuss a semantical embedding of FMUs into the the STA framework.

2.1 Stochastic Timed Automata

Formally, a TA is a finite automaton extended with a set of real-valued, time-progress-measuring counters ($\mathcal{X}$) called clocks. In addition, a TA allows for synchronization with other TAs over a finite set of so-called channels (Σ). For a set of channels Σ we let $\Sigma_o = \{a! \mid a \in \Sigma\}$ be the set of output actions over Σ while we let $\Sigma_i = \{a? \mid a \in \Sigma\}$ be the set of input actions. For a set of clocks $\mathcal{X}$ we call an element $c \bowtie n$ where $c \in \mathcal{X}$ and $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <\}$ ($\bowtie \in \{\geq, >\}$) an upper (lower) bound over $\mathcal{X}$. Let $\mathcal{B}^{\leq}(\mathcal{X})$ ($\mathcal{B}^{\geq}(\mathcal{X})$) be the set of all upper (lower) bounds over $\mathcal{X}$.

We call a mapping $\nu : \mathcal{X} \rightarrow \mathbb{R}$ for a valuation over $\mathcal{X}$ and denote all valuations over $\mathcal{X}$ by $\mathcal{V}(\mathcal{X})$.

Definition 20 (Timed Automaton)
A Timed Automaton (TA) is a tuple $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, \rightarrow, I, \mathcal{R})$, where

1. L is a finite set of control locations,
2. $\ell_0 \in L$ is the initial location,
3. $\mathcal{X}$ is a finite set of clocks,
4. Σ is a finite set of channels,
5. $\rightarrow\ \subseteq L \times \mathcal{B}^{\geq}(\mathcal{X}) \times (\Sigma_o \cup \Sigma_i) \times 2^{\mathcal{X}} \times L$ is a set of edges. We write $\mathsf{Loc} \xrightarrow{g,a,\mathcal{U}} \mathsf{Loc}'$ for an edge where Loc is the source and Loc' the target location, $g \in \mathcal{B}^{\geq}(\mathcal{X})$ is a guard, $a \in \Sigma_o \cup \Sigma_i$ is a label, and $\mathcal{U} \in \mathcal{V}(\mathcal{X}) \rightarrow \mathcal{V}(\mathcal{X})$ is a partial function giving the discrete-clock updates,
6. $I \colon L \rightarrow \mathcal{B}^{\leq}(\mathcal{X})$ is an invariant function, mapping locations to a set of invariant constraints and
7. $\mathcal{R} : L \rightarrow \mathcal{X} \rightarrow \mathbb{R}$ assign rates to the individual clocks in each location[1].

Let $\nu \in \mathcal{V}(\mathcal{X})$ be a valuation, $R : \mathcal{X} \rightarrow \mathbb{R}$ give clocks rates, $d \in \mathbb{R}$ be a real-valued number and let $\mathcal{U} \in \mathcal{V}(\mathcal{X}) \rightarrow \mathcal{V}(\mathcal{X})$ be an update-function ; then we let $(\nu + d \cdot R)$ be the valuation ν' where $\nu'(x) = \nu(x) + d \cdot R(x)$ and we let $\nu'' = \mathcal{U}(\nu)$. If $g = c \bowtie n$ is a clock bound over $\mathcal{X}$ and $\nu \in \mathcal{V}(\mathcal{X})$ then ν satisfies g ($\nu \models g$) iff $\nu(c) \bowtie n$. This generalizes in a natural way to a set of clock bounds.

The state of TA $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, \rightarrow, I, \mathcal{R})$ is a tuple (Loc, ν) where $\mathsf{Loc} \in L$ and $\nu \in \mathcal{V}(\mathcal{X})$. From a state (Loc, ν) the TA may

1. do a timed transition $(\mathsf{Loc}, \nu) \xrightarrow{d} (\mathsf{Loc}, \nu')$ if $\nu' = (\nu + d \cdot \mathcal{R}(\mathsf{Loc}))$ and $\nu' \models I(\mathsf{Loc})$ or
2. do a discrete transitions $(\mathsf{Loc}, \nu) \xrightarrow{a} (\mathsf{Loc}', \nu')$ if there exists $\mathsf{Loc} \xrightarrow{g,a,\mathcal{U}} \mathsf{Loc}'$ such that $\nu \models g$, $\nu' = \mathcal{U}(\nu)$ and $\nu' \models I(\mathsf{Loc}')$.

We define a partition over the clocks of a TA into time-independent, real-valued variables ($\mathcal{X}^{\mathtt{V}}$) and time-dependent clocks ($\mathcal{X}^{\mathtt{T}}$) s.t. $\mathcal{X}^{\mathtt{V}} = \{x \in \mathcal{X} \mid \forall \mathsf{Loc} \in L$ we have $\mathcal{R}(\ell)(x) = 0$ and x is not restricted by $I(\mathsf{Loc})\}$ and $\mathcal{X}^{\mathtt{T}} = \mathcal{X} \setminus \mathcal{X}^{\mathtt{V}}$.

[1] Allowing rates other than one is non-standard in TA semantics – in fact this renders most classical model checking questions undecidable. However, as the methods presented are simulation-based, arbitrary rates on clocks are practically feasible and semantically sane [12].

We define the infix-operator valuation-join operator $\overset{X,Y}{\otimes} : \mathcal{V}(X) \times \mathcal{V}(Y) \to \mathcal{V}(X)$ as

$$\nu \overset{X,Y}{\otimes} \nu' = \nu'' \text{ where } \nu''(x) = \begin{cases} \nu'(x) \text{ if } x \in Y \\ \nu(x) \text{ otherwise} \end{cases}$$

We shall simply write $\otimes$ for this operation and let X, Y be implicitly defined by the given valuations.

Following the compositional framework of [22] we require that a TA for any state s is

1. *input-enabled* i.e. for any $a! \in \Sigma_o$ there exists some s' such that $s \xrightarrow{a!} s'$ and
2. *action-deterministic* i.e. if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ then $s' = s''$.

Let us now define a Network of Timed Automata (NTA) with shared clocks.

Definition 21 (Network Of Timed Automata)
Let $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ be TA where $A_i = (L_i, \ell_0^i, (\mathcal{X}_i^{\mathrm{V}} \cup \mathcal{X}_i^{\mathrm{T}}), \Sigma, \to_i, I_i, \mathcal{R}_i)$ with the implicit indices ordering $1 < 2 < \cdots < n$. Let $\mathcal{X}^{\mathrm{V}} = \bigcup_{i \in 1,\dots,n} \mathcal{X}_i^{\mathrm{V}}$ be the set of shared clocks, then it holds for all $i \in 1, \dots, n$ that $\mathcal{X}^{\mathrm{V}} = \mathcal{X}_i^{\mathrm{V}}$ and for all $j \in 1, \dots, n$ where $i \neq j$ that $\mathcal{X}_j^{\mathrm{T}} \cap \mathcal{X}_i^{\mathrm{T}} = \emptyset$. Also let $\mathcal{S}(\mathcal{A}_i) = L_i \times \mathcal{V}(\mathcal{X}_i^{\mathrm{T}}) \times \mathcal{V}(\mathcal{X}^{\mathrm{V}})$; then we define the states of the network $\mathcal{A}_1 \| \mathcal{A}_2 \| \dots \| \mathcal{A}_n$ to be a pair $\langle (s_1, s_2, \dots, s_n), \nu \rangle$ where if $(\ell_i, \nu_i) = s_i$ then $(\ell_i, \nu_i \otimes \nu) \in \mathcal{S}(\mathcal{A}_i)$. A network may transit from $\langle (s_1, s_2, \dots, s_n), \nu \rangle$ by

- a timed transition

$$\langle (s_1, s_2, \dots s_n), \nu \rangle \xrightarrow{d} \langle (s_1', \dots, s_n'), \nu \rangle$$

  if for all i, $(\ell_i, \nu_i \otimes \nu) \xrightarrow{d} (\ell_i, \nu_i' \otimes \nu)$ where $(\ell_i, \nu_i) = s_1$, $(\ell_i', \nu_i') = s_2$, and

- a discrete transition

$$\langle (((\ell_1, \nu_1), \dots, (\ell_n, \nu_n)), \nu \rangle \xrightarrow[i]{a!} \langle (((\ell_1', \nu_1'), \dots, (\ell_n', \nu_n')), \nu^n \rangle$$

  if (assuming w.log. that $i = 1$)

  1. $(\ell_1, \nu_1 \otimes \nu) \xrightarrow{a!} (\ell_1', \nu_1' \otimes \nu^1)$, and
  2. for $j \in \{2, \dots, n\}$ we have

$$(\ell_j, \nu_j \otimes \nu^{j-1}) \xrightarrow{a?} (\ell_j', \nu_j' \otimes \nu^j)$$

Notice that the discrete transition-rule can be generalized beyond $i = 1$ by a temporary reordering of the indices.

$$F_s(\omega) = \sum_{i=0}^{m} \left(\mathcal{I}_{t>0} \delta_{s_i}^{A_i}(t) \cdot \prod_{j \neq i} \left(\mathcal{I}_{\tau>t} \delta_{s_j}^{A_j}(t) d\tau \right) \cdot \gamma_{s_i^t}^{A_i}(a_1! \cdot F_{s'}(\omega^1) dt \right) \tag{G.1}$$

2.2 Stochastic Semantics

David et al. provides the full stochastic semantics of Stochastic Timed Automata in [23]. Here the semantics is given as a series of repeated races among components making up the network. In essence, each sub-component will choose a delay in accordance with the probability distributions defined locally to that component. The component with the smallest delay will then win the race and gets to do a discrete step – again chosen according to a local distribution. However, this discrete step can synchronize with neighboring components, possibly altering their internal state. This procedure is then repeated over and over.

In the semantics the delays are chosen as follows: if the possible delays are bounded, the distribution is a uniform distribution between the minimal delay before some action is possible and the maximal delay where a delay is still possible.

Formally, we assume there for any state (s) of any TA $\mathcal{A}$ exists a delay-density $\delta_s^{\mathcal{A}} : \mathbb{R} \to \mathbb{R}$ and a probability mass function $\gamma_s^{\mathcal{A}} : \Sigma_o \to \mathbb{R}$. Naturally we will require these functions to be "sane" in the sense that they do not assign probability mass (density) to impossible actions (delays) i.e. $\gamma_s^{\mathcal{A}}(a!) \neq 0$ ($\delta_s^{\mathcal{A}}(a!) \neq 0$) implies $s \xrightarrow{a!} s'$ ($s \xrightarrow{d} s'$).

Let $\omega = a_1!a_2! \ldots a_n!$ be a finite sequence of output-actions: then we define the probability of a network $\mathcal{A}_1 \| \ldots \mathcal{A}_m$ generating the sequence from state $s = (s_1, \ldots S_n)$ recursively by Equation G.1. where $s_i \xrightarrow{d} s_i^d$, $s \xrightarrow{d} \xrightarrow[i]{a_1!} s'$, $\omega^1 = a_2! \ldots a_n!$ and base case $F_s(\epsilon) = 1$.

2.3 Function Mockup Unit

Similar to Broman et. al. [5], we shall here define the semantics of a single FMU as a (timed) state machine. For simplicity, we shall in the definition ignore the types of the variables, and assume wlog. that they all are reals.

Definition 22
An FMU is a tuple $\mathcal{F} = (\mathcal{S}, \texttt{init}, \texttt{V}, \texttt{set}, \texttt{get}, \texttt{doStep})$ where

- $\mathcal{S}$ is a set of states,
- $\texttt{init} \in \mathcal{S}$ is the initial state,
- V is a set of variable names,

- $\mathtt{set} : \mathcal{S} \times \mathbb{R}^{\mathtt{V}} \to \mathcal{S}$ is a value-setter function,
- $\mathtt{get} : \mathcal{S} \times \mathtt{V} \to \mathbb{R}$ is a value-getter function and
- $\mathtt{doStep} : \mathcal{S} \times \mathbb{R}_{\geq 0} \to \mathcal{S}$ is the time-progression function.

For a given FMU $\mathcal{F}$, the exact semantics is defined by the underlying implementation, and we shall hence only focus on the interaction with STAs.

Definition 23
A stochastic FMU is a tuple $\mathcal{F}_s = (\mathcal{S}, \mathtt{V}, \mathtt{set}, \mathtt{get}, \mathtt{doStep}, \mathcal{P})$ s.t.

- $\mathcal{S}, \mathtt{V}, \mathtt{set}, \mathtt{get}, \mathtt{doStep}$ are defined as for a regular FMU and
- $\mathcal{P} : \mathcal{S} \to [0, 1]$ is the probability that $\mathcal{F}_s$ starts in $s \in \mathcal{S}$ and we have that $1 = \Sigma_{s \in \mathcal{S}} \mathcal{P}(s)$.

Let $\mathcal{F}_s$ be a stochastic FMU, then we let

$$\mathtt{pick}(\mathcal{F}_s) = (\mathcal{S}, \mathtt{init}, \mathtt{V}, \mathtt{set}, \mathtt{get}, \mathtt{doStep})$$

where $\mathtt{init} \in \mathtt{init}_s$ is the initial state chosen according to $\mathcal{P}$ – by agreement for a non-stochastic FMU we let $\mathtt{pick}(s) = 1$ where $\mathtt{init}_s = \{s\}$ and $P(s') = 0$ for all other $s' \neq s$.

Notice here that our extension of FMUs with stochastic initial state has the probabilistic choices resolved once; this construction is both practically and theoretically sound. In practice, such a construction can be ensured by using seeded pseudo-random number generators. Here the seed is chosen at random initially, leading to a subsequent determined execution. In a theoretical setting, similar generative constructions have been used to define the semantics of probabilistic programs – for instance for the semantics of the IBAL language as proposed by Pfeffer et al. in [24].

It is easy to see that the semantics of an FMU fits well within the stochastic semantics for NTA as the standard specifies a collection of (complex) update-functions over a state – which in turn can be encoded as a real. However, as FMUs have no notion of discrete-update labels, to correctly embed an FMU into an NTA, we shall in the next section describe one way of encapsulating an FMU within a single STA, such as to make it compatible with the NTA framework.

3 Extension of Uppaal

To facilitate the import of FMUs in Uppaal, we have extended Uppaal with a construct for loading dynamic `C` libraries at run-time. External `C` functions in Uppaal look and act exactly like regular functions in Uppaal and only their declaration differ by the additional import environment – an example of the new syntax can be seen in Figure G.2. We shall here discuss the type-conversions between the C-like language in UPPAAL and actual C. Furthermore we introduced the type string for string constants and the type ptr_t for holding pointers to external data—a type with its binary defined by, and dependent on, the hardware platform UPPAAL is executing on and functionally equivalent to size_t known from regular C. Lastly, to facilitate single-initialization of external libraries, we have introduced the void __ON_CONSTRUCT__() and void __ON_DESTRUCT__() – that if existing, will be called upon model-initialization (and de-initialization respectively), but not for each proposition given to UPPAAL. These can be defined at a global scope as well as in the scope of each individual TA.

3.1 Type Conversion

To maintain sanity during simulation some restrictions on the types being transferable between UPPAAL and external functions are in place. Currently, the types bool, chan, clock, double, ptr_t, int and string can be used in external functions, omitting complex types constructed using the struct keyword as well as two, and more, level arrays. A chart of the type-conversion to C and other restrictions are given in Table G.1. We further emphasize some significant differences from linking directly between C programs and working calling external functions from UPPAAL.

- A bool variable in UPPAAL is either zero or one—as such, any function returning bool will return 0 if the C-function called returned 0, and 1 otherwise.
- const is enforced, implying that any variable sent as const, even if sent as an array or by reference, will not have its value changed in the UPPAAL environment—regardless of the behavior of the C-function called.
- Each import-statement has a private scope, as exemplified by Figure G.2.
- Integers in UPPAAL can be given a bounded range. If this range is violated – either when values are sent by reference or returned – the model is said to have violated model-sanity, causing a run-time error.

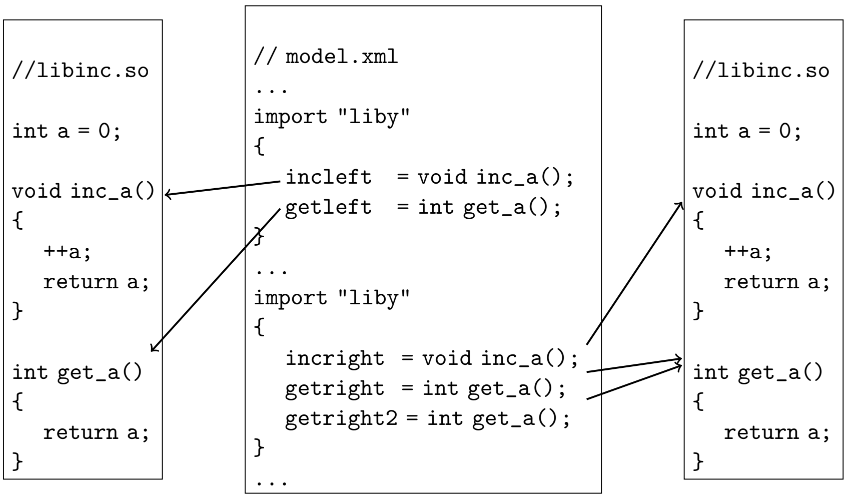
Utilizing this extension of UPPAAL and the FMI-standard, we will implement a co-simulation-algorithm directly as a timed automaton.

4 FMI/FMU in Uppaal

The FMI/FMU standard, version 2.0 is a standard for conducting co-simulation between different simulation environments. While the standard also supports

UPPAAL type	C type	By Value	Return	Array
bool	bool	✓	✓	✓
chan	const char			
clock	double			✓
double	double	✓	✓	✓
ptr_t	size_t	✓	✓	✓
int	int32_t	✓	✓	✓
string	const char			
<type>[]	<type>			

Table G.1: The type-conversion between UPPAAL and C. The complex types chan, string, clock and array-types are sent in C-convention as pointers to raw memory and are thus forced to be sent as references. All types can be sent by reference, but the immutable types chan and string are forced const. Lastly, only single-dimension arrays are supported, and only of mutable types; arrays of chan and string are currently not supported.



```
//libinc.so

int a = 0;

void inc_a()
{
   ++a;
   return a;
}

int get_a()
{
   return a;
}
```

```
// model.xml
...
import "liby"
{
   incleft   = void inc_a();
   getleft   = int get_a();
}
...
import "liby"
{
   incright  = void inc_a();
   getright  = int get_a();
   getright2 = int get_a();
}
...
```

```
//libinc.so

int a = 0;

void inc_a()
{
   ++a;
   return a;
}

int get_a()
{
   return a;
}
```

Fig. G.2: Scoping rules of external libraries when loading into UPPAAL. Each import-statement creates a new environment – this implies that the variable a has two logical instances, a_{left} and a_{right} where incleft and getleft references a_{left} while incright, getright and getright2 references a_{right}.

"Model Exchange", we shall here focus only on co-simulation.

To conduct co-simulation using the FMI/FMU standard, one needs only two components: (1) A master algorithm controlling the overview and coordination of the composed simulation and (2) one or more FMUs for the master-algorithm to coordinate and facilitate communication between. This approach shows its strength by outsourcing the heavy computation of the simulation to specialized tools while maintaining a global overview and coordination of the composed system at any time. An example of such a composed system, using Uppaal as a FMU for timed automata, was presented by Bogolomov et. al [2]. In general the MA is imposed on top of the system, enforcing a semantics particular to the given MA. In our work, we instead embed the external FMUs as timed automata. This allows us to reuse standard, well-defined, timed automata semantics, allowing us to construct a shallow MA, ensuring only that the FMI/FMU communication protocol is respected. In particular, from this approach, we adopt the so-called Interleaving Semantics.

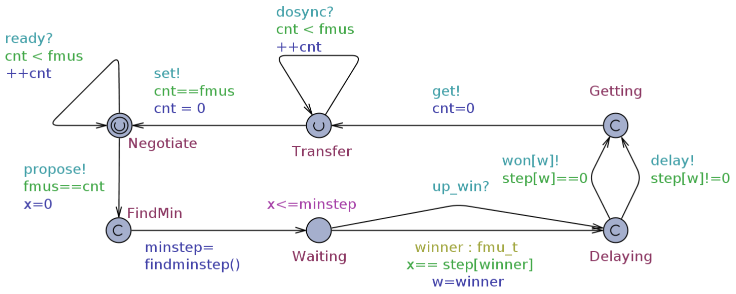
4.1 Master Algorithm and FMUs as Timed Automata

The TAs used for implementing the MA in Uppaal and importing FMUs into Uppaal are shown in Figure G.3a and Figure G.3b respectively. Let us walk through the computation of a single simulation step in the composed model. Initially, observe that

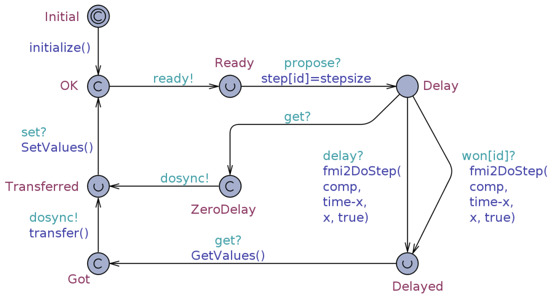
- the state of each FMU is encoded in the comp variable,
- time is a variable tracking the time progressed since the beginning of the simulation,
- x is a variable tracking the time since the last simulation step,
- step is an array containing all the proposed step-sizes and
- cnt is a variable tracking the number of FMUs that have completed a given stage of the MA – initially set to zero.

Initially the MA (depicted in Figure G.3a) and each FMU (depicted in Figure G.3b) is in the Negotiate and Initial-states. As the MA is waiting for *ready* signals from the FMUs, all the FMUs will (in random order), call the initialize-function, abbreviating the setup function calls specified in the FMU standard. After initialization, each FMU will move from the OK location to the Ready location – and while doing so, synchronize with the MA on the *ready* channel. This forces the MA to wait until all FMUs have reached the Ready-location. At this point, the MA is forced to move from Negotiate to Find-Min, synchronizing with all FMUs at once, s.t. each FMU moves from Ready to Delay. This makes each FMU propose the step-size (which could be computed by more complex functions, such as proposed by Bogolomov et. al [2]), after which it will wait for the MA to synchronize on either *delay*, *won*[*id*] or *get*. The MA will now, stepping from FindMin to Waiting, choose the minimal proposed step-size and let time progress by the given amount. Whenever time has progressed exactly minstep time-units, the MA will with even probability determine a "winner" in between FMUs proposing exactly minstep time-units as the delay. If the delay is non-zero, the MA will synchronize on the *delay* channel, in which case all the FMUs will end up in the Delayed location, ready to receive a synchronization on *get*. If the delay is zero, only the winning FMU will progress to the Delayed location while the remaining FMUs will await further synchronization. From the Delayed location, each FMU, upon synchronization with the MA on *get*, will call the appropriate getter-functions defined in the FMU-standard, synchronizing the data-arrays in UPPAAL with those in the external FMU. Notice here that if a zero-delay occurred, some FMU will have won, and all the losing FMUs will at this point in time move from the Delay to the ZeroDelay location – implying that data is not fetched from the losing FMUs as it cannot have changed since the last call to GetValues(). At this point in a simulation-step the MA has reached the Transfer location while all the FMUs are in the Got location. Here each FMU will, in random order according to a uniform distribution, transfer their local values (synchronized with the external FMU) to other FMUs in the system. The randomness here plays a key part in implementing interleaving semantics, as two FMUs can write to the same variable in a third FMU, effectively yielding a race-condition. Again, if we are in the special case of a zero-delay, only the winning FMU will transfer its values. Until all FMUs have moved to the Transferred location, the MA will wait in the Transfer location. Eventually the MA will be able to move from Transfer to the Negotiate location, triggering all the FMUs to move from Transferred to OK, pushing the updated values to the individual FMUs. This completes the cycle of a single step in the total simulation.

While our implementation here focuses on a specific MA, one can easily extend and test different MA algorithms within this framework. In [2], the authors restrict their MA to impose an ordering of the value-transfers between the FMUs – such a restriction could be implemented by imposing an order on each FMU, and checking if this order is respected for each FMU when synchronizing on *dosync*. In a similar manner, interoperability between different versions of the FMU/FMI standard can be achieved by adapting the general template in Figure G.3b.



(a) The MA as a TA.



(b) The template of a single FMU.

Fig. G.3: The TAs used for importing and simulating FMUs in Uppaal.

5 Case Study

To demonstrate our approach we shall construct a model of three small houses sharing a single heating unit. Each house is composed of two rooms which each individually can be heated. Heat can be transferred between the two rooms, but as the houses are placed apart, heat is not transferred between the houses. The shared heating unit is only capable of heating a single room at any point in time – furthermore, it takes some time for the heating unit to be transferred from one house to another.

5.1 How to model one house

Each single house is modeled using OpenModelica and entirely composed of standard components. The entire model of a house can be seen in Figure G.4. Here each of the rooms have a heat-capacity of $2649600\frac{J}{K}$ and the wall between the rooms have a thermal conductivity of $6.4\frac{W}{K}$. At the same time, each of the rooms are affected by the outside temperature, here separated by slightly better insulated walls but with a larger surface-area with a thermal conductivity of $27.20\frac{W}{K}$ in total. As it can be seen from Figure G.4, the house receives three inputs; `in_room1`, `in_room2`, and `in_outside` for the influence of the heater in either of the rooms and the influence of the (ever changing) outside temperature. As outputs, the two temperature-converters `troom1` and `troom2` give us the variables `out_room1` and `out_room2`. We will not consider the inner dynamics of the room, but simply view a room as a single mass with a heat-capacity. OpenModelica supports the export of models as Co-Simulation FMUs – in the case of our model from Figure G.4, we get an FMU with three real-valued inputs `in_room1`, `in_room2` and `in_outside` (in Watts, Watts and degrees Celsius respectively) as well as the two outputs in degrees Celsius `out_room1` and `out_room2`.

5.2 A Controller as a Timed Automaton

The controller is implemented using Uppaal timed automata and can be seen in Figure G.5. Let us informally introduce the notation used for Uppaal timed automata; circles denote *locations* and arrows denote *edges*. The key feature of the timed automata is the *clock* construction; variables that all progress at the same rate and track time. Each location can be labeled with an *invariant*, colored *purple* – a predicate that must evaluate true when the TA is in the given location. Locations can also be marked with `U` or `C` for *urgency* and *committed urgency* – forcing immediacy (and prioritized immediacy), implying that time cannot elapse when the TA is in the given location. In Figure G.5 we can see that the Heating location is marked by a double-circle, indicating that it

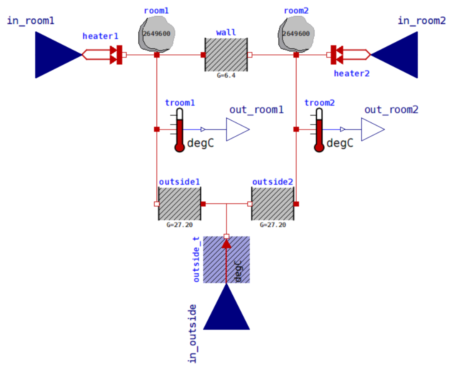


Fig. G.4: A single house with two rooms modeled using OpenModelica.

is the initial location. Edges can be labeled with `select`-statements (*yellow*), guards (*green*), updates (*blue*) and synchronizations (*turquoise*) where

- select-labels duplicate the single edge into multiple instances, one for each value possible in the declared type,
- guard-labels must evaluate to true for the edge to be admissible,
- update-labels can reset clocks and update variables declared using the `C`-language,
- synchronization-labels allows two or more TAs in the same system to move in unison when one is outputting (eg. $a!$) and multiple are inputting ($a?$) on the *channel* a.

Notice here that we only use the *broadcast* synchronization – for a given channel a, any TA is free to output $a!$ (if allowed by a guard), and all other TAs which can receive $a?$ must do so.

Aside from standard TA features, Uppaal supports a `C`-like language for complex constructions – including variable-declarations, function-calls and such.

The controller shown acts in a bang-bang manner; given a target temperature for each room (the tgt array), it randomly assigns the heating unit (the heat array) to any room with a temperature (the temp array) lower than or equal the target-temperature (right edge). If all rooms have reached their target-temperature, we can take the left edge, heating no room. When taking the right edge, if the controller decides to change the house being heated, a delay will incur between turning of the heater in one house and turning on the heater in another house. This is controlled by the variable wt which forces our controller to let time progress in the Wait-location (due to the combination of the invariant $t <= wt$ and the guard $t == wt$) – but only if the chosen house

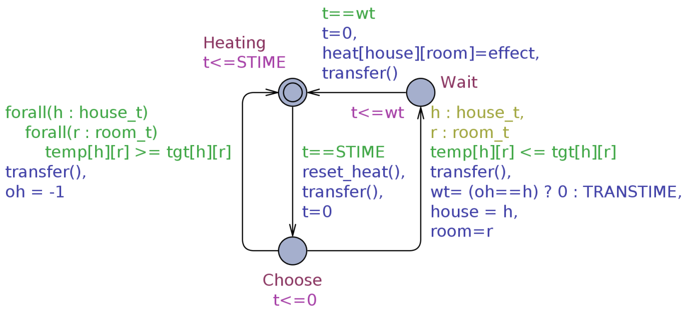


Fig. G.5: The bang-bang controller implemented in UPPAAL. The controller samples the values of the system each 60 time units and chooses at random to heat a room in a house where the measured temperature is lower than the set-temperature of the given room (right-hand side transition). If no such exist, no room is heated (left-hand side transition). Whenever a *room* and *house* is chosen, if the house differs from previous choice, the controller will wait TRANSTIME between heating one room and another.

differs, determined by the in-line if $wt = (oh == h)?0 : TRANSTIME$. Here one can also see the *select*-statement in action; an edge is created for each value in the type *house_t* ($h \in \{1, 2, 3\}$) and each value in *room_t* ($r \in \{1, 2\}$) – allowing for a concise description.

5.3 Composition of models

With the house and controller modeled, we can now focus on composing the entire system. Our system, as illustrated by Figure G.6, aside from three houses and the controller includes the weather – giving the outside temperature. For simulating the weather, we here assume a simple sinus-curve with a frequency of 24 hours, oscillating between 4 and 20 degrees Celsius. We can also see from Figure G.6 that the houses have no direct interaction with each other, and are only indirectly communicating via the temperatures they report and the choices of the controller.

Each of these individual components (even UPPAAL models [2]) can be exported as Co-Simulation FMU's – and thus composed into our system from Figure G.6 using existing tools. However, as our initial controller exhibits randomness, we would like to answer the questions: *What is the expected minimal and maximal temperatures of any room?* and: *If a room becomes too cold, will it become heated again within two hours?*

These questions both contain probabilistic measures over continuous time – and thus are not answerable by current tools. To remedy this, we below propose an extension of the Statistical Modelchecker UPPAAL SMC that allows for dynamically linking and calling arbitrary C-functions from inside the tool – a feature that facilitates Co-Simulation of FMU's.

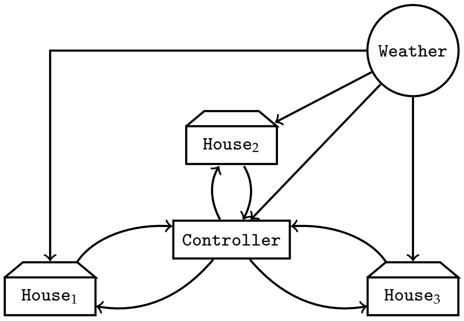


Fig. G.6: An informal overview of the composed system. The arrows indicate the flow of information.

6 experiments

This section presents the experiments that we perform on our case study when using the controller presented in Figure G.5. In this example, all the components are exported into FMUs and recomposed in Uppaal given the framework presented in Section 4 – including the controller. We shall consider the model presented in Section 5 and demonstrating three different features of Uppaal SMC; Simulation, Estimation and Statistical Modelchecking.

The composed model, FMUs and extended version of Uppaal SMC for 64-bit Linux-systems can be found at `http://people.cs.aau.dk/~pgj/UPPAAL_FMIFMU.zip`. Notice that the extended version of Uppaal SMC includes functionality to export timed automata as FMUs as presented by Bogolomov et al. [2]. We also provide a prototype tool for embedding FMUs into a Uppaal-importable timed automata for ease of use.

6.1 Simulation

In Figure G.7 one can observe the results of executing following query in Uppaal

```
simulate 1 [<=3600*24*7]{h1_output[0], h1_output[1],
                         h2_output[0], h2_output[1],
                         h3_output[0], h3_output[1]}
```

This proposition monitors the temperatures of each house over a period of seven days for a single simulation. As expected, initially our controller rapidly heats the coldest rooms in all of the houses. We can also observe that the temperature of the rooms after the first day is kept within a 19 to 22 degrees window in the current setting.

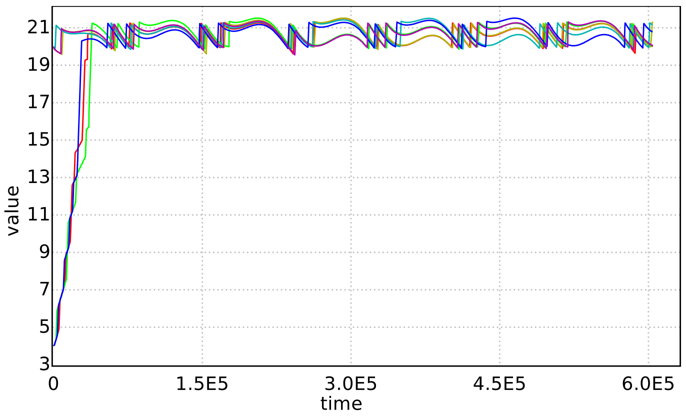


Fig. G.7: A single simulation of the system. Each color represents the temperature of a room. The starting temperature for the first room in each house is 20 degrees Celcius and 4 degrees Celsius for the second room.

6.2 Estimation

However, such a single simulation does not quantify over the probabilistic behavior of our system – one might be interested in knowing the expected maximal and minimal temperatures (disregarding the first day as our rooms are in an abnormal state).

Such a measure can be achieved by the following propositions.

```
E[<=3600∗24∗7;100] (min: mintemp())
E[<=3600∗24∗7;100] (max: maxtemp())
```

Here mintemp and maxtemp are functions defined in Uppaal C computing the minimum and maximum of the temperatures of all the rooms. The proposition computes the estimated minimal (resp. maximal) value of the expression over the course of a weeks simulation – and it does so on the basis of 100 simulations.

As we can see in Figure G.8, the performance of our constructed controller is fairly stable. Both the minimal and maximal temperature of any room stays (on average) within 18.9 degrees and 21.3 degrees Celsius. However, while the average peak temperature is stable at 21.3 degrees with less than a tenth of a degree span on the observed values, the minimal temperature over a week was observed to vary by more than a degree across all runs.

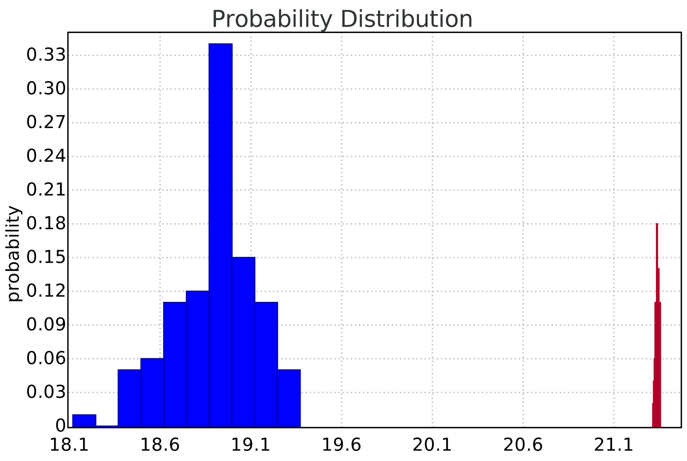


Fig. G.8: The super-imposed probability distributions of the expected minimal and maximal temperatures over all the rooms, over a week, disregarding the first day, based on 100 samples. The x-axis is given in degrees Celsius. Blue indicates minimal expectation while red indicates maximal. The mean of the minimal expectation is 18.9 while the mean of the maximal expectation is 21.3.

6.3 Statistical Modelcheking

While the estimation and simulation propositions provide some quantitative measure on different metrics of the system, we can use the statistical model checking features of Uppaal SMC to reason on the probabilistic behavior of our system.

Let us try to quantify the fairness of our controller; what is the probability that the temperature difference between the coldest room and the hottest room is greater than two degrees Celsius after the first day? This we can express as follows.

```
Pr[<=3600∗24∗7] (<>time > 3600∗24 &&
                        (maxtemp() − mintemp()) >= 2)
```

As we can observe in Figure G.9, there is a fairly high chance (more than 50%) that the temperature difference between the coldest and hottest room will grow beyond two degrees within a week.

We can also construct more complex propositions using the temporal MITL-logic. As we already know, the set-temperature cannot be respected in general in our example – however, we might be able to accept deviations as long as they are not for extended periods of time. We therefore construct the following proposition capturing: *after the first day, can it ever happen that*

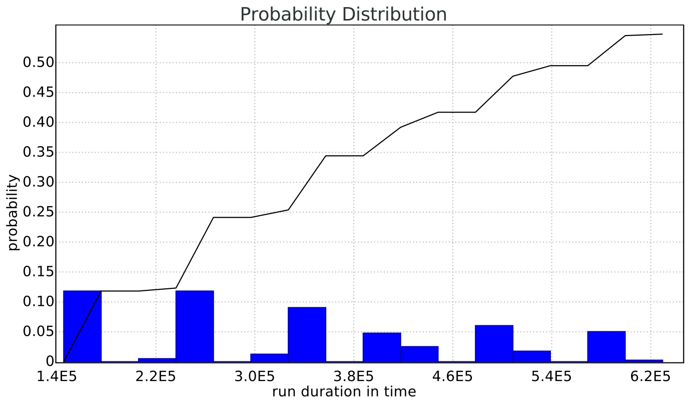


Fig. G.9: The probability distribution of having only a two-degree difference between the coldest and warmest room with the cumulative probability superimposed (black line). The measure is obtained using the statistical parameters $\epsilon = 0.05$ and $\alpha = 0.05$ – yielding 398 samples needed by Uppaal SMC. With a 95% confidence the true probability lies within $[0.50, 0.60]$ of having a greater than 2-degrees difference at any point within a week.

the second room of the first house has a temperature below 20 degrees for more than two hours at a given time.

$$\mathsf{Pr\ (<>\ [86400,604800](\ \ [\,]\,[0,7200]\ \ (h1\_output[1] < 20)))}$$

We can observe in Figure G.10 that this property is not surely satisfiable. Already in the second day of the simulation there is more than 10% chance that the controller will not be able to recover from a threshold-violation within two hours. We can also observe that this tendency seems to be repeating every night of our simulation.

7 Conclusion

In this paper we demonstrated the ease with which FMI-FMU models exported from other tools can be integrated into the setting of Uppaal SMC. We provide the flexibility of modeling different Master Algorithms (MAs) in a sound semantical framework facilitating probabilistic and temporal reasoning.

We believe that the possibility of using already existing domain models is essential in order to facilitate the use of formal methods in industrial systems, as the correct re-modeling of entire systems is a very time consuming exercise. The expressiveness of the modeling language available in Uppaal SMC allows for the efficient modeling of real-world scenarios with inherently uncertain and time-dependent behaviour – such as signal noise, human interaction and general natural phenomena. Thus this paper aims to make recent

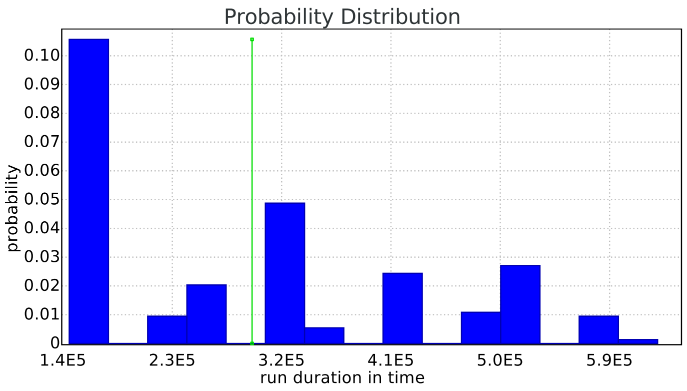


Fig. G.10: The probability distribution of not recovering from a threshold violation. The statistical parameters are $\epsilon = 0.05$ and $\alpha = 0.05$ – yielding 738 samples needed by UPPAAL SMC. With a 95% confidence the true probability lies within $[0.21, 0.31]$ of not recovering.

advances in statistical model checking and statistical validation of systems available for use in an industrial setting.

8 Future Work

Adding the possibility of calling arbitrary C-libraries during statistical simulation opens up for a number of new applications. This is true both for UPPAAL SMC and the related tool UPPAAL STRATEGO.

We consider the option of utilizing UPPAAL STRATEGO to perform controller synthesis for heterogeneous FMI-FMU systems as the most promising direction. This would make it much easier to generate optimized controllers using machine learning for systems being developed using other modeling tools. With the current approach all components in the system must be remodeled in UPPAAL STRATEGO.

Calling external C-libraries can also be applied within the domain of classical model checking in UPPAAL. Here there is the added restriction that the external calls can only be used to update the discrete variables and not the clock variables. On top of *statelessness* we also require that the external FMUs are strictly *deterministic* for the model checking to be semantically sound. For application of this in a non FMI-FMU setting see [4].

References

[1] FMI Standard Orginization, "FMI support in tools," http://fmi-standard.org/tools/.

[2] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikučionis, T. Strump, and S. Tripakis, "Co-simulation of hybrid systems with SpaceEx and Uppaal," in *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, no. 118. Linköping University Electronic Press, 2015, pp. 159–169.

[3] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, "Uppaal SMC tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, p. 397, 2015.

[4] F. Cassez, P. G. de Aledo, and P. G. Jensen, *WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with UPPAAL*. Cham: Springer International Publishing, 2017, pp. 560–577. [Online]. Available: https://doi.org/10.1007/978-3-319-63121-9_28

[5] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of FMUs for co-simulation," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, ser. EMSOFT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555754.2555756

[6] F. Cremona, M. Lohstroh, D. Broman, M. D. Natale, E. A. Lee, and S. Tripakis, "Step revision in hybrid co-simulation with FMI," in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*. IEEE, 2016, pp. 173–183. [Online]. Available: http://dx.doi.org/10.1109/MEMCOD.2016.7797762

[7] M. Pazold, S. Burhenne, J. Radon, S. Herkel, and F. Antretter, "Integration of Modelica models into an existing simulation software using FMI for co-simulation," in *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, no. 76. Linköping University Electronic Press; Linköpings universitet, 2012, pp. 949–954.

[8] N. Pedersen, T. Bojsen, J. Madsen, and M. Vejlgaard-Laursen, "FMI for co-simulation of embedded control software," in *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, no. 124. Linköping University Electronic Press, Linköpings universitet, 2016, pp. 70–77.

[9] S. Guermazi, S. Dhouib, A. Cuccuru, C. Letavernier, and S. Gérard, "Integration of UML models in FMI-based co-simulation," in *Proceedings of the Symposium on Theory of Modeling & Simulation*, ser. TMS-DEVS '16. San Diego, CA, USA: Society for Computer Simulation International, 2016, pp. 7:1–7:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2975389.2975396

[10] Object Management Group, "fUML," http://www.omg.org/spec/FUML/.

[11] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. B. Poulsen, A. Legay, and Z. Wang, "UPPAAL-SMC: Statistical model checking for priced timed automata," *arXiv preprint arXiv:1207.1272*, 2012.

[12] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards, "Statistical model checking for stochastic hybrid systems," *arXiv preprint arXiv:1208.3856*, 2012.

[13] P. E. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, and D. B. Poulsen, "Rewrite-based statistical model checking of WMTL." *RV*, vol. 7687, pp. 260–275, 2012.

[14] O. Gadyatskaya, R. R. Hansen, K. G. Larsen, A. Legay, M. C. Olesen, and D. B. Poulsen, "Modelling attack-defense trees using timed automata," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer International Publishing, 2016, pp. 35–50.

[15] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, and S. Sedwards, "Runtime verification of biological systems," *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pp. 388–404, 2012.

[16] J. H. Kim, A. Boudjadar, U. Nyman, M. Mikucionis, K. G. Larsen, and I. Lee, "Quantitative schedulability analysis of continuous probability tasks in a hierarchical context," in *Component-Based Software Engineering (CBSE), 2015 18th International ACM SIGSOFT Symposium on*. IEEE, 2015, pp. 91–100.

[17] D. Basile, F. Di Giandomenico, and S. Gnesi, "Statistical model checking of an energy-saving cyber-physical system in the railway domain," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1356–1363. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019824

[18] W. Ahmad, M. Jongerden, M. Stoelinga, and J. v. d. Pol, "Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata," in *2016 16th International Conference on Application of Concurrency to System Design (ACSD)*, June 2016, pp. 114–123.

[19] C. Jegourel, A. Legay, and S. Sedwards, "A platform for high performance statistical model checking–PLASMA," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 498–503, 2012.

[20] A. Arnold, M. Baleani, A. Ferrari, M. Marazza, V. Senni, A. Legay, J. Quilbeuf, and C. Etzien, "An application of SMC to continuous validation of heterogeneous systems," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS'16. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 76–85. [Online]. Available: http://dl.acm.org/citation.cfm?id=3021426.3021438

[21] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 2006, pp. 125–126.

[22] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, 2010, pp. 91–100.

[23] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang, "Statistical model checking for networks of priced timed automata," in *FORMATS*, ser. LNCS, vol. 6919, 2011, pp. 80–96.

[24] A. Pfeffer, "IBAL: A probabilistic rational programming language," in *In Proc. 17th IJCAI*. Morgan Kaufmann Publishers, 2001, pp. 733–740.

Paper H

WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with Uppaal

Franck Cassez, Pablo Gonzalez de Aledo and Peter Gjøl Jensen

The paper has been published in the
Proceedings of Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, LNCS Vol. 10460, pp. 560–570, 2017.

Abstract

We address the problem of computing the worst-case execution-time (WCET) of binary programs using a real-time model-checker. In our previous work, we introduced a fully automated and modular methodology to build a model (network of timed automata) that combined a binary program and the hardware to run the program on. Computing the WCET amounts to finding the longest path time-wise in this model, which can be done using a real-time model checker like UPPAAL.

In this work, we generalise the previous approach and we define a generic framework to support arbitrary binary language and hardware.

We have implemented our new approach in an extended version of UPPAAL, *called* WUPPAAL. *Experimental results using some standard benchmarks suite for WCET computation (from Mälardalen University) show that our technique is practical and promising.*

1 Introduction

Embedded real-time systems (ERTS) are composed of a set of periodic tasks (software) to run on a given architecture (hardware). The tasks are usually released at periodic time intervals. For safety-critical ERTS, each task must be completed by a deadline (relative to the release time). Checking whether a set of periodic tasks can be scheduled on a processor such that they always complete before their deadline is a *schedulability analysis*.

Tests for schedulability are based on the tasks' parameters, among them an upper bound for the *execution time* of each task. Over-estimating the execution time of a task may be safe but can also result in a set of tasks being declared non schedulable. This may lead to a choice of over-powered and over-expensive hardware.

With the ever increasing connectivity of many devices, ERTS are also subject to malicious attacks. Some of them can make use of time measurements to establish communication channels (*timing covert channel*): private information can be communicated or leaked to attackers by controlling/observing the time intervals between events (e.g., how long a computation takes).

It follows that tight bounds for the execution time of the tasks are instrumental to designing safe (schedulable), efficient and secure ERTS. Each task in an ERTS executes a program. The execution time of the program may depend on the input. The worst-case execution-time (WCET) of the program is the supremum of the execution times of the program over all the input. Computing the WCET for binary programs is a non-trivial task for at least two reasons:

- the set of input data may be very big and simulating the program over a subset of the input data only provides a lower bound of the worst-case execution-time;

- the hardware that runs the program is complex (pipelined architecture, caches) and it is effectively a *timed concurrent system* (e.g., the pipeline runs in parallel with the caches and they both have timing specifications.)

The WCET problem. Given a binary program P, some input data d and the hardware H, the *execution time* of P for the input d on H, denoted $\mathsf{Xtime}(P, d, H)$, is measured as the number of processor cycles between the beginning and end of P's computation for d (we assume P always terminates.) The *worst-case execution time (WCET)* of program P on hardware H, denoted $\mathsf{WCET}(P, H)$, is the supremum of the $\mathsf{Xtime}(P, d, H)$ for d ranging over the input data domain $\mathcal{D}$:

$$\mathsf{WCET}(P, H) = \sup_{d \in \mathcal{D}} \mathsf{Xtime}(P, d, H). \tag{H.1}$$

The WCET problem asks the following:

"Given P and H, compute $\mathsf{WCET}(P, H)$".

In general, the WCET problem is undecidable because otherwise we could solve the halting problem. However, for programs that always terminate and have a bounded number of paths, it is computable. Indeed the possible runs of the program can be represented by a finite tree (and there is a finite number states for the program and the hardware). This does not mean that the problem is tractable though: the (values of the) input data (e.g., an fixed-size array to be sorted) are usually unknown and the number of program paths to be explored may grow exponentially in the size of the program.

As mentioned before, programs run on increasingly complex architectures featuring multi-stage *pipelines* and fast memory components like *caches*: they both influence the WCET in a complicated manner. It is then a challenging problem to determine a precise WCET even for relatively small programs running on complex single-core architectures.

Computing a precise WCET for a given program is very hard and the WCET problem is usually re-stated as:

"Given P and H, compute a *tight upper bound* of $\mathsf{WCET}(P, H)$".

Tightness can be measured (see [1]) by comparing actual WCET to the ones computed using a particular method. In the sequel we use $\mathsf{WCET}(P, H)$ to denote an upper bound of the WCET for a given program.

Standard methods and tools for computing WCET. The survey article [2] provides an exhaustive presentation of WCET computation techniques and tools. A first set of methods based on simulations ([3–5]) are not suitable for safety-critical ERTS as they only provide lower bounds for the WCET.

A second set of methods rely on the construction of a Control Flow Graph (CFG) for the binary program to analyse, and the determination of *loop bounds*. The CFG is then annotated with some timing information about the cache misses/hits (some may/must analysis using abstract interpretation based techniques) and pipeline stalls to build a finite model of the system. A final paths analysis is carried out on this model e.g., using Integer Linear Programming (ILP). There are many implementations of this technique, the most prominent one is probably aiT [6, 7] which combines static analysis tools and ILP for computing WCET.

Real-time model checking based methods for computing WCET. Considering that (*i*) modern architectures are composed of *concurrent* components (the units of the different stages of the pipeline, the caches) and (*ii*) the *synchronisation* of these components depends on *timing constraints* (time to execute in one stage of the pipeline, time to fetch data from the cache), formal models like *timed automata* [8] and state-of-the-art *real-time model-checkers* like UPPAAL [9, 10] appear well-suited to address the WCET problem.

The use of network of timed automata (NTA) and the model-checker UPPAAL for computing WCET on pipelined processors with caches was first reported in [11, 12] where the METAMOC method is described. METAMOC consists in: 1) computing the CFG of a program, 2) composing this CFG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to computing the longest path (time-wise) in a NTA.

The previous framework is very elegant yet has some shortcomings: (1) METAMOC relies on a *value analysis* phase to compute the CFG but this may not terminate, (2) some programs cannot be analysed (if they contain register-indirect jumps), (3) manual annotations (loop bounds) is required on the binary program, and (4) the *unrolling* of loops is not safe for some cache replacement policies (FIFO). In our previous work [1, 13] we have reported some results on the computation of WCET using NTA that overcome the limitations of METAMOC: (1) we introduced an automatic method to compute a CFG and a reduced abstract program equivalent WCET-wise to the original program; (2) we designed detailed hardware formal models and (3) we evaluated the accuracy of our technique (comparison of measured execution times and the results of our analysis).

The technique we introduced in [1, 13] still has some drawbacks:

- the UPPAAL model (NTA) contains the CFG of the program and the machinery that is needed to simulate some instructions (written as functions in UPPAAL); some instructions (e.g., setting the overflow flag) are partially modelled because of the restricted expressiveness of the C-like operators supported by UPPAAL;

- the UPPAAL model (NTA) also contains components to explicitly model the caches as large arrays (of cache lines) which contributes a big part of the state of the system;
- as a result, we rely on UPPAAL to perform a lot of discrete computations which is not effective; moreover, the discrete state of the UPPAAL model contains a large amount of information (e.g., the full state of the caches) which also impacts the efficiency of the UPPAAL analysis engine.

Our contribution. Based on our previous work [1, 13], we propose three new contributions: (1) a generic framework for computing WCET which is language agnostic; (2) a new implementation of our framework based on an extended version UPPAAL and (3) a tool chain that combines our extended UPPAAL and an off-the-shelf binary program simulator (based on *gdb* [14]).

Outline of the paper. In Section 2 we recall how the WCET can be computed via model checking. The material in this section is based on [1, 13]. In Section 3, we introduce our new generic technique to compute the WCET of arbitrary programs. Examples are provided for an mono-processor pipelined ARM architecture. Section 4 provides details of the implementation of our technique, a tool chain architecture and some experimental results.

2 Computation of WCET via real-time model checking

In this section we introduce the basic concepts of program runs together with an abstract model of the hardware in order to compute the execution time of a sequence of program instructions.

Hardware. The hardware usually consists of a finite set $\mathcal{R}$ of registers, a multi-stage execution pipeline and caches (e.g., instruction and data caches). It typically supports a finite set of instructions, $\mathcal{I}$, e.g., `mov r1,r2` is an instruction that copies the contents of register r_2 into register r_1. The main memory component is a table of words of a given width 32-bit or 64-bit words. $\mathcal{M}$ is the (finite) set of main memory cells and we denote $\mathcal{D}$ the memory domain (e.g., 32-bit or 64-bit words). A memory state is thus a map from

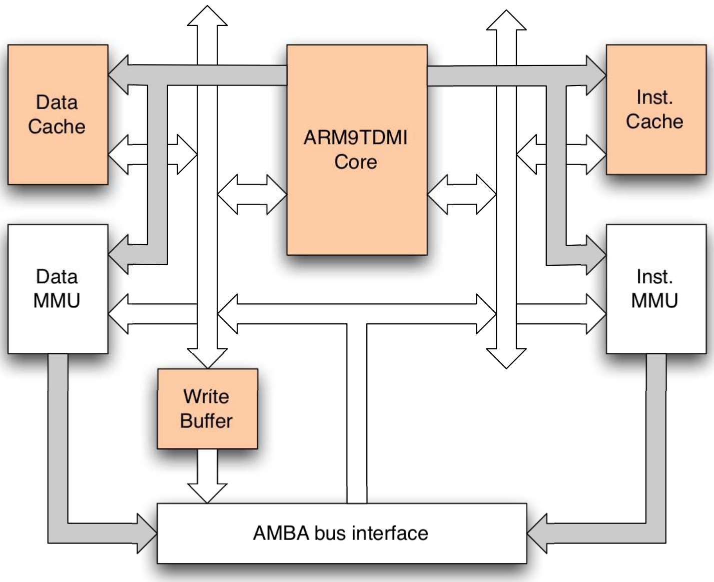


Fig. H.1: Simplified ARM920T architecture

$\mathcal{M}$ to $\mathcal{D}$. The caches and the pipeline are essential components of the hardware performance-wise but they are not necessary to define the semantics of the instructions. We omit them for now and will account for them later in this section. A *state* of the hardware is fully determined by the contents of the registers, the contents of the memory and the contents of the pipelines and caches. The hardware has a designated register, the *program counter* that points to the next instruction to process. An example of such an architecture, the ARM920T, is given in Fig. H.1. The orange blocks are the blocks we need to model to compute the execution time of program runs.

Program runs. A binary program is a map $P : \mathbb{P} \rightarrow \mathcal{I}$, with $\mathbb{P} \subseteq \mathcal{M}$, that associates with some memory locations $\ell \in \mathbb{P}$ an instruction. $P(\ell)$ is the instruction to be processed when the program counter of the hardware is at ℓ.

Given a program P, we let $\mathcal{L}_H(P) \subseteq \mathbb{P}^*$ be the set of *valid* executions of P on H. Actually we only require $\mathcal{L}_H(P)$ to over-approximate the set of feasible executions of the program P. To define this set we need to take into account the semantics of each instruction in $\mathcal{I}$, and the values of the registers of H and the memory state: this state is given by a valuation $\nu : \mathcal{R} \cup \mathcal{M} \rightarrow \mathcal{D}$. There are usually many different possible initial states of the hardware (e.g., a sorting program that sorts an array of k arbitrary elements, there are $\mathcal{D}^k$ initial possible input data).

Listing H.1: Prog1

```
1  int c_entry(int a, int b){
2          int c=1,i;
3          for (i = 0; i < 10; i++) {
4                  if(a < b){
5                          c *= 10;
6                  } else {
7                          c += 10;
8                  }
9          }
10         return c;
11 }
```

An example of a binary program compiled for the ARM920T is provided in Fig. H.2.a. This program can be obtained by compiling the C program Prog1 (Listing H.1). The Control Flow Graph (CFG) is given in Fig. H.2.b. The semantics of the program does not depend on the pipeline architecture nor on the caches: these components only impact the execution time of the program runs. However, to ensure that the WCET of each program is well-defined, we may assume that $\mathcal{L}_H(P)$ is finite. Otherwise it contains arbitrary long sequences (the alphabet $\mathbb{P}$ is finite) and the set of execution times is unbounded and the WCET is $+\infty$.

The set $\mathcal{L}_H(P)$ of program runs is finite but may contain more than one trace even if the program is deterministic. For instance in Prog1 (Listing H.1), the values of `a, b` are arbitrary at the beginning of the program because they are parameters of the function `c_entry`. This makes the test at line 4 a non-deterministic choice in our program over-approximation because the values of a and b are arbitrary (there are input parameters of the `c_entry` function). We can over-approximate the set of runs of this program by assuming that each time the test at line 4 is performed, the outcome is either true or false and both cases should be taken into account to compute the WCET. Notice that this is an over-approximation if $a < b$ evaluates to *True* (resp. *False*) the first time it must evaluate to *True* (resp. *False*) in the following iterations. Using this strategy we generate a super set of the set feasible runs of Prog1.

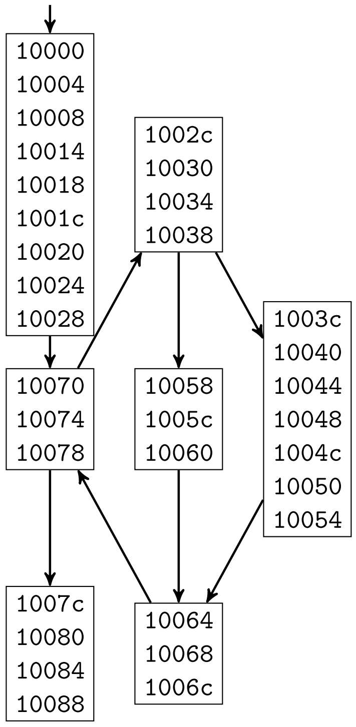
Execution time of a run. The execution time of a run $\sigma \in \mathbb{P}^*$ typically depends on the following factors:

- the time it takes for the instructions in σ to flow into the pipeline stages. This is usually non-trivial as the stages run in parallel. Moreover, the flow of instructions in the successive stages of the pipeline is governed by precedence rules: the execution of an instruction may require the availability of the result of another instruction which may temporarily block an instruction in a pipeline stage: this is known as a pipeline *stall*.
- the time it takes to fetch instructions and data from the caches and main

```
10000 <_Reset>:
10000: e1a00000 nop
10004: e59fd004 ldr sp, [pc, #4]
10008: eb000001 bl 10014 <c_entry>
1000c: eafffffe b 1000c <_Reset+0xc>
10010: 00011090 .word 0x00011090

10014 <c_entry>:
10014: e24dd010 sub sp, sp, #16
10018: e3a03001 mov r3, #1
1001c: e58d300c str r3, [sp, #12]
10020: e3a03000 mov r3, #0
10024: e58d3008 str r3, [sp, #8]
10028: ea000010 b 10070 <c_entry+0x5c>
1002c: e59d2004 ldr r2, [sp, #4]
10030: e59d3000 ldr r3, [sp]
10034: e1520003 cmp r2, r3
10038: aa000006 bge 10058 <c_entry+0x44>
1003c: e59d200c ldr r2, [sp, #12]
10040: e1a03002 mov r3, r2
10044: e1a03103 lsl r3, r3, #2
10048: e0833002 add r3, r3, r2
1004c: e1a03083 lsl r3, r3, #1
10050: e58d300c str r3, [sp, #12]
10054: ea000002 b 10064 <c_entry+0x50>
10058: e59d300c ldr r3, [sp, #12]
1005c: e283300a add r3, r3, #10
10060: e58d300c str r3, [sp, #12]
10064: e59d3008 ldr r3, [sp, #8]
10068: e2833001 add r3, r3, #1
1006c: e58d3008 str r3, [sp, #8]
10070: e59d3008 ldr r3, [sp, #8]
10074: e3530009 cmp r3, #9
10078: daffffeb ble 1002c <c_entry+0x18>
1007c: e59d300c ldr r3, [sp, #12]
10080: e1a00003 mov r0, r3
10084: e28dd010 add sp, sp, #16
10088: e12fff1e bx lr
```

(a) ARM binary for Prog1



(b) CFG of the binary program

Fig. H.2: ARM binary and corresponding CFG for Prog1

memory.
These memory transactions are usually performed in different pipeline stages and can be concurrent (e.g., an instruction in the *fetch* stage can be fetched from the instruction cache while another instruction in the *memory* stage performs some transactions with the data cache.)

In order to determine how long it takes for a run $\sigma \in \mathbb{P}^*$ to execute on the hardware H, it is sufficient to know:

- the processing time of each instruction in the different pipeline stages,
- the registers read from/written to by each instruction (to determine pipeline stalls),
- the status of the memory transactions for the instructions in σ: cache *hits* and *misses*.

Given a run $\rho \in \mathcal{L}_H(P)$, we can build an *annotated run* $\tilde{\rho}$ that contains the information required to fully determine the execution time of ρ on H. This extended run may capture the processing time of the instruction in each pipeline stage, the registers read from/written and the cache hits and misses. We let $\mathcal{L}^a_H(P)$ be the set of annotated runs associated with $\mathcal{L}_H(P)$.

For example, the following run $\rho = 10000.10004.10008.10014.10018$ in $\mathcal{L}_H(P)$ can be annotated with the time it takes to process each corresponding instruction in Prog1 (Fig. H.2.b), and whether fetching the instruction (from the instruction cache) will result in cache Hit or a cache Miss. Hence $\mathcal{L}^a_H(P)$ can be defined as sequences of pairs $(k, b) \in \mathbb{N} \times \mathbb{B}$ with the following meaning: k is the time it takes to process the instruction at p in the execution stage (E stage) of the pipeline; if b is true, fetching the instruction from the instruction cache results in a Hit otherise it is a Miss. This transformation will give an annotated run $\tilde{\rho} = (2, \mathit{True}).(1, \mathit{False}).(2, \mathit{True}).(2, \mathit{False}).(1, \mathit{False})$.

As mentioned earlier, it is noticeable that the hardware model needed to compute the execution time of a run is much simpler than the actual concrete hardware model: there is no need to model the actual processing unit (e.g., registers, memory) nor to perform actual computations (e.g., execute instructions).

Formal hardware model. As a sequence $\tilde{\rho} \in \mathcal{L}^a_H(P)$ contains enough information to compute the execution time of a program run $\rho \in \mathcal{L}_H(P)$ we can define an abstract model of the hardware as a timed automaton *transducer*, $Aut(H)$, that maps each $\tilde{\rho} \in \mathcal{L}^a_H(P)$ to a positive natural number $Aut(H)(\rho)$, which is the execution time of ρ on H. Hence the WCET of a program P on the hardware H is defined by:

$$\mathsf{WCET}(P, H) = \max_{\sigma \in \mathcal{L}^a_H(P)} Aut(H)(\sigma). \tag{H.2}$$

As $\mathcal{L}^a_H(P)$ over-approximates the set of program runs, we ensure that the value of the WCET we compute (equation (H.2)) is an upper bound of the actual WCET (this assumes that the hardware model $Aut(H)$ correctly models the timing behaviour of the hardware).

Modular computation of the WCET of a program. In practice to compute $\mathsf{WCET}(P, H)$ we need to provide a generator for $\mathcal{L}^a_H(P)$ and the model of the hardware $Aut(H)$. $\mathcal{L}^a_H(P)$ can be generated by a finite state automaton $Aut(P)$ (see [1, 13]). In general $\mathcal{L}^a_H(P)$ is a finite set of runs and can be defined by a finite computation tree.

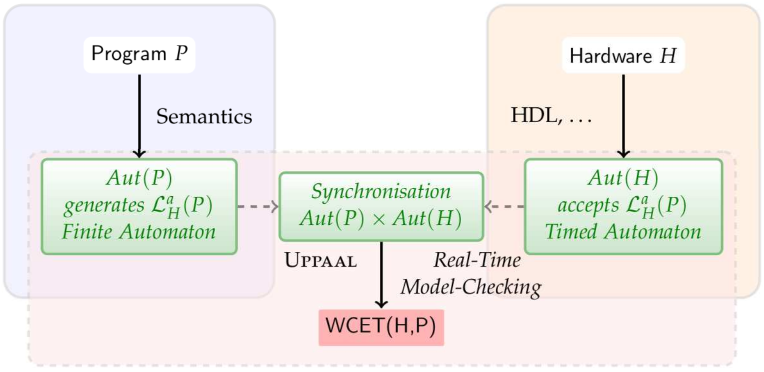

Program P
Semantics
Hardware H
HDL, ...
$Aut(P)$ generates $\mathcal{L}^a_H(P)$ Finite Automaton
Synchronisation $Aut(P) \times Aut(H)$
$Aut(H)$ accepts $\mathcal{L}^a_H(P)$ Timed Automaton
UPPAAL
Real-Time Model-Checking
WCET(H,P)

Fig. H.3: Modular Computation of WCET

In [1, 13] the modular computation of the WCET depicted in Fig. H.3 is fully implemented in UPPAAL as follows:

- a UPPAAL automaton, $Aut(P)$, that generates $\mathcal{L}^a_H(P)$ is computed based on the control flow graph of a program (for an ARM architecture.)
- the hardware model is provided for a given architecture (ARM920T). It comprises of a model of the pipeline and a model for the caches (complete model with the current state of the caches.) Notice that our method is robust against the so-called *timing anomalies* [15].
- the WCET can be computed either using a binary search or using UPPAAL *sup* operator.

This implementation has several drawbacks:

- the automaton $Aut(P)$ that generates $\mathcal{L}^a_H(P)$ is implemented using a limited C-like language. This is sometimes cumbersome and the semantics of some instructions had to be partially modelled (e.g., some bit-wise operations on registers). The result is that the UPPAAL model of the program which is a finite automaton, is hard to encode using Uppaal restricted set of C supported operations. This set was sufficient to model a large set of instructions of the ARM920T processor but may be too limited to model the semantics of more complex processors.

- the FIFO caches (instruction and data) are modelled precisely using an array to model the lines in the caches. The hardware model $Aut(H)$ contains the full state of the caches. This makes the discrete part of the state of the system $Aut(P) \times Aut(H)$ very large and impacts the efficiency of the model checking algorithm.

In the next section we describe how to overcome the previous limitations by having $\mathcal{L}^a_H(P)$ generated by a C-library outside Uppaal.

3 WUPPAAL

Program computation tree. In this section we assume that $\mathcal{L}^a_H(P)$ is available and represented as a finite tree $\mathsf{Tree}^a_H(P)$. This is based on the assumption that the number of iterations in the loops do not depend on an (arbitrary) input parameter. This is a usual assumption[1] in the WCET methods [2] as otherwise the WCET may be unbounded. Fig. H.4 shows a sub-tree of $\mathsf{Tree}^a_H(Prog1)$. We use a *sliced* version of the binary program when we build the tree. This sliced version is equivalent WCET-wise [1, 16] to the actual program. The components $\mathcal{M}^i$ in Fig. H.4 provides the values of the variables that are in the slice (some registers and other memory cells).
The following operations can be performed on $\mathsf{Tree}^a_H(P)$:

- $get\_init()$ returns the root of the tree $\mathsf{Tree}^a_H(P)$.
- $get\_next(n)$ returns the list of children of the node n (empty if n is a leaf).
- $hit\_ins(n)$ is a Boolean that indicates whether the instruction to be executed at n will result in a hit or a miss in the instruction cache.
- $get\_exec(n)$ returns the execution (in cycles) in the E stage of the pipeline for the instruction at n.

We refer to these operations as the *tree-API* in the sequel. The implementations of the Tree-API operations live outside Uppaal in the library named `libgdb2uppaal` (see Section 4 for the Wuppaal architecture). The Uppaal template in Fig. H.5 implements a full search on $\mathsf{Tree}^a_H(P)$ given the $get\_init()$ and $get\_next(n)$ functions; we assume each node of the tree has at most

[1]An exact test for this assumption does not exist as this problem is undecidable.

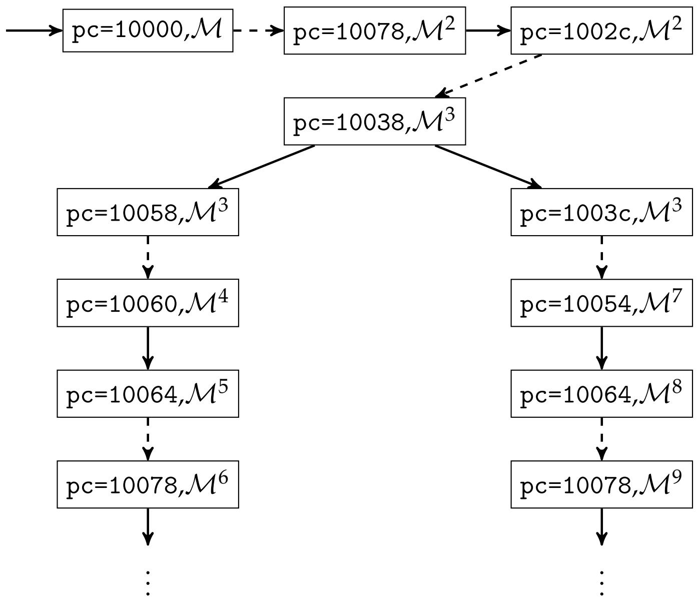


Fig. H.4: subtree of $\mathsf{Tree}^a_H(Prog1)$ where we let $\mathcal{M}$ be the memory tracked, and $r3 = 10$ in $\mathcal{M}^2$. Dashed arrows indicate sequences of deterministic instructions omitted for brevity.

2 children for the sake of simplicity. The Uppaal version of *get_init*() is `get_init(succ)` and fills in the vector `succ` with the pair $(get\_init(), \perp)$ ($\perp$ denotes the absence of node). Similarly $get\_next(n)$ is implemented by the function `get_next(n,succ)` and fills in the vector of integers `succ` with the children of n where `succ[0]` (resp. `succ[1]`) is the first (resp. second) child of n; the $\perp$ value is represented by a negative integer. The non-deterministic guarded choices in the template Program Automaton (Fig. H.5) push the children nodes to be processed to the first stage of the pipeline (see hardware model below). Each path through the template Program Automaton from the initial location (double circle) to the `END` location represents an annotated trace of $\mathcal{L}^a_H(P)$. When we model-check a safety property on this model, Uppaal generates all the traces in $\mathcal{L}^a_H(P)$.

Hardware specification. The hardware consists of a multi-stage execution pipeline and the caches (e.g., instruction and data caches). As a case-study we model an ARM920T 5-stage *execution pipeline*, the instruction cache and main memory components. The pipeline can execute concurrently the different stages (Fetch, Decode, Execute, Memory, Writeback) needed to fully process an instruction. An instruction is fetched (from the instruction cache) in stage F, decoding and operand register accesses occur in D, execution in E and if there are load/store instructions the memory accesses happen in M. The

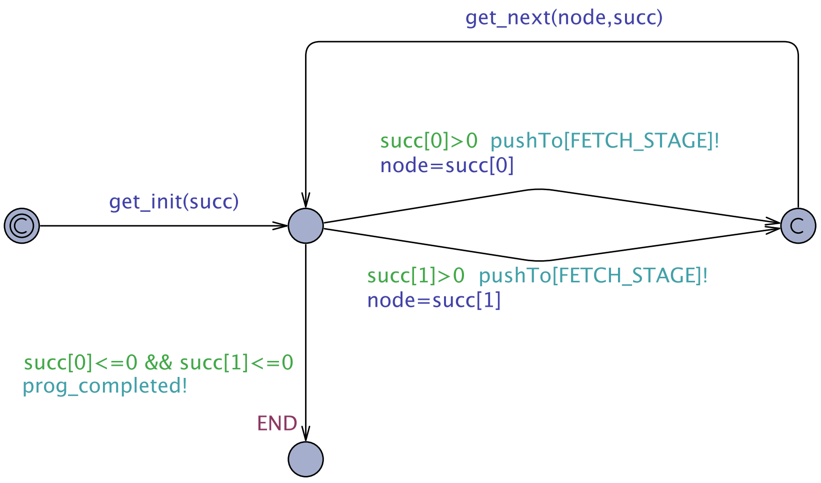


Fig. H.5: Program Automaton to enumerate $\mathcal{L}^a_H(P)$.

results are written back to registers in W. The (normal) flow of instructions in the pipeline is shown in Fig. H.6. This optimal flow may be slowed down when pipeline *stalls* occur: if the instruction $i+1$ needs a register written to by instruction i there will be a one cycle stall at cycle $j+3$ for instruction $i+1$; when the W stage is finished for instruction i, the E stage can begin for instruction $i+1$.

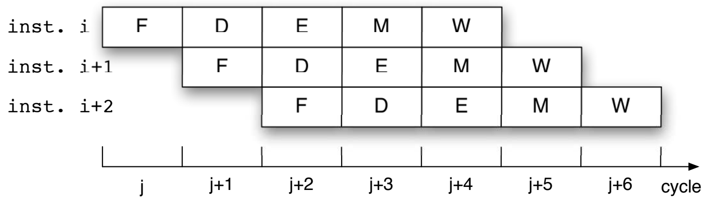


Fig. H.6: Pipeline of the ARM920T

Hardware abstract model. A formal model of the hardware for the ARM920T can be specified by a network of timed automata [1]. We provide here simpler models of the hardware because we factor out the actual state of the caches: to compute the execution time of a sequence of instructions we only need to know whether a transaction with a cache is a hit or a miss. This information is provided by each node in $\text{Tree}^a_H(P)$ ($\mathcal{L}^a_H(P)$) for a given program P. It can be computed by monitoring the addresses that are used on a given trace and using a model of the caches (e.g., number of lines, ways and FIFO replacement policy). In [17] we also proposed an abstraction/refinement scheme

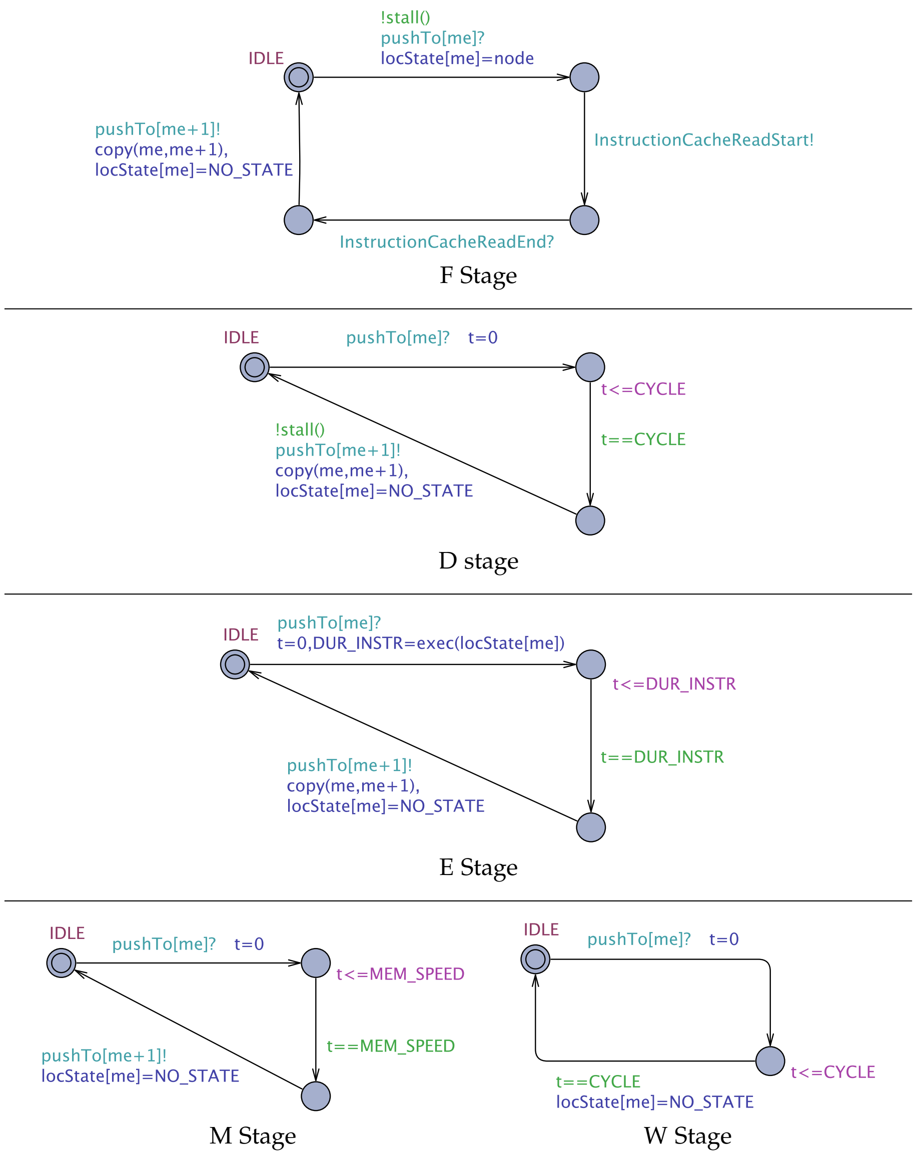


Fig. H.7: Timed Automata for F, D, E, M and W Stages (pipeline ARM920T).

to model the caches. For instance the 5-stage pipeline of the ARM920T can be specified by a network of 5 timed automata (see Fig. H.7) each of them modelling a single stage of the execution pipeline.

Each stage automaton has a unique identifier `me` (an integer). The values of this identifier for the templates (F, D, E, M, W) are respectively (0,1,2,3,4). This encodes the fact that the stages F, D, E, M, W are ordered: each node of $\mathsf{Tree}^{a}_{H}(P)$ flows from one stage k to the next $k+1$ when the `pushTo[k]` channels synchronise. For instance, the F-Stage template automaton is idle until the Program Automaton (Fig. H.5) pushes a node via the `pushTo[0]?` transition. It updates the local *state* of this stage 0 (`locState[0]=node`) where `node` is a (meta) variable used to retrieve the value sent by the Program Automaton that issues the `pushTo[0]!` command. The F stage template automaton then synchronises with the instruction cache (see Fig. H.7) to simulate the time it takes to fetch the instruction from the instruction cache.

The memory stage (M stage) assumes a constant time to read data from the data cache: each transaction takes `MEM_SPEED` cycles. We can easily model the data cache but for the sake of simplicity we use a simple version here. The other stages (D, E, M, W) are based on the same logic: they are idle until the previous stage pushes some information to them. The `copy(me,me+1)` commands transfers the information from stage `me` to stage `me+1`. When going back to the `IDLE` (initial) location, the local information of the templates are reset to the default value `NO_STATE` which indicates that the pipeline state is empty.

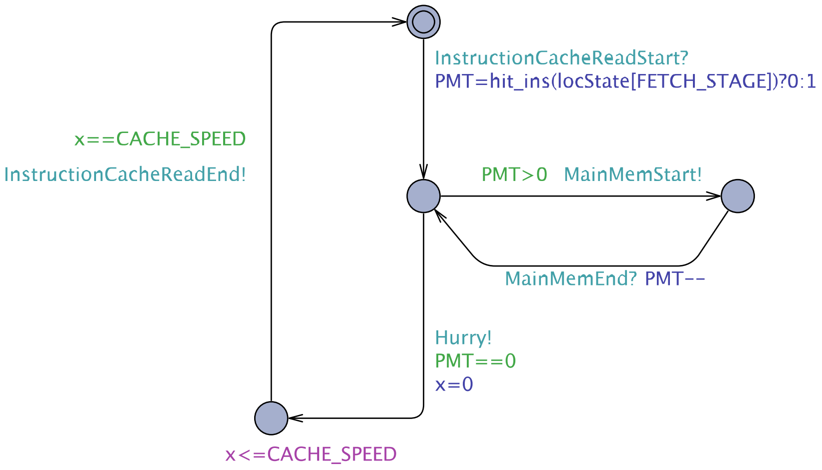


Fig. H.8: Instruction cache template.

The instruction cache is specified by the template timed automaton in Fig. H.8. The `PMT` variable holds the number of Pending Memory Transactions. This number is determined by the `hit_ins` function that can be retrieved from the annotated node in the tree.

Finally the main memory template simply simulates how long it takes to perform a transaction (read or write) with the main memory.

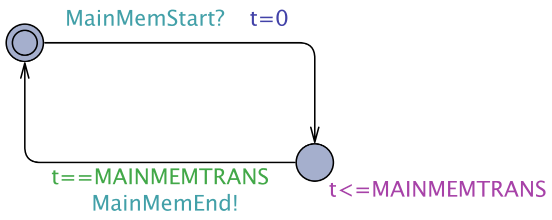


Fig. H.9: Main memory template.

4 Implementation and experimental results

Tool chain. Let us dwell on the tool chain we have constructed to demonstrate our methodology described in Section 3.

The tool-chain, visualized in Fig. H.10, is composed of five components:

- a `pre-analysis` module for constructing an annotated program that can be used to generate the program traces $\mathcal{L}_H(P)$; this step is developed in SCALA and uses some powerful Grammar and Language Processing packages KIAMA [18] and SBT-RATS! [19].
- `qemu` [20] to emulate the chosen hardware and enables us to compute the next state after executing a program instruction. As an example of usage, we set up the hardware to a given initial state (program counter and values of registers and stack), and with `qemu` we can compute the

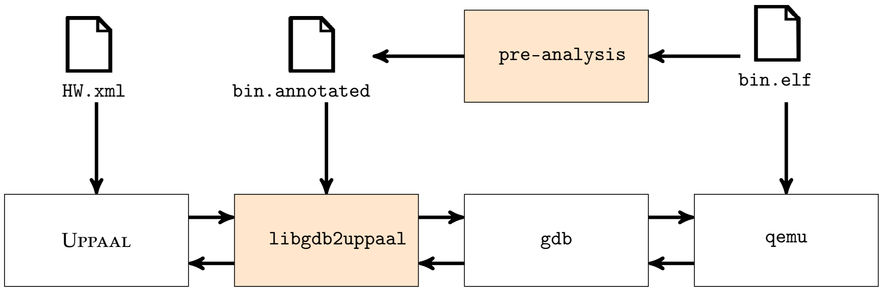


Fig. H.10: The tool chain of WUPPAAL. Orange blocks are the modules we implemented. Other blocks are existing modules.

effect of an instruction. What is communicated back (using `gdb`) is the next program counter and the next state of the registers and stack.

- `gdb` [14] for inspecting `qemu`,
- `libgdb2uppaal` to implement the tree-API given at the beginning of Section 3.
- a Timed Automaton model of the hardware `HW.xml` (an example is provided on Fig. H.7, page 219 for the pipelines and Fig. H.8, page 220 and Fig. H.9, page 221 for the main memory and instruction cache.)
- Uppaal for computing the worst-case execution time given a sequence of nodes using the Program Automaton template Fig. H.5, page 218, The Uppaal model uses an integer counter to identify the current state of the program. The `libgdb2uppaal` maintains a table that maps integers to actual program states (program counter, values of the registers and the stack). The `get_next` function in the Tree-API returns all the possible successors of a state as integers and updates the table that maps integers to program state (when a new state is encountered). The Program Automaton (Fig. H.5) will explore all the successor states.

Computing the WCET for a given binary program `bin.elf` using our framework is a two-stage process. In the first stage we compute an annotated program (e.g., a CFG and the set of variables needed to generate the annotated language $\mathcal{L}^a_H(P)$) by using `pre-analysis`.) In the second stage we use Uppaal to drive a search through the state-space, interfacing (by proxy of `gdb` and `libgdb2uppaal`) with the emulator of the hardware as described in Section 3. In the current model, we ignore the data cache but this is not a restriction as the caches can be added to the program state and modeled in the `libgdb2uppaal` library.

Support for other languages and hardware. The approach we propose is general enough to accommodate other languages and hardware. For instance, assume we want to use an x86 processor and the corresponding assembly language. What needs to be provided is a new `pre-analysis` module for this assembly language to construct the annotated program. The pre-analysis we have developed for the ARM assembly language is easy to re-use to build support for other languages.

We also need to provide an abstract model for the x86 hardware as a network of timed automata. The widgets we have proposed in Section 2 for the ARM920T pipeline can be adapted to build new formal models for an x86 platform (and of course new pipeline stages can be added if the architecture requires it).

Program	Loc	$\lvert\mathsf{Tree}^a_H(P)\rvert$	Time	WCET
duff	145	1750	4.51	61215
fibcall	48	553	2.91	19320
insertsort	84	7	2.09	210
janne_complex	67	360	3.21	12565
lcdnum	100	250	2.52	8715

Table H.1: The experimental results, time is given in seconds and includes startup overhead from initializing `gdb` and `qemu`. The *loc* measure is the number of lines of assembly.

Finally we need `qemu` (or an equivalent program) to support the emulation of the hardware. The general architecture we introduced in Fig. H.10 can be re-used as well as the the modular method depicted in Fig. H.3 to compute the WCET for programs running on the x86.

Results. We have experimented our technique using some of the standard benchmarks [21] from Mälardalen University, for computing WCET. As we can see in Table H.1, we are achieving a reasonable computation time (less than 5 seconds for all experiments), demonstrating the feasibility of our approach. We can also see that for all of the test-cases, the constructed trees are fairly small in size. In this paper we do not provide a thorough comparison with the actual measured execution times because we use simple models for the caches. The models used in [1] may be used in the future. The results in [1] demonstrated that our approach provides very accurate WCET and the new implementation should give similar results when precise models of the caches are used.

5 Conclusion

We have presented a method, based on timed automata and real-time model checking with Uppaal, to compute the WCET of binary programs. The method we designed is generic and can accommodate arbitrary hardware. The proposed tool chain allows us to achieve a modular approach to WCET-computation, reducing the overhead needed to support new binaries, and new architectures. To support different binaries we only have to provide `pre-analysis` with a different input. To support different processors, it is sufficient to provide a new hardware-model (`HW.xml`) and emulator (`qemu`).

Moreover, our technique does not rely on the computation of *loop bounds* or the assumption that the hardware is free of *timing anomalies*: this is one of the strengths of the model checking method. Another strength is that it generates a witness program trace that produces the WCET. Other interesting features of this approach includes its generality: we do not need to assume that the initial state of the caches is known. The only requirement is that the annotated language $\mathcal{L}^a_H(P)$ over-approximates the program behaviours.

Our technique is also general enough to be paired with *program refinement* techniques. As mentioned in Section 3 for Prog1, some traces in $\mathcal{L}_H(P)$ may not be feasible: if the first choice for the test $a < b$ is *True* (resp. *False*), the following test of the same condition must be *True* (resp. *False*). In that case we compute a refinement $R_1 \subseteq \mathcal{L}^a_H(P)$ of the annotated program to rule the spurious traces and analyse the refinement R_1. This can de done using the *trace abstraction approach* of [22, 23]. This enables us to define an *iterative* method to compute better and better over-approximations of the WCET and ensure that one witness trace exists.

Notice that this refinement also applies to the hardware model: we can start with a very simple model of the caches where every transaction is either a Hit or a Miss. Once a WCET is computed with Uppaal, we can check whether the witness trace is feasible in the program *and* in the caches. If the cache behaviour that is in the witness is spurious (infeasible) we can refine it as well. We have implemented a cache refinement technique in [17]. This enables us to get some control on the accuracy of the computation via model checking.

On another note, we can use our technique as a simulation based technique: the `bin.annotated` component in the tool chain Fig. H.10 can be replaced by a generator of traces. In this case we can only compute a lower bound for the WCET but we get access to the *statistical model checking* engine of Uppaal. This opens a new avenue to compute some probabilistic distributions of the WCET.

In addition, outsourcing the semantics of a binary program to a trusted emulation tool (`qemu`) eliminates errors that occurs when semantically translating binary programs into timed automata. As such a translation necessitates a very high level of detail, it can easily result in a state-space-explosion – even for simple architectures and programs. With our construction, knowledge of the hardware and static-analysis and abstraction refinement methods can be used to reduce the size of explored state-space.

References

[1] F. Cassez and J. Béchennec, "Timing analysis of binary programs with UPPAAL," in *13th International Conference on Application of Concurrency to System Design, ACSD 2013*. IEEE Computer Society, Jul. 2013, pp. 41–50.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller,

I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[3] Rapita Systems Ltd., "Rapita Systems for timing analysis of real-time embedded systems." http://www.rapitasystems.com/.

[4] G. Bernat, A. Colin, and S. M. Petters, "pWCET a Toolset for automatic Worst-Case Execution Time Analysis of Real-Time Embedded Programs," in *Proceedings of the 3rd Int. Workshop on WCET Analysis, Workshop of the Euromicro Conference on Real-Time Systems*, Porto, Portugal, 2003.

[5] B. Rieder, P. Puschner, and I. Wenzel, "Using Model Checking to Derive Loop Bounds of General Loops within ANSI-C Applications for Measurement Based WCET Analysis," in *6th Int. Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, Regensburg, Germany, 2008.

[6] AbsInt Angewandte Informatik, "aiT Worst-Case Execution Time Analyzers." http://www.absint.com/ait/.

[7] C. Ferdinand, R. Heckmann, and R. Wilhelm, "Analyzing the worst-case execution time by abstract interpretation of executable code," in *ASWSD*, ser. Lecture Notes in Computer Science, M. Broy, I. H. Krüger, and M. Meisinger, Eds., vol. 4147. Springer, 2004, pp. 1–14.

[8] R. Alur and D. Dill, "A theory of timed automata," *TCS*, vol. 126, no. 2, pp. 183–235, 1994.

[9] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Journal of Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1-2, pp. 134–152, 1997.

[10] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE Computer Society, 2006, pp. 125–126.

[11] A. E. Dalsgaard, M. C. Olesen, and M. Toft, "Modular execution time analysis using model checking," Master's thesis, Dpt. of Computer Science, Aalborg University, Denmark, 2009.

[12] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, "Metamoc: Modular execution time analysis using model checking," in *WCET*, ser. OASICS, B. Lisper, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 113–123.

[13] F. Cassez, "Timed Games for Computing WCET for Pipelined Processors with Caches," in *11th Int. Conf. on Application of Concurrency to System Design (ACSD'11)*. IEEE Comp. Soc., Jun. 2011, pp. 195–204.

[14] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 51, pp. 02 110–1301, 2002.

[15] F. Cassez, R. R. Hansen, and M. C. Olesen, "What is a timing anomaly?" in *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, ser. OASICS, vol. 23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 1–12.

[16] F. Cassez, "Timed Games for Computing Worst-Case Execution-Times," National ICT Australia, Research Report, Jun. 2010, 31 pages. Available from http://arxiv.org/abs/1006.1951.

[17] F. Cassez and P. G. de Aledo Marugán, "Timed automata for modelling caches and pipelines," in *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015.*, ser. EPTCS, R. J. van Glabbeek, J. F. Groote, and P. Höfner, Eds., vol. 196, 2015, B - International Conferences, pp. 37–45.

[18] A. M. Sloane, "Lightweight Language Processing in Kiama," in *Generative and Transformational Techniques in Software Engineering III*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2011, no. 6491, pp. 408–425. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-18023-1_12

[19] A. Sloane, F. Cassez, and S. Buckley, "The Sbt-rats parser generator plugin for scala (tool paper)," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, ser. SCALA 2016. New York, NY, USA: ACM, 2016, B - International Conferences, pp. 110–113. [Online]. Available: http://doi.acm.org/10.1145/2998392.3001580

[20] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[21] Mälardalen WCET Research Group, "WCET Project – Benchmarks." http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[22] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *SAS*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 69–85.

[23] ——, “Software model checking for people who love automata,” in *CAV*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 36–52.

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-178-1

AALBORG UNIVERSITY PRESS